C++ Ultra-Low Latency

Multithreading and Low-Level Optimizations

David Spuler

Aussie AI Labs

Published by Aussie AI Labs Pty Ltd, Adelaide, Australia.

https://www.aussieai.com

First published: July 2025.

About the Author

David Spuler is a C++ expert and serial technology entrepreneur who has combined his love of writing with AI technology in his latest venture: Aussie AI is a suite of tools for writing and editing, with a focus on fiction from short stories to full-length novels. His published works include three advanced C++ books (low latency, data structures, and safety), two generative AI LLM books, two CUDA C++ books, four non-fiction textbooks on C++ programming covering introductory and advanced C++ programming, efficiency and optimization, debugging and testing, and software development tools, and one application management book.

Other than writing, he's an avid AI researcher with a Ph.D. in Computer Science and decades of professional experience. Most recently, Dr. Spuler has been founding startups, including the current Aussie AI startup and multiple high-traffic website platforms with millions of monthly uniques, including an e-health startup acquired by HealthGrades, Inc. Prior roles in the corporate world have been as a software industry executive at BMC Software, M&A advisor, strategy consultant, patent expert, and prolific C++ coder with expertise in autonomous agents, compiler construction, internationalization, ontologies and AI/ML. Contact by email to `research@aussieai.com` or connect via LinkedIn.

About the Contributors

Michael Sharpe is an experienced technologist with expertise in AI/ML, cybersecurity, cloud architectures, compiler construction, and multiple programming languages. He is currently Senior Software Architect at PROS Inc., where he is a member of the Office of Technology focusing on developing and evangelizing AI. His AI expertise extends to monitoring/observability, devops/MLOps, ITSM, low-resource LLM inference, Retrieval Augmented Generation (RAG) and AI-based agents.

In a long R&D career, Michael has been coding C++ for almost 30 years, with prior roles at BMC Software, Attachmate (formerly NetIQ) and IT Involve. Michael has a Bachelor of Science with First Class Honors in Computer Science from James Cook University and holds several registered patents.

Cameron Gregory is a technology entrepreneur including as co-founder of fintech bond trading startup BQuotes (acquired by Moody's), a Senior Data Scientist focused on "big data" for hedge funds at fintech startup Advan Research Corporation, co-founder and Chief Technology Officer (CTO) of Trademark Vision with an AI-based image search product (acquired by Clarivate), and founder of several image creation companies including FlamingText.com, LogoNut, AddText, and Creator.me.

Cameron has been making code go fast since the 1990s at AT&T Bell Laboratories in New Jersey, is used to working with real-world data at scale, and is proficient in multiple programming languages, including C++, Java, and JavaScript. He holds a Bachelor of Science with First Class Honors in Computer Science from James Cook University.

Table of Contents

Preface

Why a Book on Ultra-Low Latency?

What a silly question! I mean, come on, why not? Everyone loves code that runs fast, and low latency programming is the epitome of all that. I've been optimizing C++ code for over 30 years now, and I wrote a book on C++ efficiency back in the 1990s. There's so much more in the newer versions of C++11 onwards, and that means even more ways to go faster!

Please Leave a Review

I hope you enjoy the book! Please consider leaving a review on the website where you purchased the book. Since few readers do this, each review is important to me, and I read them all personally.

Feedback and Contacts

Feedback from readers is welcome. Please feel free to tell us what you think of the book, the literature review, or our Aussie AI software. Contact us by email via `support@aussieai.com`.

Other Books by the Author

And some more with a particular focus on AI and fast LLM backends in C++:

- RAG Optimization: Accurate and Efficient LLM Applications
- Generative AI Applications: Planning, Design, and Implementation
- Generative AI in C++: Coding Transformers and LLMs

And if you're a fan of going super-parallel with GPU chips:

- CUDA C++ Optimization: Programming Faster GPU Kernels
- CUDA C++ Debugging: Safer GPU Kernels

About Aussie AI

Aussie AI is a platform for the development of consumer AI applications, with a special focus on AI-based writing and editing tools for fiction. Our premier applications offer an extensive range of reports and error checks for both fiction and non-fiction writing, from a full-length novel to a short report. Please try it out and let us know what you think: https://www.aussieai.com

Our AI Research

The primary focus of research at Aussie AI is on optimizing LLM inference algorithms (i.e., "running" the model after training or fine-tuning), and our research is toward the following aims:

- Fast on-device model inference algorithms, specifically for smartphones and AI PCs.
- Scaling inference algorithms to large volumes of requests.
- Efficient GPU inference algorithms (hardware acceleration).
- Non-GPU inference optimization algorithms (i.e., software methods).

Disclosure: Minimal AI Authorship

Despite my being involved in the AI industry, there was almost no AI engine usage in creating this book's text or its coding examples. Some text has been analyzed and reviewed using Aussie AI's editing tools, but not even one paragraph was auto-created by any generative AI engine. All of the CUDA C++ code is also human-written, without involvement of any AI coding copilot tools. I mean, who needs them?

However, AI was used in several ways. AI-assisted search tools, such as "Bing Chat with GPT-4", were very useful in brainstorming topics and researching some of the technical issues. The main cover art image was AI-generated, followed by human editing.

Disclaimers

Oh, and sometimes I'm being sarcastic, or making a joke, but it's hard to know when, because there's also a saying that "Truth is often said in jest!" Your AI engine certainly won't be able to help you sort out that conundrum.

Third-Party License Notices

Part I: Introduction to Low Latency

1. Low Latency Programming

What is Low Latency Programming?

Low latency programming is coding an algorithm so that it completes the task in the fastest time. In many cases, this is effectively the “user response time” or the “round-trip time” for a computation.

The main uses of low latency programming include:

- AI kernels — latency is the time between submitting a query, and starting to get the answer back.
- Embedded devices — the system must respond quickly, in real time (e.g., autonomous self-driving cars are a large embedded device).
- High-Frequency Trading (HFT) — latency is the time it takes to submit, execute, and complete a trade.
- Game engines — latency is ensuring that the characters or environment moves fast enough to be responsive to user inputs and to keep up with the frame rate.

The main programming language used for all of these low latency algorithms is my favorite one. I’ve written books on it!

C++ for Low Latency Programming

I’m a fan of C++, so you can take this with some grains of salt. The main programming languages for fast latency are:

- C++
- C
- Rust
- Assembly
- Hardware acceleration

The C++ is under the hood for most of the above cases. Most AI engines are Python at the top level, but C++ in the low-level kernels doing all those matrix multiplications. Game engines have historically been written in C++, at least for all the low-level stuff dealing with frame rates and 3D animation. Similarly, high-frequency trading is usually running in C++ at the bottom level.

You can also use C, which is the longstanding precursor to C++. The C programming language is obviously fast, as that was its key design point. C is not necessarily any faster than C++, so if you used only a C-like subset of C++, the two would be the same speed. However, using C does avoid the temptation to use some of the slower features that are available in the higher levels of C++.

Rust is a language that we refuse to talk about much, if you're any kind of C++ programmer. We'll only learn Rust if absolutely forced to do so. Apparently, Rust is also fast, and more memory safe than C++. But there's also Safe C++, profiles, hardened standard C++ libraries, and other variants of C++ to compete against Rust, so it's a whole big shemozzle.

Assembly language is faster than any of these higher-level languages. If you speak directly to the machine, there are various ways to speed up code. But it's a very low-level way of programming, and harder to learn, so the best method is to focus on optimizing only the main hot paths with assembly.

Hardware acceleration is the last option: just buy a better rig. Some of the main silicon to consider include:

- GPUs — AI, anyone? Data centers for cloud AI backends have the biggest GPUs. Or there's gaming desktop PCs with lower-end GPUs.
- FPGA — this is common in high-frequency trading and quant trading.

Plus, there's always that CPU to consider.

CPU versus GPU

With all this fuss about NVIDIA GPUs for AI, you might think that a GPU is what you need.

Not so fast!

The characteristics of AI engines and LLMs that make super-duper GPUs the mainstay of acceleration are:

- Huge numbers of arithmetic computations, and
- Highly parallelizable algorithms.

AI engines are number-crunching beasts, mostly doing vector dot product, matrix-vector and matrix-matrix multiplications. Here's the thing about GPUs:

GPUs have throughput not low latency!

You didn't hear this from me, but GPUs actually run *slow*. The clock speed of a high-end GPU is often around 1GHz, whereas a high-end gaming PC has a CPU clock speed of 4GHz or more. So, if you couldn't parallelize an algorithm, it would run slower on a GPU than a CPU. The key point is this:

Throughput + Parallelization = Low Latency

AI algorithms are very amenable to parallelization. And GPUs have high throughput of parallel operations on all those cores. A multi-core CPU has a dozen cores, but a big GPU can have thousands. Hence, it crunches data in parallel with high throughput, and the net effect is that a GPU runs AI algorithms with very low latency.

Which explains why those data center GPUs cost more than your car!

AI Engines

As already examined above, AI engines have an algorithm structure that's perfect for GPUs. The basic point about AI inference algorithms include:

- Process all of that data, and
- Hardly any alternate pathways.

Yes, for every word that an LLM throws out, it has to crunch through multiplication operations on every single number in the model. And that's just for one word. This process repeats over and over, and there are very few ways to shortcut the arithmetic without losing accuracy.

In fact, there are two main phases in AI inference with different latency characteristics:

- Prompt processing phase ("prefill") — process all the input tokens.
- Decoding phase — emit the answer words.

The prefill phase has these characteristics:

- Parallel processing of every token in the input text.
- Compute-bound (because of that parallelization).

The decoding phase has opposite characteristics:

- Sequential algorithm (one output token at a time, called "autoregression").
- Memory-bound (loading the entire model each time).

In fact, the situation with compute-bound vs memory-bound is a little more nuanced in the decoding phase. It's memory-bound overall, but the sub-components of a layer have slightly different characteristics during the decoding phase:

- Attention module — memory-bound (model weights and KV cache data)
- Feed-forward network (FFN) — compute-bound (model weights)

Hence, the double sequence of two matrix multiplications is an intense computation in the FFN (also known as the Multi-Layer Perceptron or MLP). However, the attention mechanism is memory-bound, mainly from needing to load the "KV Cache" data and less so from needing model weights. This characteristic affects the overall status of the decoding phase more than FFN computations, causing the decoding phase to be memory-bound overall.

High-Frequency Trading

HFT and quant trading algorithms have some peculiar characteristics with regard to low latency programming. The main point to consider about the algorithm is there are conceptually two main code pathways:

- Cold path — analyze, but don't trade.
- Hot path — trigger a trade.

And here's the weird part:

- Cold path — very common.
- Hot path — rarely executed.

This is different from most other types of algorithms, where the main path to optimize is also the common path. For non-HFT apps, you crank up the profiler, run the whole app, find where it's spinning the most CPU cycles, and optimize that code.

Not for HFT!

For HFT, the hot path is the rare path. Despite what people think from the name, the algorithm is actually trading much less frequently than it decides *not* to trade. Once the analysis decides to trigger a trade, that is a very hot path, and every step must execute with minimal latency. There are multiple actions for a single trade from initiation, network submission, processing, and finalization. The whole round-trip latency of this trade execution hot path is hyper-critical.

But the analysis part of the HFT code can't be slow either. The hot path is not really just "trade" and should really be thought of as "analyze-and-trade." We can't have the analysis phase running too slow, or we'll miss the opportunity to trade. So, it's true that once a trade is triggered, that pathway must be super hot, but the analysis phase cannot be a laggard either. Optimizing the analysis phase has an element like normal performance profiling of code hot spots, along with extra network latency issues from the data gathering phase via exchange network connections.

Intentional Slowness

Although latency is important, it is worth noting that there are times to go slow. The main point is that humans are slower than computers, so the algorithm often has to slow down the user interface so that the human user can keep up.

Game engines are a particular example of this. The computer has to move all of the game characters and enemies fast, yes, but also not too fast. The speed of the user's character cannot be too fast for the inputs of the user. Similarly, the enemies cannot move too fast, or the user will not be able to evade them or destroy them.

AI engines don't really have this problem in text-to-text classic LLMs. The only concern for excessive speed is not having the text output too fast to be read. However, other types of AI models such as speech and video need to have outputs in the right speed range, not too slow, but also not too fast.

High-frequency trading is one area that doesn't really have a "human in the loop." There's no real need to intentionally slow down the execution of a trade. However, there is a need to avoid over-trading too fast, lest the algorithm fail to notice some sort of failure. But this is the less common case than simply needing to go as fast as possible. Reporting a trade back to a supervising user is the last step, and not in the critical path.

2. Hardware Acceleration

Why Hardware Acceleration?

Hardware acceleration has come a long way since the Intel 8087 floating-point coprocessor in 1980. Every CPU now comes with builtin floating-point operations, and even opcode instructions that perform complex mathematics like exponentials and logarithms in hardware.

Parallelizing computations is now where the action's hot in AI, which needs many vectors and matrices running in parallel mode (i.e., tensor computations). The most powerful parallel computations are GPUs which can chomp through a continuous stream of data in parallel.

GPUs are not the only type of hardware acceleration. Even without GPUs, typical CPUs have multi-core and multi-thread parallelism. You can even do small-vector parallel instructions in the CPUs using special SIMD opcode instructions. For example, x86 CPUs have SIMD accessible via C++ AVX intrinsic functions, and Apple M1/M2/M3 chips support Arm Neon for parallelism.

Types of Hardware Acceleration

There are lots of different types of silicon chips available for your AI engine. The basic types of hardware chips are:

- Central Processing Unit (CPU)
- Graphics Processing Unit (GPU)
- Tensor Processing Unit (TPU)
- Application-Specific Integrated Circuit (ASIC)
- Field-Programmable Gate Array (FPGA)

If you want to build your own hardware, and there are plenty of research papers that do, then use an FPGA or ASIC. Even prior to the AI hype, ASICs proved their value in the Bitcoin mining boom, and FPGAs were commonly behind Azure, AWS and GCP, particularly around security/data protection.

If you're not a hardware designer, you're more likely to want the main CPU and GPU options. CPU parallelism is via AVX or Arm Neon SIMD instructions. For GPUs, you're most likely looking at an NVIDIA chip, from the P100 at the low end to the H100 at the top end (with V100 or A100 in the middle). Alternatively, the TPU is a special custom AI chip created by Google, and is in the same vein as other GPU chips.

CPU Hardware Acceleration

Many of the major CPU chips offer builtin hardware acceleration.

- x86/x64 (Intel/AMD) — AVX SIMD instructions (including AVX-2, AVX-512, and AVX-10)
- ARM — Neon SIMD instructions (e.g., on phones)
- Apple M1/M2/M3 — ARM Neon, Apple AMX instructions, or Apple Neural Engine (ANE).

AVX intrinsics can be used on x86/x64 platforms with Microsoft MSVS or GCC/Clang C++ compilers to run CPU data crunching in parallel.

The ARM Neon is a hardware acceleration processor. ARM-based architectures can run the Neon acceleration opcodes, which are 128-bit SIMD instructions that can parallelize both integer and floating-point computations. At the time of writing, the current version is based on Armv8. Notably, the Apple iPhone platform is based on ARM silicon and has Neon acceleration capabilities.

Apple M1/M2/M3 chips are based on ARM, so the ARM Neon acceleration works. There are also some additional Apple-specific hardware accelerations such as Apple AMX and Apple Neural Engine (ANE).

Detecting CPU Acceleration in C++

It is tricky to check what CPU or GPU support is available to your C++ program. There are different methods for Microsoft Visual Studio, GCC, and Apple.

Preprocessor macros. The first point is that you can only use preprocessor macros if the "single platform" assumption is true. In other words, if you're building on the single platform that you're running in production, or you're a developer toying with an engine on your own single PC.

In such cases, you can detect the current build environment using preprocessor macros. For example, if you're on a Windows box with Microsoft Visual Studio, you might try this:

```
#if __AVX2__
   // ... supports AVX2
#endif
```

This works fine if you are running C++ on your developer desktop machine, and don't plan to run it anywhere else. But this doesn't check runtime availability for AVX2 on your user's machine. It's only testing whether you've got the AVX2 architecture flag enabled in your compiler on your build machine. Hence, it's misleading and although you can do a `#if` or `#ifdef` test for whatever macro you like, it isn't very helpful for multi-platform programming.

Run-time platform testing. The `#if` method can check the major platforms that you're compiling on (e.g., Windows vs Linux vs Apple), but you cannot check what exact CPU you are running on, or what capabilities it has. The preprocessor macros are processed at compile-time, and can only detect what machine it's building on. This isn't very useful in determining if your user is running the code on a CPU that supports SIMD instructions, or if their box has a GPU on it.

Instead, you need to call C++ intrinsics to detect CPU capabilities at runtime. On the x86/x64 architecture this intrinsic uses the "`CPUID`" opcode. The C++ intrinsic calls differ by compile platform:

- MSVS: `__cpuid` or `__cpuidex` (superseding `__isa_available` in `<isa_availability.h>`)
- GCC/Clang: `__builtin_cpu_supports` or `__builtin_cpu_is` functions.

GPU Hardware Acceleration

For the sticklers, AI GPU chips are not really a "GPU" because that stands for "Graphics Processing Unit," and they aren't used for "Graphics" in an AI architecture (even when creating an image). In fact, they're really a General-Purpose GPU (GPGPU), but nothing other than AI matters in the tech industry, so we stole the acronym from the gamers.

GPUs are great big SIMD processors. There is a huge range of vectorized opcodes available for any given GPU. Each GPU isn't just one vectorized stack of silicon, but has lots of separate "cores" that process AI workloads (e.g., FMA) in parallel.

Each core runs a SIMD operation such as a small matrix multiply or FMA in a single GPU clock cycle. For example, a V100 "Tensor Core" can do a 4x4x4 half-precision (16-bit) matrix/tensor multiply in a cycle, which is a lot more advanced than a typical vectorized operation.

Hence, it's a parallel-of-parallel architecture with:

(a) all the GPU cores running in parallel, and

(b) each core doing vectorized SIMD operations.

The chips also have their own GPU RAM (sometimes called "VRAM") and there are also multiple levels of caches of that RAM. If you're assessing the specs of a GPU, consider:

- FLOPs throughput
- Cores
- RAM
- Clock speed
- Memory bandwidth rate
- Cooling systems (they run hot!)

GPU Pricing. If you're looking at renting a data center GPU, NVIDIA is top of the list for AI computations. The choice between a P100, V100, A100, or H100 is important. To run a version of Meta Llama2, a V100 is workable for that, but with not many instances per box. As of writing, pricing for a V100 runs below a buck an hour and there are 730 hours in a month, so you can do the math (pricing varies with vendors anyway). You can get an A100 for more than a buck an hour, and a H100 for roughly double that (for now). On the horizon, NVIDIA has a H200 coming mid-2024 with about 141GB RAM (versus the H100's 80GB), and also the B100 in late 2024 for even higher performance than a H200.

You can also buy a GPU chip outright from your private jet using your diamond-encrusted phone. Okay, so that's a bit of an exaggeration. Pricing changes, as of writing, you're looking at around ten grand for a V100 by itself, but pricing is higher if it's part of a "system" on a motherboard or a box (and this confuses ChatGPT if you ask it about GPU pricing).

Another option is used GPUs, which are cheaper, but might have spent their prior life in a Bitcoin-mining forced labor camp. GPUs do have a limited lifetime and can overheat with partial or total failure.

Detecting GPU Support in C++

Detecting GPU capabilities that are available at runtime in C++ is even more problematic than detecting CPU accelerators or SIMD instructions. The available options for GPU detection include:

- NVIDIA CUDA C++ compiler (nvcc)
- AMD ROCm
- Microsoft DirectML (DirectX)
- Apple Metal
- Vulkan (vkEnumeratePhysicalDevices, vkGetPhysicalDeviceProperties)
- Low-level GPU shader APIs

NVIDIA requires CUDA code to be compiled with their nvcc compiler, and the compiler itself has builtin mechanisms for testing the GPU capabilities. That results of that output can be used to set `#define` options within the C++ code too. The compiler also comes with some builtin defines.

GPU detection is not just determining if a GPU is available. More detail will typically be required, down to "is feature X available" or "which implementation for feature X is available." For example, NVIDIA has a "GPU Architecture" and a "GPU Feature List" to test for capabilities.

Assembly Language versus Intrinsics

Assembly language, or "assembler", is the low-level language for CPU machine instructions. Like C++, it is still a symbolic human-readable language, but unlike C++, it translates mostly one-to-one to machine code instructions. The syntax for assembler is much simpler than C++, and more obscure, but it's also very, very fast.

When to use assembly language. The first question to ask yourself before writing assembler in C++ is whether you need to. The use of assembler should only be considered for the most bottlenecking parts of the code, like deep inside the inner loops of a GEMM kernel. Otherwise, you're probably micro-optimizing something that's not that critical.

Another question is whether to use "intrinsics" instead of assembler. Each C++ compiler has literally hundreds of builtin low-level functions called "intrinsics" that are very fast, probably because the compiler-writers have written them in assembler. There are also lots of intrinsics to use for GPU operations and CPU SIMD extensions such as AVX-512.

There are also intrinsics that map one-to-one to x86 CPU instruction codes on that platform. Look through the long list of C++ intrinsics for your compiler platform to see if there's one that does what you need.

The use of intrinsics is via a standard C++ function call syntax, so you don't need to learn assembler to take advantage of them.

Assembly language syntax: Here are some of the basics of assembly language coding:

- Assembly code filenames usually have a suffix of "`.S`", "`.s`" or "`.asm`" (but don't need to).
- Inline assembly inside C++ could be added to base code via the inline statement `asm("string")`, `__asm__("string")`, or the alternative syntax of `asm { tokens }`, depending on the compiler.
- Comments start with a semicolon (but you can also use C++ comments for inline assembly).
- One line per assembly statement.
- Jump or branch labels need a suffix colon and should start a line (either their own line or before a statement).

Disadvantages of Assembly Language: The reason that the C language came into being was to overcome some of the low-level problems of programming in assembly or machine code. There are various downsides to using assembly language:

- Non-portable — assembly is specific to the CPU and many features depend on CPU sub-releases.
- Pitfalls — and you thought C++ had troubles.
- Maintainability — few programmers know assembly.
- Complexity — everything's harder at the low-level.

To summarize, there's only two reasons to use assembly language: speed and security (of your job).

Inline Assembly Language

Most C++ compilers support features allowing you to specify assembly language sequences in the middle of a C++ program, which is called "inline assembly language."

You don't need to put assembler into a separate code file, because you can use assembly language directives inside C++ sequences.

The directive to use to introduce an assembly language statement into C++ is somewhat compiler-dependent, but the whole concept of assembly language is platform-dependent anyway!

The "`asm`" expression is the official C++ standard version. This is like a function call with a semicolon ending it.

The `asm` statement contains the assembly language statements inside a large string constant, ending with a newline escape (i.e., "`\n`"), inside round brackets.

Multiple assembly commands can be merged by putting two string literals on subsequent lines and using the adjacent string literal concatenation feature of C++.

```
asm (
  " ; ... instructions\n" // C++ Comment
  " ; ... more instructions\n"
);
```

The Microsoft style is different, with a code block rather than an expression. You don't need to put the assembly statements inside a string literal, and you don't need the "`\n`" newline escapes, either.

The basic syntax looks like this:

```
__asm {
   ; ... instructions // C++ comment
}
```

This is the Gnu and Clang style with "__asm__" as a C++ function-like expression (similar to "asm"):

```
    __asm__ (
      " ; ... instructions\n" // C++ Comment
    );
```

Mixing C++ and assembly language is not something recommended just for fun. Not only do you need to know the assembly statements and all about the CPU registers, but you'll need to know about function calling conventions (e.g., __cdecl vs __stdcall vs __thiscall) and name mangling in C++.

Which actually sounds kind of fun.

3. System Optimizations

Optimizing the Whole System

There's a lot of moving pieces in a whole low latency system. Optimizing them is an elegant dance, where each component plays a part. There's no single answer to this, and it's an ongoing process of continuous efficiency improvement.

Instead, you need to look at all the different components in your hardware and software stack. At each layer, you need to consider:

- Better or newer components
- Configurations of the component
- Optimized programming

The good news is that optimizations to most of the layers are cumulative. You can optimize the hardware, the C++ software, and the network, and get a triple benefit.

Low Latency System Components

If you want to build a low latency system, here are some of the basic components in your stack. A single system may include:

- Hardware — CPU, GPU, FPGA, NPU, etc.
- Memory (RAM)
- Disk storage — e.g., SSD (NVMe)
- Network interface card (NIC)

The software stack looks like:

- Operating system kernel layer — Linux or bust.
- System software tools and services/daemons
- Compiler tools and system libraries
- Middleware software (e.g., Kafka)
- API/SDK clients (e.g., HFT exchange connectivity)
- Application software (your C++!)

Beyond the single system, there are various other system components:

- Network switch or router devices
- Network connections (e.g., wired, optical, microwave)
- Load balancer devices
- Backup storage devices

Combining Multithreading and SIMD CPU Instructions

You can double up! C++ multithreading software can be interleaved with CPU SIMD instructions as an optimized optimization. It's totally allowed, and you can even put it on your resume. The idea is basically this structure:

- Multithreading architecture — higher-level CPU parallelization.
- SIMD instructions — lower-level CPU vectorization.

Some of the main CPU architectures with SIMD parallelization include:

- AVX — x86 (e.g., Intel or AMD)
- ARM Neon — iOS/Mac

Note that there are variants of each of these SIMD architectures, available on different chips. For example, AVX has AVX-1 (128 bits), AVX-2 (256 bits), AVX-512 (you can figure it out), and AVX-10 (1024 bits).

Combining Multithreading and GPU Vectorization

If you've sold your car to buy a PC that has both a fast CPU and a high-end NVIDIA GPU, there's good news to think about while you ride the bus: both chips run at the same time. (Wow, in parallel, even.)

In fact, there are "threads" on both the CPU and the GPU. However, C++ CPU threads are much higher-level than the CUDA C++ threads on the GPU. The idea is:

- CPU threads — big chunks of work.
- GPU threads — very granular computations.

On the GPU, you might code vector addition with one GPU thread doing the addition in every element of the vector, up to the 1024 maximum. And if your vector has more than 1024 elements, you'd split it up into 1024 sub-sections and use "striding" to do it. But I digress.

CPU threads are not that granular, and you use them to do large chunks of work, not just one addition instruction. For example, you might have threads pulling incoming user requests off the queue, and a thread might handle the entire user request, perhaps launching some other threads on the CPU or GPU to do so.

There are some parallels (haha) between coding CPU and GPU threads:

- Both types of threads have a call stack.
- Both have "global" or "shared" memory to use across threads.
- Overhead of thread launches and exits are a thing for both CPU and GPU threads.

Note that there's also a new generation of "mini-GPUs" called a Neural Processing Unit (NPU), which aren't as powerful as a fully-fledged GPU. NPUs tend to be used on "AI Phones" and other "edge" devices, which aren't as powerful as a PC. Most of the comments about combining C++ multithreading and GPU coding also apply to the use of NPUs, except a little slower.

Going for the Triple-Double

You can even triple up your parallelism:

- Multithreading/multicore (CPU)
- SIMD instructions (CPU)
- GPU vectorization

Is there a way to do up to four levels of parallelism in just one C++ program? Yes, of course:

- Linux processes (parallelism at a higher level).
- Networking communications (the NIC runs parallel, too).

There are some optimizations of those things, too.

Advanced Linux O/S Optimizations

It doesn't end with the C++ code. There are other things you can optimize in the Linux O/S:

- Process priorities — be nice and turn yours up to eleven!
- Linux system processes — turn off the various Linux system processes that you don't need (so they don't compete for CPU time).
- Kernel bypass — direct NIC manipulations.
- Overlap communications and compute — e.g., PCIe bus GPU-to-memory upload/download.
- Networking technologies — e.g., TcpDirect and Onload; RDMA.
- Linux kernel optimizations — e.g., network buffer settings; disable writes that update the "file access date" when reading a file.
- Linux system settings — ensure you don't have accounting or security modes on.

There's also some other items on the advanced menu:

- Overclock your CPU (and the GPU)
- Buy a bigger box
- Get a faster SSD disk (e.g., NVMe)
- Assembly language
- Microwave communications
- FPGA

There's always more, but I've run out of room in your web browser.

Serving and Deployment Optimizations

If your software has to do multiple things at once, such as talk to multiple people (users), or communicate with multiple stock trading platforms, then there are many system-level practicalities that affect latency.

If your low latency application is a public-facing consumer website, there are a number of deployment issues to scale up to a lot of users.

Some of the issues to consider in the whole end-to-end latency of a request going through a system include:

- DNS lookup time
- Connection handshake time
- SSL time
- Load balancing
- Round-robin DNS
- Parallelization (multiple servers)
- Utility servers
- Caching (e.g., etags)
- CDNs
- Database lookup time
- Database indexes
- Keep-warm server architectures

Building a low-latency system is more than just coding up some C++. You have to put together a bunch of off-the-shelf components.

Network Optimization

If your algorithm has to talk between two computers, there's a network in between. The time spent sending data across the wire and back is a key part of the latency. Faster algorithms need to optimize the network traffic. The main techniques for network optimization include:

- Higher bandwidth network connections
- Advanced network protocols
- Compressing network data sizes
- Spreading bandwidth usage over time (avoiding peaks)
- Overlapping computation and communications
- Direct access to peripherals (local and remote)
- Direct access to memory (local and remote)
- Sticky sessions (keeps session data local)
- Sharing cache data between multiple servers

There's a whole book that needs to be written about network optimizations! Should be done by Tuesday.

References

These are some good articles on optimizing an entire AI LLM backend system:

1. Character.AI, June 20, 2024, *Optimizing AI Inference at Character.AI*, https://research.character.ai/optimizing-inference/
2. Apple, June 2024, *Introducing Apple's On-Device and Server Foundation Models*, https://machinelearning.apple.com/research/introducing-apple-foundation-models
3. Together AI, Nov 13, 2023, *Announcing Together Inference Engine – the fastest inference available*, https://www.together.ai/blog/together-inference-engine-v1
4. Ryan Lucchese, Niki Birkner, Yaron Hagai, Virginia Adams, August 13, 2024, *A practitioner's guide to testing and running large GPU clusters for training generative AI models*, Together AI, https://www.together.ai/blog/a-practitioners-guide-to-testing-and-running-large-gpu-clusters-for-training-generative-ai-models

And these are some references about entire HFT system optimizations:

1. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
2. Sebastien Donadio, Sourav Ghosh, Romain Rossier, 17 June, 2022, *Developing High-Frequency Trading Systems: Learn how to implement high-frequency trading from scratch with C++ or Java basics*, https://www.amazon.com/Developing-High-Frequency-Trading-Systems-high-frequency-ebook/dp/B09ZV5L2T7/
3. Irene Aldridge, April 2013, Wiley, *High-Frequency Trading: A Practical Guide to Algorithmic Strategies and Trading Systems*, https://www.amazon.com/High-Frequency-Trading-Practical-Algorithmic-Strategies-ebook/dp/B00B0H9S5K

Part II: HFT & Algo Trading

4. Trading Engine Components

Overview of Trading Engines

What high-level components does a trading system need? At the top-most level, the sequence in a HFT trading engine goes something like this:

- Ingest market data (from exchange)
- Analyze this data
- Decide whether to trade
- Submit trade order (to exchange)
- Risk management and reconciliation

Note that if you work at an exchange, the requirements are reversed, and with an even higher need for mission critical accuracy, but are also somewhat simpler at a high-level:

- Receive orders from trader clients
- Matching engine to trigger trade execution
- Send market data feed out to many traders

Software Components

Each of those components is just a small matter of more coding. We do that in C++, of course!

Market data ingestion components include:

- Exchange network protocol libraries (e.g., UDP multicast client).
- Market data normalization (converting into your own order objects).
- Snapshot synchronization support

The central management of orders, sometimes called an Order Management System (OMS), includes various software components:

- Order book data structure
- Crossing detection
- Matching engine (simulated)

A more advanced order book may also have:

- Order rule support (e.g., FIFO vs pro rata)
- Special exchange status (e.g., pre-open)
- Iceberg order detection
- Market microstructure analysis
- Generalized order book (beyond limit orders)

Algos are not a big deal in trading, and they're all published in open-source repositories on the internet (I'm kidding). Here are some of the things you'll need for your algo engine:

- Trading decision engine
- Method or API to specify algos
- Common primitives library for algos
- Transpiler from algo language to C++ (also known as "front-office programmers")
- Declarative algo specification (yeah, right, dream on!)

Also, since many algo strategies may require data from more than one financial instrument to make a decision to trade, we get to:

- Multi-instrument order book (multiple assets)
- Distributed order book management (multiple exchanges)
- Multi-instrument multi-exchange algo primitives

Trade submission to the exchange is a whole separate ball of wax:

- Rate limiter (emulated)
- Pre-trade risk management
- Trade submission
- Trade status management
- Bad trade handling

The risk management and regulatory compliance boffins want their pound of flesh:

- Risk management engine (pre-trade and post-trade)
- Position tracking
- Compliance tracking engine
- Logging
- Accounting reconciliation

All of the above stuff is just for live trading. There are some offline C++ capabilities that you'll need as well:

- Backtesting — testing if those algo devs can code worth anything.
- Historical data storage — for backtesting or stress testing.
- Synthetic data generation — robots do testing better.
- Compliance reporting — sending it all off somewhere.

If I've counted correctly, that's 33 major software components. So, no matter how great you are at C++, don't expect to knock out a new trading engine prototype over the weekend.

Low-Level Infrastructure

To implement any of these financial components, you need some helper infrastructure in both hardware and C++ software to make it run fast. Some of the C++ code libraries and templates you might need for speed include:

- TCP and UDP libraries — advanced network socket programming.
- Disk and file storage — low latency I/O with memory-mapped files and a custom filesystem.
- Statistical primitives — going way beyond the average.

Looking more specifically at C++ multithreaded coding components that aid in C++ optimizations:

- Thread pools — low-latency multithreading.
- Memory pools — preallocation of memory for objects.
- In-memory logging — save that data, but not yet.
- In-memory counters — tracking statistics for performance and accounting.
- Lock-free queues — forwarding data very quickly along the execution pipeline of components.

Hardware is important, of course, arguably even moreso than the C++ software:

- Co-located Linux servers (proximity access versus connection via microwave or fiber optics).
- Network switches
- NICs (in servers, with kernel bypass capabilities)
- FPGA servers
- GPU parallelization
- Quantum computing (it's coming!)

The need to communicate over the network also adds:

- UDP multicast for market data feed ingestion.
- Kernel bypass (hardware support in NIC hardware, plus C++ code).
- Inter-site network connectivity (around the world we go!).
- Connectivity to GPU server farm (e.g., for ML models).
- Out-of-band networking — host network connections for administration.

Safety, too! Here are some of the custom C++ libraries you may use in low-latency programming:

- Custom assertions — removeable in production code.
- Self-testing code — ditto for `#if DEBUG`.
- Testing harness — unit tests are someone else's job.
- Stress testing — using historical data feeds or synthetic data.
- Timing and benchmarking — proving your code is faster than the intern's.
- Error handling — not using standard C++ exceptions.

There are also various DevOps requirements:

- Instrumentation — tracing for errors and performance analysis.
- Monitoring — watch out for red flashing lights.
- Hardware failure detection — e.g., GPU burn.
- Kill switch — if it's redder than red.

That's another 23 low-level software components to add to the 33 C++ higher-level components in the prior section, and about five major hardware categories. Building all that should take you two weeks!

References

1. Sourav Ghosh, July 2023, *Building Low Latency Applications with C++*, Packt Publishing, https://www.amazon.com/dp/1837639353
2. Charles Cooper, 2021, *Fast implementation of an ITCH order book*, https://github.com/charles-cooper/itch-order-book
3. Ronak Chatterjee, 2023, *A high frequency trading system built with C++: High performance, low latency high frequency trading system written from scratch in C++*, https://github.com/nyarosu/hft
4. Ranjan (Man of steel), 2025 (updated), *Live High-Frequency Trading Exchange Engine*, https://github.com/ranjan2829/Live-High-Frequency-Trading-Exchange-Engine
5. Amitava Biswas, Aug 18, 2023, *Designing Low Latency High Performance Order Matching Engine*, https://medium.com/@amitava.webwork/designing-low-latency-high-performance-order-matching-engine-a07bd58594f4

5. Hotpath Optimizations

What is Hotpath Optimization?

Hotpath optimization is a multithreading C++ optimizations in HFT whereby the most important code is prioritized and super-optimized. Whereas the traditional "hotpath" in C++ code is the most heavily executed code, in HFT the hotpath is a rarely executed sequence of high importance (i.e., submitting the trade). Hence, optimizing the hotpath can mean different things:

- Profiling the most heavily executed code (traditional C++ code).
- Running the GPU profilers on CUDA C++ kernels (for AI applications).
- Optimizing the rare but most important pathway (HFT applications).

Using the various C++ profiler tools won't help you much in HFT hotpath optimization. Well, actually it can, but only if you have a way to modify the code in test mode so that it *always* runs the hotpath sequence. But take care with this idea, as maybe it shouldn't really submit a thousand live buy orders to the exchange when it's running under Valgrind in the nightly build.

Hotpath Optimization Techniques

The idea with hotpath examination is to put every single instruction under the microscope. Especially for HFT, every microsecond counts, and there are many ways to squeeze out more speed.

There are two main categories of optimizations:

- Concurrency optimizations — multithreading-related code changes.
- General C++ optimizations — all of the rest!

With regard to multithreading, the hotpath should not be subjected to any of the delays that can beset a single thread.

Some of the methods for speedup include:

- CPU pinning — give the hot thread its own core (completely avoids context switching)
- Don't use locking on the hotpath (as much as possible) via lock-free coding, read-only data structures or lock-free algorithms.
- Cache warming via prefetching of shared data needed by the hotpath.
- Keep the cache warm all the way down into the NIC.
- Use a lock-free queue data structure to avoid contention issues.
- Use custom thread pools with only preallocated memory block pools.

Other than multithreading changes, there's another few hundred other types of C++ optimizations to consider. There are a number of chapters about this.

But here's a smattering of some interesting techniques:

- Hoist code out of the hotpath by using precomputation.
- Remove slowpaths by deferring handling of error checks.
- Maximize compile-time computation (e.g., `constexpr`, TMP if you must).
- Don't allocate or free memory; use only preallocated memory or global memory.
- Use in-memory databases for any significant amounts of incoming data.
- Review data de-serialization and serialization costs.
- Don't log, or defer logging to the end, or write to an in-memory logger.
- Replace every `if` statement with branchless coding tricks.
- Examine every code statement in the entire hotpath (even at assembly level).

Odds are high that you'll find something to improve, no matter how many times you look at the same stretch of code.

Network Optimizations

In a network-heavy application, such as HFT, there is a lot of importance in the speed of networking. Many of the main optimizations are hardware issues:

- Custom NIC
- Fast switches

Note that there can be multiple networks attached to one server:

- Public network
- Private network

The purpose of a private network is to send messages only between your servers and any administrative consoles. This private or "out-of-band" network can be used for things like:

- Monitoring and administration messages
- Sending data between servers (e.g., quotes data in HFT, or KV cache data in LLM inference).

Although hardware and its related network connections are critical, let's not forget the software. Your C++ code needs to talk to the network, to receive incoming data and to emit actions (e.g., a trade in HFT) Network-related optimizations to the C++ code in the hotpath can include:

- Use kernel bypass to custom NICs for fast networking.
- Keep the client network connection warm (method depends on the API).
- Use custom wrappers for TCP and UDP network processing.

For extra speed, you may need to wrap or re-implement the TCP and UDP code. Some of the default algorithms for networking introduce some minor safety checks and other delays, which interfere with your need for speed. Linux socket programming can be a lot of fun. I can remember coding a custom version for the `select` primitive, which is loads of bitmask fiddling.

Core Pinning

Core pinning is a multithreading optimization where a thread is "pinned" to one of the cores to give it higher priority. This means that important thread that runs the hotpath can have guaranteed CPU availability, rather than waiting for the default thread scheduling algorithms. Hence, it can be a solution to avoid lock contention worries for the main hotpath thread.

Core pinning is also called "thread affinity" and has multiple other names (e.g., "processor affinity" or "CPU affinity" or "CPU pinning"), but if you hear the words "pinning" or "affinity" in relation to threads, this is it.

Pinning has other meanings in related architectures. There's a higher-level type of pinning whereby whole processes or applications are pinned to a CPU core by the operating system, rather than just a single thread, which isn't quite the same thing. Note also that CUDA C++ has another type of "pinned memory" for GPUs, but that's a memory upload optimization rather than a compute improvement.

The other side of core pinning is that you obviously don't pin the less important threads. All lower-priority threads have fewer cores available, and are downgraded.

On Windows, you can set up a process-level CPU pinning for an application via the GUI. On Linux, there is a "`taskset`" command that allows running a program with core pinning. Both Windows and Linux have non-standard system calls that can set up pinning for either a process or a thread. Programmatic C++ APIs on Linux are:

- Pinning processes — `sched_setaffinity`
- Pinning threads — `pthread_setaffinity_np`

On Windows, these are the C++ APIs:

- Pinning processes — `SetProcessAffinityMask`
- Pinning threads — `SetThreadAffinityMask`

The use of core pinning is a very powerful type of hotpath optimization. The main pathways are super-optimized because:

- No context switches
- Highest priority execution
- Guaranteed core availability (no delay)

In-Memory Logging

The last thing you want is for your hotpath to block waiting for log messages to get written to disk. Hence, your options for logging include:

- Don't log!
- Buy a faster SSD disk (what's next after NVMe?)
- Store log messages in memory

Not logging messages can be an option in some cases. This refers to tracing and debugging messages, that aren't business-critical. Some of the approaches to disable logging include:

- Compiling-out unimportant tracing.
- Disabling logging but having it still in the code.

If you use a Boolean control flag to enable or disable logging, this can be an effective solution. On the other hand, you can have a lot of these:

```
if (g_debug) {
    // Log a message
}
```

These can be inefficient on a hotpath for two reasons:

- Cost of testing the global flag multiple times, and
- Extra branches that interfere with branch prediction.

On the other hand, this can be very flexible and the above costs can be a small price to pay in some applications. You can enable or disable the global flag based on:

- Command-line options (i.e., add a "`-debug`" setting).
- Sending a SIGUSR1 signal to the process (toggle debug mode).

Whatever the choice regarding debug or tracing-related logging, you can't avoid business-related logging. For example, a HFT applications needs to track any actual trades sent, and update any risk management applications.

The solution for this is to use an in-memory logging C++ class. The features that you need include:

- Log messages are copied to an in-memory queue (preferably lock-free).
- A separate log-writing class pulls these messages off the queue.
- The thread writing log messages to disk is low-priority in the background.

In this way, you can have quite extensive logging, but the critical path is all in memory, and the slower writing to disk is deferred to a background task that can run in the quiet periods.

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Machinet, March 13, 2024, *How to optimize C++ code for use in high-frequency trading algorithms?* https://www.machinet.net/tutorial-eng/optimize-cpp-code-high-frequency-trading-algorithms
3. Ivan Eduardo Guerra, October 19, 2024, *C++ Design Patterns for Low Latency Applications Including High Frequency Trading*, https://programmador.com/series/notes/cpp-design-patterns-for-low-latency-apps/
4. Dung Le, Aug 13, 2020, *Optimizations for C++ multi-threaded programming*, https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
5. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
6. Eli Bendersky, January 17, 2016, *C++11 threads, affinity and hyperthreading*, https://eli.thegreenplace.net/2016/c11-threads-affinity-and-hyperthreading/
7. Bytefreaks, 23 November 2016, *C/C++: Set Affinity to process thread – Example Code 3*, https://bytefreaks.net/programming-2/c/cc-set-affinity-to-process-thread-example-code

6. Orders

Orders

Orders are the main low-level data for both an exchange and a trading engine. There are different types of orders:

- Limit orders (buy or sell at a set price)
- Stop orders (buy/sell when a price is reached)
- Market orders (accept the current price)
- Pegged orders

There are two "sides" for every trade, and they have separate orders: buy orders and sell orders. The prices of bid orders are called "bids" and for sell orders the term is "asks." Or you can just call them "buys" and "sells" if you prefer.

Processing of orders differs for exchanges versus trading engines, but there are many overlaps. An exchange processes a large volume of incoming orders submitted by its trading clients. A trading engine can both submit its own orders, and also tracks other orders via the exchange's market data feed.

When an order is submitted to an exchange, there are two possibilities:

- Immediate execution
- No matches yet

An immediately executing order is called an "aggressive order" and the exchange should execute it against other waiting orders. If no matches, the order goes in the "order book" for later execution, in which case it is called a "passive order".

The most common scenario is that an exchange has a number of passive orders for each stock or other financial instrument. The prices of buys are lower than the sell prices. All of the buy orders have prices lower than the lowest sell price; otherwise a trade would occur. Hence, an important measure is the highest buy price ("max-buy" or "max-bid") and the lowest selling price ("min-sell" or "min-ask"). The difference between the max-buy and min-sell is the "spread" or "bid-ask spread."

Market Data Feeds

For order processing by a trading engine, the exchange provides a data feed about all of the buy and sell orders that are currently active on the exchange for each financial instrument. the types of incoming market data feed messages includes:

- Add new order (buy or sell)
- Modify order
- Cancel order
- Trade execution
- Other administrative messages

There are a variety of low-level assumptions about the structure of the incoming data from a market data feed:

- New orders ("adds") should have a unique id that we haven't seen before.
- Modify or cancel orders should have an id that was previously added.
- Modify orders can only do certain types of updates (e.g., price changes are usually disallowed and instead the order must cancel and re-submit at the new price level).

Note that these assumptions can fail, and quite often, because a market data feed is usually implemented as a UDP multicast protocol. UDP is an unreliable protocol, where network messages can be lost or delayed, but it's faster than TCP.

To maintain accuracy, there's a separate data feed that's slower and more reliable, usually over TCP. There's a whole phase of "snapshot synchronization" logic for the entire order book, which uses this separate data feed, and it's very different from the basic UDP market data feed of incoming transactions.

Order Objects

Orders are usually represented as a simple structure. They are a frequently-used object, and the last thing we want is to have function calls for creating, copying, or moving these objects.

Bitwise copying is fine!

Here's an example order of a verbose order structure:

```
struct Order {
   unsigned long timestamp;
   int id;
   int price;  // or: double price;
   int qty;
   bool is_buy;  // Side: Buy or sell order?
};
```

Note that we will often exclude the "side" data field because we're often processing buy and sell orders separately. For example, an order book usually maintains the bids and asks at each price level as a separate data structure. Hence, context tells us whether it's a buy or sell order object, and it's more efficient not to store an explicit field. Even a small `bool` field adds more than a byte because of alignment issues.

We could use inheritance of the different types of order structures, by having an order base class, and derived buy order and sell order classes. However, that's not the typical low-latency coding style, because to take advantage of such a hierarchy, we'd need to use `virtual` functions, which are elegant but somewhat inefficient.

Note that there's only minor differences between a "`struct`" and a "`class`" in C++. A structure is really like a class, except the data member default to public, and a few other obscure difference.

Hence, you can define member functions for a structure to be class-like, or you can make a class have only public data members, like a structure. Just take care with:

- Default constructors
- Copy and move constructors
- Destructors
- Assignment operators (copy/move)

You probably just want the defaults. Also, you don't want your order objects to contain some other non-trivial data member objects, or you get the same performance problems in those subobjects.

In some data structures, you might want to store a cut-down version of an order. Your internal code probably doesn't need all of the fields that come in from the exchange data feed messages. For example, in a queue based on "order-of-insertion" you don't need to store the timestamp. Hence, it's common to have multiple different types of order objects, with fewer data members and a more compact size.

Integer Prices

Trading engines usually represent price as integers in real-world code. Some of the useful terminology in this regard is:

- Tick — the smallest possible pricing differential (depends on the stock or other instrument).
- Pip — the smallest difference in Forex pricing (foreign currencies).
- Basis point (bp) — the different differential in interest rates (usually a hundredth of a percent or 0.01%).

The value of the tick for a stock or futures contract depends on the individual stock, financial instrument, or asset. The tick for an asset might be $0.01 (i.e., a cent) or a small amount like $0.0001 for low-priced assets, or a relatively large amount such as $0.25 (quarter) for a high-priced asset.

A normal 32-bit "`int`" or "`uint32_t`" is usually adequate to represent a price. Note that the integer representation of a price indicates a multiple of the tick or pip, rather than the actual dollar price. Note that `size_t` is usually a 64-bit unsigned integer, rather than 32-bit, although it depends on the C++ implementation.

Prices cannot usually be negative (except for Swiss interest rates). Since `UINT_MAX` in `<climits>` is approximately 4.7 billion, the tick values stored as a 32-bit integer have these properties:

- $0.01 — price up to $47 million in 32-bits.
- $0.25 — price up to $1.175 billion.
- $$0.0001 — price to $470,000 in 32-bits.

To avoid issues with 32-bits, trading engines often use 64-bit integers for all prices, thereby avoiding the risk of integer overflow and also allowing for negative pricing. As required, 64-bits can be used for larger price values than those listed above, or for more granular pricing with a smaller tick value. In such cases, use `uint64_t` or `unsigned long`.

On the other hand, 32-bit integer arithmetic is marginally faster than the same operations on 64-bit integers. This also uses double the space, which affects cache lines and other cache locality, so there's a performance trade-off in terms of both space and time.

The term "tick" is used for all types of financial instruments, but there are other terms. The tick is called a "pip" for Forex currency transactions, which is usually 0.0001 or 0.01, depending on the currency pair. The term "basis point" refers to the smallest difference in interest rates, and is a fraction of a percent (a hundredth of a percent).

The reasons for the use of integers in price representations include:

- Avoids rounding errors in floating-point computations.
- Equality tests or comparisons of `float` or `double` can suffer rounding errors.
- Integer arithmetic is faster than floating-point.

Integer quantities. Quantity is also a positive integer, and is usually stored as an unsigned integer. This representation can handle a quantity up to 4.7 billion of an asset at the given price.

Negative quantities are usually invalid, and underflow of an unsigned type would wrap around to a large unsigned integer, resulting in an error. Zero quantity can be used as a representation of a used-up order, or some other marker of a "to-be-deleted-later" order, since a real order cannot have zero quantity.

Timestamp integers. Timestamps are also an integer, usually represented as an `unsigned long` to allow representation of the large numbers. Exchanges will tag an order with a timestamp, and the origination of timestamps in trading systems is usually a hardware-based timestamp. Incoming messages will often be tagged with a hardware timestamp in the NIC of the system.

Consistency Checks on Orders

Incoming market feed order messages can sometimes be lost, delayed, or corrupted. Some of the self-consistency checks to consider include:

- Positive integers for price, quantity, ID, and timestamp.
- Huge integers might be underflow (e.g., `UINT_MAX` is also `-1` converted).
- Timestamps should not be less than the previous one (but it happens with UDP).

At a higher-level, consistency should also apply to the order book data structure. Some consistency checks include:

- ID should be new for add orders.
- ID should already exist for modify or cancel orders.
- Trade executions and subsequent order updates should be consistent.
- If max-buy and min-sell cross, there should be a trade message incoming.

Another common error is that zero quantity (or price) might be used to indicate a finalized trade, a "to-be-deleted-later" efficiency optimization, or an otherwise invalid order. In multithreaded code, that order might be seen before it's removed. Hence, you need to check for this invalid case everywhere, and I do mean everywhere. Otherwise, you'll be re-processing zero-quantity orders over and over, because a zero-quantity buy or sell order can always be filled.

7. Order Book & Matching Engine

What is an Order Book?

An order book is a data structure that tracks all the currently active buy orders ("bids") and sell orders ("asks"). Typically, it has tons of data, because it has all the buys and sells from every trader who's in the market today.

Tracking the order book has a number of commonalities across all changes. There are a number of major assumptions when implementing an order book:

1. Only a single stock — each financial instrument with a price has its own order book.
2. All individual orders are seen — it's a "market-by-depth" data feed with all the juice.
3. No out-of-order transactions — we track order times implicitly as they are received via "order-of-insertion" rather than receiving an explicit timestamp and sorting based on its value.
4. No missing transactions — this is often an invalid assumption (I'm looking at you, UDP!), so the order book must be robust for problems such as IDs that are not found.

It's different for the exchange versus a trading engine client. The exchange is managing many trading clients, but a trading engine is only for one exchange. The way it works overall is:

- Exchange — real order book and matching engine to find trades to execute.
- Trading engine — an abstract "model" of what's happening in the exchange.

The exchange has to detect trades via its "matching engine" and really execute them (and do so correctly!). A trading engine is just watching what's happening, and using that information to decide whether or not to submit its own orders.

It's like the exchange has the "real" order book, and the trading engine is maintaining its own "model" of what it probably looks like.

In theory, a trading engine doesn't need its own matching engine. In practice, maintaining an order book means that you know when the bids and asks have "crossed" (i.e., a buy is high enough to match a sell), and you can thereby predict when a trade message will be coming. In fact, this can be a useful consistency self-check to confirm that your predictions about incoming trades are correct.

Matching engines and order books are intricately linked, and their code uses the same data structure. It's not much extra code, and, anyway, you'll probably also need a matching engine for:

- Unit testing of the order book
- Backtesting

Note that a trading engine's order book is about all the orders and trades, not just your own. Tracking your own submitted trades is a totally different component of your trading engine. For example, to track whether your order got executed, or to know cumulatively what your current "positions" are in the purchased stocks from your prior submitted orders, that's not in the order book.

It's some other C++ coder's problem.

Order Book Messages

The input to an order book is mainly a set of messages for:

- New buys
- New sells
- Updates (to quantity)
- Cancels (remove the order)

If you're the exchange, these submitted orders or updates to already-submitted orders are received via TCP from multiple clients. If you're one of the trading engines, you see all these messages coming across the network from a market data feed, usually via UDP multicast messages.

The main difference is in trade executions. The exchange has a matching engine that has to figure out whether to execute a trade, and then tell everyone about it. A client order book in a trading engine doesn't need to figure this out, because it gets told about trades in the incoming messages from the exchange's market data feed.

Let's examine the types of messages that occur.

Usually, the bids are priced lower than the asks, in which case there's no trades. When orders come in like this, they go into the order book, and might be executed later when things change. These are called "passive orders" and can be buys or sells.

If a new bid in an incoming buy order raises the price above the min-ask (min-sell), then that buy order will trigger a trade to occur. Similarly, if a new sell transaction has a lower ask, then it may match with one of the bids, and a trade occurs.

Market Data Feed Issues

There are two main types of data feeds, where one is like a fire hose and the other more like a faucet. The amount of data affects how "deep" of an order book data structure you can maintain from the feed:

- Market-by-Order (MBO) — full detail about every order, trade or update.
- Market-by-Price (MBP) — only trades and pricing, not every passive order.

An MBO data feed shows every order as it comes in from traders, with full detail of how many passive orders there are at every price level. MBP is more of an aggregation of pricing, showing the total quantities of bids and asks at every price level. Hence, an MBP feed is more detailed than just "quotes" about the most recent trade prices for a stock, but it's much less data than an MBO feed.

Some issues in the structure of an MBO market data feed can affect how you update your fully-detailed client-side order book in a trading engine.

- Are affected order IDs listed in a trade execution message?
- Do orders affected by a trade execution get modified by followup messages changing status or quantity?
- Is there order status tracking in the feed (e.g., partially filled, filled, or "done for day").

Some lesser-known types of data feed information:

- Broken trade — undo a prior trade.
- Execution with price — specific type of trade message.

Order Book versus Matching Engine

There's a kind of symbiotic relationship between an order book data structure and a matching engine algorithm. The tentative distinction is:

- Matching engine — detects if a trade is possible.
- Order book — is told when a trade occurs (by the matching engine).

If you're processing a market data feed, then the exchange is detecting trades (via its matching engine), and then sending trade messages down the feed. Your order book is then receiving a trade message, and doesn't need to do its own matching to detect when trades occur. In fact, many of the market data feed protocols will supply the IDs of the orders involved in a trade, so your order book may not need to implement the matching logic.

Except that sometimes it does!

You need to do matching if you don't get IDs in trade messages from the feed, which is sometimes the case. Another reason is when a trading engine needs to predict trades before you hear about them on the feed. Such issues depend on the algo that you're running.

Matching Engine Logic

Consider the situations if you get a trade message from a market data feed with only:

- Trade price
- Trade quantity

How do you know which orders did the trade? Short answer: you don't. There could be many orders with the right price and enough quantity.

In these cases, a client-side trading engine has to effectively build your own emulation of a matching engine, as part of your client-side order book maintaining code.

Note that mapping an anonymous trade message, without only price and quantity, to other orders is effectively the same logic as an exchange doing a matching process on an incoming new order.

You have to figure out which orders probably did the trade by looking at all the buys and sells at the price level, and examine all orders at that price level in the order they were received (i.e., order-of-insertion), and which orders have enough quantity, and so on.

Guess what you just coded in your order book ... a matching engine!

There are also various other scenarios where things get different with order books versus matching algorithms:

1. Exchange matching algorithm — if you're working on the exchange side, then you actually need to implement a live matching engine for real trades (scary!).

2. Backtesting — you may need to emulate a matching algorithm to simulate the effect of your fancy algorithm and its submission of orders.

3. Unit testing — you may need to emulate a matching engine, unless you're just replaying some recorded data feed messages.

And if you have to code up a real matching algorithm at an exchange, guess what data it has to maintain, for each financial instrument, to detect matches based on all the incoming trade submissions from its clients...an order book.

Data Structures for the Order Book

The first point to note about the order book is that it's an incremental algorithm. You process each market data feed message in sequence, and can maintain an up-to-date order book in this way.

Is your order book correct? Not always, since messages can get delayed or even lost (usually in a UDP multicast data feed). There's also a non-incremental secondary method call "snapshot synchronization" whereby you can correct your order book. Also, some protocols have partial status or statistics messages that can help validate your order book is still correct.

Multiple data structures are required to address the efficiency of various different types of requirements.

The overall idea would be something like:

- Hash table — mapping the IDs to order objects.
- Doubly linked list — order-of-insertion sorted list for processing orders in the order they're received.
- Heap data structure (priority queue) — maintain the maximum-buy (max-heap) and minimum-sell (min-heap).

Do you need all this stuff?

A hash table is hard to avoid because everything's keyed off the order IDs. New orders come in with a unique ID, which is then used to modify or cancel the order. The IDs also usually appear in trade messages, although not always.

Whether you need a queue or linked list for the FIFO list of orders is discussed below. The first point is that it's actually needed at a price level, and not necessarily one huge list of all orders.

FIFO Order Lists

The idea of an order list is that it has FIFO logic based on order-of-insertion, which is usually equivalent to timestamps. Assuming they have the best price, multiple orders at the same price level are supposed to match in the order they were received by the exchange.

Do you always need a linked list or queue of orders?

If you're the exchange then, yes, you definitely need to track FIFO status of orders at a price level, so as to implement "fairness" of trade execution in the matching engine. But what about on the trading engine side?

It depends on what algo you're doing. If the market data feed is giving your order IDs for trades, then you don't need to emulate a matching engine, and you don't need those linked lists, unless the algo needs a signal derived from them.

This situation is a little similar to using a Market-by-Price (MBP) data feed, which doesn't have all the individual trades. However, there are times when you want to track all the individual orders and trades, but you don't really need to know the FIFO ordering of them.

For example, you might want more price-level data than an MBP feed is giving you, such as maintaining all these price-level statistics incrementally:

- Total volume
- Total number of orders (queue length)
- Time frequency of orders (timestamp computations)
- Last order size and timestamp

An algo could be using these price-level data points, without necessarily needing the full tracking of all the orders in a queue data structure.

Price-Level FIFO Ladders

An important aspect of maintaining an order book is that matching orders should be processed fairly in a FIFO order. Hence, to process a trade in a matching engine, we need a queue of orders. This gives a FIFO ordering with the orders stored according to order-of-insertion.

But the price matters more than the ordering! To process a matching trade, we need to find all the orders that are max-buy (or min-sell), and process them in FIFO order. There are two basic ways to set up a data structure:

- One long FIFO queue
- Per-price FIFO queues

Using a single FIFO queue is inefficient when we need to process a trade, because the whole list may need to be scanned to find the orders at the right price level.

Having a separate queue (list) for each price level is much faster, because all of the items on the list have the same price level (that we want). This aspect of having lists of orders for each price level is often called a "ladder" or a "bid/ask ladder." The algorithm becomes:

1. Find the linked list or queue of orders for that price level.

2. Scan the order list processing each order record (in FIFO order).

3. Continue until we have enough quantity or the list ends.

4. Process the next-best price if we need more quantity (from Step 1).

However, there is extra storage cost because each price level needs its own object, and must contain the head and tail pointers of a dequeue or doubly-linked list. And we need a mapping data structure to find those linked lists for each price level (i.e., a hashmap or a very big bucket array).

Note that this above analysis of FIFO matching is ignoring some other types of matching rules. For example, there is pro-rata and hybrid FIFO pro-rata as other possibilities, which introduce additional complications if any order has a large quantity (which is arguably "unfair"!).

Heap Data Structures

Heap data structures, also known as a priority queue, are good at efficiently tracking the maximum (or minimum) of a set of values. They are also efficient at updating the maximum or minimum under many insertions and deletions of random values. Note that the term "heap" in this context has nothing to do with memory allocation!

There are three types of heaps:

- Max-heap — tracks the maximum value.
- Min-heap — for the minimum value.
- Min-max-heap — does both efficiently (we don't need this).

Do you need a heap for price levels? It seems like overkill to have a data structure just to calculate the maximum buy and minimum sell prices of the order book. However, see below for reasons why an incremental algorithm isn't that easy. In short: deletions are tricky to handle without a heap.

Anyway, actually you don't need a heap, because you need two! There are typically two heaps with a max-heap data structure for the buy orders to track max-bid, and another min-heap data structure for the sell orders (min-ask). These two heap data structures are completely independent.

In C++, a max-heap can be implemented using a standard library data structure via the `std::priority_queue` container class.

A min-heap can be declared using a custom comparator that reverses the logic. The default for a max-heap is `std::less`, but you can use `std::greater` to create a min-heap. Although using a custom comparator would often be inefficient, the standard C++ library probably (hopefully) has a builtin template specialization for this comparator. However, you need to check by benchmarking!

If your min-heap is slow with a custom comparator, there's another weird optimization: negative numbers to the rescue! A maximum of negative values is the minimum absolute value. You can make a min-heap from a max-heap by negating all the values on the way in, and negating again (to reverse it back to the original number) when returning an item from the min-heap.

There are alternatives to heaps, but you certainly need to track some information per price-level, and be able to access it in sorted order (e.g., find the second-best price). A heap is the most straight-forward data structure for doing this.

Ordering Out-of-Order Orders

The simplest type of order book and matching engine should be based on "order of insertion" so that older orders get processed first. However, even when trying to maintain this FIFO ordering, there can also be issues with timestamping and lost or delayed order messages. This is more of an issue for client-side coding of an implicit order book from a market data feed via UDP multicast, rather than an exchange server's incoming transactions over more reliable TCP connections.

Some orders from the exchange market data feed may be received late due to a delay and therefore appear to be "out-of-order" when they arrive at the market data ingestion component. Typically, this is due to delays or lost packets in UDP multicast messages as they are sent from the exchange's market data feed to the trading engine's client code.

How to handle this?

As a client, we want to keep our order book accurate, but not at too great of a performance cost. In the abstract, there are various ways to treat messages that come in with a timestamp that indicates they are not received correct order. Some issues include:

- Detection by tracking incoming timestamps versus previous messages.
- Ignoring timestamps and doing order-of-insertion anyway.
- Timestamp-based insertion to correctly place the orders in the FIFO list.

Detection is relatively straight-forward if we assume it's a rare event and only one transaction is received out-of-order. The idea is to simply track the incoming timestamp of the most recent order, and compare that with each incoming message. However, things get more complex if there could be multiple out-of-order transactions, which could be also the wrong order in themselves. Handling these obscure cases efficiently is more problematic.

One way to do it all efficiently is to detect out-of-order timestamps, but then ignore the issue (except maybe logging some in-memory statistics counters). In other words, just insert it into the data structure in order-of-insertion, and it will be wrongly behind some of the orders with a later timestamp.

How risky is that?

The risk is that there's a match at that price level, and some other transactions get processed by a trade, rather than this one (which was actually received earlier by the exchange). The importance of an order can also depend on its price level, since orders that are unlikely to be crossed won't see any problem at all. Another point is that lost messages occur, so there are whole orders that get missed, and the order book will get occasionally updated via the "snapshot synchronization" methods.

In other words, out-of-order trade or order messages is not the only problem that we have with our order book becoming an inaccurate model of the exchange's order book. There are various trade-offs here, and nobody likes to leave it to chance. In this case, we actually know there's a problem, whereas with a lost order, we do not.

Can we fix it?

The correction is to try to insert the order wherever it should have been. Timestamp-based insertion is inherently a slower operation on the default data structure of a FIFO queue of orders.

Insertion into an order-of-insertion queue is an O(1) insertion at the tail of the queue, which is only a couple of operations. But for an out-of-order insertion using its timestamp, you now have to scan this queue or deque. The operation of finding where exactly to insert is an O(n) linear search to examine all the timestamps on the list. It may be a rare event, and the reverse scan down the list might only be a few orders previously, but it still introduces a non-deterministic possible slowdown to insertion of orders into the order book data structures.

Another separately indexed data structure may be considered here to map timestamps to linked list locations (i.e., unsigned long to a pointer), but that adds more complexity to the situation. And it's not a hashmap, because we need lookups with relative ordering of the timestamp keys. Worse still, maintaining some other type of index will also slow down all of the other non-problematic order insertions.

Incremental Max-Buy and Min-Sell Prices

A useful optimization is to maintain the current maximum buy and minimum sell as two incrementally-updated price values. These are the same numerical types as the price type, whether it's a double or an integer. And there are two separate values:

- Max-buy price (maximum bid)
- Min-sell price (minimum ask)

The first thought is to get excited and think that maybe we don't need a data structure at all to track the price levels. Maybe we can just incrementally maintain these two values, and done.

Does it work? Let's try it out. The general idea for the incremental algorithm is:

- New buy orders — if price is more than the max-buy, update the incremental max-buy.
- New sell orders — if price is less than the min-sell, update the incremental min-sell.
- Cancel buy orders — if buy price equals the max-buy, and there are no other buy orders at that price level, find the next-highest buy price.
- Cancel sell orders — if sell price equals the min-sell, and there are no other sell orders at that price level, find the next-lowest sell price.
- Modify orders — these are not allowed to change the price of orders, so they don't affect it.
- Trades — treated like order cancels for updating the max-buy or min-sell values.

This would be a beautifully efficient algorithm ... if only it worked. It's super-fast for new buy or sell orders, and modify orders don't affect the incremental values. The problem is the deletions.

Deletions of orders, whether via a cancel message or a trade happening, need to find the new max-buy or min-sell. Canceled buy orders below the max-buy price, or sell orders above the min-sell price, don't affect the incremental values, and are efficient.

Also, even if the deletion is an order at the max-buy or min-sell, if there is even one other order at that price level, then we also don't need to do anything.

But if a deletion removes the last order at the max-buy or min-sell price, then we need to scan all the other price levels. Note that when doing any type of deletion or cancel, there shouldn't be a buy order at a higher price, or a sell order at a lower price, if we've been correctly tracking the incremental values. So, we're looking for the second-best buy or sell price.

Alas, the hash table of offer ids is no help here. We need a data structure that tracks the price levels of all offers, and one that can efficiently find the maximum or minimum, such as:

- Red-black tree (e.g., `std::map`)
- Heap (e.g., `std::priority_queue`)

On the upside, you can see that the use of the price-level data structure (heap or tree) to find the next-best buy or sell prices is a relatively rare event. Hence, this incremental optimization can be very helpful in practical terms. For example, when computing mid-quotes or the bid-ask spread, we can just access these two scalar variables, rather than querying the price level data structure.

Finding whether there's any other orders at that price level actually requires a price-level data structure anyway. We need to map the new order's price level to the linked list of other orders at that price level.

There's also another optimization here: to avoid needing to look up the price level, we could store a pointer to the price level object for the current max-heap or min-sell prices (i.e., we have then four incrementally maintained variables: two prices and two pointers to price level data structure objects).

Also, we've probably already found the price level data structure object for other reasons (e.g., to add a new order to the linked list of orders for that price level).

References

1. Sourav Ghosh, July 2023, *Building Low Latency Applications with C++*, Packt Publishing, https://www.amazon.com/dp/1837639353
2. Philippe Bourgeon, 2016, *Take Home Test (C++14 OrderBook)*, https://github.com/bgn9000/Cpp1y-OrderBook
3. Sadhbh Code, 2024, *C++20 Order Book: Order Book implementation in C++20 (Concepts & Co-Routines)*, https://github.com/sadhbh-c0d3/cpp20-orderbook
4. Code Review, 2023, *Fast OrderBook Implementation*, https://codereview.stackexchange.com/questions/285623/fast-orderbook-implementation
5. Colman M., September 25, 2024, *C/C++ Advanced Order Book Processing Example with CPU Affinity*, https://www.linkedin.com/pulse/cc-advanced-order-book-processing-example-cpu-colman-marcus-quinn-7227e/
6. Scorsone Enterprises, 2024, *Coding an Order Book in C++ (Beginner Friendly)*, https://www.youtube.com/watch?v=TRiqIkhR0XI
7. Shu Wang, 2011, *How to Build a Fast Limit Order Book*, https://gist.github.com/halfelf/db1ae032dc34278968f8bf31ee999a25
8. Quantitative Finance (Stack Exchange), 2021, *What is an efficient data structure to model order book?*, https://quant.stackexchange.com/questions/3783/what-is-an-efficient-data-structure-to-model-order-book
9. Charles Cooper, 2021, *Fast implementation of an ITCH order book*, https://github.com/charles-cooper/itch-order-book
10. Amitava Biswas, Aug 18, 2023, *Designing Low Latency High Performance Order Matching Engine*, https://medium.com/@amitava.webwork/designing-low-latency-high-performance-order-matching-engine-a07bd58594f4

8. Iceberg Orders

What are Iceberg Orders?

Iceberg orders are large orders where most of the quantity is hidden, in the same way that most of an iceberg is hidden under the water. Exchanges offer support for iceberg orders in terms of auto-replenishment of these orders with more of the hidden quantity when the displayed quantity gets traded. The replenished quantity is added as a new order with a different ID, so it's not obvious that it's a replenishment of an iceberg.

The reason that some traders would want to use iceberg orders is that a large quantity can be traded without making it obvious. Hopefully, this avoids price slippage that might otherwise occur.

The effect of icebergs is different for exchanges versus other traders because of the information asymmetry. At a high level, the management iceberg orders has aspects including:

- Display quantity is visible to other traders, but a larger total quantity is hidden.
- The exchange knows it's an iceberg, and has automatic support for this.
- Other traders don't know which orders are icebergs (in theory).

Many exchanges allow the placement of iceberg orders as a special type of bid or ask. For example, a hedge fund or institutional investor might wish to make a large trade. Even HFT traders may use their own icebergs, although they may also like to find icebergs to trade against, because it means there's a lot of hidden liquidity at a price level.

Exchanges have explicit support for icebergs as an order type. When you place an iceberg order with an exchange, you specify:

- Side (buy or sell)
- Price
- Display quantity
- Total quantity (hidden)

- Replenishment (increment)

The initial quantity is displayed and is the first order. When this order is consumed in a trade, a new order is created with the replenishment quantity. The amount of replenishment doesn't need to be the same as the initial quantity.

This is a simple type of iceberg order with a fixed replenishment quantity and zero delay. Some exchanges offer more advanced handling of iceberg orders, such as dynamic adjustment of the replenishment size, or auto-delays for when the new order appears. These more advanced features can help iceberg orders remain hidden and get the best execution prices.

Iceberg Replenishment Scenarios

When an iceberg is triggered on the side of an executed trade, the exchange automatically "replenishes" the quantity in a new order. In other words, once the display quantity is absorbed in a trade, another new order is automatically placed.

Conceptually, the new order has its own ID and is added to the end of the queue, like any other type of order. However, behind the scenes, the exchange may have shortcut this cycle with some code optimizations.

An interesting scenario arises where an aggressive order matches an iceberg order at the best price, and one or more orders at a second-best price that is also crossed and could be executed. Naively, we would say that the price level has only the single iceberg order at the displayed quantity according to the FIFO queue of orders at that price level. The replenishment orders are not yet on that queue. Hence, the matched iceberg order's initial display quantity should be used, and there being no further liquidity at that price levels, the second-best price orders should be executed to fill any remaining capacity.

But when is the iceberg replenished? Is it before or after switching to a second-best price level?

The naive scenario is not very good, because a better price was on offer by replenishing the iceberg and trading with it again, before reverting to the second-best priced orders only after the iceberg order was exhausted. This is a better algorithm for the exchange, and there's nothing really conceptually wrong with this. It's just doing the replenishments after scanning a single price level, before moving to a second-best price level.

Iceberg Algorithm Optimizations for Exchanges

There are two types of code optimizations in relation to the processing of iceberg orders:

1. Exchanges optimizing executions that trigger many replenishments.

2. Other traders trying to figure out which orders are icebergs.

On the exchange side, some of the situations that can be considered for algorithmic optimizations include:

- Repeated replenishment — iceberg orders getting replenished many times in a single trade execution with lots of new orders getting created and then executed.
- Two icebergs colliding — the aggressive order that triggered the trade was itself an iceberg, which matched with one (or more) icebergs on the other side of the trade.

Let's look at what sort of code optimizations are possible.

The first point is that the exchange knows which orders are iceberg orders. Hence, when it's scanning the list of orders to match at a price level, it can fill the non-iceberg orders, and the displayed amount of an iceberg order. Note that this is only considering a "fair" FIFO filling method, and not more complex variants such as "pro rata" algorithms where larger orders get more fill.

After the first scan of the order list for a price level, the exchange knows it has seen some iceberg orders at that price level. Hence, the exchange's matching engine knows that once the list is finished this first scan of the price level, there will be new replenishments of one or more iceberg orders at that price level. In fact, after this first scan of the best price level, if there's still quantity to be traded, then:

There's only icebergs left!

According to a naive implementation, the one or more icebergs at that price level should be repeatedly:

1. Creating a new order with a replenished quantity, and

2. Executing a trade with the new quantity.

This could happen many times. Possibly an entire iceberg order is used up, or it may have quantity left. Partial fills are also possible at the edge cases.

In practice, an exchange's matching engine may use some optimizations here, rather than repeatedly doing the same steps. After all, you can calculate how much of the available quantity should be consumed by each iceberg based on:

(a) your active order's available quantity,

(b) the iceberg's hidden remaining quantity, and

(c) the iceberg's replenishment rate.

You can do the arithmetic first, and then create new orders with new IDs. One optimization is to do a bulk-insert of all these new orders into the order book. A better optimization is not to put them into the order book at all, because they're already been traded out of the order book (before they even went in). However, the exchange still needs to emit the various new order and trade execution messages for all of these iceberg replenishment orders, so as to try to hide everything.

Finally, note that these optimizations won't apply in all situations for an exchange. For example, these optimizations are assuming that the client's iceberg orders are immediately replenishable with zero delay, which is not true of all iceberg orders.

Trader Detection of Hidden Icebergs

Other trading participants would love to find out which orders are icebergs. In theory, there's nothing to see. But HFT coders and algorithmic traders are nothing if not innovative.

Generally, the strategy for finding icebergs is to watch the market's sequence of events, via the market data feed. Anywhere that the sequence differs from what you would normally expect for a non-iceberg order, that's when you have identified a likely candidate. The main idea is:

Spot icebergs when they execute!

There's not much you can do when an iceberg is sitting passively in the order book. Similarly, a non-aggressive new iceberg order won't be easy to spot. The differences occur in the executions.

To see what can be done to detect market sequence differences, think about the process whereby an order will trigger an immediate auto-replenishment by the exchange. Some new orders have appeared and been executed in the blink of an eye. The IDs of these new replenished orders were not in the order book before, but they appear as a sequence of new aggressive orders at the same price point. Some of the main things to see is:

(a) Suddenly created orders immediately executed, and/or

(b) New orders created with the same quantity.

So, this analysis of the timing of executions and new orders gives some hints about the presence of an iceberg order at a price point. Note that this is assuming basic icebergs with replenishment of a fixed size that processes instantaneously with zero delay. However, exchanges also offer more advanced types of icebergs with dynamic replenishment quantities, time delays, and triggers based on market conditions.

Some other types of indicators that an iceberg may be present include:

- Price levels sustained despite low apparent available liquidity.
- Volume spikes at that price level.
- Recurring market maker indicators in repeated orders (not the individual traders, which is secret, but the financial institution through which they're trading, which isn't).

Probing Strategies

And finally, there's also the idea of issuing your own trades that attempt to find out if an order is an iceberg, This is called a "probing strategy" and aims to find hidden liquidity in the market. Some of the ideas include:

- Ping orders — submit small orders watching for replenishments.
- Layered orders — several orders at multiple price levels.
- Flash orders — short-duration orders to see if they get swallowed.

The overall idea is to issue these "probes" and then watch the reaction in the market data feed for what trades occur, and how quickly, and whether new orders get created.

If you think you've found an iceberg, there are two basic ways to play for an edge:

(a) Now — repeatedly trade against the iceberg or others using this extra knowledge, and/or

(b) Later — trade for price changes that will occur after the iceberg is finished.

Many of these probing methods are commonly used by algo traders. These attempts to find icebergs can have false positives, whereby it's not an iceberg, but some other algorithmic trader that's responding with new orders. Furthermore, such methods can be expensive if you fail, may change the market unintentionally, and also some types of probing may be considered "market manipulation" in some jurisdictions. Hence, if you think you can spot icebergs, maybe think about the Titanic.

Extensions

1. Examine or code up the matching engine logic for processing trades when an iceberg order is matching.
2. Examine the algorithm for optimizing iceberg matches where multiple icebergs match, and the active quantity is large enough for many replenishments.
3. Can you calculate how much each iceberg will consume of an order's quantity using only arithmetic and conditional tests? Try to avoid simulating it with a loop. Start with the case of one iceberg, then generalize.
4. Examine probing methods to detect advanced iceberg orders with delayed replenishment and non-fixed quantities.

9. Rate Limiter

What is a Rate Limiter?

A rate limiter or "throttling" component aims to avoid too many requests hitting a server in a time period. For example, a server might have a rate limit policy of "100 requests per minute" that all clients must adhere to.

Servers have two basic methods for dealing with an exceeded rate limit:

- Rejection — disallow the client's transaction with an error message.
- Smoothing — instigate a delay or other load reduction method without rejecting.

Servers don't really like having to force rate limits on their clients. After all, they want happy customers. Hence, servers will attempt to improve their capacity in other ways:

- Load balancing technologies
- Bigger servers with GPUs
- More C++ low-latency coders

But at some point, if demand for your service is unlimited, you have to say no.

Rate limiters are a general technology component and may apply to numerous types of servers and services that allow multiple clients:

- Trading exchanges limiting HFT order submissions.
- AI servers limiting the number of Norse poems people can request through their API.
- Web sites limiting the number of browser page views or online transactions.
- Email servers limiting the pass-through of emails (spam prevention).

I'm sure you can think of some more.

Client vs Server Rate Limiters

Servers and clients have different issues, but both can implement a rate limiter C++ component. The basic idea is:

- Servers — block a client from submitting too many orders (keeping it minimal).
- Clients — try to figure out when you can submit an order (so as to maximize it).

There's a significant architectural difference for the two contexts:

- Server rate limiter — per-client rate limits for many clients.
- Client rate limiter — tracks rates viz one server and one client (me!).

The scalability requirements for the server are much greater. Hence, a server will often implement its multi-client rate limiter using an in-memory database such as Redis or Memcached. The client-side rate limiter component is much less complex, and it's usually a simple C++ class.

As you can see, the objectives for servers versus clients are somewhat opposite, but the coding issues are similar. The server is effectively maintaining multiple rate limiters with one for each client. The client is maintaining one rate limiter component for its connection to the server. These are two sides of the same coin:

Client rate limiters are abstract models of the server rate limiter.

The server has the real rate limiter that will actually block orders. The client's rate limiter is a theoretical model that attempts to emulate the server-side logic to thereby predict whether we can send an order or not. Hence, to implement a client-side rate limiter you need to know as much as possible about the server's rate limiter:

- Rate limit thresholds — e.g., how many transactions in what time period?
- Rate limit algorithm — e.g., time-based or total transactions?

The rate limit thresholds are usually part of the documentation. Note that rate limits may not be time-based, such as where each client is allowed (or can buy) some "credits" and then consumes one or more credits with each submitted request.

The algorithm used by the server-side rate limiter may be trickier to discern. There's a whole bunch of theory about the best way to do this on a server. Here's a selection of algorithms:

- Fixed counts (credits)
- Token bucket
- Leaky bucket
- Fixed window
- Sliding window (log variant)
- Sliding window (counter variant)

There are other low-level features of a server-side rate limiter algorithm to consider:

- Rate limit violations — does the server reject, smooth, or delay the client transaction?
- Retry permissions — does the server's rejection include a data field with the recommended time for a retry?

And then you have to code all that into your client rate limiter.

Rate Limiter Optimizations

Client-side rate limiters are part of the "hotpath" and are performance-critical. After all, a rate limit component is queried immediately before submitting a transaction, so any latency in the rate limiter checking will directly worsen trade submission latency.

How to run fast?

Well, it depends on the server's algorithm. For example, if the server allows one request per minute, then only record the timestamp of your last submission. And when the server's method is one based on a fixed number of credits (e.g., a free trial with an upper bound on credits, or a way to purchase a number of credits), then the client can just maintain an incremental value of its own credit stache.

More interesting optimizations arise in the rate limit tracking of fixed window and sliding window algorithms. The general idea for the rate limit is:

N requests in M seconds.

In a fixed window algorithm, the allowed limit is reset every M seconds. It's like the server has an interrupt timer running every M seconds, which resets the allowed client transactions. Indeed, this is one way to implement it, although it's not the fastest for the server, because it would have to touch every client's counter every M seconds.

Nevertheless, having a timer running every M seconds is more efficient for the client-side implementation. But you have to make sure that your client-side interrupt is synchronized with the server's timestamps, or else chaos ensues.

A sliding window algorithm is a more accurate way to limit client requests. Whereas a fixed window algorithm can be manipulated by the client in a way that allows 2N transactions to be submitted, a sliding window will more correctly limit to only N client transactions.

However, it's also more complicated to code a sliding window algorithm, and requires tracking the timestamps of many requests. The methods to implement a sliding window rate limiter include:

- Naive request queue with removals.
- Fixed array of N timestamps.
- Ring buffer of N timestamps.

Compact data representation. But before we look at the code, there's a space optimization, which also helps with speed due to cache locality. The first optimization is that we can throw away most of the transaction. We only need the timestamp, so we can compact the data significantly. <

It might be desirable to store other aspects of the request, such as an order ID, especially in testing mode. But the algorithms discussed below work only on the timestamp, and don't ever need to go back to the original order or request. In production mode, a client-side rate limiter component will tell you whether or not you have permission to submit a trade, but it can't give you a list of the transactions you did previously.

Naive request queue algorithm. The naive algorithm is to realize this is an "order-of-insertion" algorithm, so we need a queue, where the orders are stored as they are received, with their timestamp being the only important field.

The basic idea goes like this:

- Remove all old transactions on the queue that are outside the M seconds time window.
- Check if there are less than N transactions still in the queue.
- If so, success, and add our request to the queue.
- Otherwise, fail with a rejected transaction (and don't add it to the queue).

But this idea is not great coding, and I'm understating it here for politeness reasons. This method is super-inefficient because we are doing:

- Insertions of new requests (even if only timestamps).
- Removals of out-of-date requests.
- Linear scanning of the request list.

Fixed array of N items. A key insight is that to manage a rate limit of N items for a fixed time period, we only need to track the last N requests. Hence, we only need to store the last N items, and we can maintain a fixed array of exactly N items, or rather, exactly N timestamp values. Throw that dynamic queue data structure to the curb!

The simplest idea is an order-of-insertion array, but we shouldn't use an array that starts from index zero. Instead, we should use a ring or circular buffer data structure.

Fixed-size ring buffer. The simple idea is to maintain a fixed-size ring buffer containing the last N timestamps. This is effectively implementing a fixed-size queue of N items in an array or vector container.

It's an implementation choice whether to use a compile-time size with `std::array` or a run-time fixed size with `std::vector` for the ring buffer. If the rate limit is rarely changing, then N is a compile-time constant and we can use `std::array`.

However, we can use `std::vector` by doing a single heap allocation with a `reserve()` call in the startup phase of the trading application, away from the hotpath. The vector method is more flexible because we can load N from a configuration file, rather than needing a re-compile.

Prefilled ring buffer. One minor optimization is to note that the client-side rate limiter in a ring buffer is doing a needless branch. There are two distinct cases:

1. Startup — the first N transactions.

2. Ongoing — the rest of the transaction requests.

If the total number of submitted transactions is less than N, then our queue is only partially filled, and the transaction is definitely allowed: we've never submitted enough transactions, regardless of the time period!

But branches are not great, as discussed in the chapter on branch prediction. In the spirit of branchless coding, let's not even check for the condition. Instead, we can pre-fill the initial ring buffer with N zero timestamp values at startup, and pretend like it already has N elements stored in it. Thus, we can remove an "if" statement (goodbye, branch, we won't miss you!).

Note that doing this also allows a secondary optimization: we no longer need both "head" and "tail" indices. The ring buffer is always full, and the most recent item is always right next to the oldest timestamp. So, we only need a single offset.

Unfortunately, we can't remove everything! If it weren't for those pesky orders, we could do it all at compile-time.

Advanced Client Rate Limiting Issues

Computing retry time. An extra feature of our rate limit algorithm is in the rejection logic: compute the wait time required until a re-submission of this trade would be accepted. This can be returned to the caller as useful information. However, it's not a simple algorithm in our fixed-size ring buffer queue. Whether we want to always compute this for the caller, or provide an API for the caller to ask for this information, is a judgement call. But if our order is going to be rejected anyway, we're no longer in the hotpath, so adding computations has a low penalty.

Server timestamps synchronization. There can be a difference in the timestamp values of the server, versus your ones. This is a rare issue, and it can cut both ways.

- False positives — you submit a trade and it gets refused.
- False negatives — you withhold a trade that would have been allowed.

The difference in timestamps can occur at both ends of the queue. Depending on which end, this rare issue could trigger a false positive or false negative.

Extensions

1. Extend the client-side rate limiter algorithm to use time delays and retries. When should a rate limiter suggest a delay versus rejecting the transaction?
2. Extend the client-side rate limiting algorithm to accept requests from multiple threads.
3. How would you handle false positives? The client rate limiter says the trade is allowed, so the trade execution component submits the trade, but the exchange server rejects its submission. Add a feature allowing the trade execution component to report "bad trades" to the rate limiter. Should it report "good trades" to the rate limiter?
4. Examine the use of lower-precision timestamp values in the ring buffer. Instead of a 64-bit `unsigned long`, can you use 32 bits? Or less?
5. Analyze the use of differences in timestamps to compact the data type. Instead of the full timestamp, can you use the number of clock ticks since the program startup timestamp?
6. Consider how to handle incoming transactions that are out-of-order according to their timestamps. For example, your code is accepting candidate trade transactions from multiple servers.
7. What statistics should be tracked and recorded to allow monitoring and management of a rate limiter software component in production?

References

1. Peer D., September 27, 2024, *Building A Custom Api Rate Limiting Tool In C++*, https://peerdh.com/blogs/programming-insights/building-a-custom-api-rate-limiting-tool-in-c (This is a server implementation of rate limiting for multiple users)
2. Mike Cheng, 2015, *ratelimiter: A C++ Rate limiter implementation*, https://github.com/mfycheng/ratelimiter (Server version)
3. Geeks for Geeks, 16 Mar, 2023, *How to Design a Rate Limiter API | Learn System Design*, https://www.geeksforgeeks.org/how-to-design-a-rate-limiter-api-learn-system-design/ (This is a server-side rate limiter; examines bucket, leaky bucket, etc.)
4. Stack Overflow, 2015, *Rate limiting algorithm for throttling request*, https://stackoverflow.com/questions/26647166/rate-limiting-algorithm-for-throttling-request
5. Learn X by Example, 2025, *Rate Limiting in C++*, https://learnxbyexample.com/cpp/rate-limiting/ (Server-side rate limiter.)

6. Arpit Bhayani, Apr 05, 2020, *System Design: Sliding window based Rate Limiter*, https://www.codementor.io/@arpitbhayani/system-design-sliding-window-based-rate-limiter-157x7sburi (Server-side rate limiting.)
7. Aman Kumar Pandey, Jan 9, 2025, *Understanding rate limiting and its implementation*, https://medium.com/@amandevbhardwaj/understanding-rate-limiting-and-its-implementation-70bb5e33f63a
8. Abhishek Dey, The Algorists, 2025, *Distributed API Rate Limiter*, https://lowleveldesign.io/SystemDesign/RateLimiter (Implements various server-side algorithms.)
9. Wikipedia, 2025, *Token bucket*, https://en.m.wikipedia.org/wiki/Token_bucket
10. Wikipedia, 2025, *Leaky bucket* https://en.wikipedia.org/wiki/Leaky_bucket
11. 4sily, 2017, *Simple rate limiter: A quick-and-dirty implementation of RPS limiter*, https://github.com/4sily/rate-limiter-cpp
12. Geeks for Geeks, 7 Nov, 2024, *Rate Limiting in System Design*, https://www.geeksforgeeks.org/rate-limiting-in-system-design/
13. Jan Gaspar, 2013, *Chapter 9. Boost.Circular Buffer*, https://www.boost.org/doc/libs/1_64_0/doc/html/circular_buffer.html (General implementation of a circular buffer that can be used in the rate limiter.)
14. Robert Mosolgo, April 5, 2021, *How we scaled the GitHub API with a sharded, replicated rate limiter in Redis*, https://github.blog/engineering/how-we-scaled-github-api-sharded-replicated-rate-limiter-redis/
15. Hiresh Trivedi, Aug 5, 2021, *Designing a Distributed Rate Limiter — Introduction*, https://medium.com/wineofbits/designing-a-distributed-rate-limiter-introduction-731afd345a66
16. Ruslan Diachenko, Feb 5, 2024, *Sliding Window Rate Limiting and its Memory-Optimized Variant*, https://rdiachenko.com/posts/arch/rate-limiting/sliding-window-algorithm/ (Client-side use of a deque for the sliding window algorithm.)
17. Stack Overflow, 2023 (updated), *Sliding window algorithm for rate limiting requests per second windows*, https://stackoverflow.com/questions/69161879/sliding-window-algorithm-for-rate-limiting-requests-per-second-windows
18. Book Notes, 2022, *Design a Rate Limiter*, https://books.dwf.dev/docs/system-design/c5
19. Ronak Chatterjee, 2023, *A high frequency trading system built with C++: High performance, low latency high frequency trading system written from scratch in C++*, https://github.com/nyarosu/hft
20. Ranjan (Man of steel), 2025 (updated), *Live High-Frequency Trading Exchange Engine*, https://github.com/ranjan2829/Live-High-Frequency-Trading-Exchange-Engine

21. Ruy Dan, 2025 (updated), *A Ring Buffer implementation with a fixed-size buffer, developed in Zig*, https://github.com/ruy-dan/ring-buffer
22. Stack Overflow, 2020 (updated), *How do I implement a circular list (ring buffer) in C?*, https://stackoverflow.com/questions/215557/how-do-i-implement-a-circular-list-ring-buffer-in-c
23. Ralf Holly, 2020, *Circular Adventures VII: A Ring Buffer Implementation*, https://www.approxion.com/circular-adventures-vii-a-ring-buffer-implementation/

10. Slowpath Removal

What is Slowpath Removal?

Slowpath removal is a multithreading optimization whereby the cold paths are removed, merged, or deferred. The idea is to give priority to the hotpath by avoiding any branches leading to the slowpath, as much as possible.

Not all code belongs on the hotpath. Some examples of slowpath logic include:

- Error handling
- Logging
- Self-testing code

Note that I really mean *removal* of these paths. There are actually two optimizations in slowpath removal:

- Avoiding the cost of testing for errors.
- Removing branches of code instructions.

We don't just want to avoid testing for errors, but we actually want there to be zero branches in the hotpath code sequence. The reasons for this include:

- Branch prediction optimizations (i.e., branch elimination), and
- Instruction cache optimization.

Another point is that to make the hotpath short, with good latency in the instruction prefetch cache, we want to minimize any slowpath code in that path. Hence, if you cannot avoid having a slowpath sequence in the hotpath, then you should encapsulate it into a separate function, and *don't inline* the slowpath function. In this way, only the test for that slowpath condition (e.g., an error flag test), and a single function call to the slowpath function, is in the instruction block along the hotpath.

If the hotpath code sequence is short and tight on the CPU, it runs a lot faster than if it has to think about alternative pathways.

Error Handling Slowpaths

Error handling is a common example of a slowpath. Most of the failures and exception states of execution are not on the hotpath, as they are uncommon events compared to success. They're called exceptions for a reason!

The problem with errors is that you have to check for them, even though they never happen. Okay, yes, so they can happen, and good programmers always check their return codes and so on. But when you're trying to go fast, you want to focus on success and winning.

The choices for error handling are therefore on the scale between two extremes:

- Repeatedly check every error (slow)
- Don't check for any errors (unsafe)

There are some trade-offs in the middle ground:

- Check for fewer errors in production, but more in offline self-testing.
- Use in-memory logging data structures to defer outputting data to log files.
- Defer error checking until multiple error statuses can be checked at once.

Deferring Error Checks

The idea of deferred error checking is to not immediately check every error status. Instead, we try to keep going and ignore possible error states, and then check for them as late as possible.

Traditional error checking is to immediately test for a failure return code. Here's an example:

```
 bool oksetup = orderobj.setup(ticker, price);
if (!oksetup) {
    // Fail...
}
bool oktrade = order.obj.submit_trade();
if (!oktrade) {
    // Fail...
}
bool oklog = logger.record(ticker, price);
if (!oklog) {
    // Fail...
}
```

The basic structure is a long `if-else-if` sequence, with error handling interleaved into the main hotpath. Yes, you could micro-optimize the above, such as by avoiding three separate Boolean variables, but you get the idea. This is a slow control flow that mixes the hotpath and the slowpath.

Faster is to run as fast as possible with all the steps, and only check for problems at the end. If we can defer error checking until after the trade has submitted, then our error handling code is completely out of the hotpath. Here's the basic concept for doing deferred error checking at the end:

```
    bool oksetup = orderobj.setup(ticker, price);
    bool oktrade = order.obj.submit_trade();
    bool oklog = logger.record(ticker, price);
    if (!oktrade || !oksetup || !oklog) {
        // Fail...
    }
```

We might optimize this using bit flags for error codes and pass-by-reference parameters:

```
    uint32_t errflags = 0;
    orderobj.setup(ticker, price, errflags);
    order.obj.submit_trade(errflags);
    logger.record(ticker, price, errflags);
    if (errflags) {
        // Fail...
    }
```

The tricky part here is whether the trade submitter or logger functions will crash when the first function fails. We have to design all the routines to be pass-through, or at least non-crashing, even if an earlier routine has had an error. This is easier said than done!

You have to take care to really defer the error checks, not just hide them. For example, if your second routine needs to check for an error status from the first function (so it doesn't crash), then you haven't really deferred the error checking until after the hotpath has finished. Instead, it's just hidden further down the call stack inside the individual functions.

Removing Error Checks

Safe C++ programming practices always have us doing a lot of extra work to check for a myriad of coding problems:

- Function parameter validation
- Function error return code checking
- Assertion failures
- Self-testing code failures
- Memory allocation failures
- File loading errors (e.g., file not found, disk full)
- Valgrind runtime checking

But if we want to go fast, many of these can be removed. Goodbye to slow code! Hello, speed.

Not all of the above error situations are that common, and many of them are under our own control, since they're really just checking for our own coding errors. Some error avoidance strategies for the critical code in the hotpath include:

- Don't use memory allocation (avoids allocation failures).
- Avoid disk-full issues with logging via good Linux admin practices and lightweight monitoring.
- Compile-out parameter validation, assertions, and self-testing code for production (but include them in unit tests and offline automated test harnesses).

If compiling out all of the safety stuff gives you concerns, here's the plan:

- Don't write buggy code!

Oh, wait! That's not so easy. But here's what we can do: mitigate against human frailty by shaking out all the bugs before they get to production.

One of the main ways to have very fast production code, but mitigate against unforeseen coding failures is to max out the use of automated testing in offline mode. Here's the basic plan:

- CI/CD — faster unit tests.
- Nightly builds — longer automated tests, static analysis, etc.

We can and should run basic unit tests as part of CI/CD, but then we should thrash the whole thing to death in nightly builds. This means to enable lots of self-testing code and other very slow tests that would cause developer productivity issues if we ran them in CI/CD. Hence, nightly builds should run stress tests under Valgrind, even running the same tests across multiple platforms, compilers, and optimization levels. We maximize the testing offline to mitigate the risk of removing these tests in production.

Never-Failing Functions

As programmers, we've had it drummed into us that every function should return a success or failure status. But, why?

Some functions should never fail. If it's a function that does not access external resources, the most common reasons for failure are internal ones (e.g., called with the wrong parameters) or very rare states (e.g., memory allocation failure). Every one of these reasons are things under our control:

- Don't call it with bad parameters.
- Don't use allocated memory.

As an example, consider a function to set up an order object to submit a trade, which is obviously on the hotpath. This is the traditional C++ style:

```
    bool ok = orderobj.setup(ticker, price);
    if (!ok) {
        // Handle the error...
    }
    // Keep going (submit the trade)
```

Here's a faster method whereby we only check for those "under-our-control" coding issues in offline regression tests. The basic idea is to have the error checks only in test modes:

```
#if SELFTEST // unit test mode
    bool ok = orderobj.setup(ticker, price);
    if (!ok) {
        // Handle the error...
    }
#else  // Production mode (hotpath)
    (void) orderobj.setup(ticker, price);
#endif
    // Keep going (submit the trade)
```

In fact, we probably should further optimize the function to have `void` return type in production, and never even think about returning an error code. We could use tricky `#if` sequences, or have two versions of the entire function. If we make the function `inline`, then the C++ optimizer might get rid of some of the unused `return` statements, but why do we need them in the first place?

The main slowness that we can't get rid of in the hotpath is return codes or exceptions from the third-party APIs, network connections, and system resources, which could really fail in production. However, we already talked about these above, and the strategies to defer these checks to later in the hotpath.

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c
3. Ivan Eduardo Guerra, October 19, 2024, *C++ Design Patterns for Low Latency Applications Including High Frequency Trading*, https://programmador.com/series/notes/cpp-design-patterns-for-low-latency-apps/

Part III: Low-Level Techniques

11. Branch Prediction

What is Branch Prediction?

Branch prediction is an optimization in the CPU whereby efficiency is improved by considering upcoming branches. The CPU in its execution logic tries to predict which of the two paths of a branch is more likely to be taken.

Most CPUs also do "speculative execution" of the future instructions, to get ahead, which must be discarded if the "wrong" branch is actually executed by the code.

For the programmer, these branch prediction capabilities give the opportunity to further optimize your code to capitalize on the CPU's abilities.

Optimization techniques for the C++ programmer include:

- Eliminating branches in the hotpath so that the code runs straight and narrow (i.e., fast!).
- Hinting to the compiler about the most likely branches of execution (e.g., `[[likely]]` and `[[unlikely]]` specifiers).
- Keep unavoidable branches in the same code neighborhood (e.g., short loop bodies).

Branch prediction has a problem in HFT: the hot path is rarely executed (i.e., actually submitting a trade). All of the branch prediction logic would try to run the cold path, as it would always be predicted. But what we want is for the branch prediction logic to always choose the hot path, even though it would mostly fail to be correct.

Thus, all of HFT is at odds with a whole swathe of computing theory about branch prediction. HFT needs a "set opposite world mode" flag, but I'm yet to find one in the GCC documentation.

Types of Branches

First things: analyze your hotpath code for branching. The main types of branches in C++ code include:

- `if` statements and `if-else` statements.
- Loop conditions and loop bodies.
- Loop control statements: `break`, `continue`.
- Function calls and `return` statements.
- `switch` statements (multi-way branching).

Some of the less obvious types of branches are:

- Ternary operator (`?:`)
- Short-circuiting in the `&&` and `||` operators

There are also hidden branches in C++ code features such as:

- Virtual function calls
- Function pointers (and function names)

Branch Compiler Hints

There are several ways for the programmer to give "hints" to the compiler and its optimizer about which pathways are more likely. As always, the compiler is free to ignore hints, so you have to check in the assembly output what effect your changes have. Some of the ways to give hints include:

- `[[likely]]` and `[[unlikely]]` path attributes (C++20).
- `likely()` and `unlikely()` condition markers (C++20)
- `noexcept` attribute (C++11)
- `[[noreturn]]` attribute (C++11)
- `[[assume(expression)]]` attribute (C++23)

GCC also has various extensions available to give hints:

- `__builtin_expect(expression, value)` (GCC extension)
- `hot` (GCC function attribute)

It's common in pre-C++20 Linux code to define your own macro versions for use with the GCC compiler:

```
#define likely(expr)    __builtin_expect((expr), 1)
#define unlikely(expr) __builtin_expect((expr), 0)
```

Branch Profiling

Branch profiling is the recording of pathway stats to analyze the most likely branches. This can also be re-used in the compiler's optimization mode, so that the optimizer can perform branch-aware optimizations. Hence, there is a two-step process whereby better branch prediction can be incorporated into your C++ executable code.

GCC has capabilities to store and use branch prediction statistics in its optimization phase. The arguments to use are:

- `-fprofile-arcs` (GCC command-line argument)
- `-fprofile-generate` (GCC command-line argument)
- `-fprofile-use` (GCC command-line argument)

Following this process will allow GCC to generate more optimal code under assumptions based on branch frequency in its seen executions. Obviously, this is an automatic method, but needs multiple steps in the build:

- Compile without branch hints
- Run the tests
- Output the branch prediction data
- Re-compile the code with branch optimizations enabled

Note that for HFT, the fully hot path (i.e., trade execution) is actually a rare branch, so this historical branch data won't be that useful. One solution is to run GCC in a test mode in which the hotpath is always dummy-executed! Other early parts of the hotpath in HFT can still benefit in both situations, such as the trading decision logic, which is always executed on incoming market data. Obviously, non-HFT applications can always benefit, as the most likely paths are also the most heavily-executed.

Branch Heuristics

In the absence of other branch prediction data, the CPU and compiler tools fall back on some heuristics. Some of the common ones include:

- The `if` code block is more likely to be executed than the `else` code block.
- Loops tend to be executed multiple times.
- Backwards branches are assumed to be loop iterations (and are preferred due to the prior assumption).

Hence, we can make some heuristic recommendations for how to organize your code:

- Put common case code in the `if` block.
- Have error handling in the `else` block.
- Don't use once-only loop executions.

Branch Elimination

The simplest way to avoid branch prediction issues is to have fewer branches. There are various ways to achieve this, ranging from minor code tricks to re-writing your entire algorithm to have fewer conditional tests.

Which branches to eliminate? The worst kinds of branches that need elimination include:

- Long if-else-if sequences
- Nested if-else statements

What data is being tested by a branch condition is also critical, and some of the problematic branches are based on unpredictable conditions:

- Branches depending on user inputs
- Branches depending on random numbers
- Branches depending on system clocks

The best types of conditional tests include:

- Compile-time known tests
- Predictable conditions

The techniques available to eliminate your least favorite branches include:

- Reorganize the overall algorithm to have fewer branches.
- Defer or combine error checking for multiple errors so that there's only one error handling branch.
- Function call optimizations such as inlining and call hierarchy flattening.
- Loop conditional test reductions such as loop unrolling and iteration bounds known at compile-time.
- Branchless programming techniques and tricks to change conditional paths to arithmetic computations.

Branchless Programming Tricks

Branchless programming is a variety of coding tricks to get rid of control flow branches. The main approach is to remove conditional tests, such as `if` statements, by using a variety of arithmetic computations instead. Code that has no branches in a long block can run very fast on a CPU because of instruction prefetching.

Advantages of branchless programming:

- Avoids branch prediction issues (CPU speedup).
- Avoids warp divergence in CUDA C++ (GPU speedup).
- Job security

Possible general software engineering disadvantages of these branchless arithmetic bit tricks:

- Code complexity — isn't it a good thing?
- Unreadable code — as if we care.
- Maintainability — is someone else's problem.

Even worse, the speed benefit might be a mirage. The issues include:

- De-optimizations from too many arithmetic operators — benchmark your tricks!
- Don't underestimate the optimizer's capability on simple code (even if it's "branchy").
- Code tricks can confuse the optimizer (undermining any benefit).
- Memory access costs may dominate over branchless code.

One of the risks with branchless code is that it runs too fast, and gets blocked by memory access delays. Hence, you may need to combine branchless code sequences with software-based memory prefetch primitives, such as with GCC builtins:

- `__builtin_prefetch()`
- `_mm_prefetch()`

Branchless Coding Techniques

Now, let's look at some of the fun tricks in branchless C++ sequences. The various types of methods for branchless coding include:

- Bit masks
- Bit arithmetic (bitshifts, bitwise AND/OR/XOR)
- Mapping Boolean flags to 0 or 1
- Mapping logical operator results to 0 or 1
- Multiplications by 0 or 1 using Booleans
- Lookup tables
- Conditional move (CMOV) assembly statements
- Ternary operator (`?:`)

Some of the more traditional C++ optimizations techniques can also reduce branching as an extra benefit:

- Loop code hoisting of conditional tests.
- Compile-time settings and configurations.

Ternary Operator and CMOV

Using the C++ ternary operator is one way to help the compiler write branchless code. Consider the basic `if` statement:

```
if (x > y) {
    max = x;
}
else {
    max = y;
}
```

This can be more concisely written with a ternary operator:

```
max = (x > y) ? x : y;
```

The ternary operator can be implemented in the compiler backend using a CMOV (conditional move) register assignment statement. This is a branchless instruction that implements the conditional assignment very efficiently.

In theory, both pieces of code are equivalent, and the compiler really should generate identical code. In practice, the use of the ternary operator makes it easier on those poor compiler engineers, because it's 100% guaranteed that an assignment is required, whereas the `if` statement requires a significant amount of extra compile-time static analysis to deduce that both assignments are setting the same variable. The C++ compiler is more likely to emit a branchless CMOV assembly statement with a ternary operator.

Boolean Flags are 0 and 1

Another way to reduce branches is to use Boolean flags in arithmetic, using them as having the values of integer 0 and 1. Here's a simple example:

```
bool inc_flag;
int x = 0;

if (inc_flag) {
    x++;
}
```

This can be implemented in a branchless manner:

```
x += (int)inc_flag
```

Note that the type cast to `int` is not really needed, but helps with readability, and ensures you don't get compiler or static analyzer warnings.

Whether that is faster is something that needs testing because it forces an addition operator into one of the pathways that previously had none, but at least its branchless so it helps with branch prediction.

That was a simple example, but many other ideas are possible. Instead of this:

```
if (clear_flag) x = 0;
```

You can try this branchless version:

```
x *= (int)!clear_flag;
```

It's not clear that this is faster, since multiplication is an expensive operation, but a good compiler can actually notice that it's a fake multiplication over two possible values (0 and 1), and the optimizer can then use a CMOV instruction. Who's to know without checking the assembly code or running a benchmark.

Logical Operators are 0 and 1

In the same vein, the Boolean values of the `&&` and `||` operators can be treated as 0 and 1 in integer arithmetic expressions. Here's an example of the maximum computation:

```
    max = (x > y) * x + (y >= x) * y;
```

Note that the optimizer can notice that a multiplication over a Boolean operand can be replaced with a CMOV, and there are two here. Again, the ternary operator's single CMOV instruction is probably faster than this possible de-optimization, because this version has either two multiplications or two CMOV instructions.

Bitwise XOR Tricks

There's the well-known XOR trick to swap two integer variables without using a temporary:

```
    x = x ^ y;
    y = y ^ x;
    x = x ^ y;
```

Don't worry; nobody understands how this works. But it uses three assignments, no temporary variable, and no branches.

Self XOR to Zero

There's also a well-known assembly language trick of zeroing a register using XOR with itself. The idea is that instead of an "`x=0`" statement, do this:

```
    x ^= x;  // Self XOR
```

The result is zero, and we don't even need to initialize the variable! However, we don't usually do this in C++, but the equivalent is common in assembly listings and compiler backend implementations.

Sign Bit Extension Masks

If you're doing any arithmetic with negative values, you can use bitwise tricks by creating two masks depending on the sign bit. The idea is that the bitmask is:

- All 0's if the number is positive (or zero).
- All 1's if the number is negative.

In other words, the bitmask is 32 bits all set to the same bit value as the sign bit. The bitmask value is either 0 or 0xFFFFFFFF, which is also that artist previously known as `-1`. One way is a ternary operator:

```
    unsigned int mask = (x >= 0) ? 0 : 0xFFFFFFFFu;
```

We can also generate this bitmask using the right bitshift operator and sign extension:

```
    unsigned int mask = x >> 31;
```

Yes, I really should portably compute the bitshift count using the standard constant `CHAR_BIT` and `sizeof(int)` as nicely done in [Farrier, 2025].

Subtraction Bit Mask

Another way to get the same result is by noting the joke about `-1` being the same value. Hence, this trick with subtraction on 2's complement signed integers works:

```
    unsigned int mask = (unsigned) ( (int)(x < 0) - 1 );
```

The comparison generates an integer 0 or 1, and then we subtract 1 to get either 0xFFFFFFFF or 0. Hence, we needed to reverse the comparison test to "<" instead.

All of the type casts are hopefully "free" without runtime costs, and are probably not necessary because implicit conversions would work, anyway.

Example: RELU Activation Function

Let's have a go at making the RELU function branchless. RELU is an "activation function" in LLM backends, and it's quite simple:

```
    if (x < 0) {
        RELU = 0;
    }
    else {
        RELU = x;
    }
```

In other words, change negatives to zero, but leave positives unchanged. Here's the ternary version (faster):

```
    RELU = (x < 0) ? 0 : x;
```

The mask-by-subtraction version combines with bitwise-and to get:

```
    unsigned int mask = (x < 0) - 1;
    RELU &= mask;
```

Another idea for a branchless version of a bitwise RELU is:

```
    unsigned int umask = (x >> 31); // All 0's or 1's
    RELU = (x | umask);
```

Actually, that's buggy, with the bit masking the wrong way around. Here's the correction:

```
    unsigned int umask = ((-x) >> 31); // All 0's or 1's
    RELU = (x | umask);
```

Beware this might be a de-optimization, because the ternary version might be a single CMOV instructions, whereas this version has three operators: negative, right bitshift, and bitwise-AND.

Sign Bitshift Portability

There's a major portability problem with this code, because right bitshift on a negative signed integer is actually undefined behavior in C++. The compiler is free to shift in zero bits or to sign bit extend on the leftmost bit position, in its sole discretion. Hence, you need to check your platform to see what the >> operator does, and whether this rightshift bitmask idea will work.

Note that we cannot fix this by doing the right bitshift on an `unsigned` type, which is guaranteed to shift in a zero bit (well-defined in standard C++, but not what we want). Note also that this is only undefined for right bitshift, not for left bitshift, which is well-defined and always shifts zero bits in on the right side (again, not what we want).

Of course, you can create the sign-based bitmask more portably by avoiding the right bitshift operator, but this loses the branchless benefits:

```
    unsigned int mask = (x >= 0) ? 0 : 0xFFFFFFFF;
```

That's safe and slow, and what's the point of that?

Lookup Tables

Precomputation of lookup tables is a fast way to get a double benefit of fast computation and branchless code. A good example in the standard C++ library are the functions for character types. Here's a slow branching version:

```
    #define islower(c)   (((c) >= 'a') && ((c) <= 'z') )
```

This has lots of computation and there are also branches in the short-circuiting logic of the `&&` operator.

A faster version uses a precomputed lookup table with 256 bytes.

```
    #define islower(c)  _islower_table[(unsigned char)(c)]
```

This is faster and branchless, at the cost of 256 bytes of global memory, and has already been done for you in the standard libraries by those uber-brainy compiler engineers.

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c
3. Paul Alexander Bilokon, Maximilian Lucuta, Erez Shermer, 27 Aug 2023, *Semi-static Conditions in Low-latency C++ for High Frequency Trading: Better than Branch Prediction Hints*, https://arxiv.org/abs/2308.14185, Code: https://github.com/maxlucuta/semi-static-conditions (Advanced branch prediction analysis, a way to do branches by self-modifying code at assembly level.)
4. John Farrier, March 2025, *Branch Prediction: The Definitive Guide for High-Performance C++*, https://johnfarrier.com/branch-prediction-the-definitive-guide-for-high-performance-c/
5. Srdjan Delić, Apr 10, 2023, *Branchless programming — Why your CPU will thank you*, https://sdremthix.medium.com/branchless-programming-why-your-cpu-will-thank-you-5f405d97b0c8
6. Jared Gorski, 11 August, 2020, *Branchless programming*, https://jaredgorski.org/notes/branchless-programming/
7. Algorithmica, March 2025 (accessed), *Branchless Programming*, https://en.algorithmica.org/hpc/pipelining/branchless/
8. Michael Kerrisk, Oct 5, 2012, *How much do __builtin_expect(), likely(), and unlikely() improve performance?* http://blog.man7.org/2012/10/how-much-do-builtinexpect-likely-and.html
9. Agner Fog, 28 May, 2024 (last update), *The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, https://www.agner.org/optimize/microarchitecture.pdf
10. GCC, March 2025 (accessed), *Common Function Attributes*, https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html
11. Algorithmica, July 2025 (accessed), *Binary Search*, https://en.algorithmica.org/hpc/data-structures/binary-search/ (Shows a branchless binary search algorithm with prefetching.)
12. Paul-Virak Khuong, Pat Morin, 15 Mar 2017 (v2), *Array Layouts for Comparison-Based Searching*, https://arxiv.org/abs/1509.05053 (Branchless and cached versions of binary search on sorted arrays.)
13. Agner Fog, 22 June 2024 (last updated), *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*, https://www.agner.org/optimize/optimizing_assembly.pdf

12. Instruction-Level Parallelism

What is Instruction-Level Parallelism?

Instruction-Level Parallelism (ILP) is a CPU optimization performed at the lowest levels of machine instruction processing. If you thought parallel programming was about multithreading, SIMD vectorization and GPU kernels, there's a whole another level deep down in the CPU.

Modern CPUs are amazingly advanced, and they have been architected to use various types of extra parallelism. Some of the types of instruction-level parallelism in a modern CPU include:

- Parallel execution units
- Pipelined execution of micro-ops
- Out-of-order execution of instructions
- Prefetching of instructions
- Branch prediction Memory data prefetching

Importantly, the CPU has total parallelism in its instruction execution units. In fact, a CPU can typically run four or more machine instructions in parallel in the same clock cycle, but using multiple execution units on different parts of the chip.

Instruction Reordering Optimizations

Instruction reordering is a type of Instruction-Level Parallelism (ILP), and is an optimization performed inside the CPU where it actually runs the machine code instructions out-of-order. The way this works in simple terms is:

- Delay any opcodes that don't have the data they need (e.g., from memory).
- Run any instructions that are ready as soon as possible.

There's a whole smash of fun to be had researching how this all works in the CPU. There are schedulers and "stations" and various queues and caches. Kudos to all those hardware engineers.

Another special type of fun is for compiler engineers. GCC does a lot of fancy optimizations in the code generation backend in terms of taking advantage of instruction orders.

But what about C++? Is there anything you can do in C++ to optimize your code? Or with inline assembly instructions?

Safety first. Most of the discussion of out-of-order execution and C++ occurs in relation to safety. Problems can arise across multiple threads if the reads and writes from our C++ statements are running out-of-order. I mean, how can it be good to just run my C++ code in any random order that the CPU chooses?

The issue of preventing out-of-order errors involves "memory order." These are especially useful for correctly implementing lock-free algorithms with atomics, but they also act as memory barriers that can prevent any undesirable types of out-of-order execution.

Speed second. But the goal is to go faster! Rather than stopping the CPU from reordering instructions by using memory barriers, let's maximize it! There are at least two major ideas:

- Minimize memory-waiting delays
- Exploit out-of-order instructions

The first point is to minimize the slowdowns whereby instructions get delayed. The main one is memory accesses, which has well-known solutions such as: cache hit maximization, cache lines, tiled memory accessing, contiguous memory blocks, reducing data sizes, etc.

Other than cache locality, there's not a lot of discussion anywhere in books or on the internet about exploiting out-of-order instruction execution to make code run faster. But there's some discussion of this in Agner Fog's astounding CPU resources; see (Fog, 2024). The key point is:

Free extra parallelism!

The average CPU has hidden parallelism in terms of its various computation pathways. For example, the CPU can run these two computations in parallel:

- Integer arithmetic — Arithmetic-Logic Unit (ALU)
- Floating-point arithmetic — Floating-Point Unit (FPU)

That's not the full list!

Modern CPUs now have more than one ALU, so they can perform two or more integer additions or comparisons in parallel. Some CPUs can also run different types of integer arithmetic, such as addition and multiplication, on separate pathways. Similarly, some of the SIMD operations run separately from the non-SIMD instructions.

Out-of-Order Execution Optimizations

So, you can see the opportunity here, right? Not only can the CPU run the same operations in parallel via SIMD instructions, but it can run two (or more!) different types of computations in parallel.

Unfortunately, the opportunities for huge improvements to your C++ are somewhat limited. For example, if you have a computation with both integer and floating-point computations, can you parallelize them? Yes, but only in limited circumstances, where:

- The two computations don't depend on the results of the other.
- Not requiring memory accesses for the computations.
- Computation operands are values already in CPU registers.

If there's a dependency, they can't run in parallel. And if they both require memory requests, that's the bottleneck regardless of whether the instructions can run in parallel. The data needs to be already loaded from memory into CPU registers to run fast.

That's quite a list of limitations, but it's not insurmountable. The optimization methods include:

- Prefetching the memory (e.g., `__builtin_prefch()` with GCC).
- Removing "dependency chains" from the code sequence of arithmetic instructions.

One common way to remove data dependencies is to use multiple separate variables for intermediate results.

Multiple Accumulator Optimizations

A simple example of using parallel arithmetic computations in a CPU is using multiple accumulator variables for vector dot product. Here's an unrolled version for the dot product:

```
float vector_dot_product_unroll2_ILP(
   const float v1[], const float v2[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i += 2) {
        sum += v1[i] * v2[i];
        sum += v1[i+1] * v2[i+1];
    }
    return sum;
}
```

The problem is there's a data dependency between the two additions. The two multiplications can run in parallel, if the CPU can do so, but the second "`sum+=`" operation must await the completion of the first one. The solution that increases the opportunity for CPU instruction-level parallelism is:

Multiple separate accumulators!

Hence, the code becomes:

```
float vector_dot_product_unroll2(
    const float v1[], const float v2[], int n)
{
    float sum = 0.0f, sum2 = 0.0f; // Two accumulators!
    for (int i = 0; i < n; i += 2) {
        sum += v1[i] * v2[i];
        sum2 += v1[i+1] * v2[i+1];
    }
    return sum + sum2; // Add the accumulators
}
```

This new version now allows the compiler to use out-of-order execution or other instruction-level parallelism optimizations, because the two "+=" operations are now independent inside the loop body.

This function also needs other optimizations applied to it, which are orthogonal to this idea of breaking data dependency chains, such as marking the pointers are "restricted" and using AVX SIMD vectorized instructions.

References

1. Agner Fog, 22 June 2024 (last updated), *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*, https://www.agner.org/optimize/optimizing_assembly.pdf
2. Agner Fog, 28 May, 2024 (last update), *The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, https://www.agner.org/optimize/microarchitecture.pdf
3. Daniel Lemire, April 2018, *Is software prefetching (__builtin_prefetch) useful for performance?* https://lemire.me/blog/2018/04/30/is-software-prefetching-__builtin_prefetch-useful-for-performance/
4. Johnny's Software Lab, March 31, 2024, *The pros and cons of explicit software prefetching*, https://johnnysswlab.com/the-pros-and-cons-of-explicit-software-prefetching/
5. Katecpp, Oct 5, 2015, *Improve performance with cache prefetching*, http://katecpp.github.io/cache-prefetching/

13. Cache Locality

What is Cache Locality?

Cache locality is the idea of staying "local" in our accesses to memory locations to maximize the benefits of some hardware caches in the CPU. There are two general categories of cache locality:

- Instruction cache locality — machine code instruction execution.
- Memory cache locality — data access from memory locations.

There's a lot going on in the CPU in terms of caching accesses and also prefetching possible future accesses. Cache locality is the idea of ensuring that our C++ code maximizes the value of those hardware cache optimizations.

Caching occurs primarily at a lower-level than multithreading, which means that each thread's execution can benefit from these optimizations. Most of the methods to improve cache locality are related to the general code structure, rather than specific ways to do thread synchronization or other multi-threading requirements. The general ideas include:

- Tight code blocks and loops — instruction cache locality.
- Localized and predictable memory access sequences — data cache locality.

You can do both together if you like, since they have orthogonal speedups. Easier said than done!

There are various tools you can use to examine the rates of cache hits and cache misses in the instruction or data caches. Some of the main ones include:

- perf (Linux)
- cachegrind (valgrind)
- Intel VTune
- gperftools
- uprof (AMD)
- likwid-perfctr

Depending on how you look at it, these speedups make cache locality either more or less important in multithreaded applications versus sequential code. It's more important in multithreading because we have lots of threads in different places doing different things, all of which need to have good cache locality. Or maybe it's less important, because the CPU has to throw away all of those per-thread hardware caches at every context switch, so why bother with cache locality? I'll leave it to you to judge that.

Instruction Cache Locality

The instruction cache stores recently executed machine code instructions in a CPU hardware cache. There's also a separate mechanism of "instruction prefetching" to try to load the next instruction that will be executed. As part of this prefetching method, there's also "branch prediction" in the CPU, which attempts to predict which of two branch directions will get chosen.

To get the best out of these instruction speedups, our C++ code should generally use:

- Short and tight loops
- Fewer branches

Keeping loops short will mean that the CPU stays within the same block of code, maximizing the chances that it already has an instruction in its cache. Interestingly, this means that some common code optimizations can be bad for instruction cache locality:

- Inlining of functions
- Loop unrolling

Both of these can cut both ways, since they both reduce branches, but also lengthen code blocks. Whenever you're tempted to maximize your use of such optimizations, think about the plight of the poor instruction cache as it tries to keep up.

Branches are another separate issue from short code blocks. In fact, long code sequences of compute instructions are fine for branch prediction. To maximize the CPU's branch prediction capability, we should either have few branches, or at least have very predictable branches. At the limit, we could use branchless programming, which is a set of tricks to get rid of branches. See Chapter 4 for more on branch prediction and branchless coding methods.

Data Cache Locality

There are numerous improvements that you can make to improve cache locality for the memory access caches. And there are rather a lot of different caches for CPU memory accesses:

- L1 and L2 caches (per-thread)
- L3 cache (shared)
- TLB cache (virtual address accesses)
- NUMA multi-core caching

There are some general recommendations for the entire application, that aim to reduce memory cache misses:

- Use less memory!
- Fewer memory allocations
- Smaller data sizes

But particular algorithms can also be modified to keep nearby memory in the caches. Data structures can affect the level of cache locality, with improvements such as:

- Separate cold data from hot data — improve cache locality for hot data.
- Structure of Arrays (SoA) vs Array of Structures (AoS) — which one is best depends on the context.
- Contiguous data structures — arrays and vectors, not linked lists or trees.
- Compact data structures — smaller memory sizes are easier to maintain in the cache.

The code execution of various algorithms can alter the sequence of memory accesses, and thereby maximize cache locality. Some well-known improvements include:

- Loop segmenting — process short sub-sequences of a longer array.
- Tiling algorithms — process 2D "tiles" in a matrix or multidimensional data structure (also called "blocking").

The goal of these algorithm modifications is to iterate over a small sub-section in the data, keeping cache locality during that "hot" computation, and then move on to the next part. This works particularly well with matrix multiplication, because it involves multiple computations with every element of the matrix.

There are also some dynamic approaches whereby you can manually ensure that data is already in the cache when you need it:

- Memory prefetching
- Cache warming

See Chapter 3 for more about prefetching and cache warming.

Memory Hierarchy

To fully understand the caches, we need to know of all the different types of memory used in a C++ program. Handling memory properly is one of the most important parts of C++ optimization, because memory access is much slower than the CPU. Memory is the bottleneck, and you need to know where the compiler puts everything.

Learn to love the linker-loader!

When your program starts running, the "loader" puts all sorts of things in different places. The basic moving parts that happen *before* execution starts are:

- Instructions — the code's machine instructions.
- Global read-write memory — initialized or zero-initialized global variables.
- Read-only data — string literal data.

To get deeper into the memory segments used by the linker-loader, these are the main ones:

- text — stores the machine code instructions (read-only, executable)
- bss — all zero'd global data such as global arrays without non-zero initializers (read-write)
- data — Initialized non-zero global variable data (read-write)
- rodata — read-only data such as string literals or constants (read-only)

Yes, the "text" segment has a confusing name, and it's sometimes called the "code" segment. According to Wikipedia, BSS stands for "Block Started by Symbol," but you didn't need to know that.

All of the above segments are statically resolved, for the most part, by the linker. However, once the program gets going, there are more dynamic allocations of memory within its virtual address space.

The main types of dynamic memory are:

- Stack memory — the function call stack with parameters and local variables (also `alloca`).
- Heap memory — dynamically allocated by the C++ `new` operator or the older `malloc` function.
- Thread-local storage — via the "`thread_local`" keyword (C++11).

See Chapter 8 for more about reducing stack and heap memory, and now let's discuss thread-local storage.

Thread-Local Storage

Thread-Local Storage (TLS) is memory that is exclusive to a particular thread. The other threads do not have access to it. In C++, this is defined via the "`thread_local`" keyword, available since C++11. The usage is simple:

```
    thread_local int tls_variable;
```

There are also some earlier and non-standard versions:

- `_Thread_local` — older version of specifier.
- `__thread` — GCC non-standard modifier with similar semantics.
- `__declspec(thread)` — on Microsoft C++.

The key features of `thread_local` variables are:

- Accessible in one thread only.
- Persistent memory storage.
- Variables, objects or arrays only (cannot have a `thread_local` function).

Per-thread access. If you declare a variable as "`thread_local`" then the C++ compiler has to ensure the semantics. Accesses to that variable in C++ must go to the version of that variable for the current thread. Typically, this means that the variable has multiple copies, with different addresses for each thread.

How is it implemented? It's not necessarily using any particular hardware support behind the scenes, and it's not necessarily using any magic per-thread caching.

The C++ compiler can allocate different addresses per thread to the same data, and then ensure that accesses within each thread get the correct version. After all, the C++ compiler knows that a particular variable is "`thread_local`" because it's a type specification.

Persistent memory semantics. The thread_local specifier is very similar to the static keyword in terms of its memory persistence. Its effect is similar to:

- Global variables (with external scope linkage)
- `static` file-scope variables
- `static` local variables (in a function)
- `static` data members (in a C++ class)

A `thread_local` variable is created when a thread starts and destroyed when the thread finishes. This has some implications:

- At most one copy is created at program startup.
- Dynamically created (along with the thread itself).
- Does not persist across thread shutdown and restarts.

Note that persistence and scope are different things. Persistence is whether the data is maintained across multiple accesses, whereas scope is simply whether its name can be referenced within code statements.

For example, if you use a `thread_local` variable as a local variable in a function, its value will persist across invocations to that function, and always have the same address. However, it's scope is limited to within the function, where its name is accessible. This is the same as a `static` local variable, but with the extra semantics that only one thread can see this version. If multiple threads call the function, they'll get different versions of the `thread_local` variable inside the function.

Thread-local variables occupy a special niche in the programmer's bag of tricks. You don't need to wrap accesses with any locking or other synchronizations, which is nice. They are like atomics, in that they cannot be messed up by another thread, but unlike atomics because they are not shared across threads. The main usage is to have some shared code, but also have a special non-shared variable, especially where you want the variable to persist, such as having per-thread counters, flags, intermediate calculations, and so on.

References

1. Wikipedia, May 2025 (accessed), *.bss*, https://en.wikipedia.org/wiki/.bss
2. Milan Stevanovic, 2014, *Advanced C and C++ Compiling*, Apress, https://www.amazon.com.au/dp/B01HXFLQH0/
3. John R. Levine, 1999, *Linkers and Loaders*, Morgan Kaufmann, https://www.amazon.com/dp/1558604960
4. CPP Reference, May 2025 (accessed), *Storage class specifiers*, https://en.cppreference.com/w/c/language/storage_class_specifiers.html
5. Microsoft, 2021 *Thread Local Storage (TLS)* https://learn.microsoft.com/en-us/cpp/parallel/thread-local-storage-tls
6. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/

14. Cache Warming

What is Cache Warming?

Cache warming is a specific type of prefetching optimization aimed at keeping the various memory caches fresh. It typically involves scanning through all the memory data required for the "hot path," even though there's no real intention to use the data (until later). The hot path needs a warm cache, so that when the hot path is executed (e.g., a trade execution in HFT), then memory accesses are very fast.

There are multiple ways to trigger prefetching of data to keep the cache warm:

- Low-level C++ prefetching primitives.
- Copy to `volatile` temporary variables.
- Explicit dry-run parameters in the code.

Unlike other types of CPU prefetching, cache warming is something your C++ code does directly, rather than a hardware-enabled feature. It's up to you to determine what data is needed the most in hot path computations, and then pre-load that data on every pass-through. You effectively do a "dry run" of the hot path, but access the memory to ensure it's maintained in the cache.

Note that cache warming is not always a guaranteed win. Using the "dry run" approach can end up with a lot of extra conditional tests:

```
if (!dry_run) {
    // Do something
}
```

This can negatively impact performance in two ways:

- Runtime cost of testing the flag, and
- Extra branches of code that slow down CPU branch prediction.

As with everything in multithreading, you really need to time it to see if these costs are less than the gain from faster memory cache accesses.

Memory Prefetch Primitives

Although you can "manually" prefetch data in basic C++ code, there are also some builtins that are convenient for larger amounts of data. Some of the C++ primitives to use for cache warming include:

- `__builtin_prefetch` (GCC)
- `_mm_prefetch` (GCC)

Prefetching is more effective on some data structures than others, with a general preference for contiguous data blocks. Cache locality issues in the "cache lines" with size 64-256 bytes are another reason. As a practical example, contiguous arrays are better than dispersed data structures liked links lists and trees. This means that `std::vector` contiguous memory layouts can be more effectively prefetched than the spread-out memory used by `std::list` objects.

Volatile Temporary Variables

Another approach for manual prefetching is the use of `volatile` specifier on temporary variables. By assigning data to a `volatile` temporary variable, the optimizer cannot remove an apparently unused assignment. For example, consider if we do this:

```
    int temp = my_order_book[0];
```

The C++ compiler may notice that "`temp`" is not used anywhere else, so it can throw away that entire assignment statement into nowhere. The solution is to use the `volatile` specifier:

```
    volatile int temp = my_order_book[0];
```

The compiler is forced to load the data into memory even when it seems to be unused by the remainder of the code, because assigning data to a `volatile` variable is itself a side-effect.

Note that we only want to declare temporary variables as `volatile`, but not the shared global data arrays we're trying to prefetch. We don't want the main data structures to have this status. If our main global variables or arrays were declared as `volatile`, this would actually interfere with having them loaded from the memory caches. They would be uncached!

Dry-Run Executions

A simple approach to cache warming is to still execute all the steps, even if you're not going to do anything. For example, in HFT, you could call the "execute trade" function even if the decision is to *not* trade any stocks.

The method is simply to pass a Boolean flag indicating a "dry run" or "test run" or "warm-up run" or whatever term you like. A simple conceptual example:

```
if (!dry_run) {
    orderobj.setup(ticker, price);
    execute_trade(orderobj);
}
```

A better way to get more cache warming is to populate all the objects as if you were going to actually do a trade. At the very last step, the dry-run flag is tested, and no trade gets submitted.

```
orderobj.setup(ticker, price);
if (!dry_run) {
    execute_trade(orderobj);
}
```

But we really want to warm up the entire path, even the trade execution logic. Hence, we go deeper by passing the flag inside:

```
orderobj.setup(ticker, price);
execute_trade(orderobj, dry_run);
```

And our trade execution code looks like:

```
void execute_trade(Order &order, bool dry_run)
{
    if (!dry_run) {
        g_order_count++;  // Count total
        // Other accounting stuff..
        // Submit the order...
    }
}
```

That isn't really much better, is it? We didn't warm anything extra, but just pushed the test inside the function.

Double Data Trouble

We really need to actually prefetch some data! One way is to double up all our data. The basic data for order count tracking is like this:

```
int g_order_count = 0;
```

One common trick is to use an array of two values with two meanings:

- Live data
- Dry-run data (unused)

Hence, our order count becomes:

```
int g_order_count[2] = { 0, 0 };
```

Then we can try this:

```
if (!dry_run) {
    g_order_count[0]++;  // Live run
}
else {
    g_order_count[1]++;  // Dummy
}
```

The point of the dummy is that we access the `[1]` array element in order to warm up the `[0]` element (without changing it). This works because of "false sharing" with "cache lines," which is often a slowdown problem, but here they offer an advantage. We can warm the cache by touching adjacent array elements, without disturbing the main data. (Note that here we don't use the `alignas` trick to avoid false sharing, because we actually want it to occur!)

In the spirit of branchless programming, we can make this code tighter by mapping the Boolean flag to 0 and 1 integer values:

```
g_order_count[(int)dry_run]++;
```

Note that we have actually added extra computation to our hot path! Instead of a global variable increment, it's now an array index lookup plus the increment.

We need to measure our optimizations to ensure that the gain from memory cache warming is greater than the extra cost of these array indexing operations. (We've also added a large amount of extra computation to our cold path, including whole extra function invocations, but we care less about that.)

Our conceptual trade execution routine starts to look like:

```
void execute_trade(Order &order, bool dry_run)
{
    g_order_count[(int)dry_run]++;  // Count total
    // Other accounting stuff.. same tricks
    if (!dry_run) {
        // Submit the order...
    }
}
```

The idea is that our "dry run" mode has run over as much of the code as possible, only stopping short of actually submitting the order. By maintaining the two copies of all data, with dry-run and live values, we can prefetch all of those arrays into memory caches.

Problems with Cache Warming

The above cache warming double-array trick has used false sharing of cache lines for good, not evil. And yet it has a problem: false sharing.

Our use of false sharing was harmless (and helpful) because we assumed only a single thread was in use. There's no cache invalidation slowdown when it's only one thread. The cache warming idea for the L1 and L2 caches requires a single thread, although the L3 cache can be warmed for multiple threads. Hence, this cache warming idea has limitations:

- Single thread required for all order submissions (if you want L1/L2 cache warming).
- Thread pools and other multi-thread design patterns are therefore problematic.

We cannot really have a thread pool model where each consumer thread could potentially submit a trade. The above cache warming logic only works for one thread. If we try to use multiple threads, our cache warming logic is actually a cache freezing de-optimization, because we've got the "false sharing" problem for real.

Even worse, consider what happens if we try to use a thread pool model with the following modifications:

(a) multiple consumers, where each thread tries to decide whether to trade,

(b) single trade submission thread.

In other words, multiple decider threads, where each decider then hands off to the single trading thread (which is kept warmed).

But then we've made another conceptual error. The hot path should really include the decision logic, as the overall latency is from receiving incoming data to submitting a trade. However, we haven't kept the cache warm for these multiple "decider" threads, particularly so for all the data they use in deciding whether to trade, so the decision modules won't run fast.

Possible solutions include:

- Single thread for all decision and order submission (with L1/L2 warming), or
- Keep multiple threads warm (tricky!), or
- Modify the cache warming code tricks to use reads only, not writes (avoiding the cache invalidation problem), or
- Only warm up the L3 cache (for multiple threads).

But these solutions have additional problems:

- Single order thread idea lacks a failover or backup plan.
- Single order thread cannot issue two trades without blocking.
- Warming multiple threads means each thread needs its own copy of the data.

None of these solutions are great, so that's why they pay you the big bucks.

Further Optimizing Cache Warming

Another further iteration of advanced cache warming would be to actually submit a dummy order, such as if the exchange connectivity allowed the sending of test-only transactions. Doing this would allow us to keep warm any of the data structures that are actually inside the client API of the exchange connection.

The advantage of the use of dry-run cache warming is that all the various data structures used to prepare a trade are kept warm in the memory caches (L1/L2/L3). The downside is extra processing that occurs whenever you're not trading. In other words, there are extra computations done on the "cold path" every time, just to keep the "hot path" all snuggly and warm.

The code to traverse all the memory data structures can be a significant cost in itself, although it only occurs during the cold path. There are several advanced tweaks to optimize your cache warming code:

- Exploit cache line sizes for quicker loading of contiguous data.
- Limit cache warming to the total L1/L2/L3 cache size.

A further optimization of cache warming is to use "cache lines" to your advantage. The L1/L2 caches don't work on individual bytes, but on blocks of memory called "cache lines", which are usually sized between 64 bytes and 256 bytes (e.g., Intel is usually 64 bytes, Apple M2 is 128 bytes, some other CPUs are 256 bytes). Hence, to load a "cache line" of 64 bytes on an Intel CPU, you need to load one of the bytes from the 64-byte block. Your C++ code doesn't need to explicitly touch every element of a vector to have the entire vector hot as a fresh-baked oven loaf in the cache. Admittedly, this doesn't speed up the hot path itself, but only the preliminary cache warming code.

An important limitation of cache warming is the maximum sizes of the L1, L2, and L3 caches. If you're trying to warm up the CPU cache for your 7B AI model, that's 7 billion floating-point numbers, and trying to keep them all in the CPU cache isn't going to work. On the other hand, you can probably preload an entire 7B model into the CPU RAM (i.e., global memory, not the caches), or into the GPU's VRAM, but that's preloading not cache warming, and it's a slightly different story.

If you know your CPU's cache size, you can optimize your cache warming strategy by only trying to prefetch that much data. If you load more data than the cache size, the newly warmed data is just evicting other data from the cache that you prefetched earlier in the warming code. Hence, prefetching exactly the amount of data equal to your CPU cache size is the optimal cache warming strategy.

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c
3. Edelweiss Global Markets Oct 14, 2024, *Cache-Warming*, https://edelweissgm.github.io/hft/2024/10/14/CacheWarming.html
4. Ibrahim Essam, Jul 19, 2024, *Cache warming and memory access*, https://ibrahimessam.com/posts/cache/
5. Nimrod Sapir, 2019, *High-Frequency Trading and Ultra Low*, Latency Development Techniques, https://corecppil.github.io/CoreCpp2019/Presentations/Nimrod_High_Frequency_Trading.pdf, Code: https://github.com/DanielDubi/StaticFlatMap
6. Daniel Lemire, April 2018, *Is software prefetching (__builtin_prefetch) useful for performance?* https://lemire.me/blog/2018/04/30/is-software-prefetching-__builtin_prefetch-useful-for-performance/
7. Johnny's Software Lab, March 31, 2024, *The pros and cons of explicit software prefetching*, https://johnnysswlab.com/the-pros-and-cons-of-explicit-software-prefetching/
8. Katecpp, Oct 5, 2015, *Improve performance with cache prefetching*, http://katecpp.github.io/cache-prefetching/

15. AVX Intrinsics

What are AVX Intrinsics?

AVX intrinsics are SIMD parallel instructions for x86 and x64 architectures. They are actually machine opcodes supported by the x86/x64 CPU, but are wrapped in the intrinsic prototypes for easy access from a C++ program.

The main advantage of SIMD instructions is that they are CPU-supported parallel optimizations. Hence, they do not require a GPU, and can even be used on a basic Windows laptop. The main downside is that their level of parallelism is nowhere near that of a high-end GPU.

There are multiple generations of the AVX intrinsics based on the x86/x64 CPU instructions. Different CPUs support different features, and exactly which intrinsic calls can be used will depend on the CPU on which your C++ is running. The basic AVX types are:

- AVX — 128-bit registers = 4 x 32-bit `float` values
- AVX-2 — 256-bit registers = 8 x 32-bit `float` values
- AVX-512 — 512-bit registers = 16 x 32-bit `float` values
- AVX-10 — 512-bit registers (with speedups)

The AVX intrinsics use C++ type names to declare variables for their registers. The `float` types used to declare the registers in AVX using C++ all have a double-underscore prefix with "`__m128`" for 128-bit registers (4 `float`s), "`__m256`" for 256 bit registers (8 `float`s), and "`__m512`" for 512 bits (16 `float`s).

Similarly, there are also register type names for `int` types (`__m128i`, `__m256i`, and `__m512i`), and types for "`double`" registers (`__m128d`, `__m256d`, and `__m512d`).

AVX intrinsic functions and their types are declared as ordinary function prototypes in header files. The header files that you may need to include for these intrinsics include `<intrin.h>`, `<emmintrin.h>`, and `<immintrin.h>`.

Useful AVX SIMD vector intrinsics for `float` types include:

- Initialize to all-zeros — `_mm_setzero_ps`, `_mm256_setzero_ps`
- Set all values to a single `float` — `_mm_set1_ps`, `_mm256_set1_ps`
- Set to 4 or 8 values — `_mm_set_ps`, `_mm256_set_ps`
- Load arrays to AVX registers — `_mm_loadu_ps`, `_mm256_loadu_ps`
- Store to `float` arrays — `_mm_storeu_ps`, `_mm256_storeu_ps`
- Addition — `_mm_add_ps`, `_mm256_add_ps`
- Multiplication — `_mm_mul_ps` (SSE), `_mm256_mul_ps` (AVX-2)
- Vector dot product — `_mm_dp_ps`, `_mm256_dp_ps`
- Fused Multiply-Add (FMA — `_mm_fmadd_ps`, `_mm256_fmadd_ps`
- Horizontal addition (pairwise) — `_mm_hadd_ps`, `_mm256_hadd_ps`

Note that the names of the intrinsic functions have meaningful suffixes. The "`_ps`" suffix means "packed-single-precision" (i.e., `float`), whereas "`_pd`" suffix means "packed-double-precision" (i.e., `double`).

AVX Operations

The main SIMD instructions are called "vertical" instructions, by convention. They take one vector and a second vector (e.g., both are 128-bit), apply an operation element-wise in parallel, and put the result into a third register. In other words, they return the result of a "pair-wise" or "element-wise" operation on two vectors into a third vector.

For example, vertical addition requires two input vectors and will output a third vector with the sums. AVX-512 SIMD addition will add two 512-bit registers full with `float` values on a paired element basis (i.e., adds up 16 pairs of the 32-bit `float` values), yielding a third 512-bit vector with the result (16 `float` values).

Binary operations. The full list of binary AVX operations is very long. Supported AVX operations include:

- Multiplication
- Addition
- Subtraction
- Division
- Maximum
- Minimum
- Fused Multiply-Add (FMA)
- Bitwise operations

Unary operations. AVX unary intrinsics apply a particular function to all elements of an AVX register in parallel, and return the resulting register. Supported AVX unary operations include:

- Clear to zero
- Set to a constant
- Casts
- Conversions
- Popcount (POPCNT)
- Leading-zero count (LZCNT)

Mathematical Functions. Simple float-to-float mathematical functions are effectively a type of unary operator. AVX supports a variety of functions with vector hardware instructions, such as:

- Absolute value: `abs`
- Error function: `erf`
- Reciprocal
- Rounding, ceiling, floor
- Roots: `sqrt` (square root), cube root
- Inverted roots (e.g., `invsqrt`)
- Exponential: `exp`, `exp10`
- Logarithm: `log`, `log10`
- Trigonometric functions
- Hyperbolic functions
- Statistics (e.g., Cumulative Distribution Function)

AVX Horizontal Intrinsics

Horizontal operations refer to arithmetic across the values within one vector. AVX intrinsics exist to do "horizontal" operations across the same vector, such as adding horizontal elements of a vector, or finding the maximum of pairs of elements within a vector.

Horizontal SIMD instructions are typically designated with a "h" prefix (e.g., "horizontal add" is "hadd"). More specifically, the intrinsic for 128-bit horizontal add is "`_mm_hadd_ps`" and it is "`_mm256_hadd_ps`" for 256-bits.

However, do not make the mistake of assuming that these horizontal AVX intrinsics are a "reduction" of a vector down to a single float (i.e., vector-to-scalar). I mean, they really should do exactly that, but that would be too good to be true.

The horizontal intrinsic functions are still effectively "pairwise" operations for AVX and AVX-2, except the pairs are within the same vector (i.e., horizontal pairs). If you want to add all elements of a vector, or find the maximum, you will need multiple calls to these intrinsics, each time processing pairs of numbers, halving the number of elements you are examining at each iteration. Hence, for example, summing all the `float` values in a vector with AVX or AVX-2 uses a method of "shuffle-and-add" multiple times.

Thankfully, AVX-512 actually does have horizontal reductions that process all the elements in their 512 bit registers. Hence, the 512-bit horizontal add uses a different naming convention and uses the prefix of "reduce add" in the intrinsic name (e.g., `_mm512_reduce_add_ps` is a summation reduction). In other words, this reduction operates in parallel on all 16 `float` values in an AVX-512 register, and the `_mm512_reduce_add_ps` intrinsic can add up all 16 `float` values in one operation. This horizontal reduction summation is useful for vectorizing functions such as average, and could be used for vector dot products (i.e., do an AVX-512 SIMD vertical multiplication into a third vector of 16 `float` values, then a horizontal reduction to sum those 16 `float` values), although there's an even better way with FMA intrinsics.

Supported AVX horizontal operations for pairwise horizontal calculations (AVX or AVX-2) or vector-to-scalar reductions (AVX-512) include floating-point and integer versions, with various sizes, for primitives, such as:

- Addition
- Maximum
- Minimum
- Bitwise operations

Portability Checking of AVX Versions

The power of AVX support has changed over the years, with different CPUs having different capabilities, not only with AVX, AVX-2 and AVX-512, but also their sub-releases. And it's also a little unclear into the future, with reports that some of the newer Intel chips have AVX-512 disabled.

If you write some code using AVX-512 intrinsics, and compile your C++ into an executable with the AVX-512 flags on, and then it runs on a lower-capability CPU without AVX-512, what happens? Do the AVX-512 intrinsics fail, or are they simulated somehow so that they're slower but still work? Answer: kaboom on MSVS. In the MSVS IDE, if you try to call these intrinsics on a CPU that doesn't support it, you get "unhandled exception: illegal instruction."

In other words, the C++ compiler still emits the AVX-512 instruction codes, but they aren't valid, so it excepts at runtime.

Hence, the calls to AVX-512 are not emulated at run-time on lower-capability CPUs. And they aren't checked, either. That's up to you!

Dynamic test required: Firstly, you cannot use the preprocessor. You can't test `#if` or `#ifdef` for whether you've got AVX-512 in the CPU or not. You can use the preprocessor to distinguish between different platforms where you'll compile a separate binary (e.g., ARM Neon for phones or Apple M1/M2/M3 chipsets). But you cannot choose between AVX/AVX-2/AVX-512 at compile-time, unless you really plan to ship three separate binary executables. Well, you probably could do this if you really, really wanted to.

The other thing you don't really want to do is low-level testing of capabilities. You don't want to test a flag right in front of every AVX-512 intrinsic call. Otherwise, you'll lose most of the speedup benefits. Instead, you want this test done much higher up, and then have multiple versions of the higher-level kernel operations (e.g., vector add, vector multiply, vector dot product, etc.)

What this means is that you have to check in your runtime code what the CPU's capabilities are, at a very high level in your program. Hence, it is important to check your platform has the AVX support that you need, such as via the "`cpuid`" intrinsic at program startup. Then you have a dynamic flag that specifies whether you have AVX-512 or not, and you can then choose between an AVX-2 dot product or an AVX-512 dot product, or whatever else, during execution. Obviously, it gets a bit convoluted when you have to dynamically choose between versions for AVX, AVX-2 and AVX-512 (not to mention all the AVX sub-capabilities and also AVX-10 coming soon).

Example: Basic AVX SIMD Multiply

Let us do a basic element-wise SIMD multiply using AVX (version 1) and its 128-bit registers. This will do a paired vector multiply an array of 4 `float` numbers (i.e., 4 x 32-bit `float` = 128 bits). Each `float` in the resulting array is a pairwise multiplication of the elements in the two operands.

This is how SIMD instructions work, by operating on each element of the array (i.e., "pairwise" or "element-wise"). For example, a "vertical" multiply will take the 4 `float` values in one input array, and multiply each of them by the corresponding `float` in the other input array of 4 `float` numbers, and then will return a resulting output array with 4 `float` values.

For testing, let us assume with want to create an AVX function that multiplies 4 `float` values element-wise. The test code looks like:

```
    float arr1[4] = { 1.0f , 2.5f , 3.14f, 0.0f };
    float arr2[4] = { 1.0f , 2.5f , 3.14f, 0.0f };
    float resultarr[4];
    // Multiply element-wise
    aussie_multiply_vectors(arr1, arr2, resultarr, 4);
```

Testing the results of the multiply as an element-wise multiply of each pair in the 4 `float` values (using my home-grown "`aussie_testf`" unit testing function that compares `float` numbers for equality):

```
    aussie_testf(resultarr[0], 1.0f * 1.0f); // Unit tests
    aussie_testf(resultarr[1], 2.5f * 2.5f);
    aussie_testf(resultarr[2], 3.14f * 3.14f);
    aussie_testf(resultarr[3], 0.0f * 0.0f);
```

Here's the low-level C++ code that actually does the SIMD multiply using the "`_mm_mul_ps`" AVX intrinsic function:

```
    #include <xmmintrin.h>
    #include <intrin.h>

    void aussie_avx_multiply_4_floats(
        float v1[4], float v2[4], float vresult[4])
    {
        // Multiply 4x32-bit float in 128-bit AVX registers
        __m128 r1 = _mm_loadu_ps(v1);    // Load floats
        __m128 r2 = _mm_loadu_ps(v2);
        __m128 dst = _mm_mul_ps(r1, r2); // AVX Multiply
        _mm_storeu_ps(vresult, dst); // Convert to floats
    }
```

Explaining this code one line at a time:

1. The header files are included: `<xmmintrin.h>` and `<intrin.h>`.

2. The basic AVX register type is "`__m128`" which is an AVX 128-bit register (i.e., it is 128 bits in the AVX version, not AVX-2 or AVX-512).

3. The variables "`r1`" and "`r2`" are declared as `_mm128` registers. The names "`r1`" and "`r2`" are not important, and are just variable names.

4. The intrinsic function "_mm_loadu_ps" is used to convert the arrays of 4 float values into the 128-bit register types, and the result is "loaded" into the "r1" and "r2" 128-bit types.

5. Another 128-bit variable "dst" is declared to hold the results of the SIMD multiply. The name "dst" can be any variable name.

6. The main AVX SIMD multiply is performed by the "_mm_mul_ps" intrinsic function. The suffix "s" means "single-precision" (i.e., 32-bit float). This is where the rubber meets the road, and the results of the element-wise multiplication of registers "r1" and "r2" are computed and saved into the "dst" register. This computation is analogous to the basic C++ expression:

```
dst = r1 * r2;
```

7. The 128-bit result register variable "dst" is converted back to 32-bit float values (4 of them), by "storing" the 128 bits into the float array using the "_mm_storeu_ps" AVX intrinsic.

AVX Memory Alignment Issues

The above example glosses over the issue of managing "alignment" of memory addresses on byte boundaries with the "alignas" specifier. Some of the AVX SIMD intrinsic calls require that addresses are 16-byte aligned (i.e., this is effectively 128-bit alignment), which is not guaranteed by the C++ compiler. However, we've tolerated non-aligned addresses by using the "_mm_storeu_ps" intrinsic, which works with either aligned or non-aligned addresses.

Note that alignment restriction requirements of AVX are somewhat in flux. Not all AVX intrinsics require alignment, and they are "relaxed" in many cases. There have also been some bugs in compiler toleration of non-aligned addresses in C++ intrinsics. Where required, the alignment needs are:

- AVX-1 — 16-byte alignment (128-bit).
- AVX-2 — 32-byte alignment (256-bit).
- AVX-512 — 64-byte alignment (512-bit).

Since we can sort out alignment at compile-time using the C++ "alignas" specifier and "aligned" type attributes, there is no performance penalty (except in terms of space) for ensuring greater compatibility across CPU platforms and compiler versions by preferring aligned addresses.

You can create your own macros to easily test pointer addresses for alignment by checking their remainder with the `%` operator. These examples use bitwise-and to replace the slow remainder operator:

```
#define aussie_is_aligned_16(ptr) \
    ((((unsigned long)(ptr)) &15ul) == 0)
#define aussie_is_aligned_32(ptr) \
    ((((unsigned long)(ptr)) &31ul) == 0)
```

Although our code to multiply 4 `float` values tolerates non-alignment, it's a minor slug. The "`_mm_storeu_ps`" AVX intrinsic is slower if the addresses are not aligned, so we should fix the alignment for performance reasons. There's also another "store" intrinsic to convert from 128-bits to 4 floats called "`_mm_store_ps`" (without the "u") that runs faster, but does not tolerate non-aligned `float` arrays.

Actually, "`_mm_storeu_ps`" is supposed to be equally as fast as the alternative "`_mm_store_ps`" if the address is correctly aligned, so we can still use that intrinsic if we prefer safety, but we need to change the variables to be aligned on 16-byte boundaries for a speedup.

To ensure alignment in C++, there is an "`alignas`" specifier for variable declarations. We can use "`alignas(16)`" to force C++ to create the variables with 16-byte alignment of the address where they are stored.

For example, our unit test harness code could have ensured 16-byte alignment of all memory addresses via:

```
// Test with 16-byte alignment
alignas(16) float arr1[4] = { 1.0f , 2.5f , 3.14f, 0.0f };
alignas(16) float arr2[4] = { 1.0f , 2.5f , 3.14f, 0.0f };
alignas(16) float resultarr[4];
```

There are various non-standard alternatives to "`alignas`" in the various compilers. For example, MSVS has "`__declspec(align(16))`" with two prefix underscores, and GCC supports "`decltype(align(16))`".

The AVX code for an alignment-requiring version is not much different, with minor changes to the names of the C++ intrinsics:

```
void aussie_avx_multiply_4_floats_aligned(
      float v1[4], float v2[4], float vresult[4])
{
    // Use 128-bit registers to multiply 4x32-bit floats...
    __m128 r1 = _mm_loadu_ps(v1);    // Load floats 128-bits
    __m128 r2 = _mm_loadu_ps(v2);
    __m128 dst = _mm_mul_ps(r1, r2);  // Multiply
    _mm_store_ps(vresult, dst);  // Aligned convert to float
}
```

Ideally we'd like to ensure that the function is only called with aligned addresses at compile-time. The first attempt is to declare "`vresult`" above as "`alignas(16)`" for type checking of alignment issues, but it fails for function parameters. Fortunately, there's another way using type attributes:

```
__attribute__((aligned(16)))
```

Another method is to define our own assertion that uses bitwise tests on the address instead:

```
#define is_aligned_16(ptr) \
    ((((unsigned long int)(ptr)) & 15) == 0)
```

This tests the address is a number that is a multiple of 16 using bitwise-and with 15, but this is at runtime and costs extra cycles.

AVX-2 SIMD Multiplication

Here is the AVX-2 version of pairwise SIMD multiply with intrinsics for 256-bit registers, which is eight 32-bit `float` variables.

```
void aussie_avx2_multiply_8_floats(
    float v1[8], float v2[8], float vresult[8])
{
    // Multiply 8x32-bit floats in 256-bit AVX2 registers
    __m256 r1 = _mm256_loadu_ps(v1);   // Load floats
    __m256 r2 = _mm256_loadu_ps(v2);
    __m256 dst = _mm256_mul_ps(r1, r2);  // Multiply (SIMD)
    _mm256_storeu_ps(vresult, dst);  // Convert to 8 floats
}
```

This is similar to the basic AVX 128-bit version, with some differences:

- The type for 256-bit registers is "`__m256`".
- The AVX-2 loading intrinsic is "`_mm256_loadu_ps`".
- The AVX-2 multiplication intrinsic is "`_mm256_mul_ps`".
- The conversion back to float uses AVX-2 intrinsic "`_mm256_storeu_ps`".

AVX-512 SIMD Multiplication

Here is the basic 16 `float` SIMD vector multiplication using 512-bits in AVX-512.

```
void aussie_avx512_multiply_16_floats(
    float v1[16], float v2[16], float vresult[16])
{
    // Multiply 16x32-bit floats in 512-bit registers
    __m512 r1 = _mm512_loadu_ps(v1); // Load 16 floats
    __m512 r2 = _mm512_loadu_ps(v2);
    __m512 dst = _mm512_mul_ps(r1, r2); // Multiply (SIMD)
    _mm512_storeu_ps(vresult, dst);  // Convert to floats
}
```

Note that AVX-512 will fail with an "unhandled exception: illegal instruction" (e.g., in MSVS) if AVX-512 is not supported on your CPU.

Example: AVX 128-Bit Dot Product

The AVX instruction set has a vector dot product intrinsic that wraps an x86 dot product instruction. There are versions of the dot product intrinsic for AVX (128-bit), AVX-2 (256-bit) and AVX-512 (512-bit).

For basic AVX (128 bits), this is a full vector dot product of two vectors with 4 x 32-bit `float` numbers in each vector. One oddity is that although the result is a floating-point scalar (i.e., a single 32-bit `float`), it's still stored in a 128-bit register, and must be extracted using the "`_mm_cvtss_f32`" intrinsic.

The example code looks like:

```
float aussie_avx_vecdot_4_floats(float v1[4], float v2[4])
{
    // AVX dot product: 2 vectors of 4x32-bit floats
    __m128 r1 = _mm_loadu_ps(v1);   // Load floats
    __m128 r2 = _mm_loadu_ps(v2);
    __m128 dst = _mm_dp_ps(r1, r2, 0xf1); // Dot product
    float fret = _mm_cvtss_f32(dst);  // Extract float
    return fret;
}
```

Example: AVX-2 256-Bit Dot Product

Here is my attempt at the 256-bit version of a vector dot product of 8 `float` values using AVX-2 instructions, which seems like it should work:

```
float aussie_avx2_vecdot_8_floats_buggy(
    float v1[8], float v2[8])
{
    // AVX2 dot product: 2 vectors, 8x32-bit floats
    __m256 r1 = _mm256_loadu_ps(v1); // Load floats
    __m256 r2 = _mm256_loadu_ps(v2);
    __m256 dst = _mm256_dp_ps(r1, r2, 0xf1); // Bug!
    float fret = _mm256_cvtss_f32(dst);
    return fret;
}
```

But it doesn't! Instead of working on 8 pairs of `float` numbers, it does the vector dot product of only 4 pairs of `float` values, just like the first AVX code. The problem wasn't related to alignment to 256-bit blocks, because I added "`alignas(32)`" to the arrays passed in. It seems that the "`_mm256_dp_ps`" intrinsic doesn't actually do 256-bit dot products, but is similar to the 128-bit "`_mm_dp_ps`" intrinsic that does only four `float` numbers (128 bits). These are based on the `VDPPS` opcode in the x86 instruction for 32-bit `float` values and there is `VDPPD` for 64-bit `double` numbers. However, it seems that "`_mm256_dp_ps`" is not using the 256-bit version. Or maybe my code is just buggy!

References

1. Intel (2023), *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, August 2023, 248966-Software-Optimization-Manual-V1-048.pdf
2. Agner Fog (2023), *Optimizing subroutines in assembly language*, https://www.agner.org/optimize/optimizing_assembly.pdf
3. Félix Cloutier (2023), *x86 and amd64 instruction reference*, https://www.felixcloutier.com/x86/
4. Microsoft (2023), *x86 intrinsics list*, https://learn.microsoft.com/en-us/cpp/intrinsics/x86-intrinsics-list
5. Intel (2023), *Intel Intrinsics Guide, Version 3.6.6*, May 10th, 2023, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html
6. Intel (2023), *Intel C++ Compiler Classic Developer Guide, version 2021.10*, https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-10/overview.html, PDF: https://cdrdv2.intel.com/v1/dl/getContent/781922?fileName=cpp-compiler_developer-guide-reference_2021.10-767249-781922.pdf
7. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft

16. Contiguous Memory Blocks

Why Contiguous Memory Blocks?

A critical part of optimizing low-latency engines is to store data in a contiguous memory block so that they have a sequential address space. Processing large chunks of data in parallel is the main optimization used in both GPU and CPU SIMD acceleration. All of the vectors, matrices, and tensors need their underlying data in a block for efficiency.

Processing data that is in adjacent addresses is much faster than jumping all over the place. Vectors should obviously be stored as a simple contiguous array in memory. Less obviously, similar comments apply to the linearized memory storage of matrices and tensors.

The use of contiguous memory is an important optimization for both sequential and parallel algorithms. The reasons that memory blocks are more efficient include:

- Data locality (cache hits)
- Data block GPU uploads (model weights from memory-to-cache)
- Predictive cache pipelining (in CPU sequential accesses)

Data locality refers to using data in the same or similar address locations. This is helpful for the cache hit rate because data that is already in the cache is much faster to access that a non-cached RAM memory address.

GPU uploads from CPU RAM to the GPU's Video RAM (VRAM) is done in blocks. Obviously, we don't want to be uploading random bits of data from different parts of the RAM.

Non-GPU architectures also benefit from the use of contiguous memory. This is obviously true of CPU SIMD instructions (e.g., AVX on x86), but even in sequential execution, the CPU has its own RAM caching methods and often has other optimizations of memory accesses. Predictive cache pipelining is where the CPU attempts to predict what the next memory location will be, and load it in a pipelined speedup, before being asked. This pipelining of memory accesses is much faster than doing completely sequential address lookups.

Typically, predictive cache pipelining uses the simple heuristic that the next address is the most likely next request, which assumes that data is being processed in the order of the addresses. Hence, scanning an array in reverse is the worst possible order for these CPUs. Similarly, jumping around to different memory addresses, such as scanning the column of a matrix using a large "stride," is also inefficient.

Low-Level Memory Block Functions

Memory block operations in the standard C++ libraries are implemented using fast assembly language behind the scenes. The main functions in the standard C++ library that operate at a low level on binary bytes in a memory block are:

- `memset()`: set bytes to a value, usually used to clear bytes to zero.
- `memcpy()`: copy bytes.
- `memmove()`: copy bytes, but tolerates overlapping regions.
- `memcmp()`: compare a sequence of bytes.
- `memchr()`: search for a byte in a sequence.

These functions are lower-level than the modern C++ versions, such as `std::copy`, `std::move()`, and their "backward" versions. The above listed memory block functions are not aware of object-level semantics, and won't run any special functions on memory containing objects.

Note that unlike the standard string functions (such as `strlen`), these functions do not assume a block is null-terminated by a zero byte. Zero is simply a binary value, and these functions don't stop at a zero byte. All of these functions operate on a block of memory with a known maximum byte length.

Each compiler environment typically offers some extra non-standard byte-wise functions that are also fast. Some of the less standardized C++ intrinsics that operate on memory blocks include:

- `_memccpy()`: copy bytes up to a specified sentinel byte.
- `memicmp()` or `_memicmp`: compare bytes ignoring letter case.
- `bcopy()`: copy bytes
- `bzero()`: clear bytes to zero.
- `bcmp()`: compare bytes.
- `_byteswap_uint64()` (Microsoft intrinsic): Swap the bytes of an integer.
- `__builtin_bswap16()`: GCC function to swap the bytes in an integer. There are versions for 32-bit and 64-bit.

Fast Memory Block Operations

The slow way to do things in arrays is one element at a time. The faster way is to use the standard memory block functions on the whole array. There are a number of standard functions that operate on array data or memory blocks and they are very fast.

Initialize with `memset` byte fill. The `memset` function sets all of a memory block to a byte value. It is widely used as a fast way to initialize a block of memory to all zeros.

```
    memset(&x, 0, sizeof(x));
```

Almost all usages of memset will be for the zero byte. The only other usage I've seen is to fill memory with a dummy non-zero byte as a form of mutation testing to catch uses of uninitialized memory.

```
    memset(&x, 0x55, sizeof(x));
```

Fast array copying with `memcpy`. The fast way to copy an entire array is with `memcpy`. Rather than copy each element of an array, one at a time, in a loop, the `memcpy` standard library function can be used to copy the entire array in one statement:

```
    memcpy(destarr, srcarr, sizeof(srcarr));
```

Note that this is a bitwise copy of the array intended for simple data types. For example, it won't run copy constructors if applied to an array of objects.

The `memcpy` function does a very fast memory block copy. It is like `strcpy` in that the destination is the first parameter. `memcpy` will copy everything, even null bytes and hidden padding bytes. It keeps going even if it finds a null byte, so it is not like `strcpy`, and will always copy a fixed number of bytes. `memcpy` is a super-fast byte copy, but is unsafe, because it does not have well-defined behavior if the source and destination blocks overlap.

Safer byte copy with `memmove`: The `memmove` function is a safer version of `memcpy`, which also works correctly if the memory blocks overlap. If the source and destination blocks don't overlap, it's the same as `memcpy`, except probably slightly slower. If they do overlap, then `memmove` conceptually will copy the source to a temporary area, and then copy it to the destination block.

Copying arrays using `struct` assignment. An alternative method of copying arrays is to make a tricky misuse of `struct` assignments. This is similar to how `std::array` works, which could also be used in a similar vein, but this example totally avoids any constructor, copying or move costs (also works in C).

This method is not portable, is very unreadable and uses pointers incorrectly by converting between two different pointer types. However, it can be faster than `memcpy` because it makes use of the assignment operator rather than calling a function. On the other hand, `memcpy` is an intrinsic function that might be inlined to assembler instructions by the compiler, so this trick might be a waste of time. Benchmarking is recommended here.

To copy an array using this method it is necessary to declare a new dummy `struct` type that is the same size as the array that is to be copied. Then we use type casting to fool the compiler into thinking it is copying structures when really it is copying arrays. The method is illustrated below:

```
struct dummy_transfer { // The new struct type
    int a[MAX]; // This field gives the right size
};

int a[MAX], b[MAX]; // The array variables being copied
static_assert(sizeof(struct dummy_transfer) == sizeof(a));
*(struct dummy_transfer *)a = *(struct dummy_transfer *)b;
```

The assignment statement first type casts both "a" and "b" to be pointers to the new `struct` type, and then dereferences these pointers so that the compiler believes it is assigning between two structures. The assertion is an efficient compile-time safety net to ensure that the copying statement will work. Of course, a better way entirely is probably to put the array inside a class object, with lovely encapsulation and modularity, and then we can simply copy the objects.

`memcmp` byte comparisons. The `memcmp` function does a byte-wise comparison in a memory block. Its return value is like `strcmp`, returning 0 for equality, and a negative or positive value otherwise. Note that `memcmp` is not like `strcmp`, and will not stop when it finds a zero byte.

Memory Block Function Pitfalls

The standard memory block functions are fast, but they are not always safe. Here are some of the common pitfalls that commonly occur in everyday coding.

`memset` sizeof problem. Here's another glitch in using memset inside functions:

```
    void zero_array(int arr[10])
    {
        memset(&arr, 0, sizeof(arr));  // Bug
    }
```

The problem is not memset, but the sizeof operator on function parameters. An array parameter in a function is like a hologram and isn't really there. It's not really an array, but a pointer, and `sizeof(int[10])` is the same as `sizeof(int*)`. Hence, `sizeof(arr)` is probably only 4 or 8, rather than 40 or 80, leaving most of the array uninitialized. Personally, I recommend a memset debug wrapper function to catch this kind of problem at runtime, or maybe a tricky preprocessor macro can detect it at compile-time with a `static_assert` somehow.

`memset` portability issue. Even though it's a fast zeroing method, the use of `memset` to zero bytes has an obscure portability problem on any architecture where all-bytes-zero is not the same as all data types zero. However, on most standard platforms, all-bytes-zero is correct for all types: integer zero (ignoring endianness), floating-point zero (positive zero is all bits zero), and the null pointer.

memcpy overlapping blocks error: The only downside with `memcpy` is that it can fail with overlapping ranges for the source and destination blocks, so if you are shuffling arrays up or down one element using `memcpy`, then you have to be careful, because the results on overlapping ranges are undefined. Here's a buggy example of using `memcpy` to remove the first character of a string in place:

```
    memcpy(s, s+1, strlen(s+1)+1);  // Bug
```

The problem is that the blocks starting at "`s`" and "`s+1`" are overlapping. It is implementation-defined whether it will be correct. The fix is simply to use `memmove`, which always works correctly for overlaps:

```
    memmove(s, s+1, strlen(s+1)+1);  // Correct
```

memcmp return value. A pitfall with memcmp is that you cannot assume that it returns 1 or -1, but must compare the return result to zero (like the strcmp function).

```
    if (memcmp(&a, &b, sizeof(a)) == 1)  // Bug
    if (memcmp(&a, &b, sizeof(a)) > 0)   // Correct
```

memcmp object equality testing. Looking at the memcmp function, you might think of it as an opportunity to do a fast equality/inequality test on large objects by simply doing a byte-wise test. You would not be the first to think that.

Consider if you have a complex number class:

```
    class MyComplex {
        float real,imag;
        // .. etc
    }
```

The brute-force equality test is:

```
    bool is_equal(const MyComplex &a, const MyComplex &b)
    {
        return (a.real == b.real && a.imag == b.imag);
    }
```

Our idea to optimize this with memcmp looks like:

```
    bool is_equal(const MyComplex &a, const MyComplex &b)
    {
      return memcmp(&a,&b,sizeof(MyComplex)) == 0; // Bug!
    }
```

Unfortunately, there are multiple obscure pitfalls with this approach:

- Padding bytes
- Two types of floating-point zero
- Multiple types of floating-point NaN (not-a-number)
- Bitfields

Padding byte problems. If float is 4 bytes, but the machine has 8-byte alignment, then the "real" and "imag" data members will be stored on 8-byte alignment addresses, and there will be another 4 bytes each of dummy padding.

It doesn't even have to be on a machine with alignment issue, but can occur with a bigger object if we've mixed different size objects (e.g., `char`, `int`, and pointers). The padding bytes will be uninitialized (e.g., for local objects or if allocated with "`new`"), in which case they can contain random values. Since `memcmp` does not skip the padding bytes, its test will fail.

Now, we could possibly work around this portability issue via the use of `memset` in the constructor, or `calloc` memory allocation, to zero all of the bytes of an object including the padding bytes.

Negative zero problems. Unfortunately, the next problem is not a portability problem, but a fundamental issue with floating-point numbers. There are two zeros! There's the normal zero with all bits zero, and there's negative zero, with the sign bit set, but all other bits zero. Hence, the bitwise testing of both float numbers fails if there's ever a negative zero.

`NaN` problems. Similarly, but perhaps less seriously, the representation of `NaN` (Not-a-Number) in floating-point is also not fixed. There are multiple values of `NaN`, both positive and negative. So, `memcmp` would say the float values differ, even if both are `NaN`. I think this `NaN` issue is less serious than negative zero, because if your computations are generating `NaN`, then they're probably already failing, and an incorrect `memcmp` equality test won't matter as much.

Bitfield problems. If our structure has any bitfield data members, this `memcmp` idea fails too. Bitfields are a standard C++ feature that is defined with a suffix colon and a number of bits like:

```
    unsigned int myflag:1; // Boolean bitfield with 1-bit
```

With bitfields it's implementation-defined how this is represented numerically, and there might be undefined bits in the same byte, or extra padding bytes again.

Still want your `memcmp` speedup? I've just shown you about 15 pitfalls, but maybe you still want to live on the edge and get that speedup? You can use `memcmp` to do fast array or object comparisons if you're really, really sure that you have:

- Zero byte initializations. All allocated arrays or objects must be first zero'd by `memset` or `calloc`. You cannot rely on constructors, and it's hard to put a `memset` as the first action of the constructor due to initializer lists and base classes. You may have to intercept all of the runtime uses for the `new` and `new[]` memory allocation operators with your own wrapper that does `memset` on the block, rather than use constructor tricks.

- Padding. It's also unclear if you can actually rely on `static` or global variable initialization to carefully zero all the padding bytes in an array or object. Probably it works on most platforms, but I doubt it's fully portable. To be sure, use `memset` on the global variables during program startup.
- No bit-fields used. That's easy, at least.
- Floating-point computations should avoid negative zero and `NaN`.

Raw Subarray Memory Blocks

Passing raw subarray types to functions can be a fast alternative to some of the modern C++ contiguous containers (i.e., `std::array`, `std::vector`). However, the passing of a container object by reference with "`const&`" parameters is also very fast, so don't assume that raw arrays are always faster.

If a function accepts a raw array type, it is possible to pass it any array as an argument, or any pointer of the right type. In this way, it is possible to pass memory blocks or "sub-arrays" to a function by passing the address of a particular array element. A function to operate on a particular type of array can be written, and used to operate on various arrays.

```
void clear(int a[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        a[i] = 0;
}

void test_subarrays()
{
    int a[100];
    clear(a, 10); // clear first ten, 0..9
    clear(a + 50, 10); // clear 50..59
    clear(&a[50], 10); // clear 50..59 (equivalent)
}
```

Multidimensional subarrays. It is also legal to pass multi-dimensional arrays to functions. However, the sizes of all but the first dimension must be specified in the function receiving the array.

For example, to pass a two-dimensional array to a function, the function header would look like:

```
    void fn(int a[][SIZE2]);
```

The reason for this restriction is that the compiler cannot determine the address for an arbitrary array element if it does not know the sizes of all but one of the dimensions.

Because the sizes of most of the array dimensions must be specified in the function declaration it is very difficult to write a function to act on sub-arrays of multi-dimensional arrays. For example, this idea would be useful to define library functions to operate on matrices with different dimensions. Ideally, we would like one function to calculate the determinant of a matrix for any dimension (i.e., an `n`-by-`n` matrix where `n` varies). Consider how we want a determinant function to look:

```
    double determinant(double matrix[][], int n); // Bug
```

Ideally, the dimensions of the matrix are not specified at compile-time, but are specified at run-time by the `n` argument. This is not possible as a simple C++ declaration because the second dimension (i.e., `n`) needs to be specified in the definition of the two-dimensional array type. The best solution is to use dynamic multi-dimensional arrays.

Dynamic Memory Management Pitfalls

Memory management is really not the strong suit of C++. If your program is crashing or behaving badly, it's highly likely to be some kind of memory problem. There are so many pitfalls in C++ dynamic memory management, and even in static or global (non-dynamic) memory, that it's hard to list them all.

C++ programs have access to a large block of free memory, called the heap. The actual size of the available memory depends on the system. This memory is available to a C++ program which can allocate itself chunks of memory from this heap. This is useful when a C program does not know beforehand how much data is being stored, and hence, how much memory is required. Instead of allocating a large array for the worst case, the program can allocate itself blocks of memory as required.

Blocks of dynamic memory can be allocated in two main ways:

- The C++ style "`new`" or "`new[]`" operators
- The older style `malloc()` and `calloc()` functions (inherited from C)

Other ways to allocate dynamic memory include:

- `strdup()`: make an allocated copy of a string.
- `realloc()`: a companion to `malloc/calloc` that is rarely used.

Once the memory is no longer needed it is "freed" back to the heap. Again, there are two main ways:

- The C++ style "`delete`" and "`delete[]`" operators
- The older style "`free`" function

Some of the main memory problems in a C++ program can include:

Uninitialized new memory. The `new` operator does not initialize the new chunk of allocated memory. Accidentally using it is a common bug.

Uninitialized malloc memory. The `malloc` function also does not initialize its allocated memory. Again, use of a memory block that is allocated by `malloc` but hasn't been properly cleared is a common bug. One of the mitigations is to use `calloc` instead, because `calloc` does zero the bytes of every block it allocates.

Mismatched new/delete with malloc/free. Memory allocated with `new` should be deallocated by `delete`, but `malloc`'d memory should be `free`'d. Never the twain shall meet, or else kaboom.

Mixing new/new[] and delete/delete[]. Memory allocated by `new` should be released by `delete`, but memory allocated by the array version "`new[]`" should be freed by the `delete[]` array version. Again, they're not supposed to mix.

free(nullptr) is harmless. If it's so harmless, why is it a pitfall? Sure, `free(nullptr)` is officially defined by the standard to do nothing. But if your coding is doing this, it sure walks and talks and quacks like a buggy duck.

`strdup(nullptr)` is not harmless. This is probably a crash, but even on systems where it's not, it's clearly a bug in your code if you're trying to duplicate a null pointer.

Pitfalls for Non-Dynamic Memory Blocks

There's so many pitfalls in management dynamic memory, with either new/delete or malloc/free, that surely we've run out? No, don't worry, it's comforting to know that there are still a bunch more insidious problems in other types of non-allocated memory.

Here's a list of some more fatal memory stomps that aren't about allocated blocks on the heap:

- Buffer overrun of a global, local, `static`, or stack buffer variable.
- Returning the address of a local variable on the stack (i.e., non-`static` variable).
- Trying to write to addresses of string literals (often a crash if they're non-writable, but maybe worse behavior if it can be modified).
- Modifying `arr[10]` in an array of size 10 (raw arrays or `std::array`).
- Uninitialized local variables or local buffers on the stack (non-`static`).
- Using an uninitialized local pointer variable to access some random address in Timbuktu.
- Null pointer dereferences. Oh, well, at least you initialized it.
- Returning the address of a "`static`" local variable (aliasing problems).
- Using a negative array index.
- Modifying a string literal (they're in read-only memory on Linux).

The standard C++ library functions can also have problems:

- `strcpy()` on overlapping string arguments: `strcpy(s, s+1);`
- `strncpy()` can leave strings without a null byte terminator.
- `memcpy()` on overlapping memory blocks (use `memmove` instead).
- Trying to `free()` or `delete` a global, `static`, stack or instruction address will crash.
- Double `fclose()` on file pointers from `fopen`.
- Ignoring the return value of `erase()` in an iterator loop.

17. Memory Pools

What are Memory Pools?

Memory pools are a C++ optimization where you take control of the memory allocation used for a class of objects. The basic idea is to store all objects of the same type in a big array, next to each other, rather than being spread out over the heap wherever the `new` operator decides to put them.

Memory pools are a general optimization that can be used in C++ with the `new` operator, and also in C programming with `malloc`.

Some of the related data structures include:

- Bucket array
- Hive

A bucket array is like a memory pool, in that it's a big memory block, and you put your objects in there. However, a bucket array usually handles erasing an object by simply marking it as invalid using a Boolean flag. The memory for an erased object is not usually re-used when you insert a new object.

A hive is a generalization of a bucket array, whereby a hive can dynamically expand and contract the number of buckets. Notably, there's a `std::hive` class to use in C++26, which would make a good basis for an advanced type of memory pool.

However, we're going to examine some of the simpler types of memory pools first.

Why Memory Pools?

Other than being a fun and gritty project in low-level C++ coding, the goal is speed, and this is achieved in various ways:

- Preallocation — no need to allocate memory on a low-latency hotpath.
- Fewer allocation calls — one big chunk rather than lots of small ones.
- Fewer deallocation calls — reusing memory addresses within the pool.
- No memory fragmentation — we don't mix small and large memory allocations.
- Less memory overhead — hidden heap memory "control blocks" are not needed.
- Cache locality — all objects are stored contiguously.

In fact, you can even get the number of memory allocations for your class down to zero, if you really want to, by using a global memory pool object. Even the memory pool is not on the heap! But this only works for a fixed-size memory pool, and thus, only if you're really sure you won't need too many objects.

Memory fragmentation is also a slowdown that can be avoided or reduced with memory pools. The problems with fragmentation arise in two ways:

- Frequent allocations and de-allocations, and
- Different-sized memory blocks.

A memory pool is helpful in both respects. The memory pool avoids lots of allocations by using one big block, and avoids deallocations by re-using the locations inside the block. And because the memory block stores lots of blocks of the same size, we aren't mixing up different size allocations.

Disadvantages of Memory Pools

Firstly, this whole idea of memory pools is only about reducing allocated memory on the heap. This optimization is not relevant for objects stored on the stack (i.e., local variables), or static objects, such as global scope objects or `static` data members.

Memory pools are not the only option for optimization memory allocation. In fact, the use of an open-source drop-in replacement for the standard C++ memory allocators is another significant option:

- jemalloc — the original FreeBSD allocator, now a Facebook favorite.
- tcmalloc — from Google, with an Apache 2.0 license.

The other disadvantages of memory pools include:

- Fixed maximum number of objects (in the basic versions).
- Only works for single-sized objects (e.g., one class).
- Need one memory pool object for each type of object (via templating).
- Not useful for optimizing variable-sized objects (e.g., strings).
- Allocating too much memory in one massive chunk.

However, we can work around a lot of these disadvantages by using a templated class for our memory pool. The optimization of memory pools is a general algorithm that works for all types of objects.

Memory Control Block Overhead

Whenever you allocate memory on the heap, using the new operator or the old-style malloc function, it returns you the address of the block. But that's not actually the start of the *real* memory block.

There's actually an extra memory control block stored before that address. It contains meta-information about the memory block, which is used by the C++ standard library to keep track of things. For example, the size of the memory block is stored in that control block.

Whenever you deallocate a memory block by sending the address to `delete` or `free`, the standard library knows to look backwards a few bytes. Hence, it can find the size of the memory block, which helps it to deallocate the full block of memory. You don't need to worry about it, because the standard library takes care of it.

Hence, if you create a memory pool from one big chunk to contain 100 objects, rather than 100 separate calls to the `new` operator, there are 99 fewer memory control blocks. This is why memory pools reduce the memory overhead from your objects.

Fixed-Size Memory Pool Algorithms

For simplicity, we're going to limit our first memory pools to just one huge block in memory. This means that we can choose the overall capacity of the memory pool, but we can't increase it later by adding a second big block. This makes our memory pool more like a vector or array, rather than a dynamic bucket array or hive.

Even with these restrictions, there are still quite a few choices to make about designing our memory pool algorithm. Some of the alternatives include:

- Boolean flag — storing an "active" flag in each object.
- Index array — maintaining a list of indices of free blocks as a "free list" (instead of a per-object flag).
- Pointer array — tracking the free list via pointers.
- Permutation-based free list approach.

In the first case, we only have one array, and each block contains the "active" flag along with the stored user objects. In the other cases, we maintain two arrays, for one of the user's objects, and another as the free list (with either indices, pointers, or permutations).

Disadvantages of Boolean Flag Method

The first point to remember is that this memory pool is a significant optimization. It achieves all the advantages of a memory pool as outlined above: preallocation, fewer allocations and deallocations, less memory fragmentation, and so on. Hence, it's a good start, and a worthy improvement to our classes.

We could stop now, and go home with a smile on our face.

However, it's not optimal. There are even better ways to code up a memory pool. The suboptimal features of this version of a memory pool include:

- Mixing hot and cold data
- Alignment issues for some types
- Extra padding bytes needed
- Slow insertions

One problem with the above approach is that it mixes "hot" and "cold" data. Your objects are probably hot areas of processing that are doing whatever you need. The Boolean flags are only used by the memory pool when inserting and deleting objects, and are thus cold data for the main processing algorithms. It would be better for cache locality if the cold data was separated from our hot objects.

Memory size is also not optimal. By adding a single Boolean variable to each object, it's not just 1 byte extra, because the compiler probably may have to add a number of padding bytes to meet the alignment requirements (depending on what's inside your objects). This will increase the memory size, and worsen cache locality when processing multiple objects.

However, the main problem with the Boolean flag approach is that it's slow. In fact, it has worst case *O(n)* performance for an insertion, because it might have to scan the entire array to find a free block. This worst case won't happen initially, but the performance can degrade as the memory pool fills up, and we do lots of insertions and deletions.

We can do better!

Boolean Flag Array Method

One way that we can address some of these issues is by separating all of the Boolean "`active`" flags into a different array. Rather than storing a flag in each object, we just store the user's object in the main block, and have a second block that contains the Boolean flags.

The advantages are that it fixes the hot-cold data problem, addresses alignment concerns, and the compiler won't need to add extra padding to the array of user objects. The array of Boolean flags should be one byte per object, but stored in a different array.

Firstly, we move the "active" flag out of the structures:

```
    struct Node {
        unsigned char data[sizeof(T)];  // Raw storage
    };
```

And put it into a separate array:

```
    bool activearr_[N];
```

The handful of places that used the "`active`" flag need to be changed to the "`activearr_`" array member.

We can also fix the alignment issues using the `alignas` and `alignof` specifiers:

```
    alignas(alignof(T)) std::array<Node, N> arr_;
```

Bit packing. This active flag array method can be further improved by using bit packing. We only need one bit flag per object, rather than one byte each. Hence, we can pack them all into an array of 64-bit `unsigned long`, and can check for a free block using one integer comparison, testing 64 memory blocks at a time.

In practice, this version is pretty fast. Even so, it is technically still an *O(n)* worst case algorithm for insertion or deletion with large numbers of objects. And there are a few ways to fix that.

Index Array Memory Pool

The faster solution is to maintain an array of integer indices for the free locations. The advantages of this index array approach over the earlier "active" flag method include:

- Insertion and deletion always have *O(1)* complexity.
- Separates hot data from cold data.
- No extra padding bytes needed.

Here's the basic definition of the class:

```
    template<typename T, int N>
    class IndexMemoryPool {
        struct Node {
            unsigned char data[sizeof(T)]; // Raw storage
        };
    private:
        alignas(alignof(T)) std::array<Node, N> arr_;
        int freelist_[N];  // free indexes (stack-like)
        int ct_;
        int ctfree_;
    // ...
    };
```

Some of the basic primitives are simple:

```
bool empty() { return ct_ == 0; }
bool full() { return ct_ == N; }
int capacity() { return N; }
int count() { return ct_; }
int count_free() { return ctfree_; }
```

The index array is a "free list" that tells us where to find a free memory block. After a lot of insertions and deletions, if functions a lot like a stack of free locations. At the start, it's a fixed-size stack that's full with the index of every element available.

```
IndexMemoryPool() : arr_(), ct_(0), ctfree_(N) {
    for (int i = 0; i < N; i++) {
        freelist_[i] = i;  // Store all indexes
    }
}
```

When we allocate a new block, that's a "pop" of the stack, because we're removing from the free list:

```
int pop_free_index()
{
    assert(ctfree_ > 0);
    int index = freelist_[ctfree_ - 1];
    assert(index != -1);
    freelist_[ctfree_ - 1] = -1; // Clear it
    ctfree_--;
    return index;
}
```

The allocation of a block is mostly a call to this "pop" of the free list:

```
T* alloc() {
    if (full()) return nullptr; // fail!
    int index = pop_free_index();
    assert(index != -1);
    ct_++; // Incremental count
    return reinterpret_cast<T*>(&arr_[index]);
}
```

And the reverse is true when the caller releases a memory block. This is a push operation of a newly free index onto the stack.

```
    void push_free_index(int index)
    {
        assert(ctfree_ < N);
        freelist_[ctfree_] = index;
        ctfree_++;
    }
```

And here's the version for release the memory:

```
    void erase(T* addr) {
        assert(ct_ >= 0);
        Node* nptr = reinterpret_cast<Node*>(addr);
        if (nptr >= reinterpret_cast<Node*>(&arr_[0])
          && nptr <= reinterpret_cast<Node*>(&arr_[N - 1])
             ) {
            // Valid pointer...
            int offset = nptr - &arr_[0];
            push_free_index(offset);
            ct_--;  // Incremental count
        }
        else { // Invalid pointer...
            assert(false);
        }

    }
```

In summary, note that the push and pop of the free list stack is very efficient with O(1) complexity. This index array version has constant-time efficiency.

Boolean Flag Memory Pool

This is the simplest approach, but not the fastest. Let's examine it to get some of the basic ideas.

Some of the interesting features of this code include:

- Boolean flag — stored as a data member in every memory pool record.
- Pointer arithmetic — used in computing the offset when erasing an object.
- Incremental count — increment on allocation, decrement on release.
- Compile-time pool size — this uses `std::array` rather than `std::vector`.

Here's the basic layout of the memory pool class.

```
template<typename T, int N>
class MemoryPool {
    struct Node {
        T data;
        bool active;
    };
private:
    std::array<Node, N> arr_;
    int nextfree_;
    int ct_;
    // ...
};
```

The constructor has to set all the "`active`" flags (although using `memset` would be faster than a loop):

```
MemoryPool() : arr_(), nextfree_(0), ct_(0) {
    for (int i = 0; i < N; i++) arr_[i].active = false;
}
```

The code maintains the index of the "next free" object. Initially, it's increasing as the first blocks get used, but later it's necessary to scan linearly.

```
int find_next_free(int offset) {
    if (offset == -1) offset = 0;
    int i = offset;
    do {
        if (!arr_[i].active) return i; // Found
        i = (i + 1) % N;
    } while (i != offset);
    return -1;  // It's full!
}
```

Here's the code for the allocation of a memory pool block:

```
T* alloc() {
    if (full()) return nullptr; // fail!
    assert(nextfree_ != -1);
    int oldindex = nextfree_;
    arr_[oldindex].active = true; // Not free
    nextfree_ = find_next_free(nextfree_);
    ct_++; // Incremental count
    return reinterpret_cast<T*>(&arr_[oldindex]);
}
```

And here's the code whereby a block is released by the caller. Note that the index computation requires pointers converted to the correct type. This code has some safety checks that are quite expensive, and might later be removed for production usage.

```
    void erase(T* addr) {
        assert(ct_ >= 0);
        Node* nptr = reinterpret_cast<Node*>(addr);
        if (nptr >= reinterpret_cast<Node*>(&arr_[0])
          && nptr <= reinterpret_cast<Node*>(&arr_[N - 1])
            ) {
            // Valid pointer...
            int offset = nptr - &arr_[0]; // Pointer arith
            assert(nptr->active);
            nptr->active = false;  // Free now
            ct_--;  // Incremental count
            if (nextfree_ == -1) { // Was full?
                nextfree_ = offset;
            }
        }
        else { // Invalid pointer...
            assert(false);
        }
    }
```

Constructor inefficiency. This implementation has a high-level slug if the memory pool is instantiated for use with a non-trivial class type. The definition for `std::array` will cause the constructors for every single object to run needlessly on the empty storage bytes, when the memory pool is first created or defined. The solution here is simply to use bytes instead of the class type for the storage declaration:

```
    struct Node {
        unsigned char data [sizeof(T)]; // Raw storage
        bool active;
    };
```

But we also need to be careful of memory alignment in this situation. The template could be instantiated on any type, some of which will need aligned addresses. Character addresses won't get automatically aligned, so we have to use `alignas` specifier. However, it's hard to fix in this implementation, because I cannot use `alignas(alignof(T))`. The extra "`active`" flag in the structure is messing everything up. But that's only one disadvantage of this method.

Memory Pools Versus Containers

Why do you need a memory pool? Why not just use the standard C++ containers for your objects? Isn't a memory pool about the same as `std::vector`?

Yes and no.

Yes, a memory pool for your objects is very similar to managing them all in a standard vector. After all, the memory pool code can use a `std::vector` object inside it as the big pool. So, yes, you can manage your objects in a standard vector if you:

- Use a single `reserve` or `resize` call to allow the vector to allocate memory in one call.
- Keep track of objects going in and out of the vector.

In other words, it's almost the same thing as writing a memory pool, except it's mixed in the middle of your application's main logic.

Hence, no, it's not quite the same thing. There are two types of containers:

- Contiguous storage containers — it's very similar.
- Maps, sets, hash tables — memory management performance gains.

We'll examine vectors and arrays in a minute, but first let's look at the other containers. There are two aspects to use normal memory allocation and storing your objects in these advanced containers:

- Allocating memory for your objects — you've improved nothing (it's one allocation call per object).
- Extra container allocations — the container also needs memory allocation and a memory pool doesn't help with that.

But for the containers based on contiguous memory, the issue is less clear cut. The standard containers based on contiguous storage include:

- `std::vector`
- `std::array`
- `std::inplace_vector` (C++26)

When you compare a memory pool to using a standard vector of your objects, there is less gain to performance. However, creating a memory pool as a standalone class has several practical advantages:

- Separate memory management optimizations from business logic.
- Ensures only a single (huge) memory allocation occurs (or only a few if it's dynamic).
- Callers of the interface or API don't need to know about the memory management aspects.

Creating a memory pool as a separate idiom is good for encapsulating the performance optimization aspects of memory management. It encourages modularity by isolating high-level business logic from low-level resource management.

Advanced Memory Pools

Higher-level improvements to the public memory pool interface are also possible. Most of the discussion here has been about a memory pool for one type of class, with a focus on reducing the number of distinct blocks requested on the heap. More advanced memory allocators are well-known, and they offer a variety of generalized performance optimizations and convenience features:

- Thread safety (e.g., a single mutex or a lock-free version).
- Intercepting the class-specific `new` and `delete` operators.
- Passing arguments to object constructors via parameter packs and `std::forward()`
- Placement `new` operator — does not really allocate memory!
- Custom allocators — memory pools via allocator functor objects.

Additional memory management features that could be added to a memory pool include:

- Dynamic expansion with multiple chunks rather than a fixed-size pool.
- Multiple object types supported in the memory pool.
- Dynamic size of objects allowed by allocating multiple large "pools" or memory chunks.
- Downsizing the memory pool if fewer objects are required.

Even more general than memory pools is the concept of "custom allocators." The idea of custom allocators is not just to enhance memory handling of a few classes, but to take over the whole memory allocation shemozzle from the standard library.

Extensions

1. Build your own simple memory pool templated class.
2. Add a memory pool to your object class by overloading a set of class-specific `new` and `delete` operators, sending these allocation requests to the memory pool instead.
3. Code up multiple types of memory pools and measure their performance.
4. Generalize your memory pool class to dynamically manage multiple big chunks of memory, rather than just one.
5. Implement an advanced dynamic memory pool using the new container class `std::hive` (C++26) as the underlying data structure, rather than a vector or array.

References

1. Sourav Ghosh, July 2023, *Building Low Latency Applications with C++*, Packt Publishing, https://www.amazon.com/dp/1837639353
2. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
3. Devansh, Feb 27, 2024, *A quick introduction to Memory Pools: Optimizing Memory Management in Software Engineering*, https://machine-learning-made-simple.medium.com/a-quick-introduction-to-memory-pools-cc3198d004db
4. happyer, Apr 23, 2024, *Memory Pool Techniques in C++*, https://medium.com/@threehappyer/memory-pool-techniques-in-c-79e01f6d2b19
5. Bernardo Palos, 2025, *Memory Pools in C++_ What They Are and How to Implement Them*, https://palospublishing.com/memory-pools-in-c_-what-they-are-and-how-to-implement-them/
6. Stack Overflow, 2019, *C++11 memory pool design pattern?* https://stackoverflow.com/questions/16378306/c11-memory-pool-design-pattern
7. Boost, 2011, *Boost.Pool*, https://www.boost.org/doc/libs/1_53_0/libs/pool/doc/html/index.html
8. Roger Ferrer Ibáñez, Nov 19, 2017, *A very simple memory pool in C++11*, https://thinkingeek.com/2017/11/19/simple-memory-pool/

9. Contributors, May 2025 (accessed), *jemalloc memory allocator*, https://jemalloc.net/, https://github.com/jemalloc/jemalloc (originally from FreeBSD, then updated by the Mozilla Foundation and Facebook, Inc.)
10. Google, May 2025 (accessed), *TCMalloc*, https://github.com/google/tcmalloc (Apache 2.0 License)

18. Data Compression

What is Data Compression?

Data compression is a common efficiency requirement in low-level programming. This is the general class of algorithms that aim to make data smaller.

Generally, there are two main phases:

- Compression — reducing data to a smaller size.
- Decompression — inflating the data to its original values.

There are two main classes of data compression algorithms:

- Lossless — the original data is fully restored.
- Lossy — uncompressed data differs and is partially "lost" in some sense.

Obviously, lossless algorithms are preferable, but some types of lossy algorithms can achieve much better compression ratios. Hence, there is a trade-off between data size and accuracy of the uncompressed data.

The best example of lossy compression in the modern era is "quantization" in AI, which is a type of "model compression" technique. The billions of 32-bit numbers that are an LLM's "weights" can be shrunk down to fewer bits, often as few as 4 bits each, while still retaining the general capabilities and knowledge of the original model. In this way, LLMs are much smaller to transport over the network, to store on disk, and also faster to run on CPUs or GPUs (less memory usage).

Note that this is not a typical data compression algorithm, because quantized models are not "uncompressed" back to 32-bit numbers, but are used in their low-bit formats.

Related Data Algorithms

Data compression is somewhat related to other common programming tasks, but it is not the same as:

- Encoding — converting data to a simpler representation.
- Encryption — hiding the data with varying levels of difficulty.
- Hashing — mapping data to a hash value.
- Data compaction — using smaller data records.
- Data structures — organizing data for fast search or other operations.

Encoding or encryption of data are different requirements to compression. In fact, both encoding and encryption can increase the data load and slow things down., but have other advantages.

Encoding neither makes the data smaller nor hides it from prying eyes. The goal of encoding is to make it easier to use data, whereas data compression has the central aim of reducing size. There are some common encoding algorithms:

- Base64 (or UUencode) — transmit binary data over text-only streams using the subset of printable characters.
- UTF-8 — internationalization encoding for European letters and Double-Byte Character Sets (DBCS).
- Rot-13 — simple semi-encryption method.

Note that most of these encoding algorithms will actually increase the size rather than decrease. This is true of both UTF-8 or UUencode.

Encryption is a different task, and it's not related to data compression. The purpose is to maintain privacy and security of the data payload, and the way to do that often increases its size. Some of the well-known encryption algorithms for "hash" creation include:

- SHA (Secure Hash Algorithm)
- MD5 (Message Digest 5)
- AES (Advanced Encryption Standard)

These algorithms have been used for things like password encryption in the past. More recently, these encryption algorithms became known as cryptographic algorithms, because they're used in Bitcoin mining and other "crypto" creations.

Low Latency Data Compression

When there's too much data, you want to compress it. In low latency programming, some example situations where data compression algorithms can be useful include:

- Disk storage — e.g., compressing large volumes of historical data in files or databases.
- Network data transmission — e.g., sending data off to a different site for ML model processing.

The goals of data compression include:

1. Reduced space storage on disk or in memory, and/or

2. Faster network transmission

Note that the goal is not to make processing faster. In fact, to process the data later, you would have to uncompress it first. Both compression and uncompression are extra processing costs, so there is a trade-off when considering the benefits of data compression algorithms.

Trading algorithms involve the processing of a lot of data, often aggregated from multiple financial exchange locations. Full market data with deep order book details grows quickly in size, and compressing this data speeds up historical storage, algorithmic analysis for trading storage, and backtesting with this data.

Data Compression Algorithms

There is a long history of data compression algorithms from the 1970s and earlier. The lossless algorithms from that era include:

- Run-Length Encoding (RLE)
- Huffman coding
- Lempel-Ziv algorithm (LZ)

Run-length encoding is well-known and unsophisticated. It can do well if the data has long "runs" of the same value, which is relatively common in images, but not in text.

Huffman coding is a more complicated data compression algorithm. It involves building a separate data structure that represents the bitwise encodings for different patterns. Huffman coding is quite successful as an algorithm, but its main downside is the need to convey the dictionary as a separate data structure before decoding can begin on the main compressed data.

The Lempel-Ziv algorithm was a clever idea of overcoming Huffman coding's main disadvantage by building a dictionary incrementally inside the compressed data, thereby alleviating the need to pre-send a separate data structure. Several variants for LZ data compression have been used:

- LZ77 — the original 1977 version.
- LZ78 — an improved version in 1978 by the original researchers.
- Lempel-Ziv-Welch (LZW) — a further improved and popular algorithm.

LZ77 is the original LZ algorithm from 1977, using a "sliding window" over the text. Each entry in the compressed file includes a character count and an "offset" back to a prior occurrence of the same text string. In this way, the explicit dictionary required for Huffman encoding is no longer needed, because the strings in the prior text are used as if they were a dictionary.

The LZ78 variant is a modification of the LZ77 algorithm, which uses an explicit dictionary data structure. However, this dictionary is used internally by both the encoder and decoder, but does not need to be sent from the encoder to the decoder. Instead, the decoder can rebuild the dictionary in an incremental fashion as it decompresses the text.

The LZW algorithm became the most widely used data compression algorithm. It was used in numerous Unix tools and notably in the GIF image format.

The improved LZW algorithm modified the LZ78 algorithm to introduce faster handling of some failed-match sequences. Initially, it has a "predefined" dictionary for all possible symbols, which is used if a text string has no match in the normal LZ78 dictionary. For 8-bit data, there are 256 of these predefined symbols, being all the single-character strings. Both the encoder and decoder start with this initial dictionary.

The main downside of the LZW algorithm for modern computing is that it's an inherently sequential algorithm, since both the decoder and encoder must incrementally build the dictionary from the data sequence. This means that it's difficult to fully parallelize LZW encoding or decoding on multiple CPU threads or using GPUs.

Parallel Data Compression Algorithms

Many of the traditional data compression algorithms are difficult to parallelize. The problems that need to be overcome include:

- Inherently incremental algorithms (sequential nature)
- Variable-length bit strings

Incremental algorithms. Compression that must start at the beginning of the data stream is problematic for parallelization. An example of this is the "sliding window" approach in LZ77. The dictionary approach in LZ78 is also difficult to parallelize, because it requires a phase to construct the dictionary, before all sub-texts can be compressed.

Bit position offsets. It might seem that the Huffman algorithm could have parallel decompression once the dictionary has been sent to the decoder. However, another problem arises: given a chunk of data to decode, how does the decoder know which bit in the first byte to start with. It could be any of the bit positions. The variable-length bit strings used by compression mean that you cannot determine this without processing from the beginning of the input string.

Naive parallelism. You can parallelize all of these algorithms by doing a "restart" in every chunk, which is sometimes called a "segmented compression" algorithm. The LZW algorithm even has a "clear code" for exactly this purpose. But that's not a very good method! I don't see much difference between that idea and just splitting the data into a bunch of files, and then compressing each file. Furthermore, the code that's working on each chunk is running an inherently sequential algorithm, rather than one that's vectorized over SIMD or GPU kernels. Somehow that doesn't seem optimal to me!

Parallelizing LZW compression. The LZW algorithm also suffers from these problems that limit parallelism. The encoder and decoder both build the dictionary in an incremental manner from the start, limiting parallelism. And most LZW algorithms used variable-length bit representations, which also depend on prior input data.

There are some parallelizations possible for an LZW data compression algorithm, but these are not inside the main data compression loop. Rather, the features that wrap around the main LZW logic can be separated.

Parallel optimizations to the LZW algorithm include:

- Run the bit packing/unpacking in a separate thread from the encoder/decoder.
- Run the I/O in parallel to the encoding/decoding (i.e., file input, file output) by reading or writing chunks to files in separate threads.

Even this bit packing parallelization runs into problems and requires modification. There's no difficulty if the LZW algorithm uses fixed-width 12-bit codes. However, variable-length LZW variants can change the bit positions, leading to difficulty with parallel packing and unpacking of chunks.

Chunk headers. One possible solution to the variable bit position issue is to extend the algorithm so that every chunk has a "header" of data. A simple version would have an extra two bytes that encode: (a) what bit position to start processing for the rest of the data, and (b) how many bits are being used for packing.

However, this simple two-byte header still has a problem: the number of bits for encoding numbers in LZW can increase at any point in the middle of a chunk. A more complex header is needed, such as the two bytes indicating the bit position and number of bits, along with another two bytes indicating how far along in the chunk until the bit sizes increase.

Newer parallel algorithms. This chapter has only scratched the surface of parallelizing data compression algorithms. It is perhaps unsurprising that older algorithms designed for sequential processing are difficult to parallelize. However, there is an immense body of research in this area, and some newer parallel data compression algorithms:

- Parallel gzip
- Zstandard
- Parallel bzip2
- Parallel LZ4
- Parallel LZMA
- BTW parallel Huffman

There are many options and many research papers to read.

References

1. Michael Dipperstein, 2019, *lzw: An ANSI C implementation of the LZW compression algorithm*, https://github.com/MichaelDipperstein/lzw
2. Mark R. Nelson, 1999, *LZW data compression/expansion demonstration program*, https://people.cs.pitt.edu/~kirk/cs1501/assignments/lzw/lzw.cxx
3. Geeks for Geeks, 21 May, 2024, *LZW (Lempel–Ziv–Welch) Compression technique*, https://www.geeksforgeeks.org/computer-networks/lzw-lempel-ziv-welch-compression-technique/
4. Wikipedia, June 2025 (accessed), *LZ77 and LZ78*, https://en.wikipedia.org/wiki/LZ77_and_LZ78
5. Facebook, 2025, *Zstandard - Fast real-time compression algorithm*, https://github.com/facebook/zstd
6. LZ4 contributors, 2025, *LZ4*, http://www.lz4.org/, https://github.com/lz4/lz4
7. Dingwen Tao, 2019, *GPU-Accelerated Lossless Compression Survey*, https://github.com/dingwentao/GPU-lossless-compression
8. Flanglet, 2025, *Kanzi: Fast lossless data compression in C++*, https://github.com/flanglet/kanzi-cpp
9. Martin Vorbrodt, 2019, *Extremely Fast Compression Algorithm*, Vorbrodt's C++ Blog, https://vorbrodt.blog/2019/03/22/extremely-fast-compression-algorithm/

Part IV: Low Latency Data Structures

19. Modern C++ Containers

Standard C++ Containers

Contiguous data containers. The general-purpose containers with contiguous data are called "sequence containers" and include several that are well-known and often used:

- `std::string` — dynamic character arrays.
- `std::vector` — dynamic everything arrays.
- `std::array` — static fixed-size arrays.
- `std::bitset` — fast bit vectors.

Associative containers and sets. The associative key-value data structures are more commonly called a "map," "dictionary," or "symbol table" design pattern. Note that the "set membership" idiom is usually very similar to the associative containers, because the search is the same, but the sets don't have a payload at the end.

The main types of modern C++ containers for searching include the choice between two main types of underlying data structures:

- Red-black balanced binary trees — logarithmic complexity for search, insert and delete.
- Hash tables (with chaining) — constant-time average complexity (fast!), but linear worst-case (slow!).

The containers include these red-black tree versions:

- `std::map` — key-value lookup (dictionary idiom).
- `std::set` — key-only set membership lookup.

And these are the hash tables (my favorite data structure!):

- `std::unordered_map` — dictionary hash table for key-value pairs.
- `std::unordered_set` — hash table for set membership.

There are also variants that allow duplicates, which means multiple copies of the same key stored separately in the container. Examples include:

- `std::multiset`
- `std::multimap`
- `std::unordered_multiset`
- `std::unordered_multimap`

Linked list containers. Some of the containers to manage data in dynamically-allocated linked lists include:

- `std::list` — double-linked list
- `std::forward_list` — singly-linked list

Note that the hash table containers (e.g., `std::unordered_map`) also belong on this list because they use "chaining" for collision resolution. This approach effectively hangs linked lists off every bucket of the hash table.

Sorted "flat" containers. There are some newer containers in C++23 that are "flat" in the sense that they maintain data in sorted order. These classes include:

- `std::flat_set`
- `std::flat_map`
- `std::flat_multiset`
- `std::flat_multimap`

Special semantics containers. Some of the general-purpose containers with different semantics to searching include:

- `std::stack` — dynamic FIFO structure.
- `std::queue` — queue data structure (single-ended).
- `std::dequeue` — double-ended queue.
- `std::priority_queue` — implements the "heap" data structure.

View containers. The various types of "view" containers include:

- `std::string_view`
- `std::span`
- `std::mdspan` — multidimensional view class.

Bit-level data structures. Modern C++ supports both class libraries and utility functions for a variety of low-level bit manipulation tasks. Some examples include:

- `std::bitset`
- Bit manipulation utilities in `<bit>`

Small utility data structures. Some of the more generic types of "mini-data structures" include:

- `std::pair`
- `std::tuple`
- Ranges
- `std::optional`
- Permutations

Multithreading data structures. Parallel coding with synchronization and locking is supported in modern C++ with libraries such as:

- `std::thread`
- `std::mutex`
- `std::lock`
- `std::condition_variable`
- `std::atomic`
- `std::latch`
- `std::barrier`

And that's not the full list of primitives available in the concurrency library. Many of these multithreading capabilities have been available since C++11.

Upcoming C++26 containers. Some of the upcoming containers include:

- `std::hive` (C++26)
- `std::inplace_vector` (C++26)

What's missing? I feel ungrateful to even be writing this list, given the amazing amount of work that's gone into coding up all the above data structures in the standard C++ library.

Nevertheless, some of my favorites aren't on the list yet! Data structures that are missing from the standard C++ containers library include:

- Sorted array — indirectly supported only (e.g., `std::sort`).
- Tries — 26-way tree for storing text keys based on letters.
- B-tree — multi-way tree data structure good for disk storage.
- Graphs — depth-first search, breadth-first search, topological sort.
- Tri-state Boolean — indirectly supported via `std::optional`.

General Container Optimizations

Containers have a lot of commonalities in their performance patterns. Some general comments apply to multiple types of container classes, and making them run faster. Consider the following when implementing the usage patterns of your containers:

- Choose an initial size — avoid container auto-resizing slowdowns.
- Minimize insertions and deletions — yeah, right, those actions are why we use containers!
- Auto-resizing of containers — watch out for silent slugs!
- Remove all elements with `clear()` rather than a loop.
- Container destruction can be slow.

Choose your containers wisely:

- Prefer hash tables when you need fast searching (e.g., the standard container class `std::unordered_map`).
- Don't use a key-value associative container if you only need a set.
- Consider whether you need sorted or unsorted scanning of all elements.
- Prefer the various contiguous-memory standard C++ container classes such as `std::array` and `std::vector` for good cache locality.

Optimizations in relation to the types of data to use in containers:

- Choose scalar types — objects have more risks of slowdowns from calls to constructors, destructors, move operators, etc.
- Prefer integer keys — faster than `std::string` or `char*` in key-value pairs.
- Reduce the sizes of keys and values — minimizes overall container memory size and improves cache locality.

Choosing Containers

This should be a short section of the book because it's very easy: use the two containers `std::vector` or `std::unordered_map`, and forget the rest. Oh, maybe `std::queue` and `std::stack` if you must.

I'm only half joking, because there are two things that you often want to do quickly:

- Scanning — `std::vector` is an array with contiguous data (cache locality).
- Searching — `std::unordered_map` is a hash table with *O(1)* average complexity for search, insert, and delete.

So, that's covered most of the basic data processing requirements. You're either scanning through a set of data to work on it repeatedly. Or you're looking a key up in a dictionary, so you need search to be fast.

What about the other dynamic classes? Somebody's spent a whole lot of time on them, so surely they're useful for something?

There are situations where you might want to consider alternatives to arrays and hash tables. For example, there's `std::map`, which uses red-black trees and has logarithmic complexity for searching, inserting and deletion. But this is not as good as *O(1)* of a hash table. The situations where a hash table might not be the best include:

- Scanning of the whole data set that is stored in sorted order — neither `std::vector` nor `std::unordered_map` are good at this.
- Real-time latency-critical situations — where the worst-case linear performance of searching a hash table is too risky.

But if you ask me, you can still use only arrays and hash tables in combination. Hash tables aren't great at scanning because it's a non-contiguous linked list scan.

Here's a funny thought:

1. Insert repeatedly into the hash table, and then

2. Linearize the hash table in an array.

Those are your main trade-offs. Beyond that, if you're only searching a set of keys, but don't need to map the key to any other data, then use a set rather than a dictionary (officially called an "associative container"). There's a (slow) red-black binary tree in `std::set`, but fortunately there's a (faster) hash table for that called `std::unordered_set`.

Linearizing Containers

One common optimization is to perform some "preprocessing" before doing a lot of sequential processing of the data. This applies when the startup does a lot of insertions, but the main processing is mostly about scanning the data. In this case, we can switch to a linearized version of a dynamic container for faster scanning. Here's example code for linearizing a linked list:

```
// Linearize linked list to vector
std::list<int> mylist;
std::vector<int> vec;
// ....
int n = mylist.size();
vec.reserve(n);
for (auto& iter : mylist) {
    vec.push_back(iter);
}
```

This code to linearize is not particularly efficient, because it's forced to linearly scan the linked list, and then insert into the vector one-at-a-time. However, I can't see a way to do a bulk-insert out of a linked list.

As an alternative, if we no longer needed the linked list version, we could use the `merge()` member function (C++17) to transfer items from the list container to the vector. This is particularly effective because `merge()` changes the internal container pointers, but doesn't call any copy or move methods.

Changing Containers

Another idea is to convert our insertion-friendly container to one that's best for fast searches. One idea that goes from binary trees to hash tables is this:

- Handle the insertion phase with `std::map` — logarithmic insertion complexity with red-black trees.
- Convert to `std::unordered_map` (hash table) for faster searches.

Note that C++17 has the `std::merge()` member functions for splicing one container into another. There's also `extract()` to remove a single item. Note that these routines don't move or copy any user data, but only update container internal pointers. This avoids the need for erasing data from one container and re-inserting all the data into the other container.

On the other hand, hash tables also have fast insertion with constant time on average, which is better than logarithmic (on average), so why do we need the red-black trees at all? One reason is that hash tables can degrade to linear performance in the worst case. Another reason is that the trees are good at fast processing of the data in sorted order, whereas hash tables have unsorted data.

Maybe we should do the reverse, handling insertions with our hash table, and then converting to a red-black tree for scanning in sorted order. No, not really. If we want sorted scanning of data, we'd probably do better to export the hash table to a `std::array` or `std::vector`, and then use `std::sort()` on the array or vector.

So many choices, so little time!

Useful Member Functions

Optimizing containers is about choosing the best one for your requirements, and then making the best usage of the interfaces that are provided. You don't need to write your own if you can do better with the standard containers.

Memory management of the various containers can be further optimized in a number of ways. Firstly, you can consider things like whether the container is "full" and what "capacity" it has. The main member functions include:

- `size()` — number of elements in the data structure.
- `capacity()` — maximum allowed with current memory.
- `reserve()` — request an amount of memory.
- `resize()` — reorganize to a bigger or smaller size.
- `clear()` — quickly remove all elements.
- `shrink_to_fit()` — request a smaller memory size.

The hash table containers, such as `std::unordered_map`, also have member functions to control the number of buckets and the resizing policies:

- `bucket_count()` — size of the hash table array.
- `bucket_size()` — length of a chain at an index.
- `load_factor()` — number of keys divided by hash table size.
- `max_load_factor()` — read or set the load factor that will trigger a rehash.
- `rehash()` — manually trigger a hash table size change and rehash (at your discretion).

You can use these member functions to track how effective the hash table is performing. This also allows taking control of the policy of when it will auto-resize and rehash into a bigger hash table with more buckets.

These C++17 member functions are useful sometimes for removing or moving multiple elements in a container:

- `extract()` — pulls a node out of the container data structure.
- `merge()` — efficiently combines two containers.

Hidden Auto-Resize Slugs

The auto-resizing capabilities of many C++ containers makes them dynamic and easy to use. However, it also hides a common efficiency that has existed since the earliest days of the STL: hidden calls to special functions. In fact, there are multiple reasons that you might want to avoid container auto-resizing:

- Slow performance — every object might get moved.
- Iterator invalidation — all objects could be at new addresses.

Auto-resizing of a container is probably something you want to avoid for performance reasons. In the worst case, it can trigger a significant delay when you're inserting into a container. The cost of an auto-resize may include:

- Memory allocation — e.g., allocating a new memory block or a hash table array.
- Move assignment calls — not for all container classes.
- Re-hashing — re-computing this for all the objects.

Note that some containers will call the move assignment operators, whereas others will resize the container without actually putting the stored objects in new locations. Here's how it works for some:

- `std::vector` — calls move assignments if the allocated block changes.
- `std::unordered_map` — zero move operator calls.

The situation with the hash table is complex, but basically it moved internal pointers around, but not your objects. The hash container doesn't need to move the objects inside the nodes on the chained linked list, so doesn't call move operators for the user's objects on those nodes. However, it does have to do other container-internal computations:

- Re-compute the hash function for every node's key, and
- Re-attach the node to a different chained linked list.

There's no overall mechanism to control the resizing properties of all containers, but we can use various different methods. The main solutions are:

- Reserve maximum memory, or
- Manually manage the resizing process.

Initialization with maximum size. The first idea for avoiding auto-resizing is to guess the maximum number of elements we could possibly need to store in the containers, and call the `reserve()` function at the definition of the container object. For example, the code could be:

```
    std::vector<int> v;
    v.reserve(1000);
```

But not this, which will run 1,000 default constructors in a vector of non-scalar type:

```
    v.resize(1000);  // Slow!
```

And this also would create 1,000 new objects and run their constructors:

```
    std::vector<int> v(1000);  // Slug!
```

This reservation of memory is a type of "preallocation" optimization. We ensure that all memory that could be required is allocated during the initialization phase, which ensures that no memory allocations are performed later in the hotpath.

Detecting auto-resizing. Alternatively, we can detect when an insertion is likely to trigger an auto-resize. The standard container interfaces allow us to know this, before we do an insertion:

```
    if (v.size() + 1 > v.capacity()) {
        // Resizing likely on insertion!
    }
```

Unfortunately, there's not a lot that we can do in this situation. I mean, we could just "not insert" as a strategy, but that doesn't sound great.

Deferring container auto-resizing. Alternatively, we could detect the situation 10 insertions ahead of time, still insert the single item, and then do something later to manage the resizing, perhaps in a lower-priority thread.

```
    const int n_lookahead = 10;
    if (v.size() + n_lookahead > v.capacity()) {
        // Resizing will be soon!
    }
```

In standard C++ classes that are more dynamic than the basic `std::vector`, such as `std::unordered_map`, we can defer the auto-resizing to a more convenient time. This is only possible for the dynamic classes based on linked lists or binary trees. Note that the hash table classes actually used linked lists, because of linear chaining as the collision resolution mechanism.

We can initialize the hash table to a particular size in the constructor. The bucket count is an optional integer parameter to the constructor.

```
    std::unordered_map<std::string, int> hmap(1000);
```

This only works well if we know the maximum size that we need. For more dynamic handling, we can also use the bucket management functions in the `std::unordered_map` interface to detect when the hash table is getting full, and take appropriate action.

The "load factor" is the number of elements stored in the container, divided by the hash table array size (i.e., the number of "buckets"). There's no target load factor in the standard definition, but an implementation will typically aim for a load factor around 0.5 to 1.0. The container implementation also has a "maximum load factor" that will trigger a rehash into a bigger hash table when it's exceeded.

When the load factor is near the maximum value, this means the class will soon be increasing the hash table size, and possibly re-hashing every single element.

Here's the idea coded up:

```
// Detect rehash risk
std::unordered_map<int,string> h;
int n_lookahead = 10;
float load_estimate = (h.size() + n_lookahead)
                    / (float) h.bucket_count();
if (load_estimate >= h.max_load_factor()) {
    // Rehash is likely!
}
```

In the case of a hash table, we can actually ensure that it won't rehash by manipulating the maximum load factor setting. The `max_load_factor` method has overloads allowing us to both get and set the value.

Hence, a solution that defers rehashing: increase the maximum load factor setting, insert our new object, and then reset the maximum load factor:

```
float old_load_factor = h.max_load_factor();
h.max_load_factor(old_load_factor*2.0f); // Skip rehash
h.insert({ x, s });  // Insert the object without fear!
h.max_load_factor(old_load_factor);  // reset
```

Note that we have to be careful, lest we introduce another hidden slug: never-resizing our hash tables.

Don't defer it forever!

If you forget to ever rehash your hash table, it won't crash, but becomes a hidden slowdown. The use of chaining means that the standard hash table containers won't fail if they never get auto-resized, but they will degrade to the linear performance of a linked list for all operations.

Hand-Coding Containers

The standard containers are elegant and beautiful, but they are designed to be very general. Hence, they can sometimes be slower than you could achieve on your own. Some of the problems with standard container performance include:

- Too many allocations and deallocations with `new` and `delete`.
- Non-contiguous storage in dynamic containers (e.g., linked lists, binary trees).
- There is no easy way to change the overall algorithm — e.g., you can't change `std::unordered_map` to not use linked list chaining for collision resolution.
- General containers may not meet the requirements of your specific application.

In short: sometimes you can do better!

20. Move Semantics

What are Move Semantics?

Whoever invented move semantics deserves the Nobel prize. Move semantics refers to a beautiful and elegant addition to C++ class definitions added in C++11. The syntax is concise and the internal definition is semantically consistent in many ways. But the most beautiful part of move semantics: it's all about making C++ even faster!

Move semantics were about making C++ more efficient at a very high level. The issues were unnecessary calls to class constructors and copy assignment operators in a number of situations, such as:

- Temporary object creation
- Returning a class type from a function
- Overloaded operator return types

Most of the changes in C++11 that brought in move semantics were done in a way that maintained backward compatibility. The new features available in classes included:

- Move constructors
- Move assignment operators

Whereas the new special members needed to be added to existing classes, there were also a number of automatic compiler optimizations that were enhanced to take advantage of move semantics:

- Copy elision
- Return Value Optimization (RVO)
- Named Return Value Optimization (NRVO)

Some parts of copy elision rely on move operations, whereas other cases of copy elision and RVO are actually independent of move semantics, and can be used without move special functions. But the optimal choice is to use all of them together.

Copy Elision

Copy elision is an automatic C++ compiler optimization that "elides" (removes) various "copy" operations on objects. I guess "copy removal" just didn't have the same ring to it?

Copy elision works in particular situations in the C++ language. These situations include:

- Class-type `return` statements — the main situation.
- throw expressions (and handlers)
- Coroutines

The effect of copy elision is to avoid a full object copy. Instead, the place where the new object is used simply refers to the old object, which would have been copied without this optimization.

Technically, there are other unusual situations, and there are two variants of copy elision:

- Removal of copying, or
- Downgrading copying to a move operation.

You don't need to modify your code to get the benefits of copy elision. In fact, you also don't need to turn the optimizer up to eleven. Copy elision is a normal part of the C++ standard.

Return Value Optimization

Returning an object type is a special case where the old code used to be inefficient. The good news:

- Return Value Optimization (RVO) is an automatic compiler optimization.
- Nothing you need to do!

Well, actually you do need to declare a move constructor and a move assignment operator to get the full benefits, but you were doing that already, right?

Why was RVO needed? Because return statements used to cause lots of copying for objects. This could be worked-around by declaring a reference object parameter, which was returned back, instead of having an object return type. But that's inconvenient, and there are also cases where it's not possible:

- Binary operator overloads (non-assignment) — e.g., binary "+" operator.
- Unary operator overloads (non-increment/decrement) — e.g., unary "-" operators.
- Postfix increment/decrement operators — must return the old object (not the current one).

Operator overloading was one of the most beautiful parts of C++ signatures. Shame that it used to be inefficient, but now it's not.

Any function can return a class object, rather than a pointer or reference, but the effect is that the function itself needs to declare a local object to be returned. Consider this code:

```
MyClass func(int x)
{
    MyClass ret(x);  // Create object
    return ret;    // Copy object
}
```

And then it gets copy constructed again when we call the function:

```
MyClass m = func(3);
```

Move semantics solve this problem, in combination with copy elision. This special case is called Return Value Optimization (RVO), and allows the compiler to do "one-two-skip-a-few" for object copying.

To get even more technical, this situation is called Named Return Value Optimization (NRVO), when a function returns a named local variable (i.e., "`ret`" here). The non-named version of RVO occurs when the function returns an unnamed object, such as a temporary object created as the result of a construction or operator.

Some types of RVO are implementation-specific and optional for the compiler to do. However, NRVO is "mandated" by the C++17 standard when returning a named local object variable. I guess unnamed RVO will be mandated at some time in the future, too.

RVO is very efficient in that it doesn't just convert copying to moving, but can in fact avoid the complete creation of temporary variables. The compiler can optimize the above code so that the `return` statement constructs or moves the object directly into the place where it was called from. This means not only we avoid various copies/moves, but also the avoidance of that temporary object's constructor and destructor, too.

Moving Multiple Objects

Moving multiple objects arises as an inefficiency in C++ because there's no multi-move semantics. Some examples where you want to move multiple contiguous objects to a different memory location include:

- Move capabilities for a custom multi-object container.
- Shuffling objects along in a sorted array on insertion or deletion.
- Auto-resizing a `std::vector` container (bigger or smaller).

There's no multi-move constructors or assignment operators in the standard C++ language, so there's only single object moving methods. In practice, you can move multiple objects in various ways, such as:

- Moving them one-by-one
- `std::move(begin, end, dest)` overload

Note that this is the `std::move` overload that does real runtime work, not the simpler version that's just a type-cast to an R-value reference.

Unfortunately, all of these ideas are calling the move constructors for every single object. This is fine for scalar types or classes with simple inlined versions, but it's still not optimal.

The workaround for your own class is simply to define a non-special member function to do fast moving, which you can call explicitly. But this doesn't solve the general problem of using your new class in a container that may need to bulk-move your objects at some point.

Generic Move Operator

Some types of objects are "relocatable" and can used an optimized move method. The basic ideas of move semantics refer to the difference between a "shallow copy" (also called a "bitwise copy" or a "byte copy") versus a "deep copy". The basic idea is this:

- Copy assignment or constructor — deep copy
- Move assignment or constructor — shallow copy

The copy constructor has to make a full copy of every data member of the other object to create a new object. The old object is unchanged.

The move constructor needs to transfer all of the data members from the old object to the new object. And then the old object needs to be "cleared" in some way, which leaves it in a "valid" state (so that its destructor doesn't crash or deallocate memory it no longer owns). Hence, why not do these steps in general as an optimization:

- Shallow move old data members to new object — bitwise copy of all bytes.
- Clear old object's data members — zero the old bytes.

This idea of a relocatable object that is a C++ class object is similar to the type trait "`std::is_trivially_move_constructible`" (C++11). However, this isn't quite what we want, which is a way to specify that our object is relocatable. The type trait instead only detects some cases where this is true. Perhaps we could set this type trait to "true" for our own class, and the standard container classes will honor this type trait setting, but I have my doubts.

Instead, let's think about generalizing the idea to all relocatable class types. We can even code up the idea:

```
template<typename T>
T& generic_move_assignment_buggy(T& newobj, T& oldobj)
{
  memcpy(&newobj, &oldobj, sizeof(T)); // Move bitwise
  memset(&oldobj, 0, sizeof(T));
  return newobj;
}
```

Well, that has an aliasing bug if the new and old object are the same. So, let's fix that first:

```
template<typename T>
T& generic_move_assignment_safer(T& newobj, T& oldobj)
{
    if (&newobj != &oldobj) {  // Avoid aliasing
            memcpy(&newobj, &oldobj, sizeof(T));
            memset(&oldobj, 0, sizeof(T));
    }
    return newobj;
}
```

Does this idea work?

The short answer is: yes and no. Yes, this idea can be used very often, and is efficient.

Let's look at the good news first. This approach works for all these situations:

- Scalar types — moving an integer is a bitwise copy anyway.
- Simple object data members — if this move approach also works for the sub-object.
- Virtual functions — yes, the hidden "`vptr`" pointer in the old object is also moved by the bitwise copy.

However, technically the full answer is "no," because there are some problem areas when using this approach:

1. Self-referring pointer data members.

2. Virtual function problems — `vptr` is nulled in the old object.

3. Virtual destructor problems — a problematic special case.

4. External pointers into the old object (invalidated).

5. Obscure portability problems with zero byte representations.

Self-referencing data member problems. This is a problem when the object is relying internally on its own address. Self-referring internal pointers (or references) are data members inside the object that point to another part of the object. These are uncommon, and seem like bad programming style anyway.

Note that pointers pointing outside of the object are just fine. In fact, that's why this copy-and-zero approach is efficient, because we don't need to copy and reallocate any pointer data members. A bitwise copy of a pointer or reference is still pointing to the right place.

Virtual function problems. The `memset()` function has cleared every byte to zero, including any of the hidden "`vptr`" pointers to the virtual function table. When there's any virtual function in a class, then it has a hidden pointer inside the object. There are also other places that may have another `vptr`, including:

- Base class — but it usually shares a single `vptr` with the derived class.
- Multiple inheritance — requires multiple `vptr`'s in the object.
- Subobject data members — if they are of a class that has its own virtual functions.

If your code calls any of these virtual functions after it's been nulled, I'm betting against you. Nevertheless, we might be able to work around this by simply not calling any virtual functions after this move sequence.

Virtual function problems. Destructors make it a little more difficult, because it's hard to stop the C++ compiler from calling them. And every class with any other virtual function is supposed to make its destructor also virtual. Just ask Scott Meyers in the very first edition of his *Effective C++* book, which was good advice in the 1990s, and still remains so.

Hence, if our object has a virtual destructor, it may try to access the null `vptr` at some point. There's no simple workaround to "just avoid calling the destructor," since it's called implicitly.

External pointers into the object. I feel like we can live with this idea. If there are any pointers or references to refer to the old object's internal data, they are now invalidated. But that's true anyway, because the whole idea of a moved object is that it's going away.

All bytes zero portability. There's a theoretical portability problem when using `memset` to clear an object to have all its bytes equal to zero. I'm not sure it even applies anymore, as I don't know of any platform where this is a real problem. The concern is whether clearing all the individual bytes to zero will actually clear multi-byte data to its equivalent zero or null value.

In practice, these are all true:

- Characters — byte zero is always character zero.
- Integers (signed and unsigned) — all bytes zero is integer zero.
- Floating-point — all bits zero is floating-point positive zero in the IEEE 754 standard.
- Pointers — all bytes zero is the `nullptr` in any platform I know.

Hence, I'm not sure it's a real problem, but every book on C++ portability I've read has mentioned it, so now I have, too.

Workaround for fast move problems. I hate to give up on a really efficient idea, so we can point to the limitations where we need to ensure:

- "Relocatable objects" with no internal pointers or references.
- No virtual functions
- No virtual destructor

Maybe we can work around the virtual function problems by not clearing the `vptr`. Here's the idea:

```
    memset((char*)&oldobj + sizeof(void*), 0,
                    sizeof(T) - sizeof(void*));
```

This assumes that there's only one `vptr`, and it's shared by the base class and derived class. Unfortunately, this idea still fails for subobjects with their own virtual functions and multiple inheritance where objects can have more than one hidden `vptr`. Anyway, it's a worthy try, and we could always ban virtual functions, which aren't that efficient anyway!

Multi-move generic function. This idea can be generalized to moving a contiguous array of multiple objects at once. The need for such a "multi-move" capability is less often required, but can arise when containers resize, and we also need it to implement sorted array insertions and deletions.

The above "generic" version only works for one object. Let's think about generalizing the idea of bytewise moves and then clearing to zero.

Here are some thoughts:

1. The idea still generally works on a mult-object block, because it's similar to moving one object at a time.

2. Overlapping ranges of objects are a problem, because the `memset` will wrongly clear some of the newly moved objects.

Amusingly, note that we did deal with the "overlapping blocks" problem in the single-object generic move. It's the same as the "aliasing" check!

Detecting overlapping ranges more generally is a bit more intricate to code. Here's my attempt at updating the generic move method to support multiple objects:

```
 template<typename T>
T& generic_multimove_assignment(T * destarr,T* srcarr,int n)
{
    if (destarr == srcarr) {  // Same exact block
        // Nothing to do
    }
    else {
        T* enddest = destarr + n;
        T* endsrc = srcarr + n;
        if (enddest > src && enddest < endsrc) {
            // Overlapping (moving left safely)
            memmove(destarr, srcarr, n * sizeof(T));
            int num_overlap = enddest - src;  // Ptr arith
            // Clear non-overlapping part
            memset(enddest, 0, (n - num_overlap)*sizeof(T));
        }
        else if (endsrc > dest && endsrc < enddest) {
            // Overlapping (move right safely)
            memmove(destarr, srcarr, n * sizeof(T));
            int num_overlap = endsrc - dest;  // Ptr arith
            // Clear non-overlapping part
            memset(src, 0, (n - num_overlap) * sizeof(T));
        }
        else {
            // Non-overlapping blocks (move all)
            memcpy(destarr, srcarr, n * sizeof(T));
            memset(srcarr, 0, n * sizeof(T)); // Clear old
        }
    }
    return newobj;
}
```

Compiler support? Even with the restrictions to scalar and relocatable objects, and other problems listed above, this idea of just moving memory blocks around is so efficient that maybe the compiler should provide this as an option automatically? Is this the default assignment operator? No, not quite, because the default move constructor or assignment operator is a "member-wise move" of all of the data members. This is the same as a bitwise move if all data members are trivial, but any complex classes as subobjects will need their own move constructors called.

I like this whole idea a lot more than the normal move member functions, where you have to fiddle endlessly with every single data member. Come on, the single object version is only two statements! Hence, I'm hereby recommending to the standards committee that, like the "`=default`" specifier, there needs to be a new "`=fast`" specifier added to the C++26 language.

21. Arrays

Arrays are wonderfully efficient! They're the most basic data structure known to humanity. The main features to note about an array include:

- Contiguous memory storage — great for cache locality.
- Single type of data — no need to be worried about the type.

In modern C++, there are several ways to create an array data structure:

- `std::array`
- `std::vector`
- `std::inplace_vector` (C++26)

There are also some older methods of using arrays that still work in modern C++ code:

- Fixed-size array variable: `int arr[10];`
- Allocated fixed-size array: `new int[10];`
- Old-style allocated array: `malloc(sizeof(int)*10);`

Note that the size of arrays in these examples don't need to be a compile-time constant in C++. They can be a variable, where the size of the declared array is sorted out at run-time.

Array Operation Complexity

There are two main types of arrays to store objects: sorted and unsorted. Well, actually, there's other types of arrays with different semantics (e.g., stacks, queues, heaps, ring buffers), but let's just look at searching and sorting for now.

Are they fast? Here's the 10,000 foot view:

- Unsorted arrays — very fast insertions/deletions, but slow searches (linear) and even slower to sort the data.
- Sorted arrays — faster search (logarithmic), slower insertions/deletions, and great if you need sorted data.

In more detail, here's the overall complexity analysis of the basic searching methods:

- Searching — unsorted is *O(n)* (linear search) and *O(log n)* for sorted (binary search).
- Inserting — unsorted is *O(1)* (add to the end), but *O(n)* if sorted (shuffle required).
- Deleting — this is *O(1)* if unsorted (tricky swap method!), but *O(n)* if sorted (also shuffles).
- Print unsorted — both are *O(n)* with a linear scan of the array.
- Print sorted — unsorted is *O(n log n)* because it requires a sort, but only *O(n)* if already sorted.

And some other algebraic operations:

- Maximum/minimum — unsorted is *O(n)* because it requires a scan, but only *O(1)* if already sorted (choose first or last element).
- Top-k elements — unsorted requires an *O(n log n)* sort or at least a "partial sort"; only *O(k)* for a sorted array.
- Sum or average — both are *O(n)* because the whole array must be scanned.

Modern C++ Arrays

We're going to implement our own sorted and unsorted arrays to examine the algorithms. Standard C++ already has two types of unsorted arrays in `std::array` and `std::vector`. We could just wrap around those types, but I'm going to use low-level raw arrays to show the algorithms in more detail.

Sorted arrays are trickier. Note that there's no "sorted array" class in the standard C++ library.

However, there are some primitives we can use to achieve sorted arrays:

- `std::sort()` — modern C++ version with a hybrid quicksort/heapsort algorithm.
- `qsort()` — old-style quicksort with function pointers (not recommended).

There is also some builtins for "binary search" on a sorted array:

- `std::binary_search()` — modern C++ implementation for array.
- `std::equal_range()` — binary search that handles duplicate elements in the array.
- `bsearch()` — old-style binary search with function pointers (not recommended).

If we are inserting into a sorted array, we don't need binary search exactly, because we're assuming the element isn't already in the array. Instead, we need a "binary-like search" method of finding the index location to insert a new item. In other words, we need to find the spot where the item fits in the array, but do it logarithmically, rather than using a slow linear scan.

Writing a binary-like search algorithm to find the insertion point is very fiddly coding! Fortunately, the standard C++ library has two methods that code it for us:

- `std::lower_bound()` — generalizes binary search for insertions.
- `std::upper_bound()` — similar version that finds the location above.

Strictly speaking, `std::binary_search()` in the C++ standard only requires a "partitioned" array rather than a "sorted" array. But for a scalar type with well-defined comparisons, this is the same thing.

Custom Array Implementation

Anyway, let's look at some of the basic operations in our custom versions of array algorithms. We'll examine the unsorted array version, but the sorted version is almost identical. Here's the overall class members:

```
template<typename T, int N>
class UnsortedArray {
private:
    T arr_[N];
    int capacity_ = N;
    int count_ = 0;
    //...
};
```

Note that "`capacity_`" is somewhat redundant if we're templating based on a compile-time array size. However, it would be useful if we were dynamically constructing our arrays at runtime.

Here are some of the basic "getter" functions:

```
    int size() { return count_; }
    int count() { return count_; }
    int capacity() { return N; }
```

And here are some of the basic utility functions:

```
    bool empty() { return count_ == 0; }
    bool full() { return count_ == N; }
```

Container Deletion Pitfalls

While we're on the topic of deletions, let's look at some common mistakes with deletions from C++ containers. There are at least two major pitfalls in using the `erase()` method to remove an object from a C++ container. Here's the basic first attempt:

```
    for (auto iter : container) {
        if (want_to_delete(*iter)) {
            container.erase(iter);  // Kaboom!
        }
    }
```

This will crash with a big mushroom cloud. The problem is that we've assumed the iterator stays valid, whereas the `erase()` method actually returns an updated iterator that we need to use. We can't use a range `for` loop to do this, so we have to use `begin()` and `end()` manually:

```
    for (auto iter = container.begin();
                iter != container.end(); ++iter) {
        if (want_to_delete(*iter)) {
            iter = container.erase(iter);  // Use return value
        }
    }
```

This is not a crash, but still a major bug. The iterator loop skips over the next item after the erased object. There are two increments in the deletion sequence:

1. `erase()` returns the next valid iterator (after the removed object), and

2. `++iter` skips to the next element (again!).

To get it correct, we need to change the idiom to avoid `++iter` if we erase anything.

```
for (auto iter = container.begin();
              iter != container.end(); /*Not here!*/ ) {
    if (want_to_delete(*iter)) {
        iter = container.erase(iter);  // Use return value
    }
    else {
        ++iter;  // Only if not erasing!
    }
}
```

And now the code finally works!

Bypassing Interfaces

The `std::array` and `std::vector` classes are designed to allow you to get access to the stored data via the `data()` member function. It's also guaranteed that the data is stored in contiguous memory locations. Note that this is also true of `std::string`, which has a `data()` member and also `c_str()`, which returns the same address.

The `data()` method allows direct access via pointers or low-level array types to the data in the standard array or vector containers. Whether doing this is any faster is unclear, and needs benchmarking, since many of the member functions are simple pass-through inlined functions that work on the internal data anyway.

But there's certainly a few pitfalls! The address returned by the `data()` member is not guaranteed forever. There are at least two major types of bugs:

- Object is destroyed, or
- Object is moved or modified.

Since you have a pointer to an object's data, you want that object to stick around. But the object can disappear in a few ways:

- Stack object goes out of scope (triggering the destructor and unwinding the stack).
- Allocated object is deallocated by the `delete` operator.
- Object is moved by a container (e.g., an auto-resize or other "iterator invalidation" situation).

Even if the object stays around to watch your skills, there's another problem. If the underlying object is modified, then the internal address of the data that you have may become invalid. The issues are very similar to the well-known "invalidated iterator" problems with containers. Changes to the container that probably invalidate the `data()` pointer include:

- Insertions and deletions
- `reserve()`
- `resize()`
- `shrink_to_fit()`

Any of these members that modify the object are allowed to move the data. For example, they might allocate a different memory block, and move the whole array away from your pointer. But there are a huge number of other situations under which an iterator into a container may become invalidated, which presumably also invalidates an old address returned from the `data()` member function.

Watch out!

22. Unsorted Arrays

Unsorted Arrays Overview

Unsorted arrays are not an all-star data structure, and don't get a lot of use for basic search requirements. The main features include:

- Slow search lookups in cases like associative arrays or sets (linear scan cost).
- Fast insertions and deletions (constant cost, without any "shuffle").
- Sorting an unsorted array is costly with *O(n log n)* complexity.

Unsorted arrays are very useful if we want fast insertions and deletions, but rarely need to search or sort the array. Insertion is very fast with constant time, just by adding the new element at the end of the array. Deletions can also be implemented in constant time, but only via a trick of swapping the to-be-deleted element with the last element.

Interestingly, we can always fix our unsorted array by sorting it, and that turns out to be a decent idea. Let's examine the two ways to get a sorted array:

- Build an unsorted array, then sort it, or
- Incrementally maintain a sorted array.

The first plan costs *O(n)* in total to do all the *n* insertions (unsorted), and then costs *O(n log n)* to sort it with `std::sort`. The second plan costs *O(n)* for every one of the *n* insertions into a sorted array, and so we get to *O(n^2)* quadratic complexity for the incremental sorted array approach. In summary, our analysis suggests:

- Unsorted array (sort it later) — complexity of *O(n log n)*.
- Sorted array (incremental) — quadratic *O(n^2)* complexity.

An unsorted array might be the way to go? However, as discussed above, it's not as bad as that sounds if we have scalar types in a sorted array, because the "shuffle" is a single memory block copy.

Note that an unsorted array is actually sorted in a weird way: by the order of insertions. Hence, if you have an ordered sequence of data, they are mapped into the array sequence according to the order in which they are processed. If these objects have an associated timestamp, your supposedly unsorted array may well be sorted implicitly according to the timestamp field.

Unsorted arrays are underestimated, and can be efficient in practice. An array that is unsorted functions as a list of items, but is stored in contiguous memory, which can make scanning the array efficient in terms of cache locality (e.g., faster than linked lists in `std::list` or red-black binary trees in `std::map`).

Unsorted arrays can be useful for semantics other than basic search lookups. An array can efficiently implement a fixed-size stack, but a fixed-size queue is better implemented using a ring buffer that progresses around the array in a circular fashion. You can also put a balanced binary tree or a heap data structure into an array, but we're getting far away from a basic unsorted array in doing that.

Linear Search of Unsorted Arrays

Linear search is the worst part of unsorted arrays. There's not really a better way to search an unsorted array. Here's a simple hand-coded linear search of the array to demonstrate the algorithm that's happening:

```
int find_linear_search(const T &item)
{
    for (int i = 0; i < count_; i++) {
        if (item == arr_[i]) return i;  // found
    }
    return -1; // not found
}
```

The above assumes we're stored our data in a raw array type as the data member. If we choose to store the data as `std::array` or `std::vector`, we could use standard member functions to search the array, such as `find()`.

Note that if we were doing a lot of searches of an array without many insertions or deletions, here's an idea: pre-sort the array! This gives us this approach:

1. Pre-sort the array with `std::sort`

2. Use binary search on our newly sorted array (logarithmic complexity).

Template Value vs Reference Parameters

Templating based on a type has a common conundrum about how to choose between passing function parameters by reference or value. The desirable efficient that we want is usually:

- Small integer types — pass-by-value.
- Large class types — pass-by-reference.

Which signature should we use?

```
int find_linear_search(const T &item)  // Const reference
int find_linear_search(T item)  // Pass-by-value
```

Which one we desire for larger non-class types, such as `long` or `double`, is somewhat implementation-dependent and you really need to benchmark to check! Unfortunately, there's no way to alter the signature of a templated function according to a compile-time setting. I don't think there's a way to do it in type traits.

However, the most common modern C++ style is to use `const` reference parameters. The reasons are:

- Large class types — `const&` references are much faster.
- Small integer types — it's not much worse.

In one sense, I'm not sure about the last point, because:

1. It's a micro-optimization, and

2. The compiler may auto-optimize it anyway.

But there is a simple solution whereby you can use `const&` reference parameters for generic types, but use pass-by-value for small integers. Template specialization to the rescue! Just define specialized versions of templated functions for the handful of small integer types:

```
int find_linear_search(int item)  // Pass-by-value
{
    // etc...
}
```

Now you only have to define about 27 more versions for every type.

Fast Linear Search

You're thinking that this doesn't exist, and the heading is an oxymoron. But there are situations where linear search on an unsorted array can be faster than the alternatives:

- Small number of elements
- Sentinel search optimization
- Low-level support for searching
- Parallel linear search

Let's examine all of these techniques in turn.

Sentinel linear search optimization. This is an optimization attributable to Knuth (1973) in the Mix programming language. The idea is to remove the conditional test in the loop (i.e., removing "`i < count`") by guaranteeing a successful search. The trick is to add an extra element at the end of the array, which equals what we're searching for.

Note that this requires that we declare our array data member with one more item than the capacity. We always need a spare element at the end, even if the array is full to capacity.

```
        T arr_[N + 1];  // Extra dummy element
```

Sentinel-based searching is only good for arrays of scalar types, because it requires making a copy of the search element, which is created at the end. The sentinel search of an unsorted array still has linear complexity, but has a lower complexity constant because each loop iteration is faster in practice.

Low-Level Search Support

Some types of CPU have explicit instructions that support scanning a memory block for a value. If we're using an array of characters or bytes, there are these candidates:

- `std::find` — on an array, vector, or string type.
- `strchr` — old-style character strings (null-terminated)
- `memchr` — low-level memory blocks of bytes.

The modern C++ code using `std::find` looks something like this:

```
bool find_standard(const T& item)
{
    auto iter = std::find(arr_, item);
    return iter != arr_.end();
}
```

The version that returns the integer index of the element in the array is:

```
int find_standard_index(const T &item)
{
    auto iter = std::find(arr_, item);
    if (iter == arr_.end()) return -1;  // Fail
    return iter - arr.begin();  // Pointer arith
}
```

Note that this idea only works for arrays of contiguous memory. Pointer arithmetic doesn't work well on general iterators for dynamic memory containers.

Parallel Linear Search

There are multiple ways that we could parallelize our linear search algorithm. It just depends on our budget! Here are some options:

- CPU SIMD instructions (e.g., AVX or ARM Neon)
- Multithreading (on CPU)
- GPU hardware

SIMD instructions allow use to test multiple values in parallel on a CPU. For example, an x86 CPU from Intel or AMD allows the AVX sets of instructions, and there are a few versions:

- AVX — 128 bits (4 x 32-bit integers).
- AVX-2 — 256 bits (8 x 32-bit integers).
- AVX-512 — 512 bits (16 x 32-bit integers).
- AVX-10 — 1024 bits (32 x 32-bit integers).

CUDA C++ GPU linear search. If we have an NVIDIA GPU, this advanced type of parallelism is much more extensive. In fact, we can create 1024 threads, and each thread can compare only a few elements with our search key.

This sounds like an almost constant-time algorithm on the GPU, but it's not quite that good. In practice, there are two phases:

1. Compare each loop element in parallel, and

2. Collate the results.

The GPU can compare all the array elements 1024 at a time. Hence, it's not constant time, but it's still linear time divided by 1024.

Also, at the end we have a synchronization problem with detecting which of the threads had a successful result of the comparison. It's not quite as bad as a "horizontal reduction" of the array (e.g., max or sum), but we have to synchronize the results in shared memory or global memory.

We could use "warp shuffle" instructions that coordinate via faster GPU registers, but these only work within each warp of 32 threads, so it ends up being like a horizontal reduction over each warp.

Unsorted Array Insertions

Inserting into an unsorted array is very fast because we can just insert it at the end. This is very efficient with constant time complexity.

The code example for insertion at the end:

```
void insert_end(const T & obj)
{
    if (full()) {
       throw std::overflow_error("Insert full array");
    }
    else {
       arr_[count_++] = obj;
    }
}
```

There's nothing much to this code: only one statement! It's very efficient to insert at the end of an array.

Insertion at an Index

Inserting in the middle of an unsorted array seems to be an *O(n)* operation. If we needed to insert into the middle, it would seem slower because of the need to shuffle the other elements out of the way. And that would certainly be true of a sorted array, where a shuffle is needed to maintain the sorted array.

But, no, we're talking about an unsorted array here. Let's ban the shuffle.

There's a move trick to insert into the middle of an unsorted array at a given index in *O(1)* time. The trick is to note that in an unsorted array we only need to move a single element out of the way. The idea is two short phases:

1. Move the existing element "out of the way" and to the end.

2. Insert the element at that location.

Here's a coded version of the "move away to the end" optimization. One fast way is to use `std::move`, which is like a type cast with no runtime code, and this causes move assignment on a complex object (or simple byte copying on a scalar type). Here's the code:

```
void insert_at_offset(const T & obj, int offset)
{
    if (full()) {
       throw std::overflow_error("Insert full array");
    }
    else {
       // Move to end
       arr_[count_ + 1] = std::move(arr_[offset]);
       arr_[offset] = obj;  // Insert at location
       count_++;
    }
}
```

Note that this only works for an unsorted array, not a sorted array. If we wanted a sorted order, or we need the implicit order-of-insertion in an unsorted array, then this "move to end" idea cannot be used as it will ruin the ordering.

Fast Unsorted Array Deletion

There's a trick for deleting an arbitrary element from an unsorted array that is often missed in articles. Unsorted array deletion need not be *O(n)* complexity, but can be done in *O(1)* time.

Deletion of an item from an unsorted array is a two-phase operation: find and destroy. Here's the code to find the element, which uses linear search to find its offset, and is thus *O(n)* unavoidably:

```
void delete_key(const T& item)
{
    int offset = find_linear_search(item);
    if (offset == -1) {
      throw std::invalid_argument("Delete not found");
    }
    else {
      delete_offset_swap(offset);
    }
}
```

The naive idea for deleting from an unsorted array that we've found here is to remove the element and "shuffle" the rest of the elements downwards (to the left) so that there's no "gap" in the array. Doing a shuffle isn't so bad for scalar types, where it's probably just one call to `memmove` behind the scenes. But for non-scalar objects, we're moving a lot of objects. Either way, our unsorted array deletion with a shuffle has cost complexity of *O(n)* time.

There is a faster way!

First, let's get rid of the special cases: if there's only one element in the array, just erase it, and set the count to zero. And if the erase location is the end-most object, just erase it there, and decrement the count. Otherwise, if the object we want to remove is at the front or middle of the array, we do a tricky swap with the end element:

- Swap `arr[i]` with `arr[n-1]`
- Erase at `arr[n-1]`
- Decrement `n`

This swap idea has changed our unsorted array deletion from *O(n)* time to the optimal *O(1)* complexity. There's no loops anywhere!

Note that we can use `std::swap` here, and we may need to explicitly run the destructor of objects being destroyed (optional for scalar types). Here's what the code looks like:

```
void delete_offset_swap(int offset)
{
        if (empty()) {
            throw std::underflow_error("Delete empty arr");
        }
        else if (count_ == 1) { // ***
            if (!std::is_trivially_destructible<T>::value) {
                arr_[0].~T(); // Explicit destructor
            }
            count_ = 0;
        }
        else {
            if (offset != count_ - 1) {
                // Swap with the end element
                std::swap(arr_[offset], arr_[count_ - 1]);
            }
            if (!std::is_trivially_destructible<T>::value) {
                arr_[count_ - 1].~T(); // Expl destructor
            }
            count_--;
        }
    }
```

The above code uses "type traits" from modern C++ to detect whether or not we need to explicitly run the destructor when destroying an object in the array. This is very efficient because type traits are evaluated to compile-time constants, so the compiler should optimize out the path if not needed (i.e., using "dead code elimination"). There are several options available in the type traits library, depending on exactly what types we want to support in our array:

- `std::is_trivially_destructible<T>::value`
- `std::is_destructible<T>::value`
- `std::is_scalar<T>::value`

Actually, the above code has a minor inefficiency. The giveaway is that two code sequences with `is_trivially_destructible` are similar. Can you see it? We don't need to expressly test for `count==1` (marked with stars), because the general code in the else clause also works for that special case as well.

And also, what was I thinking? There's no need to swap the element to the end, only to destroy it there. That's two hidden moves inside `std::swap`, when we only need one moved element.

The better idea than swapping is to destroy the object where it is, and then move the end element down:

```
    if (!std::is_trivially_destructible<T>::value) {
        arr_[offset].~T(); // Destroy in place
    }
    if (offset != count_ - 1) {
        // Move down the end element
        arr[offset] = std::move(arr_[count_ - 1]);
    }
    count_--;
```

Note that `std::move()` here is only a compile-time type cast operation. It will ensure that the move assignment operator is used on complex class types, and is also efficient for scalar and other trivial types.

Yes, moving the end element to the middle of the unsorted array changes some addresses. It will certainly invalidate iterators over the container. But so would the shuffle of elements, so we're okay there.

Note that this only works for an *unsorted* array data structure. If we did this on a sorted array, we'd ruin the sorting order in the array by moving the biggest element into the middle of the sequence. Sorted arrays need to do the shuffle.

One final point is that this fast deletion trick with swapping will break the unofficial ordering of the array by its insertion order. If we have timestamps associated with our array elements, swapping the end element into the middle will ruin that implicit ordering.

23. Sorted Arrays

Sorted Arrays Overview

There is no standard C++ sorted array class, so we've got to implement our own. The C++ containers that can be used for sorted arrays include:

- `std::vector` — variable size.
- `std::array` — fixed size at compile-time.
- Native C++ arrays — old-style non-container builtin arrays.

A sorted array has a good search lookup cost, being logarithmic in the total number of elements, by using the "binary search" lookup algorithm. However, that's not as good as a hash table (e.g., `std::unordered_map`), which has *O(1)* average search cost.

Insertions and deletions have a poor *O(n)* theoretical complexity, although the first phase of finding where to insert or delete is also logarithmic, using an algorithm very similar to binary search.

The linear cost arises because once they find the location, they then need to shuffle elements:

- Make a gap (insertion), or
- Close a gap (deletion).

If we're using a class object for our array, such as `std::array` or `std::vector`, we can use the `insert()` method. This is doing a shuffle behind the scenes.

The main advantage of a sorted array is that it's, well, sorted, so if we want to process the array elements in sorted order, then it's already done for us. That's desirable because raw sorting of an unsorted array is expensive with its well-known *O(n log n)* complexity (e.g., `std::sort` typically uses a quicksort-heapsort hybrid).

If we need to use sorted data, there are other options in the C++ containers. The `std::map` container is implemented as a balanced binary tree, called a "red-black tree," and this has logarithmic complexity for all major operations: search, insertions and deletions. However, a sorted array has good memory cost because it uses contiguous storage, so it should not be underestimated!

Shuffling Array Elements

Shuffling of array elements along by one location is required for both insertion and deletion in sorted arrays. Shuffle right to create a gap for a new insertion, and shuffle left to close a gap after deletion. We can also use this idea for unsorted arrays, but there are faster tricks, as examined later in this section.

In practice, shuffling of sorted arrays is quite efficient for scalar types via a memory block copy, using the `memmove()` standard function. Note that `memmove()` is an older function that does a bytewise copy of the memory that ignores object constructors and move operators. Presumably, the standard `insert()` method is using fast byte copies for scalar types.

Here's an obscure pitfall: we cannot use various other copying methods because the shuffle involves overlapping source and destination memory blocks. There does not seem to be a version of C++ copying that permits overlaps. These functions would be incorrect and lead to undefined behavior on overlapping memory blocks, which is definitely true of any array shuffle:

- `std::memcpy` (old C-style)
- `std::copy_n`

However, we can use the overloads of the `std::move` function that work on ranges of multiple objects. These version of `std::move` have a real runtime cost, unlike the basic version, which is a compile-time type-cast that converts to a movable R-value reference (with no runtime code generated). We also need to pay attention to whether we are shuffling to the left or right, because these functions don't work for all overlapping arguments.

- `std::move` or `std::copy` — moving or copying left (i.e., close a gap for deletion).
- `std::move_backward` or `std::copy_backward` — for moving or copying right (i.e., create a gap for insertion).

Note that using the `std::copy` or `std::copy_backward` functions also work here, but the copying operation is slower than moving for non-scalar types. Hence, the `std::move` versions are more general, but still have some downsides:

- Expensive for non-scalar objects.
- Iterators are invalidated on the array.
- Invalidates any pointers or references to specific objects.

Unfortunately, the shuffle cost is terrible for complex objects that will require their move operators called for every single object. I can't say that I recommended sorted arrays for those types. Note that there are also various types of objects where we could still use a memory block move to do a "shallow move" of the objects (i.e., "relocatable objects"), rather than individually moving each element. However, using this idea requires tricks to prevent the C++ container from doing its move thing, such as using a low-level raw array rather than `std::vector`.

Binary-Like Sorted Array Insertion

Sorted arrays are logarithmic for searches, but not quite as good for insertions and deletions. Inserting a new element into a sorted array is a three-phase algorithm:

1. Find the location to insert,

2. Shuffle elements to the right (create a gap), and

3. Insert the new element at the location.

There are three ways to find the location in a sorted array:

1. Linear search from the front.

2. Linear search from the back.

3. Binary-like search (faster!)

Linear search over a sorted array doesn't use equality, but finds the first element the bigger than the new element. Or to go in reverse, start at the end and look for the first element that's smaller than the new one.

The advantage of starting at the end is that we can shuffle as we go, but it'll have terrible cache locality problems in accessing memory addresses in reverse. CPU memory prefetch algorithms usually assume a forward access order.

Anyway, neither of the linear algorithms are fast and they aren't typically used. Instead, binary-like search for the insertion point is much faster, with a logarithmic complexity.

Binary-like search for insertion involves splitting up the array into two intervals, and choosing between the two based on the midpoint value. This is not exactly the same as binary search, because we're assuming that the element is not already in the array. Hence, it's like binary search, but we're looking for smaller versus bigger elements in comparison to the new element, rather than seeking equality.

You can code your own binary insertion algorithm, or the standard C++ library has two functions to help:

- `std::lower_bound()`
- `std::upper_bound()`

These functions are general methods on many containers, but they require the underlying data to be sorted, or "partitioned" is the official term. It means the same as "sorted" for everyone except polymaths who like Abelian groups.

Hopefully, this is coded in the standard library via a binary-like search method, and is therefore fast. It should have logarithmic complexity. However, if we follow it up with an `insert()` call, then that's an array shuffle that's likely to be linear in cost.

Note that there's no equivalent "binary deletion" algorithm when we're deleting from a sorted array. That just uses normal binary search to find the element, such as `std::binary_search`, if it's there, and then we can remove it. Insertion is different to deletion in that sense.

Sorted Array Deletion

Deletion of an element in a sorted array is easier than insertion. There are two major phases:

1. Find the element using binary search.

2. Shuffle the elements left to close the gap.

Note that we're using real binary search, not the binary-like search for insertion, because we assume the element is present. We can't delete an element that's not in the array. Hence, we can use `std::binary_search` to find the element.

The deletion phase is a left shuffle of all the array elements. As discussed above, we can do a byte copy such as `memmmove()` or `std::move`, which both are well-defined with overlapping memory blocks.

These methods can be efficient for scalar and other trivial types where bitwise shallow copying is allowed, but may trigger a cascade of move constructors or move assignments on complex classes. Thus, sorted arrays can be potentially inefficient for non-scalars because of the hidden costs of shuffling objects.

Batched Multiple Insertions in Sorted Arrays

The optimization here is that we can perform two or more sorted array insertions faster if we do them together. There are several possibilities to consider:

1. Adjacent sequences — if we have two or more items that "fit" between two elements of the array, we can insert them in a block, and do only one shuffle.

2. Sorted sequence — if we have a sorted list of two or more new elements to insert, we can insert them by doing a single scan of a merge sort (i.e., merging two sorted arrays into one longer array).

These ideas are certainly more efficient than naive repeated insertions, but they are special cases. However, looking at those efficiency gains, we get the inspirational idea of a more general way to handle a sorted array with lots of insertions:

1. Defer insertions by storing to-be-inserted data in a separate location.

2. Batch insert all of the new data later (e.g., every 100 insertions).

For example, we could store the first 100 to-be-inserted elements in a separate array of 100 elements. Or reserve an extra 100 elements at the end of the main vector.

However, correctness before efficiency. Our to-be-inserted array elements are supposedly already inserted into our sorted array, so they should be found by search, and should be able to be deleted, too. Hence, both search and deletion algorithms will need to look in two locations, which gets complex and bug-prone, not to mention less efficient.

Even assuming we've fixed those bugs, the overall efficiency of this batched-insertion method is actually not that great. We've overlooked the practical problem that the batched insertion step needs the 100 extra elements to be in sorted order before the merge. So, we'd have to run `std::sort()` on the new array of 100 extra elements, before merging them into the main sorted array.

Alternatively, we could maintain our 100 elements as a sorted array, but then your boss might notice this oddity that could be hard to explain on a whiteboard: maintaining insertions into a sorted array to optimize insertions into a sorted array.

I'm not sure how much we've actually improved things? My brain is about to explode figuring it out, but feel free to talk amongst yourselves. I'm going to analyze deletions instead.

Batched Multiple Deletions in Sorted Arrays

The deletion of an element in a sorted array is a "find-and-destroy" sequence that is quite inefficient. The finding of the element is fast using binary search, with a logarithmic cost. However, the shuffle required to remove an array element in the middle of the array has linear cost.

If we're doing a lot of deletions, the cost is significant from a lot of shuffling. If we're inserting and later deleting *n* elements into a sorted array, and each deletion is linear, then it's quadratic in complexity.

Can we do better?

Yes, and there are multiple ways to do so. Some of the options include:

- Deleting a range of elements
- Deferred deletions

The simplest idea is to delete two or more array elements at once. This reduces multiple deletions to a single shuffle operation. However, it's not always possible to know that our many deletions will be in a subrange or even pairs of adjacent elements.

The more general case of multiple random deletions can be optimized via deferred deletion algorithms. The idea of deferred deletions is to track multiple deletions, possibly in some other ancillary data structure, and then finalize the deletions all at once.

Deferred Deletions with Extra Data Structure

The idea of optimizing a large number of deletions is to group multiple deletions and then perform them together. For example, if we track the locations to be deleted, and can then detect adjacent pairs of elements (or more), then these subarrays can be deleted together. This is efficient because we shuffle once per deleted subarray rather than once per deleted array element.

Here's an idea: store the array indices for deferred deletion. The approach involves:

1. Store indices of to-be-deleted elements in a secondary data structure, and

2. Later, we scan this data structure to look for adjacent indices, which can be deleted together, which is faster.

In addition to the gain of processing merged pairs or longer subarrays of elements, the shuffle can also be optimized to move smaller chunks around, by removing the "gaps" where the to-be-deleted elements are located.

This approach is workable, and fortunately there's only two problems:

1. Bugs, and

2. Slugs.

The downsides of this approach of storing the indices of the to-be-deleted elements include some major potential bugs:

- Insertions will mess up the indices in the secondary data structure.
- Searches will still find the to-be-deleted items.

Fixing these problems is not easy, and certainly not efficient. And even if we only had multiple deletions in a row, this approach is also not especially efficient in general, with both space and time overhead:

- The space overhead of the secondary data structure.
- Extra time cost of updating a secondary data structure (and destroying it).
- The algorithm to find adjacent indices is inefficient (e.g., sort the indices).

The other major problem with this approach of using an extra secondary data structure containing deletion offsets is simply: it's unnecessary. There's a better way.

Deferred Deletion & Vector Defragmentation

There's a simple way to handle deferred deletions in a sorted array, and it doesn't even require an extra data structure. The basic strategy looks like:

1. Mark each deleted element as "to-be-deleted" (later).

2. Ignore all to-be-deleted elements (e.g., when searching).

3. Remove all the to-be-deleted elements together (using vector defrag).

How do you mark an array element for deletion? There are two basic strategies:

1. Add a new Boolean flag in the array, or

2. Special values

Extra Boolean Flag Method. Obviously, if our array elements are large objects, then we could just add another `bool` data member to mark its status. But if our array elements are small, such as an array of integers or timestamps or other scalars, then adding an extra data field is inefficient in terms of both space and time.

Extra space cost will be at least a byte per array element, and likely more due to alignment considerations. Larger array elements also reduce cache locality and will impact speed.

Special Values Method. The idea of using a special value is to re-use an existing data field in the array, rather than adding to its size. Some common special values to consider include:

- `0`
- `-1`
- `nullptr`
- Negatives

Usually, we will just use a single, fixed special value such as `0` or `-1`. However, if we want to mark the data, but still be able to know what the original data value was at that location, one way is to negate it (reversibly). The slow way to do this is to multiply by `-1`, whereas the faster way is to toggle the sign bit.

No matter what method is used, the key point is that we are able to look at an array element and decide whether it's valid or a to-be-deleted array element. We'll need to use that function in all of the other array operations.

Searching and Insertion with Deferred Deletions. Note that it's quite tricky to ignore the to-be-deleted elements during search and insertion. The binary search algorithm may still "find" the number in a to-be-deleted array element, in which case you need to check adjacent array elements, as there may be a non-deleted array element with the same value.

Sorted array insertion with some to-be-deleted elements should still work. The sorting of the array key should be maintained across both valid and already-deleted elements. Inserting a new element will change all the array indices, but note that we're not tracking these indices for our deferred deletion algorithm, so this shuffling from insertion doesn't cause problems with the deletions done later.

An interesting wrinkle in this method occurs when an insertion matches the location of a to-be-deleted element via the binary insertion method. In this case, the new element can simply replace the to-be-deleted item, and no shuffle is needed, leading to a very efficient insertion. Furthermore, even if not, our shuffle for insertion can be shorter, as it only needs to shuffle the elements until the first to-be-deleted item is found (i.e., only shuffle up to a "gap" in the array).

Searches and insertions are not the only code to modify. You also need to ignore the to-be-deleted elements in any other array operations, such as printing the array elements or other linear scan. It's actually quite error-prone to have to remember to handle already-deleted elements in every other operation. Easy to leave an insidious bug this way!

Vector Defragmentation. The final stage of this deferred deletion algorithm is to clean up the array to remove all of the to-be-deleted array elements. Until this is done, the array could be wasting a significant amount of space.

The idea of vector defragmentation is to scan the entire array and compact all the valid array elements together. This is accomplished via a simple two-pointer algorithm. At the end, we need to resize our full array down to the reduced number of stored elements.

Phew! That was a lot of special cases to handle for our delayed deletion algorithm. Hope it's worth it!

Many Searches, Insertions & Deletions

The general case is a sorted array that's undergoing a large volume of searches, insertions, and deletions. A sorted array is not necessarily the best data structure for that, but the assumption is that we need a sorted array for some other reason, such as fast scanning of the entire array through its contiguous memory.

Searching is not the problem. The binary search algorithm is very efficient with logarithmic complexity in both average and worst-case cost.

Insertions and deletions in a sorted array are much worse, since both involve a "shuffle" that is linear in cost. In both cases, the main optimization to consider is a deferred algorithm, where multiple insertions and deletions can be delayed, and then performed as a group. Overall, this deferred batching idea doesn't seem to work very well for insertions, but works extremely well for deletions.

In practice, the shuffle is not that bad, because it's just a big memory block copy using `memcpy()` or `memmove()` or similar functions. Thus, if the array elements are a scalar, or any other similar "plain old data" object type that doesn't require a move constructor or move assignment operator, then it's not really *O(n)* complexity to do insertion or deletion in a sorted array. Hence, the benefits of using deferred deletion with vector defragmentation may not be as great as they seem.

Extensions

1. Benchmark the sorted array implementation with a raw array versus using `std::vector` as the internal data array, especially to see if our hand-coded binary search is fast or not.
2. Explore the use of "shallow copying" on sorted arrays containing "relocatable objects" in the shuffle needed for insertions and deletions in a sorted array data structure.
3. Explore the efficiency of calls to move constructors in a "shuffle" for a sorted array implemented using `std::vector` or `std::array`.
4. Implement the binary-like search algorithm to find the insertion location in a sorted array. (Note that deletion is just the normal binary search to find the element.)
5. Benchmark inserting into an unsorted array and then sorting using `std::sort`, versus incrementally maintaining a sorted array. Do the results differ for a scalar integer type versus arrays of an object like `std::string` (which has move operators)?
6. Implement a hybrid binary-linear search where the binary search reverts to linear search once the interval is small enough.

7. Implement an AVX SIMD version of linear search over integers that tests a number of integers in the array at once.
8. Implement a "cache-aware" binary search that chooses the middle index at the start of a cache line (where possible), and tests all values in that cache line immediately using an unrolled linear search.
9. Implement a binary search that is both cache-aware and uses AVX SIMD instructions to test all elements in the same cache line more efficiently.
10. Implement a sorted array with deferred insertions and deletions.
11. Is there a better way to optimize insertions into a sorted array via batched insertions or deferred insertions (in the general case)? What about if we exclude searches and deletions, so that it's only a sequence of many random insertions? Maybe we can build some other data structure with better insertion complexity, such as a red-black tree (`std::map`), and then linearize it into the array with a tree traversal at the end.

24. Order of Insertion

Whenever you hear the words "order of insertion" in a set of requirements, it should be associated with certain ideas. Note that this is exactly the same as First-In-First-Out (FIFO), which means that any type of queue is good at this:

- Linked list queue — `std::queue` container.
- Doubly-linked list queue — `std::deque` container.
- Array queue or dequeue — a ring buffer.

However, order-of-insertion is not necessarily a queue data structure. If the requirements include insertion or deletion in the middle of the sequence, then it's not really a queue (nor even a dequeue).

These types of requirements that combine order-of-insertion traversal along with generalized insertions and deletions can arise in several practical contexts:

- Least-Recently-Used (LRU) cache.
- Operating system paging algorithms.
- Order book updates (trading engine).
- Rate limiting (throttling) of requests.

These all have a time element that causes them to have queue-like need for insertion-ordering. However, there needs to also be key-based searches, insertions and deletions, so a basic queue is not adequate.

Hash Table with Order-of-Insertion

As an example, let's consider a dream list of requirements for such a data structure:

1. Fast search, insert and deletion, and

2. Traversal in order-of-insertion.

To get to the first three, with fast search, insertion, and deletion, you should immediately think: hash tables.

Hash tables have average case O(1) complexity for search, insertion and deletions. Admittedly, hash table can degrade to linear complexity in the worst case. Furthermore, hash tables have a poor traversal cost generally, and totally fail at maintaining any order in the traversal. We can't maintain "order of insertion" with just a hash table.

Hence, to implement traversal in the insertion order we need another data structure. The first idea is to have two totally distinct containers, and search them both when we're doing our operations. A better idea is that in our hash table nodes, we can insert a pointer to some other node in another data structure, so that we don't need to do two lookups.

Two options come to mind:

- Array or vector — contiguous data with good cache locality.
- Doubly-linked list — non-contiguous linked data structure.

Let's look at each of these options.

Contiguous Array Version

The idea is to maintain traversal in the order of insertion by maintaining the items in a separate std::vector or std::array container. For example, you could maintain an array of pointers to the hashed nodes in the array. And each hash node would need either a pointer back to the array or an index offset of where the element is found in the array.

The use of an array or vector makes the traversal of items super-fast, by scanning the array, in contiguous memory locations. Okay, so actually the cache locality isn't that great, since scanning the pointers in the array has good locality, but then it's jumping via the pointers to the nodes in the hash table, which are in different places in memory.

It's easy to maintain order-of-insertion in the array, simply by always inserting at the end. Our array or vector data structure has a count of how many elements are in the array, and we can insert a new item at the end.

Problems arise with deletion, however. If the need for deletion was only to remove an item from a fixed-size array to make room for the next one, then we could address this by using a ring buffer implemented as an array (i.e., a fixed-size queue in an array).

However, if we want to remove arbitrary items from our hash table, and hence from our array, the use of a contiguous array causes difficulties. The difficulty is not in finding the location for removal, but at the end of this sequence:

1. Search the hash table for the key.
2. Find the pointer or index into the array in the hash node.
3. Remove the node from the hash table container.
4. Remove the pointer from the array or vector container.

However, once we try to remove the entry from the array, there's a gap. There are three possible approaches:

1. Mark the item as "deleted" (i.e., leave a gap).
2. Shuffle the array elements down.
3. Move the end array element down into the gap ("swap and pop").

None of these solutions are great. They all lead to suboptimal complexity in one or other of the methods.

Marking each item with a "deleted" flag works fine on deletion, but the insertion-order scan has to skip extra unused elements. There are a few ways to mark the elements:

- Boolean flag inside each element.
- Separate array of Boolean flags.
- Packed bit vector representing the Boolean flags.

Furthermore, with the marking-as-deleted method, the array will fill up, and need to have its gaps removed eventually. This is a costly type of "garbage collection" or "memory reclamation" algorithm that will have linear complexity. And until it's cleaned up, the method will waste extra memory space for all the deleted gaps.

Shuffling all of the elements down to fill the gap does maintain the correct order in the array. However, it's an O(n) operation and will also invalidate all the pointers into the array from other non-removed elements in our hash table. So, we'd need some way of finding all those elements (e.g., reverse pointers), and also the cost for updating them all.

Finally, the “move end element down” array trick is an O(1) method to cover our gap, and would only require updating one non-removed hash node, which is also O(1). Admittedly, the need to store reverse pointers from the array back to the hash nodes adds O(n) more space. However, it fails completely, because the array is no longer sorted in order of insertion.

Is there a way to salvage the dream of maintaining a contiguous array that is sorted by insertion order? There are some tricks to try, like permutation arrays, but I can’t see a good solution.

Doubly-Linked List Version

A more natural solution is to thread a doubly-linked list through our hash nodes. The advantages of a doubly-linked list are:

1. No fixed size limits.

2. Easier deletion with O(1) complexity.

3. Maintains order-of-insertion naturally.

Note that the linked list has to be doubly-linked so that deletion is easy once we find a node to remove. If it’s only a singly-linked list, then we cannot find the element before the current node, so we can’t easily unlink the current node.

The doubly-linked list method is not without downsides. There are problems with time and space:

- Extra space for previous and next pointers in each node.
- Non-contiguous memory usage for scanning (it’s a linked list!)

To implement the interleaved doubly-linked list, each node in our hash table needs to have “next” and “previous” pointers. We also need to track the head and tail of this list at the container level.

The idea is that a scan in order of insertion is just to run down the doubly-linked list in one direction. Hence, when we insert a new item it has to be inserted at the end of the list.

The reason that this method is better than an array or vector is that it's easy to remove in a linked data structure. There's no "gap" when we remove an item from a linked list. We just update the pointers to the adjacent list elements to point around the removed list node.

Could we use a separate doubly-linked list, such as the `std::list` container, rather than manually threading pointers through our hash table? Yes, but this wouldn't really avoid the space cost of storing "next" and "previous" pointers in each hash node, but just move them elsewhere. Additionally, we'd need a pointer to the list node in the doubly-linked list stored in the hash nodes. And each insertion would need two separate memory allocations for the hash nodes and linked list nodes. Hence, threading our doubly-linked list through the nodes themselves seems more efficient overall.

25. LRU Cache Data Structure

What is an LRU Cache?

Least-Recently-Used (LRU) caches are a common requirement in low-latency programming. There are several important applications of an LRU cache:

- Operating system paging algorithms
- Memory access caches (low-level)
- Order book updates in trading

The idea of an LRU cache is to maintain a cache of recently used data, such as memory we've just accessed, or a piece of data we've just updated. But we don't want an unlimited size data structure, so when it gets full, we evict the data that was "least recently used" (i.e., the oldest data).

Note that an LRU cache is a more specific type of cache that just mapping keys to the values they were set to. The operations we need to support include:

- Add a new key to the cache (with its corresponding value).
- Update a key when it gets re-used again (more recently).
- Remove the least-recently-used item in the cache (to make room for insertions).

Sounds like a queue? No, it's not!

Not a Queue or Deque

An LRU cache has features that sound like a queue with FIFO ordering. We want to evict the oldest items from the cache, which sounds exactly like maintaining a queue of elements, and deleting from the tail of the queue will remove the oldest element.

These features are very queue-like and maintain a FIFO-like order-of-insertion:

- Add a new item to the end of the queue (the newest item).
- Remove from the front (to evict the oldest item).

The feature that's not like a queue occurs on the "update" of a key that's already in there, which occurs if a cached item is then accessed a second time. This requires two problematic operations:

- Search — find the item already in our LRU cache, and
- Deletion — remove the item from the middle of the queue.

It's starting to sound less-and-less like a queue. There's no fast searching method for `std::queue` and `std::deque`, and we'd have to use a linear scan.

Deletion is also a problem. We need to move an item from the middle of the queue back to the head of the queue. This is not like a standard queue, which only allow deletions from the end. A standard dequeue container also allows deletions from the front, but this doesn't help us.

Hence, we can't just use a queue or dequeue, but need something fancier as our implementation of an LRU cache.

Overall, an LRU cache has similar requirements to the general case earlier: fast searches, insertions, and deletions. We also need to maintain order-of-insertion for cache evictions, but we need to remove arbitrary nodes from that sequence, so a standard queue or dequeue won't work.

Note that, unlike the general case, we don't actually need to traverse the sequence in order, but only use it for evictions.

Nevertheless, the basic idea of an LRU cache implementation is similar to the general case of a data structure that maintains ordering by insertion sequence:

- Hash table for fast searches, insertions, and deletions.
- Maintain order-of-insertion sorting via an array, vector, or linked list.

Adding a new node into the cache is simply an insertion into the hash table, and adding it to the head of the array or list. This item is the "most recently used" so it will now be the last to be evicted from the cache.

If our cache is full, adding a new node means removing the oldest. It's easy to remove the "least recently used" by removing it from the hash table, and removing the end element from the list (effectively, like a queue).

We could seemingly implement this queue-like functionality with two possible approaches:

- Statically with a fixed-sized array (i.e., a ring buffer wraparound), or
- Dynamically via a linked list.

Only one of these ideas will work!

Array Implementation Fails

Let's consider a contiguous array implementation first, which would be desirable for cache locality efficiency. In other words, we use a hash table for searching, insertion and deletion, but also maintain a separate array or vector data structure to track insertion order.

In practice, we'd need to use a wrap-around of elements in a ring buffer structure, implemented via an array or vector container.

This is workable for many of the LRU cache requirements. Search and insertion is very fast in the hash table. We don't actually search the array, which is fortunate, and inserting into an array with order-of-insertion is just adding it to the end (fast!).

However, deletion is a problem. We run into a significant efficiency problem arises when we need to update a cache item that's already in the cache from a prior access: Every update of a value already in the cache needs to do two things to the array:

(a) delete the node in its previous place in the array, and

(b) re-insert the node at the head (it's now the most-recently used item).

The key point is that the "previous place" for an item could be anywhere in the array or ring buffer. So, we need arbitrary deletions at any location. For the reasons discussed in the general case, an array or vector that implements a ring buffer or a fixed-size array will fail in this situation.

Removing an item from the middle of the array is problematic and needs an inefficient shuffle method to fill the gap, followed by trying to update pointers to all the array elements that were moved by the shuffle. Alternatively, moving the array's end element down to cover the gap fails because it completely messes up the order of elements in the array.

A ring buffer implemented in an array or vector is no better at handling random deletions. Removing from the middle of a wraparound sequence in a ring buffer is actually the exact same situation, except rotated, and has the same problems.

One solution is to not allow cache updates. If an item is already in the cache, we could simply *not* update its position in the sequence. However, this is no longer an LRU cache, but more like a Least-Recently-Loaded (LRL) cache, or really a FIFO queue version of a cache.

The requirements for an LRU cache are somewhat different to a FIFO queue. For example, all frequently-used items will get evicted from the cache in a fixed order, getting no benefit over infrequent accesses. The efficiency of the cache does not adapt to access patterns. Overall, it seems that a contiguous data structure is not effective for an LRU cache.

Linked lists to the rescue!

Doubly-Linked List LRU Cache

Fortunately, an LRU cache is also fast to implement with a hash table and doubly-linked list. Note that a singly-linked list fails to provide efficient deletion, so we have to double up. Hence, the basic idea is:

- Hash table — good at efficient search, insertion and deletion (but without ordering).
- Doubly-linked list — maintains data according to order-of-insertion.

There are two ways to implement our doubly-linked list:

- Second container — using the standard `std::list` container separately (it's doubly-linked).
- Threaded intrusively — use a doubly-linked list that is threaded through the hash table nodes.

The first solution is workable if we maintain a pointer or iterator into the linked list from our hash table nodes. We could make our list contain copies of the whole keys (if small), or pointers to the hash table nodes if the keys are a complex object (i.e., don't copy it). But overall, the two container approach is inefficient because we're doubling the number of allocated nodes by doing memory allocation once in the hash table, and again in the `std::list` container.

A better solution is to intrusively thread our own hand-coded doubly-linked list through our hash table nodes. This requires extra space for "next" and "previous" pointers in our hash table nodes, but doesn't require a second memory allocation, and also maintains only one copy of the keys.

Let's run with that idea and examine the efficiency of the operations:

- Search — use the hash table to get O(1) average search cost (we don't search the linked list).
- Insertion — fast O(1) insertion into the hash table, and also O(1) insertion at the end of the doubly-linked list.
- Deletion — fast (O1) deletion from the hash table, and also O(1) deletion in the middle of a doubly-linked list (hooray!).
- Traversal (insertion-ordered) — linear scan of the linked list (easy).

The linked list needs to be doubly-linked because deletion from the middle of a singly-linked list is problematic. Efficient deletion from the middle of a singly-linked list needs to go backwards to find the previous node, which doesn't work with one-way pointers.

Deletion from the middle of a doubly-linked list is easy by resetting two pointers, in the node prior to us, and the node afterwards. This is fiddly but has only O(1) complexity, with just a few pointer operations. Unlike the array version, there's no "shuffling" or other hidden costs, so deletion is also fast, and maintains the order-of-insertion requirement.

The deletion algorithm for doubly-linked lists is fiddly with some edge cases, but not that difficult. Once the list node to remove is found, we need to update the pointers in both the previous and the next node on the list. We also need to handle special cases like when the array is empty, or has only one element, or when deletion is at the head or tail of the array.

References

1. Geeks for Geeks, 27 Dec, 2024, *LRU Cache - Complete Tutorial*, https://www.geeksforgeeks.org/lru-cache-implementation/
2. Shaila Nasrin, Jan 18, 2025, *LRU Cache Implementation in C++*, https://medium.com/learn-coding-concepts-with-shaila/lru-cache-implementation-in-c-8a52f259206f
3. CPP Scripts, May 2025 (accessed), *C++ LRU Cache: Mastering Efficiency with Ease*, https://cppscripts.com/cpp-lru-cache
4. Peter Goldsborough, May 2025 (accessed), *lru-cache: A feature complete LRU cache implementation in C++*, https://github.com/goldsborough/lru-cache
5. Tim Day, 2012, *LRU cache implementation in C++*, https://timday.bitbucket.io/lru.html

26. Fast Ring Buffers

What is a Ring Buffer?

A ring buffer is an array-like data structure where the data moves around in a “ring” so that the end wraps around to the beginning. It’s also known as a “circular buffer” and is often what is meant when people talk about a “fixed-size queue.”

A ring buffer is stored in a single array or vector of contiguous data, but is not accessed in the same idiom. The data is processed in a FIFO (First-In-First-Out) idiom, where items are added to the “tail” of the queue, and removed from the “head” for processing.

Hence, a ring buffer is a good choice of data structure for implementing a fixed-size queue or dequeue (double-ended queue).

Some of the main design decisions when implementing a ring buffer involve error handling:

- Overflow — inserting into a full buffer
- Underflow — removing from an empty buffer

Should the ring buffer throw an exception, or just return a Boolean failure status to the caller?

Simple Ring Buffer

A basic ring buffer data structure has three main elements:

- Array or vector of objects (fixed-size)
- Head index (integer)
- Tail index (integer)

Here's some code using `std::array` for a ring buffer:

```
template<typename T, int sz>
class RingBuffer {
private:
    std::array<T, sz> arr;  // Fixed-size array
    int head;
    int tail;
    // ....
};
```

New objects are inserted at the tail, and retrieved for processing from the head. In a typical implementation, the progression goes from left to write, using a "+1" idea for the next location. Technically, the ring buffer data could be handled in reverse order, but the forward progression around the ring is simpler and allows marginally more efficient arithmetic because there are no negatives to handle.

Thus, the basic primitives needed by a ring buffer:

- Insert at the tail
- Remove at the head

Here's the basic insertion method:

```
bool push(const T& x) {
    int newtail = (tail + 1) % sz;
    if (newtail == head) {
        // Overflow (full)
        return false;
    }
    tail = newtail;
    arr[tail] = x;
    return true;  // success
}
```

And here's the "top" method for an interface that allows "top" to access, and "pop" to remove:

```
T top() {
    if (is_empty()) {
        // Underflow
        return T(0);
    }
    return arr[head];
}
```

The “pop” method actually removes the item from the ring buffer:

```
void pop() { // Just remove (no return)
    if (is_empty()) {
        // Throw exception? (optional)
        return;
    }
    else {
        head = (head + 1) % sz;
    }
}
```

And there are also various simple primitives:

- Capacity — the fixed-size of buffer.
- Empty — zero elements
- Full — fixed-size array is full.

The code is reasonably simple:

```
int capacity() const { return sz; }
bool is_empty() const { return head == tail; }
bool is_full() const { return (tail+1) % sz == head; }
```

Pros and Cons of Ring Buffers

The main advantage of a ring buffer is that it has contiguous data. This means that our fixed-size queue should be faster to access than one stored as a linked list using `std::queue`.

The main disadvantage of a ring buffer is that it has a fixed size, unlike `std::queue`, which grows dynamically. This ring buffer size doesn’t necessarily need to be known at compile-time, but does need to be set when you initialize the ring buffer. There are also more advanced types of ring buffers which use multiple arrays, which can be dynamically grown in size.

The other disadvantages are that the ring buffer is very specific to a FIFO access pattern. It’s not a fast data structure for these operations:

- Searching for a value
- Sorting data
- Inserting at a random location (rather than the tail)
- Deleting from a random location (rather than the head)

Insertions and deletions are slow because they require a "shuffle" of all objects. Note that there's an interesting wrinkle: we could make insertion and deletions fast if we don't mind violating the FIFO ordering and moving objects around (invalidating any pointers or iterators referencing them). The idea is that the ring buffer becomes like an unsorted array (with wraparound):

- Fast random insertion — move the current element at the insertion location to a free location at the end of the ring buffer, then insert.
- Fast random deletion — move the last element to the location we are deleting from.

It's not all bad news. The data in a ring buffer is mostly stored contiguously, so there are some operations that still have good cache locality properties:

- Scanning or visiting all data elements
- Random access of data by integer index

A linear scan of all the elements can be quite fast, provided you don't mind that it's unsorted (or rather, it's sorted by order-of-insertion). The data elements are always in one or two contiguous data blocks, which is better than dispersed data structures like linked lists or binary trees. However, it's not quite as fast as an array or vector scan of objects, which is always one contiguous block.

Accessing one of the objects via an integer ordinal is still quite fast (i.e., 0...`n-1`). Mainly, it's just some integer arithmetic with head and tail to find its array offset in the ring buffer.

Incremental Count Optimization

Computing the count of how many elements are currently inside the ring buffer is somewhat tricky: In the above computations, we can compute the "count" of how many elements are in the buffer using arithmetic on head and tail indices.

```
int count() const {
    return (tail >= head) ? tail - head
                          : sz - (head - tail);
}
```

An alternative that can be faster, if the `count()` method is called often, is to maintain an incremental count, and store it in the ring buffer.

The idea is pretty simple:

- Insertions — `count++` (except if full)
- Deletions — `count--` (except if empty)
- Count — just return the `count` variable.

Hence, the computations during insertion and deletion are only a single integer increment or decrement, and the `count()` function becomes a simple getter of an integer data member. In addition, the availability of a "count" variable actually allows some optimizations to some of the other methods:

- `empty()` — test `count==0`
- `full()` — test `count==capacity`

These are much faster than the earlier versions using head and tail index arithmetic. Hence, these efficiency gains may override the extra costs from incrementally computing the count during object insertions and removals.

Avoiding Three Integers

If we use an incremental count optimization for the number of items in the ring buffer, we end up with three integer values:

- Head
- Tail
- Count

It turns out that we don't need all three, because they are inter-related numbers. We can calculate the "tail" variable from the "head" and the "count" value.

```
    tail = (head + count) %sz;
```

There are actually some other numbers that are also related, which we could also use. For example, the total number of insertions and deletions of objects is related to the head and tail values, and the count is simply the difference between them.

Alternative Variable Pairs. It turns out that a ring buffer can be defined by any two variables from a set of several related calculations.

Some of the possible pairs include:

- Head and tail
- Head and count
- Tail and count

Note that there are two main implementations of the initialization of head and tail values. These yield implementations that differ by one in all calculations, so you have to consistently choose between them:

- `head = tail = 0`
- `head = 1, tail = 0`

The meanings of head and tail differ slightly in these two variants. Hence, the inter-relationship with the count is also different by one. Care must be taken to avoid off-by-one errors!

Combining Two Variables. The optimization ideas above reduced our three variables (head, tail, and count) down to two variables. Any pair of them will do, since they are inter-related.

But what about reducing it to one variable? Having only one integer variable in our ring buffer might be desirable because:

- Efficient single arithmetic operations.
- One integer value as an atomic for lock-free versions.

Can it be done?

The key point to note is that we really do need two distinct values. However, we can put them together into a single integer with encoding and packing ideas. For example, we could store the head as 16 bits and the count as 16 bits, and put both in a 32-bit unsigned integer.

Note that this limits the capacity of the ring buffer to 2^16 which is 65,536. We could also pack them into a 64-bit `unsigned long` if we needed more capacity.

Modulo Arithmetic Optimizations

The `%` operator for modulo arithmetic (or remainders) is one of the slowest operations in C++. The typical code we want to optimize in a ring buffer or fixed-size queue uses this idiom:

```
head = (head + 1) % N;
```

Modulo arithmetic is based on division, which is also slow, even on integers. Hence, our ring buffer can be improved by getting rid of the percent!

How? There are several options:

- Bitwise arithmetic
- Type casts
- Ternary operator
- Branchless coding
- Unsigned arithmetic

Bitwise-and trick. Firstly, if we choose the buffer size `N`, to be a power-of-two, then we can use bitwise arithmetic. A remainder of a power-of-two is the bitwise-and of the number one less. These are equivalent:

```
head = (head + 1) % 16;    // Modulo
head = (head + 1) & 15;    // Bitwise-and
```

Validating power-of-two. One thing you might want is a safety net to ensure nobody uses the ring buffer for a size that's not a power-of-two. We want this:

```
static_assert(is_power_of_two(N)); // How?
```

We can use the Kernighan bit trick:

```
static_assert( (N & (N-1)) == 0);  // Kernighan
```

How does this work?

It's just magic, and let's forget about it.

No, actually, the Kernighan trick is that "`N&(N-1)`" clears the value of the rightmost bit of a number. Hence, if the number without the rightmost bit equals zero, then there's only one bit set in the number. And the set of numbers with only one bit set: powers of two.

Note that lots of parentheses are necessary around the bitwise operator to avoid an operator precedence glitch. Also note that the Kernigan trick fails with a false positive if `N` is zero or negative, so we should add some more safety checks at compile-time:

```
    static_assert(N > 0);
```

Type casts. The use of bitwise-and is limited to powers of two, which is annoying, but there's an even more specific way to do this for some of them: type casts. If we can choose the size as 256 (8-bits) or 65,536 (16=bits), we can do this:

```
    head = (unsigned char)(head + 1);    // 8-bits
    head = (unsigned short)(head + 1);   // 16-bits
```

Note that type casts are often effectively free after C++ does its optimization thing. The register allocation algorithm can just choose to use a value in a different way, and propagate that forward to other arithmetic. Thus, a type cast operation may result in zero runtime instructions.

Ternary operator. But why are we using arithmetic in general, when there's actually only one case where we want to reset the value. Another way is to use the ternary operator instead of arithmetic. The calculation becomes:

```
     head = (head + 1 == N) ? 0 : head + 1;
```

We can also implement this logic in two instructions, which is worth a try:

```
     head++;
     if (head == N) head = 0;
```

Or if you like short-circuiting operators, you can do this:

```
     (++head) == N && (head = 0);
```

The compiler probably treats that the same, but you never know, and you might want to check the assembly output (e.g., using "`gcc -S`").

Branchless coding tricks. Another trick is to notice that we just want to zero the value in one specific case. Hence, we can use the branchless coding trick of using logical operators as 0 or 1 integers. The goal of branchless coding is to remove all control flow branches, so that the CPU's branch prediction logic can run fast. Note that the ternary operator is actually like an `if` statement, and it has two branches. The branchless version with only fixed arithmetic is:

```
    head = (head + 1) * (head + 1 != N);  // Branchless
```

The way this works is to multiply the value by 0 or 1, depending on the logical test. Again, we can also try this as two statements:

```
    head++;
    head *= (head != N);  // Branchless
```

Note that I doubt the branchless versions are very efficient, because they've added a multiplication operation. The ternary operator version is likely better, and isn't that bad despite its branches, if you look at the assembly. Most compilers will convert it to a single CMOV (conditional move) CPU instruction, which makes it effectively branchless, too.

Unsigned arithmetic. One final trick is to note that we have modulo arithmetic for free in the CPU: unsigned integer arithmetic. Overflow of unsigned integers is not an exception in C++ and when you think about it, implements the exact semantics of modulo arithmetic. Hence, here's the idea:

```
    unsigned char head;
    ...
    head++;
```

It works! And there's not a single percent operator anywhere! All this time and we had cheap modulo arithmetic hiding in plain sight.

We really need to time this, because it isn't 100% guaranteed as faster code. A lot of the uses of `head` will involve converting it from `unsigned char` to an integer offset, such as for array indexing in the vector of objects that makes up the ring buffer. A variation of this idea would be to store the head and tail as integers or unsigned integers, so that they can be used as the fastest type of normal integer, but still use unsigned arithmetic overflow tricks for modulo arithmetic.

This is the idea for an N=256 size ring buffer:

```
    int head;
    ....
    ((unsigned char*)&head)++;
```

This relies on the platform being "little endian" with the lowest-order byte stored on the left, which is true in most modern CPUs (but not if you're sending integers over the network in "network byte order"). And, yes, you got me, I really should use `reinterpret_cast` here rather than the old C-style type cast.

Obviously, these tricks of using `head` and `tail` as unsigned integers only work for a limited set of sizes:

- N=256 — `unsigned char` (8-bits)
- N=65,536 — `unsigned short` (16-bits)
- N=4.7 billion — `unsigned int` (32-bits)

We can even do decrement and negative calculations this way, since underflow is also not an exception, whereas the `%` operator and negatives don't talk to each other at parties.

Move Semantics

If our ring buffer contains complex objects, there are many more considerations for making it efficient. One of the biggest inefficiencies in a ring buffer class is inserting and deleting any non-trivial objects. If we do it wrong, we're calling copy assignment operators and copy constructors to make new objects in the array, and running the destructor when we release an object.

Move semantics to the rescue!

The first point to note is that it doesn't matter for simple data types in our ring buffer. Any scalar values like integers or floating-point numbers don't have any copy constructors or destructors to worry about. In fact, this is also true of simple structures and classes, so long as they are "plain-old data" or POD data types.

But anything more complicated than this will have costly calls to copy constructors and copy assignment operators.

To optimize this, we need to talk about:

- Move constructor and move assignment operator
- R-value references
- Copy elision
- Return Value Optimization (RVO)

In practice, the problems arise in both our "push" and "top" versions. The "pop" routine causes a copy assignment operator invocation:

```
bool push(const T& x) {
    // ....
    arr[tail] = x;  // Copy assignment
    return true;  // success
}
```

And the "top" member has the problem of returning an object type, which will use a copy constructor call at the `return` statement.

```
T top() {
    // ...
    return arr[head]; // Copy constructor
}
```

The automatic compiler optimization of "copy elision" might help improve the performance of the "top" method. Returning an object is exactly the situation it's meant for. However, we can use move semantics explicitly to ensure it's improved:

```
bool pop_top_move(T& outobj) {
    if (is_empty()) { return false;    }
    ct_incremental--;
    int oldhead = head;
    head = (head + 1) % sz;
    outobj = std::move(arr[oldhead]); // Move assign
    return true;  // success
}
```

Note that `std::move()` is a compile-time type-cast here, without any runtime cost. And it's required to convert to an R-value reference, as otherwise the assignment statement would still call a copy assignment operator.

Constructor Problems

One of the performance problems with our ring buffer implementation is that `std::array` calls the constructor for every object whenever a new ring buffer object is defined or created. This occurs with this use of `std::array` for our ring buffer:

```
std::array<T, sz> arr;  // Fixed-size array
```

How to avoid these constructor calls? After all, our ring buffer is supposedly empty with zero objects initially. Some of the solutions that don't work and will still call constructors:

- Raw arrays
- Pointer to std::array

Using a raw array like this will still call all the constructors when our ring buffer is created:

```
T arr[sz];
```

Similarly, we could use an allocated copy of `std::array`, since it's really an object not an array. It works like this:

```
std::array<typename T,sz> * arrptr;
....
arrptr = new std::array<T,sz>;  // in constructor
```

This allocates our big array in the constructor rather than as a non-allocated data member. This adds an extra inefficiency from the extra allocated block, and doesn't work anyway. The `new` operator will still run all the individual object constructors.

What about using `std::vector` instead?

Standard Vector Problems

Using `std::vector` can be better than `std::array`, because it delays both its memory allocation and its construction of objects:

```
std::vector arr<T>;
```

Unfortunately, I'm not a big fan of this approach, because it has other difficulties:

- Extra memory allocation call (inefficient).
- Bounds checking failures in debug libraries.

The first point is that `resize()` has the same problem with too many constructor calls. Doing this in the constructor will still call all the constructors:

```
arr.resize(sz);  // Constructors!
```

So, we can call `reserve()` instead of `resize()`. That won't call constructors:

```
std::vector arr<T>;
// ....
arr.reserve(sz);  // No constructors!
```

This has hopefully allocated the memory for all the objects, without running their constructors. But this can run into various problems when we try to use the vector elements. The problem is on this type of statement in our `push` method:

```
arr[tail] = x;
```

And the same problem still occurs with our code that gets items out of the ring buffer. Note that the issue is not move semantics, because this has the same issue:

```
outobj = std::move(arr[oldhead]); // Move assignment
```

The issue is bounds checking on the `[]` operator for `std::vector`. In theory, the `reserve()` function has allocated valid memory for enough objects. However, the `size()` function is still zero, so the runtime bounds checking will trigger on any debug run of the code.

Yes, maybe some platforms this will work, with no bounds checking. But you can run into portability problems. For example, it makes the code fail with spurious runtime errors on any type of "hardened" standard C++ library.

Explicit Destructor Calls

Another problem with our ring buffer implementation when instantiated with class types is destructor calls. Instead of too many constructor calls, we have too few destructor calls. The problems include:

- Destructor calls missed after move assignments (e.g., popping).
- Destructor calls on destroying the whole ring buffer.

One solution: don't bother. If the object that's used in a ring buffer doesn't have important destructor actions after a move (and it shouldn't), or if destroying the whole ring buffer is in the shutdown sequence of the application, then you can maybe just forget about this problem.

Another solution is to explicitly call the destructor ourselves. You can call the destructor of a class like any other member function using the `~T()` syntax. For example, in the `pop` function, we can do:

```
    arr[head].~T();  // Explicit destructor
```

Basic types don't need destructor calls, so we ideally want to distinguish trivial types from fancy class objects. We can also use type traits to do this, which are wonderfully efficient compile-time operators that work during instantiation of the template. Here's how it works:

```
    if (!std::is_trivially_destructible<T>::value) {
        arr[head].~T();  // Explicit destructor
    }
```

The alternative is to note that trivial types have no-op destructors, and the compiler would remove them anyway. Hence, the above type trait test may be unnecessary, but it's a fast compile-time test anyway, so either way is fine.

Note that we are assuming here that the class being used has a destructor that works properly after an object has been moved away. In other words, it doesn't do something silly like assuming a pointer in the object is non-null.

The move assignment operator also needs to properly clear all the non-trivial data members, such as pointers, to zero or null values, so that the destructor doesn't access bad memory after a move.

Class Interface Bypass

There are a couple ways to bypass the class interfaces, and thereby avoid the inefficiencies of construction and destruction. This makes the caller of our ring buffer manage when the objects are created and destroyed. The main ways are:

- Blocking non-trivial types
- Raw character buffer arrays
- Pointers to objects

Trivial types only. We can make our ring buffer, or other home-grown containers, faster simply by disallowing their use with complex objects. We can efficiently trigger compiler warnings with the type trails, so that users of the template know to only use scalars or other POD types. Here's some examples using the various different settings:

```
static_assert(std::is_pod<T>::value); // Plain-Old Data
static_assert(std::is_trivial<T>::value); // Trivial
```

Raw character-array memory buffers. The idea is to use a character array as a raw buffer, rather than `std::array` or `std::vector`, for our container class (e.g., our ring buffer). To bypass class constructions by using raw memory buffers, we have choices like:

```
char arr[sizeof(T) * sz];  // Static data member
char *arr = new char[sizeof(T)*sz]; // Dynamic alloc
```

This raw byte idea is workable, but every use of the array has to involve index calculations and type casts to object-type pointers. It's fiddly and annoying, but it's faster, because it avoids constructor calls, and doesn't need all the extra messing around to avoid `std::vector` bounds checking. There are also concerns with:

- Uninitialized bytes in the buffer
- Alignment of addresses

We really should also initialize the bytes in our array buffer to all nulls in the constructor using `memset` on the whole array. To do this, we also need to make sure that all the classes using the ring buffer have properties like:

- All-bytes-null is a stable but invalid initial status of the object.
- Destructor doesn't fail on an all-bytes-null object.

We also need to manually take care of alignment of the addresses, since the compiler thinks we only have characters, which don't have alignment issues. There's the `alignas` standard specifier and various non-standard implementations for older language versions.

If we're really careful, maybe the initialization is not needed and we can leave out the `memset` call in the constructor. There's some new "uninitialized memory" primitives coming in C++26 that may also help to do so. You can maybe avoid needing the null byte initialization, but I'm betting against you when I run `valgrind` on your code.

Pointers. As much as I admire the design of move semantics, there is a simpler way to avoid the overhead of objects moving in and out of our ring buffer. Old-school coding still works: store pointers to the objects in the ring buffer instead of full objects. The upside is avoidance of object copying and moving overhead.

The downside of pointers is the extra level of indirection, and double hit to memory with poor cache locality because of that. And pointers have a few pitfalls with a bad reputation as being unsafe, but I'm sure you've heard that before.

Extensions

1. Implement a reverse ring buffer that uses decremented indices for head and tail, rather than addition, so that it grows from right-to-left instead of left-to-write.
2. Implement a dequeue in a ring buffer by adding "insert-at-head" and "remove-from-tail" operations for the ring buffer (rather than the normal insert-at-tail and remove-from-head idiom). The trick is we'll need to subtract one from indices and go in reverse.
3. Implement a ring buffer with initialization of "head=1" and "tail=0" (rather than "head=tail=0"). All calculations will differ by one, such as the "empty" calculations is not "head==tail" anymore.
4. Implement a ring buffer using two full-size integers that count the number of insertions and deletions. Note: the relationship between head and tail versus insertions and deletions is not that difficult!

27. Perfect Hashing

What is Perfect Hashing?

Perfect hashing is the extreme of hashing, where we guarantee that there's no collisions. Hence, the hash function is "perfect" because no pair of two keys map to the same hash value. This makes for a super-fast hash lookup with guaranteed *O(1)* search performance, and no need to look up a second hash location ever.

Perfect hashing is faster than normal hash tables. Regular hashing is fast on average, with *O(1)* average search, but collision resolution mechanisms like linear chaining or probing can have worst case *O(n)* search cost. Perfect hashing has guaranteed *O(1)* search complexity for best, average, and worst case. In fact, we don't even code up a collision resolution method at all.

Unfortunately, the good news stops there, because this only works in a very special situation: where the set of keys is known at compile-time. This hash table can only contain a fixed set of keys that we know whenever we build the perfect hashing code.

If there are any insertions or deletions, this idea doesn't work at all, and may require us to re-run and re-compile our perfect hashing engine if they occur. Thus, we can tolerate insertions and deletions but only if they are *rare*. Some examples of rarely changing sets of strings we might want to look up with perfect hashing include:

- Special keywords in a programming language tokenizer (e.g., 100 reserved words).
- Common English words in a grammar checker (e.g., 1,000 basic words).
- Stock tickers on an exchange's market data feed (e.g., about 5,000).
- Vocabulary words of an AI model (often 50,000 to 100,000 words).

Yes, the last one is a bit tricky, because tickers might change daily, in which case we might need to re-run our perfect hashing in every overnight build. Also, finding a perfect hash function for 100,000 LLM vocabulary strings in a reasonable amount of time might be a struggle.

Disadvantages of Perfect Hashing

We already mentioned the main disadvantage of perfect hashing, which is that it requires a known set of keys, or at least a very rarely changing set of keys. Other disadvantages include:

- Cost to build — expensive to scan the search space to find a perfect hash map.
- Scalability problems — cannot handle a large number of keys because the search space becomes too large.
- Static data — insertions and deletions invalidate the hash map.
- Recomputations — increasing the key set requires a total re-run of the whole shemozzle.

Perfect hashing also has some of the disadvantages of a basic hash map like `std::unordered_map`, such as:

- Unsorted data
- Scanning all data is somewhat inefficient (and in unsorted order)
- Cache locality issues because objects are stored randomly in the hash table.

Perfect hashing is not perfect for every case. Some alternatives data structures to consider for search lookup optimization include:

- Bloom filters
- Tries
- Automata (precomputed)

Or you could just put all your keys in an array and use a GPU to check them all in parallel.

Perfect Hash Functions

Special hashing algorithms can be used in any situation where the search data is known at compile-time. The most efficient solution is to use hashing with a specially developed hash function, designed to prevent all collisions. This is called a *perfect hash function* and can only be developed for unchanging data. If a perfect hash function can be found, the symbol table can be searched with one computation of the hash function and one key comparison to determine if the key is actually there at the index.

The most difficult aspect of using this method is the search for a perfect hash function for a particular set of data. There are a few common methods of doing so:

- Inspired guesswork
- Brute-force computation
- Use a perfect hashing tool (e.g., GNU gperf)

In some cases, the programmer can work out a function that has no collisions by guessing at a function. For example, if the programmer notices that all keys have a different first letter then it is easy to compute a perfect hash function as a mapping from the 26 letters to a different unique integer, the hash value. There's a curious fact unknown to most AI engineers, that humans are very resourceful and this method of "guessing" the function works surprisingly well.

The brute-force approach involves trying to generate the hash function using a computer which tries a number of different hash functions of a particular meta-pattern, applies the hash function to each key, and report when no collisions occur.

Further Optimizations of Perfect Hashing

The general complexity of perfect hashing is *O(1)*, which is true of the best case, average, and worst case complexity. Hence, it's fast for large sizes, but we still might want to optimize it a little more! There are two places to try to speed up:

- Lookup function (online)
- Perfect hash function creation (offline)

The basic method of perfect hashing can be optimized so that lookup is even faster. Some of the ways that we might super-optimize the search phase include:

- Not checking the key is present.
- Using a power-of-two hash table size.
- Larger hash table size.

Avoiding string comparisons. The sequence for a perfect hash lookup:

1. Calculate the perfect hash function.

2. Find that location in the hash table.

3. Compare the string at that location with our search key.

But why are we doing this string comparison at the end? That's quite slow. Well, sometimes we don't need to, and it depends on context. For a grammar checker or LLM tokenizer, we need to detect whether or not the key is there, because multiple words could map to the same hash location.

On the other hand, a market data feed from a US stock exchange might only contain our set of ticket names, so we can *assume* that only one string could possibly be at the hash table location. In other words, we're assuming that every string is found, and there are zero failed searches, so our hash table is mapping of the string to a set of data structures (e.g., our order book for that stock). That's all fine, and it will go faster, but the code will break completely if the exchange adds a new stock ticker!

Another way we could avoid the string comparison is to use two or more perfect hash functions. This data structure is known as a Bloom filter, and combines multiple bit vectors with multiple hash functions. Bloom filters are a probabilistic data structure that can confirm 100% that a key is invalid, but can only confirm that a key is likely to be valid, but not with 100% certainty.

Power-of-two hash table size. The size of the array that is our hash table is one main parameter for a perfect hash function, so we have some control over it. Note this basic point: the hash table size must be more than the number of keys, or else it's a little hard to avoid collisions! In fact, it's easier to find a perfect hash function if the size is significantly more than the number of keys, so that there are some empty slots.

But what size? For some reason lost in the mists of time, everyone wants to choose a prime number, preferably a Mersenne prime, because that supposedly makes hash maps more evenly spread. But in the case of perfect hashing, we are looking for exact mappings with zero collisions, so it's perhaps not so important to use a prime.

Instead, we should use a power-of-two hash table size, because that allows the arithmetic in our perfect hash function to be faster. The reason is that most perfect hash functions look like this:

```
    offset = some_big_number(key) % N;
```

The `%` remainder operator is extremely slow, even on integers. The only reason it is used here is to ensure that the hash function maps to between `0` and `N-1`, where `N` is the hash table size.

We can use "strength reduction" to use a faster arithmetic operation, such as:

- Bitwise-and operator — if `N` is a power-of-two (e.g., for `x%16` do `x&15`).
- Type cast to `unsigned char` — if `N` is 256 (8 bits)
- Type cast to `unsigned short` — if `N` is 65,536 (16 bits).
- Overflow of `unsigned char` — if `N` is 256 (8 bits)
- Overflow of `unsigned short` — if `N` is 65,536 (16 bits).

We've already examined a lot of these optimizations to modulo arithmetic in detail for the discussion of ring buffers in Chapter 21.

Larger hash table size. An important point about hash table sizes is that bigger can be better. This is true for both the offline computation of the perfect hash function, and the online search lookup. Bigger hash tables have more "gaps" and are an easier search space to find a solution. In terms of online search performance, a bigger table worsens cache performance, but that's not likely to be great for a hash table anyway. Furthermore, these extra gaps also mean that unsuccessful searches will be faster on average, because those keys that map to a gap can avoid the string comparison at the end. And memory is cheap, after all.

Offline search optimizations. The search for a perfect hash function can be very expensive, and even impossible. Some of the ways to speed things up include:

- All of the hash function optimizations.
- Splitting up the search space (partitioning).

The first point is that any optimization to the perfect hash function computation applies a thousand-fold to the offline search. For example, we also get faster computations possible in the offline search for a hash function if we only look at power-of-two table sizes. In fact, our offline code does a lot more of those computations.

Search space partitioning optimizations. The search space is combinatorial and explodes with large key sets. One approach is to split the keys into multiple perfect hash tables, such as by partitioning the key sets. Some of the ways to consider partitioning include:

- First letter — we can use 26 different perfect hash tables.
- Two letters — this gives 26*26=676 separate hash tables.
- Length of keys — e.g., stock tickers are at most 5 letters long.
- Preliminary hash — a simple hash function to start with (e.g., first two letters modulo a size smaller than 676).

Note that this means running the perfect hash engine multiple times to find a different perfect hash function for each partitioned set of keys. However, running 26 searches for smaller sets of keys will often run faster overall than trying to find one super-perfect hash function for every single key.

Example: ANSI C Keywords

As an example of the various approaches, let us attempt to develop a perfect hash function for a set of C's 32 keywords for a programming language tool:

```
auto break case char
const continue default do
double else enum extern
float for goto if
int long register return
short signed sizeof static
struct switch typedef union
unsigned void volatile while
```

Using my own version of "inspired guesswork", involving a couple of hours of poring over ASCII tables, I managed to come up with a reasonable perfect hash function. The basic approach I took was to break up the words into groups of about five keys by using a test of the string length, and also by making single character comparisons on the larger groups of keys with the same length. Once the group was small enough I looked for letters in the keys that were unique, often the first or second letter, and then examined the ASCII binary values of these letters. This way, the hash function extracts certain bits from each letter, and generates a small integer, which is then mapped into an "interval" of values for that particular group. The function, which produces hash values in the range 0..36, is as follows:

```
int my_hash(char* s)
{
    switch (strlen(s)) {
    case 2: // Only "if" and "do"
        return (s[0] & 01) + 2; // 2..3
    case 3:
        return (s[0] & 01) + 8; // 8..9
    case 4:
        if (s[1] == 'o') // goto, long, void
            return (s[0] & 03) + 26; // 26..29
        else // auto, case, char, else, enum
            return ((s[1] & 14) >> 1) + 30;
    case 5: // break, const, float, short, union, while
            // First letter is unique
            return (s[0] & 07)+(s[0] == 'c') + 10; // 10..16
    case 6:
            if (s[0] == 's') // signed,sizeof,static,struct
                return (s[5] & 03) + ((s[5] & 8) >> 3)
                    + ((s[5] & 16) >> 2) + 18; // 18..22
            else // Letter not 's' - double, return, extern
                return (s[0] & 03) + 23; // 22..24
    case 7: // "typedef", "default"
            return (s[0] & 16) != 0;
    case 8: // continue, register, unsigned,volatile
            // First letter is unique
            return ((s[0]&04) >> 1)+(s[0] & 01) + 4; // 4..7
    default: // Can't be a C keyword
            return 0; // Pick any number
    }
}
```

The second approach is to make the computer perform a brute-force search for a perfect hash function. The following program takes a set of keys from a file and develops a hash function of the following form:

```
( Σ C[i] * key[i] ) mod N
```

The code attempts brute-force computations with many combinations of the constants C[i] and N. If one of these hash functions produces no collisions, a perfect hash function has been found. The source code below implements this concept.

```
//------------------------------------------------------------
// PERFECT HASH FUNCTION BRUTE-FORCE SEARCH
//------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

//------------------------------------------------------------
#define LEN 10 // Maximum length of a word

//------------------------------------------------------------
char words[MAX][LEN]; // words being hashed
int C[LEN]; // coefficients of hash function

//------------------------------------------------------------
#define MAX_MULTIPLIER 1 // Let C[i] range 0..MAX_MULTIPLIER
// 0 means skip, 1 --> use addition
#define MAX_MODULUS 1000
int G_MODULUS;
int G_MODULUS_START_MULTIPLIER = 5;
int G_MODULUS_TOP;

//------------------------------------------------------------
```

```
// Apply the hash function coefficients to a key
//------------------------------------------------------------
int compute_hash_perfect(char* s, int modulus)
{
    unsigned int hash = 0;
    for (int i = 0; i < LEN && s[i] != 0; i++) {
            hash += s[i] * C[i];
    }
    return hash % modulus;
}

//------------------------------------------------------------
// Try all the combinations of coefficients
// This function finds the perfect hash function!
//------------------------------------------------------------
void perfect_hash_find_best(int nwords, int nstart)
{
    bool done = false;
    bool flags[MAX_MODULUS]; // has a key hashed here yet?
    int modulus = nstart * G_MODULUS_START_MULTIPLIER;
    do {
        // Do one possible modulus (table size)
        for (int i = 0; i < LEN; i++) C[i] = 0;  // Clear
        do {
            // Update C[i] coefficients for next attempt
            C[0]++;
            for (int i = 0; i < LEN; i++) {
                if (C[i] <= MAX_MULTIPLIER) break;
                C[i] = 0;
                if (i + 1 < LEN) { C[i + 1]++; }
            }

            memset(&flags, 0, sizeof flags);

            // Scan all strings to count collisions...
            bool collision = false;
            for (int num = 0; num < nwords; num++) {
                int val = compute_hash_perfect(
                              words[num], modulus);
                if (flags[val]) {
                    collision = true;
                    break;
                }
                flags[val] = true;
            }
            if (!collision) { // report success!!
                printf("NO COLLISION: ");
                for (int i = 0; i < LEN; i++) {
                    printf("%2d ", C[i]);
                }
                printf(", MODULUS = %d ", modulus);
                if (modulus == nstart)
                   printf(" PERFECT!!! (n=%d)",(int)nstart);
```

```
                    printf("\n");
                    break; // exit do loop. Do next MODULUS
                }
                // Finish only when all multipliers
                // are up to MAX_MULTIPLIER
                done = true;
                for (int i = 0; i < LEN; i++) {
                    if (C[i] < MAX_MULTIPLIER) {
                        done = false;
                        break;
                    }
                }
            } while (!done);
            if (done) {
                printf("FAILED with MODULUS %d\n", modulus);
            }
            modulus--; // Try the next modulus value
        } while (modulus >= nstart);
    }
```

As shown in the source code above, the program is set to find all hash functions where the coefficient is either 0 or 1. These functions are a useful special case, as no multiplications are actually needed (all the characters with a 1 coefficient are simply added). When the program is run as shown on the ANSI C keywords as inputs, the best hash function it produces has modulus 134 (i.e., hash table size 134) and the following coefficients:

```
NO COLLISION:  1  0  1  1  1  1  1  0  0  0 , MODULUS = 134
```

This information can be coded up into a simple perfect hash function. Unfortunately, the memset and strncpy calls are necessary to ensure that characters beyond the end of the string are considered zero, as is assumed by the hash function generator.

```
//--------------------------------------------------
// Computer-generated hash function for C keywords
//--------------------------------------------------
int computer_hash(char* s)
{
    char s2[10];
    memset(s2, 0, 7);  // zero the first 7 letters
    strncpy(s2, s, 7); // copy up to 7 letters
    return ((int)s[0] + (int)s[2] + (int)s[3]
          + (int)s[4] + (int)s[5] + (int)s[6]) % 134;
}
```

This is not a *minimal* perfect hash function for these 32 keys. If the records to be stored with these keys are quite large, the space wastage of 134 hash table entries may be too large. A simple method of overcoming this is to add an array of 134 small integers (i.e., using the char type), where each entry in this array sets each C keyword to a unique value in the range 0..31. On the other hand, this may be a de-optimization as a sparse hash table can be more efficient than a minimal perfect hash function. If the table is large, it becomes likely that an unsuccessful search will map to a location containing a null pointer entry, and this avoids the need for the key comparison.

Perfect Final Thoughts

These computations we found here are not *minimal* perfect hash functions. If the stars align, you can sometimes find a mapping that works with the hash table size exactly equal to the number of keys. It might take a lot of CPU juice to find one, though. Good luck with that!

All of the hash functions in this section (both human and computer-generated) have multiple limitations, such as:

- ASCII-specific — not portable to the EBCDIC set or other character sets.
- Little endian — I haven't checked portability to big endian machines.

Finally, if you'd rather use a tool for perfect hashing than have as much fun as I just did, you can use the GNU gperf tool, which is a perfect hash function generator. GNU gperf will output the perfect hash function in C++ for you, and is highly customizable.

Extensions

1. Generalize the perfect hash functions to use parallel arithmetic in the hash function computation, such as AVX or ARM Neon SIMD instructions on a CPU or GPU kernel calculations.
2. Parallelize the search for a perfect hash function on either a CPU (e.g., AVX or ARM Neon functions) or on a GPU (e.g., in CUDA C++).
3. Implement multiple perfect hash functions on the same set of keys to get a Bloom filter data structure, where the string comparison can be omitted during lookup.
4. Try out the GNU gperf tool for one of the data sets.

28. Matrix Multiplication

Matrix-Vector Multiplication

Matrix multiplication by a vector gives another vector. Let us consider the simple case first, where the matrix is square with dimensions *NxN* and the vector is also of size *N*. The matrix has *N* rows and *N* columns, and the input vector has *N* elements. The resulting output vector will also have *N* elements. Conceptually, in pseudocode:

```
    MAT[N][N] * VIN[N] -> VOUT[N]
```

It's not immediately obvious, or at least, I don't remember my High School Math teacher mentioning it, but matrix-vector multiplication is a bunch of vector dot product computations. We need to do a vector dot product for each of the elements of the output vector. Each element is a dot product of a matrix row times the input vector. Note that the dimensions match for a dot product, with *N* matrix rows and *N* elements in the input vector.

Rectangular matrices. The general case of a rectangular matrix multiplied by a vector is a little trickier, but not a lot. If our matrix is *MxN* and the vector is size *N*, then the output vector has size *M*. Note the two of the dimensions must match: the columns of the matrix and the elements of the input vector are both *N*. However, this dimension *N* "disappears" and the output vector has size only dependent on *M*. The pseudocode:

```
    MAT[M][N] * VIN[N] -> VOUT[M]
```

The rectangular matrix-vector multiplication is almost identical to square matrix-vector computations. Each element of the output vector is a dot product of a matrix row with the input vector. Again, we note that the dimensions of the matrix rows (*N*) must match the size of the input vector (*N*), or else we cannot compute it. I mean, we *could* still compute it with mismatched dimensions, such as by assuming that the shorter one (matrix row or input vector) had zeros in the missing elements, but that sounds a little buggy.

Complexity of Matrix-Vector Multiplication. The algorithmic complexity of matrix-vector multiplication is quadratic in *N*, whereas matrix-matrix multiplication is cubic in *N*. The basic matrix-vector multiplication scans *N* rows of the matrix, with each row element performing a computation against each of the *N* elements in the vector, giving two nested loops with an overall *O(N^2)* cost.

Memory layout: One important point for the efficiency of matrix-vector multiplication is that the default memory layout has contiguous addresses for both the matrix row and the vector. Obviously, a vector is just a sequence of memory with all the elements in series. Not so obviously, a row of a matrix, when stored as a C++ two-dimensional array, is also a contiguous set of data (i.e., a matrix row is like a vector). Hence, the dot product multiplication of a matrix row and the input vector is simply scanning forward along contiguous addresses for both of its inputs, which makes it easy to vectorize.

Optimizing Matrix-Vector Multiplication

The version of matrix-vector multiplication with row-wise vector dot products needs three parameters, because it outputs to another separate destination vector.

```
void aussie_matmul_vector_basic_out1(const ymatrix m,
          const float v[], int n, float vout[])
{   // Basic matrix-by-vector using vector dot product
    for (int i = 0; i < n; i++) {
      const float* rowvector = &m[i][0];
      float sum = aussie_vecdot_basic(rowvector, v,n);
      vout[i] = sum;
    }
}
```

Nested Loop Matrix-Vector Version: The same matrix-vector multiplication algorithm in the form of two nested loops is below. This is flattening the call to the lower-level vector dot product function and putting its inner summation loop directly inside the outer main loop. The basic C++ code looks like:

```
void aussie_matmul_vector_basic_out2(const ymatrix m,
      const float v[], int n, float vout[])
{   // Basic matrix-by-vector using nested loops..
    for (int row = 0; row < n; row++) {
        float sum = 0.0f;
        for (int col = 0; col < n; col++) {
            sum += (m[row][col] * v[col]);
        }
        vout[row] = sum;
    }
}
```

Optimizations of matrix-vector multiplication. Various ways to optimize the naive nested loop matrix-vector multiplication suggest themselves:

- Hoisting loop-invariant code (loop code motion) of the "`m[row]`" expression.
- Loop pointer arithmetic for both loops.
- Loop unrolling of the inner loop to unroll 4, 8 or more iterations.
- Loop tiling to unroll a 2x2 tile/block.
- Vectorization using the AVX1/AVX2 vector dot product versions we already examined.

I tried coding several more of these optimizations and here are the benchmarks:

```
Matrix-Vector (MatMulVec) benchmarks (N=2048, ITER=300):
Matrix-vector nested loops: 3480 ticks (3.48 seconds)
Matrix-vector nested loops hoisted: 3489 ticks (3.49 sec)
Matrix-vector nested ptr-arith: 3415 ticks (3.42 seconds)
Matrix-vector unrolled inner (4): 1166 ticks (1.17 seconds)
Matrix-vector unrolled inner (8): 938 ticks (0.94 seconds)
Matrix-vector nested tiled 2x2: 1995 ticks (2.00 seconds)
Matrix-vector vecdot AVX1 DP: 1414 ticks (1.41 seconds)
Matrix-vector vecdot AVX2 FMA: 929 ticks (0.93 seconds)
```

Interestingly, code hoisting and loop pointer arithmetic were a waste of effort. Loop tiling did better than the original, but probably its speedup is primarily from the effect of loop unrolling rather than data locality or cache hit rates, since simpler loop unrolling did better. Note that the AVX1 version used the "dot product" intrinsic but AVX-2 used the FMA intrinsic. Simple loop unrolling also did as well as AVX2 hardware vectorization, probably because the versions of AVX1 and AVX2 were simply calling the vector dot product functions, so they still had function call overhead. Hence, this algorithm can be further optimized by inlining to fix the AVX function call overhead, combining AVX intrinsics with unrolling for the inner loop, and then some minor final tweaks such as pointer arithmetic.

Tiled Matrix-Vector Multiplication

A more detailed analysis of the matrix-vector algorithm shows that it is not optimal in at least three areas:

- Data locality
- Pipelining AVX intrinsic arithmetic
- Redundant loads

The data locality of the 2x2 tiled version is better, but more improvement is possible, starting with the use of AVX intrinsics inside the "sub-kernel" for the tile. The AVX instruction sequences of "load, calculate, store" in the earlier non-tiled AVX-optimized versions are not allowing for the natural instruction pipelining of the AVX intrinsics to calculate multiple sums or FMA operations with near-parallel pipelining. And the entire input vector is getting re-loaded repeatedly for every row in the matrix. So, we need to examine improvements on three aspects.

A tiled sub-kernel is the main way to fix data locality and pipelining. Improving data locality is somewhat inherent to tiling. The pipelining can be improved by unrolling the tiled sub-kernel and reordering the loads and stores so they don't block the arithmetic of AVX intrinsics.

Can we avoid redundant vector loads? Since it's unavoidable to access every element of every row at least once, the redundant loads of the vector suggest that we should modify the algorithm so as to work on a subsection of the vector for each of the matrix rows. This suggests an inversion of the main nested loops of the algorithm. However, that runs into the major problem that it destroys cache locality, by scanning down the column of the first matrix. I benchmarked this loop interchange idea, and it actually increased execution time. Maybe we should use the transpose of the first matrix, so that it's in column-major order when scanning its columns? No, that's actually just going back to the original algorithm without the loop interchange.

Anyway, a better plan seems to be to reduce the redundant loading by using temporary calculations inside the tile sub-kernel. Here is what a basic tiled/blocked algorithm using 2x2 tiles looks like in basic sequential C++:

```
void aussie_matmul_vector_tiled_2x2_better(const ymatrix m,
        const float v[], int n, float vout[])
{   // Tiled/blocked matrix-by-vector using 2x2 tiling
    aussie_assert(n % 2 == 0);
    for (int row = 0; row < n; row += 2) {
        vout[row] = 0.0f;
        vout[row + 1] = 0.0f;
        for (int col = 0; col < n; col += 2) {
          vout[row] +=
              (m[row][col]*v[col]) // row+0,col+0
            + (m[row][col+1] * v[col+1]) // row+0, col+1
              ;
          vout[row+1] +=
              (m[row+1][col]*v[col]) // row+1, col+0
            + (m[row+1][col+1] * v[col+1])  // row+1, col+1
              ;
        }
    }
}
```

One minor improvement would be to use memset to clear the whole output vector to zero, rather than individual assignments, which I added to the 4x4 tiled version. There is another minor improvement is removing the "common sub-expressions" of v[col] and v[col+1] and I tried this with no improvement noted in the 2x2 tiled version, but about 10% improvement in the 4x4 tiled version. The computations of m[row] and m[row+1], etc., can also be hoisted out of the inner loop, giving another 10% gain for the 4x4 tiled version. The C++ code for the 4x4 tiled version with a fully unrolled 4x4 sub-kernel now looks like:

```
void aussie_matmul_vector_tiled_4x4_CSE2(
    const ymatrix m, const float v[], int n, float vout[])
{   // Tiled/blocked matrix-by-vector using 4x4 tiling
    aussie_assert(n % 4 == 0);
    memset(vout, 0, sizeof(float) * n);
    for (int row = 0; row < n; row += 4) {
        const float* rowvec = &m[row][0];
        const float* rowvec1 = &m[row + 1][0];
        const float* rowvec2 = &m[row + 2][0];
        const float* rowvec3 = &m[row + 3][0];
        for (int col = 0; col < n; col += 4) {
            float fcol0 = v[col];
            float fcol1 = v[col + 1];
            float fcol2 = v[col + 2];
            float fcol3 = v[col + 3];
            vout[row] +=
              (rowvec[col] * fcol0) // row+0, col + 0
              + (rowvec[col + 1] * fcol1) // row+0, col + 1
              + (rowvec[col + 2] * fcol2) // row+0, col + 2
              + (rowvec[col + 3] * fcol3) // row+0, col + 3
              ;
            vout[row + 1] +=
              (rowvec1[col] * fcol0) // row+1, col + 0
              + (rowvec1[col + 1] * fcol1) // row+1, col + 1
              + (rowvec1[col + 2] * fcol2) // row+1, col + 2
              + (rowvec1[col + 3] * fcol3) // row+1, col + 3
              ;
            vout[row + 2] +=
              (rowvec2[col] * fcol0) // row+2, col + 0
              + (rowvec2[col + 1] * fcol1) // row+2, col + 1
              + (rowvec2[col + 2] * fcol2) // row+2, col + 2
              + (rowvec2[col + 3] * fcol3) // row+2, col + 3
              ;
            vout[row + 3] +=
              (rowvec3[col] * fcol0) // row+3, col + 0
              + (rowvec3[col + 1] * fcol1) // row+3, col + 1
              + (rowvec3[col + 2] * fcol2) // row+3, col + 2
              + (rowvec3[col + 3] * fcol3) // row+3, col + 3
              ;
        }
    }
}
```

Matrix-Matrix Multiplication

Now let's look at matrix-matrix multiplication, whereas above we looked at matrix-vector multiplication. The proper MatMul and GEMM kernels are coded for full matrix-matrix multiplication.

Matrix multiplication results in another matrix as the output. For the simple case of two square matrices of the same size, the resulting output matrix is also of the same dimensions. In pseudocode:

```
    M1[N][N] * M2[N][N] -> MOUT[N][N]
```

For multiplying two rectangular matrices, or sizes *MxN* and *NxP*, we get an output matrix of size *MxP* (i.e., the inner *N* dimensions disappear). In pseudocode style:

```
    M1[M][N] * M2[N][P] -> MOUT[M][P]
```

Note that *P=1* is the case of matrix-vector multiplication, because an *Nx1* matrix is actually a vector with *N* rows of a single element (i.e., one column).

Algorithmic Complexity. The naive implementation of a matrix-matrix multiplication via three nested loops is a cubic algorithm, with *O(N^3)* complexity. The well-known Strassen algorithm has complexity about *O(N^2.7)*, which looks like such a massive improvement. Other algorithms such as the Coppersmith-Winograd algorithm and numerous sub-variants have better asymptotic complexity, but with a high constant overhead, making them impracticable for anything but very large values of *N*.

Basic Matrix-Matrix Multiplication. The basic naive algorithm for matrix multiplication is three nested loops. There is nothing fancy here: this is just coding up the basic matrix multiplication method that you forgot the second you finished your Senior math exam.

If you don't believe me, check it out on Wikipedia.

Here's the C++ code:

```
void aussie_matmul_matrix_basic(const ymatrix m1,
          const ymatrix m2, int n, ymatrix mout)
{
    // Matrix-Matrix mult basic naive n^3 algorithm
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            float sum = 0.0f;
            for (int k = 0; k < n; k++) {
                sum += (m1[row][k] * m2[k][col]);
            }
            mout[row][col] = sum;
        }
    }
}
```

The two outer loops are scanning the rows of the first matrix, and the columns in the second matrix. The innermost of the three loops is doing a vector dot product computation over the "`k`" index variable. However, it's not a normal vector-vector dot product. Instead, it's the dot product of one "horizontal" vector, which is a row of the first matrix, and of a second "vertical" vector, which is a column of the second matrix. Hence, the number of rows in the first matrix must equal the columns of the second matrix, which is true here because we're assuming that both matrices are square. Hence, the "`k`" variable is spinning down the `n` elements of a row and a column at the same time. Every element of the *NxN* output matrix requires a vector dot product calculation like this.

Vectorization. None of these matrix multiplication algorithms are especially good, because they are all *sequential*, rather than parallel algorithms. Neither the naive cubic version nor the Strassen algorithm are what we need. What we need for GPUs and CPU SIMD intrinsics are vectorizable algorithms for matrix-matrix multiplication. Unfortunately, the above simple triple-nested matrix multiplication algorithm is *not* one of them, because non-contiguous storage of the second matrix hampers vectorization.

Memory layout problems for matrix-matrix multiplication: The layout in memory for matrix-matrix multiplications is not as fortuitous as it was for matrix-vector multiplications. Each computation in matrix-matrix multiplication is a vector dot product of a row of the first matrix with a column of the second matrix. Each row of the first matrix is happily stored in contiguous memory, but the columns in the second matrix are not. In fact, the "stride" between two elements of a column of a matrix is a very large number of bytes in the default memory layout.

The default storage of matrices and two-dimensional arrays in C++ is called "row-major" storage layout. Row-major storage has each row in contiguous memory. The rows are stored one at a time, top to bottom, and adjacent elements in a row are also adjacent memory addresses. Columns are a second-class citizen in row-major layout, and you have to jump around to find adjacent elements of a column vector.

The alternative storage method is "column-major" storage layout where the columns are stored in contiguous memory, and it's the rows that are in the smoker's carriage at the back of the train. However, column-major is not the default C++ storage mode.

Hence, to vectorize a matrix-matrix multiplication, we want to keep the first matrix in row-major storage, but we need to rearrange the storage of the second matrix to be column-major storage, rather than the default row-major storage. Column-major storage would help vectorize the columns with each column element in adjacent memory locations. The first matrix is fine, but we want the second matrix to be stored in a mirror image of itself.

Hmm, a mirror and a matrix. What does that sound like? A transposed matrix.

Pseudo-Transposed Second Matrix. The simplest way to get column-major order of a matrix (especially if square) is to use the transpose of the matrix, and modify the internals of the matrix multiplication function to pretend that the transpose is actually the column-major storage of the original second matrix. I call it the "fake transpose" method, which is a bit of a misnomer because it is the actual transposed matrix, but we modify the matrix multiplication code to access it with reversed logic indices.

Confusing? Yes, I felt the same way, but if you follow it through carefully, you can see that the transpose is really very similar to storing the original matrix in column-major order, where each column element is stored in adjacent memory. The columns of the original problematic matrix become fake rows in the fake transpose, stored in sequential memory addresses. So, for square matrices, we can take the transpose of a matrix, and it's like the matrix has been converted into column major storage. However, we also need to change the C++ code in the matrix multiplication kernel, because it assumes row-major order storage of both matrices, but now we've got row-major storage only for the first matrix, and column-major storage for the second one (our fake transpose).

The main point of optimization with a transpose is that the column becomes a contiguous vector from a row in the transposed matrix.

Here's what the matrix multiplication algorithm looks like when it's working on a "fake" transpose:

```
void aussie_matmul_matrix_fake_transpose(const ymatrix m1,
    const ymatrix m2, int n, ymatrix mout)
{
    // Matrix-Matrix naive n^3 algorithm on a TRANSPOSE
    for (int row = 0; row < n; row++) {
        const float* rowvec = &m1[row][0];
        for (int col = 0; col < n; col++) {
            float sum = 0.0f;
            const float* colvec = &m2[col][0];  // Row!
            for (int k = 0; k < n; k++) {
                sum += (rowvec[k] * colvec[k]);
            }
            mout[row][col] = sum;
        }
    }
}
```

Note that the above code assumes the transpose has already been computed. However, it is viable to compute a new transpose matrix in a preliminary step and still be faster, because transposing a matrix only adds an extra *O(N^2)* time to compute the transpose (and *N^2* storage space to store it temporarily), whereas the main matrix multiplication is *O(N^3)* time.

Perhaps surprisingly, this transpose method is much faster even without any vectorization. Because the column vectors are accessed in sequential order from contiguous memory, there is much better data locality for the memory cache, and also for any predictive pipelining happening in the cache. Here's the benchmark comparison:

```
Matrix-Matrix multi (MatMul) benchmarks (N=2048, ITER=1):
Matrix-matrix mult basic: 69479 ticks (69.48 seconds)
Matrix-matrix fake transpose: 47469 ticks (47.47 seconds)
```

The transpose method is 31% faster with an unchanged basic MatMul algorithm. And all we did was permute two indices in a two-dimensional array. This code does exactly the same arithmetic computations as the naive version, but accesses memory in a different order, giving us a cache speedup.

There are various other small coding optimizations that can improve the transposed MatMul method further. The loop body could be partially unrolled by 4 or 8 iterations (or more).

Here's the C++ code of the version with an unrolling factor of 8 iterations:

```
void aussie_matmul_matrix_fake_transpose_unrolled8(
  const ymatrix m1, const ymatrix m2, int n, ymatrix mout)
{
    // Transpose Matrix-Matrix mult 8 iteration unroll
    aussie_assert(n % 8 == 0);
    for (int row = 0; row < n; row++) {
            const float* rowvec = &m1[row][0];
            for (int col = 0; col < n; col++) {
                float sum = 0.0f;
                const float* colvec = &m2[col][0];
                for (int k = 0; k < n; k += 8) {
                    sum += (rowvec[k] * colvec[k])
                        + (rowvec[k + 1] * colvec[k + 1])
                        + (rowvec[k + 2] * colvec[k + 2])
                        + (rowvec[k + 3] * colvec[k + 3])
                        + (rowvec[k + 4] * colvec[k + 4])
                        + (rowvec[k + 5] * colvec[k + 5])
                        + (rowvec[k + 6] * colvec[k + 6])
                        + (rowvec[k + 7] * colvec[k + 7])
                        ;
                }
                mout[row][col] = sum;
            }
    }
}
```

Here are the benchmark results:

```
Matrix-Matrix mult (MatMul) benchmarks (N=2048, ITER=1):
Matrix-matrix fake transpose unroll 4: 15221 ticks (15.22 s)
Matrix-matrix fake transpose unroll 8: 12151 ticks (12.15 s)
```

Further tweaks are possible. The internal loop could be fully unrolled for a known vector size. Also, the initialization "`sum=0.0f`" could be removed by peeling the first iteration and starting the loop at "`k=1`".

Pointer arithmetic could be used to avoid loop indices and the double bracket accesses. However, these are small fry, and we're now on the hunt for the Spanish mackerel of MatMul optimizations: *vectorization*.

Vectorized MatMul

Cache speedup is not the only benefit of the transpose method. Once we have column-major storage for the second matrix, then both the rows of the first matrix, and the columns of the second matrix are in contiguous memory. The computation is a normal vector dot product again on two vectors stored as arrays in memory (i.e., "`rowvec`" and "`colvec`" in the C++ code above). Hence, we can just use all of our standard vector dot product speedups again, including vectorization and hardware acceleration.

As an example, here's the AVX-2 vectorization of the transpose method using the FMA 256-bit intrinsics to do the vector dot product in parallel. This parallelizes the dot product by 8 elements at a time:

```
void aussie_matmul_matrix_fake_transpose_vecdot_AVX2(
  const ymatrix m1, const ymatrix m2, int n, ymatrix mout)
{
    // AVX2 Matrix-Matrix multiplication
    aussie_assert(n % 8 == 0);
    for (int row = 0; row < n; row++) {
        const float* rowvec = &m1[row][0];
        for (int col = 0; col < n; col++) {
            const float* colvec = &m2[col][0];
            mout[row][col] = aussie_vecdot_FMA_unroll_AVX2(
                                     rowvec, colvec, n);
        }
    }
}
```

Here are the benchmark results:

```
Matrix-Matrix multi (MatMul) benchmarks (N=2048, ITER=1):
Matrix-matrix fake transpose AVX1: 19522 ticks (19.52 s)
Matrix-matrix fake transpose AVX2: 12747 ticks (12.75 s)
```

If anything, these AVX results are disappointing. Basic loop unrolling techniques (in the prior section) did better than AVX1 and the same as AVX2 vectorization. However, we haven't used AVX optimally inside the sequential code here. The AVX intrinsic calls should be moved up into the loop body without any function call overhead (i.e., inlining the function manually).

I coded up that idea, and it made almost zero difference! I guess the C++ compiler is already inlining it, or function call overhead is a tiny percentage.

Further parallelization speedups would include using AVX-512 or AVX-10 intrinsics for vectorizing 16 elements in parallel. Also desirable are various further optimizations of the sequential code around any AVX intrinsics. The inner "`col`" loop could be fully or partially unrolled with multiple AVX sequences and/or optimized with pointer arithmetic.

Loop Tiled/Blocked MatMul

The triple-nested MatMul version with the vectorized inner loop is still nowhere near what is possible. There are three more ways to increase throughput:

- Data locality within the matrices.
- Pipelining of the SIMD instructions.
- Avoiding repeated loads of the same data.

The data locality of the basic AVX transposed MatMul algorithm is still far from optimal, although we fixed the most egregious problem by using the transpose. The algorithm is simply scanning down all of the dimensions, without really any attempt to maintain data locality.

The method of calling AVX intrinsics is simply doing "load, FMA, store" repeatedly along blocks of 4 or 8 elements, which does not allow for the natural pipelining of the FMA instructions. The loads and stores are interrupting the flow of computation.

Secondly, if you look carefully at the "load" operations that are happening in the sequence, you realize that it is repeatedly loading the same regions of the matrices.

Tiling or blocking the MatMul loops are far more effective. The basic idea is that instead of scanning sequentially, we process smaller square or rectangular "tiles" or "blocks" of the data, one at a time.

Data locality is the main aim of a tiled algorithm, but it also helps us achieve better pipelining of SIMD instructions, because we can load all the data in, and then perform multiple arithmetic operations on it without any intervening loads or stores.

And since a tiled MatMul is iterating more carefully over smaller blocks of data within the matrices, there's also less redundant loading of the data overall.

Fast Matrix Multiplication Theory

The main techniques for faster matrix multiplication of general matrices include:

- Strassen's algorithm
- Winograd's algorithm
- Fast Fourier Transform (FFT) methods

Matrix multiplications can also be sped up by restricting our algorithm to only use matrices that are of special types:

- Low-rank matrix factorization
- Sparse matrices
- Special matrix methods (e.g., Butterfly matrices, Monarch matrices, etc.)

Each of these specialized matrix types can have a faster matrix multiplication kernel than using the all-purpose GEMM kernel. For example, sparse matrices can be stored in a compacted permuted-tuple format, with parallelization of permutation arrays for computation.

Approximate Matrix Multiplication. Approximate Matrix Multiplication (AMM) refers to a variety of complicated model optimization techniques that replace matrix multiplications with various approximations that avoid the cost of arithmetic multiplication, trading off some accuracy.

These methods are usually distinct from quantization methods, are not specific to certain subclasses of matrices, and evoke more advanced mathematics in the theory of matrices.

Note that these algorithms apply at the high-level of how matrices are multiplied with other matrices or with vectors (e.g., avoiding some vector dot products), whereas there are also low-level optimizations of the arithmetic operation when multiplying two numbers.

These two classes of approximation research are not the same, and are actually orthogonal to each other.

Multiplying by Transpose

The transpose of a matrix is commonly used in matrix multiplication algorithms, both as part of the algorithms and as a speedup. For example, this occurs in AI engines with the QKV matrix computations inside the attention heads, where the transpose of K is used, usually denoted as K^T in the algebraic formula.

Note that this is the actual algebraic use of the *real* transpose, as opposed to the unique idea of using a "fake transpose" to get column-major storage of matrices for easier vectorization.

The code to compute the transpose of a matrix is shown below for a square matrix:

```
void aussie_matrix_transpose_basic(const ymatrix m1,
     int n, ymatrix transpose)
{
    // Transpose: put transpose into the output matrix
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            transpose[j][i] = m1[i][j];
        }
    }
}
```

The funny thing is that if we want to multiply a "real" transpose as the second matrix in some computation, then the original non-transposed matrix is the "fake transpose" of the "real" transpose.

How awkward!

But it's actually good, because we usually already have the original matrix in memory, and we don't even need to compute the (real) transpose. Instead, to do a MatMul of a matrix with this real transpose, we can instead use the original matrix as the second operand in the kernel that is based on the column-major storage of a fake transpose. Oh, dear, I feel like it's all circular and I'm digging myself into a word pit here! But it all works out in the end, and it's fast, which is really the one and only thing.

References

1. Ulrich Drepper (2007), *What Every Programmer Should Know About Memory*, November 21, 2007, http://people.redhat.com/drepper/cpumemory.pdf
2. Kazushige Goto (2008), *Anatomy of High-Performance Matrix Multiplication*, ACM Transactions on Mathematical Software, Volume 34, Issue 3, Article No.: 12, May 2008, pp 1–25, https://doi.org/10.1145/1356052.1356053, PDF: https://www.cs.utexas.edu/~flame/pubs/GotoTOMS_revision.pdf
3. Harald Prokop (1999), *Cache-Oblivious Algorithms*, Masters Thesis, MIT, June 1999, http://supertech.csail.mit.edu/papers/Prokop99.pdf
4. Intel (2023), *Intel® 64 and IA-32 Architectures Optimization Reference Manual: Volume 1*, August 2023, 248966-Software-Optimization-Manual-V1-048.pdf
5. Agner Fog (2022), *Vector Class Library (VCL)*, https://www.agner.org/optimize/vcl_manual.pdf
6. Sergey Slotin (2022), *Matrix Multiplication*, Algorithmica, https://en.algorithmica.org/hpc/algorithms/matmul/ Code: https://github.com/algorithmica-org/algorithmica/blob/master/content/english/hpc/algorithms/matmul.md

Part V: Multithreading Optimizations

29. Multithreading Optimizations

C++ Multithreading Optimizations

Multithreading is the art of parallelizing on a multicore CPU, often as part of low latency programming. Threads have been around since at least the 1990s (e.g., POSIX threads), even before most CPUs even had "cores," but recent advancements have made them much easier to code.

C++11 introduced a standardized thread library called `std::thread` (along with the supporting extra classes `std::mutex` and `std::atomic`), and C++17 then introduced a lot more advanced parallelization modes.

What is Multithreading?

In this discussion, threads run on the CPU, and you can have many threads per CPU (or per "core"). Multithreading and multicore programming are largely the same thing, or at least they're in the same ballpark.

Other types of threads can differ quite a lot. For example, there is also a slightly different idea of "threads" on GPUs in the CUDA C++ programming language.

You can run 1024 threads on an NVIDIA GPU, but you might not want to do that on your CPU lest you run out of stack space. CUDA C++ allows 1024 threads by having a quite restricted amount of GPU memory (sometimes called VRAM) allocated to the call stacks for each GPU thread in a grid.

Hence, stack overflow is a thing on GPUs, too.

How Not to Multithread

If you're looking for a short career as a multithreading programmer, here are some suggestions:

- Launch as many CPU threads as you possibly can, ideally one per vector element, just like you do in a low-level GPU kernel for AI inference.
- Put huge buffer objects as local variables on your call stack, and launch multiple threads of that.
- Fix your huge local buffer variables by making them `static`, because that function won't ever get run twice at the same time.
- Use mutexes around every access to all your variables, just to be safe.
- Recursion will get you fired in any coding job, except university lecturer, so it's best to pretend you've never heard of it.

High-Level Multithreading Optimization

The first point above all else: *multithreading is a high-level optimization in itself.* Hence, you want to be judicious in your choices of where to use your threads, and at what level.

Some of the issues that control the overall concurrency that is achieved via a multithreaded architecture include:

- Abstraction level choices for splitting the work across threads.
- Thread pool design pattern — avoid creating and destroying threads.
- Thread specializations — e.g., producer and consumer threads model.
- Message-passing design pattern to avoid locking — e.g., with a paired future and promise.

Focusing on the data can also be useful to optimize:

- Multithreading-friendly data structures — e.g., queues (esp. lock-free versions).
- Maximize read-only "immutable" data usage — aims to avoid blocking concurrent readers.
- Advanced data structure read-write models — copy-on-write, versioned data structures.
- Shard data across threads — reduces needed synchronizations (or other types of data partitioning).
- Reduce disk writes — e.g., use in-memory logging with late disk writes.

Ways to optimize by focusing on the execution pathways include:

- Slowpath removal — keep the hot path small and tight.
- Defer error handing — most error code is uncommonly executed (i.e., a slowpath), so avoid, defer or combine error detection code branches.
- Cache warming — keep the hotpath bubbling away.
- Full hotpath optimizations — e.g., for HFT, the hotpath is not just "trade" but actually the full latency from data feed ingestion to execution, so it's actually "receive-analyze-decide-and-trade."

Some of the more pragmatic points include:

- How many threads?
- How long should each thread run?
- When to exit a thread versus waiting.

There's no wrong or right answer to these questions, as they depend on the application and the problem you're trying to solve.

Low-Level Multithreading Optimization

There are various ways to modify how you run threads in order to optimize their concurrency speed. These are not as impactful as the higher-level thread choices, but are still important.

Some methods to change the lower-level thread architectures include:

- Core pinning (processor affinity) — every popular thread can have a favorite core.
- Early unlocking — e.g., copy data to local variables, release lock, then do the computations.
- Cache locality improvements (L1 cache and memory prefetch cache)
- Branch reductions —the instruction pointer on the straight-and-narrow.
- Lock-free algorithms — avoid mutex overhead and blocked thread delays.

Ways to avoid slow-downs in multithreading, and therefore increase speed:

- Minimizing thread launch and shutdown overheads.
- Releasing locks early by avoiding unnecessary computation, I/O waits, etc.
- Minimizing context switches
- Memory reductions (e.g., allocated memory optimizations; reduce thread-specific call stack size).
- Avoid spinlocks (busy wait) or mitigate with exponential backoff methods.
- Avoiding "false sharing" from overlap of CPU memory prefetch cache lines (e.g., use `alignas(64)` to separate unrelated atomics).
- Check `std::lock_guard` is not unnecessarily delaying the unlock (i.e., till it goes out-of-scope).

Sequential C++ Code Optimizations

An important point about the code running in any thread is that: *it's just C++ code*. Each thread is running a sequential set of instructions, with its own call stack. Hence, all of the many ways to optimize normal C++ code also applies to all of the code in the thread.

Hence, all of the basic ideas for C++ code optimizations apply:

- Compile-time processing — `constexpr`, `constinit`, etc.
- Operator efficiency — e.g., replace multiply with bitshift or addition.
- Data type optimizations — e.g., integers versus floating-point.
- Memory optimizations — improve with cache warming (prefetching), memory reductions.
- Loop optimizations — e.g., loop unrolling, code hoisting, and many more.
- Compiler hints — e.g., `[[likely]]` statements.
- Function call optimizations — e.g., inlining, always_inline, etc.
- C++ class-level optimizations — e.g., specializing member functions.
- Algorithm improvements — various non-concurrency improvements, such as precomputation, caching, approximations, etc.

So, the bad news is that once you've coded your multithreaded algorithm, you still have to go and do all the other types of sequential optimizations.

Oh, come on, who are we kidding? — it's loads of bonus fun.

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Dung Le, Aug 13, 2020, *Optimizations for C++ multi-threaded programming*, https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
3. Geeks Programming, March 2025 (accessed), *Performance Boosting with C++ Multithreading Techniques*, https://geeksprogramming.com/performance-boosting-cpp-multithreading-techniques/
4. Karthikeyan Akkapalli, Aug 25, 2024, *Multithreading in C++: Concepts, Challenges, Advanced Techniques, and Applications*, https://medium.com/@karthikeyan_akkapalli/multithreading-in-c-concepts-challenges-advanced-techniques-and-applications-b97cbdcf31c7
5. Deb Haldar, August 17, 2017, *Top 20 C++ multithreading mistakes and how to avoid them*, https://acodersjourney.com/top-20-cplusplus-multithreading-mistakes/
6. Nimrod Sapir, 2019, *High-Frequency Trading and Ultra Low*, Latency Development Techniques, https://corecppil.github.io/CoreCpp2019/Presentations/Nimrod_High_Frequency_Trading.pdf, Code: https://github.com/DanielDubi/StaticFlatMap
7. Machinet, March 13, 2024, *How to optimize C++ code for use in high-frequency trading algorithms?* https://www.machinet.net/tutorial-eng/optimize-cpp-code-high-frequency-trading-algorithms
8. Ivan Eduardo Guerra, October 19, 2024, *C++ Design Patterns for Low Latency Applications Including High Frequency Trading*, https://programmador.com/series/notes/cpp-design-patterns-for-low-latency-apps/

30. Common Multithreading Bugs & Slugs

Multithreading Bugs Overview

Modern C++ is hard enough, and multithreading adds another layer of complexity. You're not alone, and bugs abound in parallel multithreaded C++ code! Various beginner bugs and simple misunderstandings include:

- Linux linking problem with the "pthreads" library (needs "`-pthread`" linker option).
- `main()` does not wait for other threads and needs to call `join()`.
- Calling `join()` inside the new thread causes a deadlock.
- Crashing on join() because the thread is no longer "joinable" (test via the `joinable()` method).

Here are some simple mistakes you can make when trying to convert your application to multithreading:

- Not using any synchronization for your threads (Yikes!).
- Not locking in all the places.
- Forgetting locking for `cout` and `cerr` output.
- Not unlocking on all paths.
- Double-locking a mutex.
- Double-unlocking a mutex.

Once you get into running multiple threads, here are some common gotchas in terms of assumptions and misunderstandings:

- Assuming that the standard C++ containers are always thread-safe.
- Assuming that all simple `int` or pointer operations are atomic without using `std::atomic`.
- The `volatile` specifier is not a synchronization method.

Let's examine some of these simpler multithreading mistakes.

Main Thread Exits Early

Here's a simple "Hello World" program using standard threading. It looks totally fine, right?

```
#include <iostream>
#include <thread>

void thread_function()
{
    std::cout << "Hello world!" << std::endl;
}

int main()
{
    std::thread t1(thread_function);
    return 0;
}
```

Can you see the bug? The program won't print anything.

Why? Because there's nothing stopping the `main()` function, which just keeps going and exits immediately. It doesn't wait for the other thread to even start, let alone finish, but is indifferent to its plight.

That's one of the things to understand, but there are actually a few fundamental points to note here:

- Launching a new thread is a non-blocking operation.
- Exiting the program kills all unfinished threads.
- To wait for a thread, call `join()`.

Hence, to fix the program, you need to do this in the `main()` function:

```
std::thread t1(thread_function);
t1.join();   // Wait!
```

After this change, the main thread will politely wait for the other thread to print its message and finish. The `join()` function has the following features:

- Blocking call that waits for the other thread to finish.
- Immediate return if the other thread has already finished.

Self-Join Deadlock. Note that you cannot call `join()` from inside the new thread itself. This causes an immediate deadlock, because the `join()` call in the thread is waiting for itself to finish, but it cannot finish because it's waiting (is anyone else a fan of *Catch 22*?). I feel like this self-join situation is a bug that the standard threads library could check for, and maybe it does in the newer "hardened" versions of the standard C++ library.

Anyway, just don't do that. It's the main thread that needs to join the new thread from the outside, not the other way around.

Joinable Safety Check. In the above simple code, it's not necessarily needed, but safer thread code would validate that the thread is allowed to join before trying to do so, because it crashes if you're wrong! For example, a "detached" thread is non-joinable. Here's the simplest check:

```
    if (t1.joinable()) t1.join(); // Safer
```

Note that in addition to `join()`, there's also a method called `detach()`, but the former is much simpler. The main thread still needs to wait for a detached thread before exiting, but requires additional synchronization via some other method, because you can't `join()` a detached thread, as we just discussed.

Linux Linking Problem

You may find that a standard C++ program using the standard thread library does not compile with GCC on Linux, or at least on older versions. The problem is that standard C++ threads are implemented as POSIX threads on Linux with GCC.

The problem is that the POSIX threads library (usually called "pthreads") is not getting linked properly. You need to add an extra "`-pthread`" compiler flag to the linking step (without an "s"). The error looks like this:

```
    .../thread:127: undefined reference to `pthread_create'
```

And the fix is to add this linking flag for GCC:

```
    -pthread
```

Here's the line in my `Makefile` for my testing build:

```
    LINKFLAGS=-L/usr/lib64/ -g $(PFLAGS) -pthread
```

Volatile Misunderstanding

This is a common mistake made about a longstanding feature of C++ (and also C). The "`volatile`" specifier in C++ is *not* for synchronization. In particular, the wrong use of this specifier is not useful in multithreading because it:

- Does not do anything with other threads.
- Does not make a variable atomic.

Not only won't it do anything useful for your multithreading synchronization, but it will actually slow your code down because it interferes with the optimizer.

The purpose of `volatile` is much more mundane than multithreaded code, and relates only to sequential programming, with these features instead:

- Indicates that this variable or address has "side effects" that the compiler does not know about.
- Blocks the compiler from "optimizing out" reads or writes to this variable.

The main real-world uses of the `volatile` specifier include:

- Mapping an I/O device to a variable or memory address.
- Stopping compiler optimizations in benchmarking of low-level arithmetic.

The first one of these is the reason that it exists in the C++ language (and originally in C, too). The idea is to tell the compiler that a variable or address represents an input or output device. So, if the compiler sees the same variable or address read twice, it doesn't optimize the second one out, which would be faulty if that address represents incoming data from a peripheral device or network feed. Similarly, if you write the same value to that variable, intending to send two bytes to an output device, the compiler is stopped from blocking you.

The use in benchmarking is a programmer trick that really misuses a language feature. But there's nothing wrong with that, because the standard semantics for `volatile` are well-defined and have existed in the language since forever. It was standardized into the C language in the ANSI C standard of 1989/1990, and was formally incorporated into C++98.

The `volatile` specifier is a wonderful feature of C++ that I've used often. But, as mentioned above, don't use `volatile` as a synchronization method, because nowhere in the above list of its features is anything related to multithreading or concurrency.

Advanced Multithreading Bugs

As you progress to greater multithreading knowledge, the bugs get harder:

- Race conditions — a variety of orders that can have different results.
- Deadlock — often from wrongly-ordered acquisition of multiple locks.
- Livelock — a weird kind of near-deadlock cycling.
- Memory order errors — with atomics and lock-free data structures.
- High-level concurrency issues — sigh, the low-level concurrency code was working so well.
- Thread starvation — a low-priority thread never gets any juice.
- Priority inversion — weirdly, a low-priority thread gets *all* the juice.

That's more than enough! However, there's another important category of C++ multithreading bugs:

All the other C++ bugs you already know about.

Multithreaded code still uses basic sequential C++ code in every thread. There might be a few bugs to watch out for in that!

Multithreading Slugs

There are plenty of ways to improve the performance of a C++ multithreading application. In fact, you could write a whole book on it!

Some of the higher-level slugs to avoid include:

- Using sequential code instead of multiple threads (the horrors!).
- Launching too many threads (leads to thread overhead).
- Too many runnable threads per core.

Some possible slowdowns in your locking strategy:

- Coarse-grained locking for an entire data structure (per-container locking).
- Using a single per-class mutex as `static` data member (per-class locking).
- Using unique locks for read operations, instead of shared read-write locks.
- Using a mutex for a simple integer counter (or a Boolean status flag), when atomics would be enough.

Some of the low-level slugs in locking synchronization include:

- Overlong lock holding with `std::lock_guard` destructor unlocking.
- Not freeing a lock when no longer needed (e.g., when doing computation).
- Holding a lock while doing the last computations, instead of copying data to local variables (and then unlocking before the computations).
- Holding a lock before an I/O operation or blocking kernel system call.

Some other ideas for areas to address for performance:

- Thread function arguments are pass-by-value by default (e.g., for objects).
- Not using a thread pool instead of launching/destroying lots of threads.
- Don't do core pinning (thread affinity) with core zero (it's the main Linux kernel core).
- Blocking calls to `select()` in socket programming.
- Not doing any real work in the main thread (it's a useful worker, too!).

Fake Multithreading

One weirdly common slug is "redundant thread computations" due to a simple programming bug. This means that you have multiple threads repeating the exact same work in multiple threads, but nobody notices because it's a slug rather than a bug.

For example, if you're optimizing a "vector-add" operation that takes two vectors and outputs a third vector, and the vectors are very long (e.g., in AI), then you might try to have different segments of a vector processed in different threads to parallelize the operation. But if you mess up the indices, such as if your boss calls you away to an important meeting while you're coding, there might be a problem with the loop indices.

If you actually send work to each thread that has the full index range, rather than a sub-segment, then each thread scans the entire vector and outputs the entire third vector. This is insidious because the results should be correct, but it's re-computing the same arithmetic operations multiple times in parallel.

There's nothing wrong with your high-level design except that the code still has `n` instead of `i` in the code that assigns jobs to threads. You can go crazy and optimize your multithreaded vector-add operation with producer-consumer thread pools and lock-free queues, and then add work stealing for load balancing, but if your indices are wrong, it's all moot. Slugs and bugs can live together!

References

1. Deb Haldar, August 17, 2017, *Top 20 C++ multithreading mistakes and how to avoid them*, https://acodersjourney.com/top-20-cplusplus-multithreading-mistakes/
2. Akhil Robertson Cutinha, Jan 10, 2021, *Common Multithreading Mistakes*, https://medium.com/swlh/common-multithreading-mistakes-e36ca8e98e7a
3. Rainer Grimm, February 10, 2021, *Resolving C/C++ Concurrency Bugs More Efficiently with Time Travel Debugging*, https://www.modernescpp.com/index.php/resolving-c-c-concurrency-bugs-more-efficiently-with-time-travel-debugging/
4. Geeks for Geeks, 27 Feb, 2024, *Threading Issues*, https://www.geeksforgeeks.org/threading-issues/
5. Matrix Media Solutions, September 20, 2024, *Debugging Threaded Code: Tips and Techniques for Identifying and Resolving Concurrency Issues*, https://www.matrixnmedia.com/debugging-threaded-code-tips-and-techniques-for-identifying-and-resolving-concurrency-issues/
6. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, 2023, *Common Concurrency Problems*, in book *Operating Systems: Three Easy Pieces*, 2018, https://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf, https://www.amazon.com/exec/obidos/ASIN/198508659X/

31. Thread Overhead

What is Thread Overhead?

Thread overhead is the extra cost of creating and destroying threads, at the start and end of multithreaded algorithm execution. This is effectively an extra cost that you wouldn't have in a single-threaded C++ application, and is offset against the performance gain of parallelizing your code into multiple threads.

Hence, the two main components of thread overhead are:

- Launching new threads
- Destroying a finished thread

Note that these costs do not involve any other thread, but are specific to a single thread. There are some other less obvious causes of extra thread overhead:

- Constructors of `thread_local` objects (thread-local storage)
- Destructors of thread-local objects on thread shutdown

These per-thread costs are analogous to the startup and shutdown costs of C++ global objects in a non-threaded program. A normal C++ program has extra code that runs before `main()` for global object constructors, and destructors that run after the application finishes.

Measuring Basic Thread Overhead

You're supposed to use a "thread pool" to avoid all the basic overhead of starting and stopping threads. I was wondering how much that overhead would actually be, so I decided to time it, using a dummy example.

Here's my simple benchmarking function that just consumes some time, but uses `volatile` to avoid getting optimized away:

```
void thread_function(int n)
{
    for (volatile int i = 0; i < n; i++) {
        // nothing
    }
}
```

I wanted the code to be doing some real instructions, rather than just sleeping for a delay, such as with the `this_thread::sleep_for()` function, in case it made any difference to the status of the thread before shutdown.

Here is the instrumentation I used to try to measure thread startup and shutdown overhead using the high-resolution clock in the `<chrono>` library:

```
{
before = std::chrono::high_resolution_clock::now();
before_thread = before;
std::thread t1(thread_function, n);
after_thread = std::chrono::high_resolution_clock::now();
t1.join();
after_join = std::chrono::high_resolution_clock::now();
}
now = std::chrono::high_resolution_clock::now();
```

And the computations of the different costs in microseconds are:

```
diff_thread_function =
    std::chrono::duration_cast<std::chrono::microseconds>
       (now - before).count();
startup_thread_function =
    std::chrono::duration_cast<std::chrono::microseconds>
       (after_thread - before_thread).count();
shutdown_thread_function =
    std::chrono::duration_cast<std::chrono::microseconds>
       (now - after_join).count();
compute_thread_function =
    std::chrono::duration_cast<std::chrono::microseconds>
       (after_join - after_thread).count();
```

Lambda and Functor Threads

A thread can be defined in other ways than a normal function call, such as function pointer (not much different), a lambda function, and a functor (function object). Hence, I decided to test the various different ways that a thread body of executable instructions could be defined, such as:

- Standard function (i.e., with a name)
- Lambda function (anonymous function)
- Functor (function object)

The named function is shown above with the timing instrumentation around it. Here's the lambda function version with the anonymous `[]` syntax:

```
std::thread t1( [](int n) {
        for (volatile int i = 0; i < n; i++) {
            // nothing
        }
    }, n);
```

And here's a functor for your viewing pleasure, which is a "function object" where the `operator()` has been defined:

```
struct Functor {
    void operator()(int n) {
        for (volatile int i = 0; i < n; i++) {
            // nothing
        }
    }
};
Functor functor;
std::thread t1(functor, n);
```

Timing Results

Timing on Linux with GCC, these are the non-threaded timings:

```
Basic Function: 34 microseconds
Basic Function (Repeat): 36 microseconds
Basic Inline: 34 microseconds
Basic Ptr-to-Fn: 34 microseconds
Basic Functor: 32 microseconds
Basic Lambda: 34 microseconds
Basic std::function: 33 microseconds
```

And these are the threaded calls on Linux with GCC:

```
Thread Func (First): 228 us (init: 144, compute: 84, end: 0)
Thread Func (Repeat): 43 us (init: 3, compute: 39, end: 0)
Thread Func (Repeat): 41 us (init: 2, compute: 39, end: 0)
Thread Lambda: 42 us (init: 2, compute: 39, end: 0)
Thread Functor: 40 us (init: 2, compute: 37, end: 0)
```

Note that these are microseconds! The overhead from setting up the threads library with 200 microseconds is not even half a millisecond. And that's only the first call, with the rest of the threads seeming to have only 2 or 3 microseconds of startup overhead on Linux!

Timing on Windows (MSVS) for the non-threaded function calls:

```
Basic Function: 107 microseconds
Basic Function (Repeat): 102 microseconds
Basic Inline: 78 microseconds
Basic Ptr-to-Fn: 97 microseconds
Basic Functor: 136 microseconds
Basic Lambda: 87 microseconds
Basic std::function: 182 microseconds
```

And here are the Windows timings of the thread launches for the same functions:

```
Thread Func (1st): 3387 us (init: 74, compute: 3312, end: 0)
Thread Func (Rep): 649 us (init: 41, compute: 607, end: 0)
Thread Func (Rep): 729 us (init: 35, compute: 694, end: 0)
Thread Lambda: 621 us (init: 34, compute: 586, end: 0)
Thread Functor: 539 us (init: 30, compute: 509, end: 0)
```

A few conclusions can be drawn:

- Thread launch overhead is about 26% on Linux (43 vs 34) and 500% (649 vs 107) on Windows (admittedly, an unfair comparison of a Linux server versus a Windows laptop!).
- The first thread launch has a large extra time cost, which disappears on a repeat, perhaps from initialization of the thread mechanisms (or perhaps it's just a cold cache?).
- There's not much difference between running a thread body with a standard named function, lambda function, or functor (function object) on either platform.

Limitations. This is a dummy function and the overhead would be proportionally less if the thread did more computation. This is a single test of a single function for a single iteration count. There might be a few statisticians who want to object to that level of sampling.

Furthermore, as you can see, my timing method isn't particularly effective at separately computing the startup and shutdown costs of a thread. A lot of the startup cost and shutdown cost seems to be hidden inside the compute time, while the main thread is waiting with the `join()` call. Nevertheless, the total costs are quite indicative of the extra overheads, especially on the very first thread launch.

Synchronization and Context Switch Overhead

The above discussion is about the overhead of threads starting and stopping. There are various other types of overhead that should be optimized in the middle of the thread's execution:

- Synchronization overhead — extra cost of mutexes, locks, atomics, etc.
- Thread wait durations — blocked while awaiting a mutex or lock.

There are also some slowdowns that arise because your code is now split up into multiple threads, which have to be scheduled and time-sliced by the OS and the hardware. Some of the general areas of cost include:

- Context switches — cost of swapping threads in and out of CPU.
- Scheduling costs — the OS choosing which thread to run next.

There are also some slowdowns that occur in hardware caches during these switches, because the OS does not store and reinstate any of the hardware caches. The new thread starts with cold hardware caches, leading to cache misses with performance problems in several areas:

- Memory cache invalidation — context switches lose low-level L1/L2/L3 CPU cache advantages.
- TLB cache loss — the virtual address cache is lost.
- CPU instruction pipeline —stalls because execution location has moved.
- Instruction prefetch — cleared because a new thread starts elsewhere.
- Memory prefetch cache — the new thread is unlikely to be accessing the same memory locations.
- NUMA cache issues — loss of cache coherence in multicore NUMA.

In other words, everything that the CPU does to make executing code run fast gets undermined by a context switch.

What causes a context switch? Context switches can arise at the end of a time-slice in scheduling, or can occur whenever the threads uses a primitive that can block the thread. Some examples that trigger a context switch include:

- Synchronization — waiting for a lock or mutex.
- System calls — those that block, such as for I/O or networking.

A context switch involves storing all the status of the current thread and then overlaying a new context for the new thread. This has its own cost, and also triggers a flush of various CPU hardware caches, so the new thread starts its time-slice with cold caches. Hence, context switches are expensive, and minimizing the number of context switches is an important part of optimizing multithreading code.

32. Thread Pools

What are Thread Pools?

Threads going swimming in warm ocean water. Who doesn't love the beach?

Thread pools are a design pattern in C++ multithreading that avoids the cost for creating and destroying threads by using long-running threads. Instead of incurring this thread overhead, a "pool" of available threads have been pre-created, which sit there until work is available to be done. The main characteristics are:

- Idle threads wait for work (e.g., off a task queue).
- Threads are not destroyed after completing a chunk of work.

Thread pools are mostly used in a "producer-consumer" design pattern, although thread pools can also be used in other ways. There are effectively two thread pools in this design pattern:

- Producer thread pool — or sometimes a single producer.
- Consumer thread pool — always multiple, or what's the point?

Typically, one or more producer threads adds work items to a queue, such as when it receives new data from a network source. Another group of consumer threads is idle while waiting to pull work off the queue. Consumers do the work, return the results, and then add themselves back to the pool of idle consumer threads awaiting more work.

Work Queue Implementation

The typical features of the thread-safe queue used in a producer-consumer work queue include:

- Vector of worker threads
- Queue of arbitrary tasks (e.g., usually implemented as lambdas, functors or `std::function` wrappers)
- Stop flag for shutting down

The main interfaces are:

- Enqueue work (push) — by the producer.
- Deque work (pop) — by the consumer worker threads.
- Shutdown — tell all the threads to stop.

For a more advanced thread pool, some extra convenience features of the work submission interface to consider include:

- Work functions with arbitrary arguments (via parameter packs, variadic functions)
- Perfect forwarding of function arguments (e.g., `std::forward`)

The work queue can be implemented in various ways:

- Use `std::queue` or `std::deque` inside the thread pool object.
- Hand-coded locking queue with mutex and condition variable.
- Lock-free queue with atomics and "Compare-And-Set" (CAS) primitives.

Thread Pool Example

I tried hard to make this example simpler; I really did! In fact, my aim was to use only explicit function names, and avoid any uses of the syntactic sugar for:

- Lambda functions
- `std::function`
- Functor mechanics

However, it was a triple fail. Perhaps the last point was unavoidable, since a worker task is a function object. But I also had to add a little lambda function just to get the worker thread function to run and another one for the predicate in the condition variable wait. I also used `std::function` for the type of the function objects.

Anyway, here's the first attempt at a "simple" thread pool with these features:

- Wraps around a `std::queue` of tasks — not anything home-grown.
- Each task is a function object — so they can be put on a queue.
- Vector of threads — each one waits forever for a task.
- Mutex and condition variable for synchronization — i.e., basic locking, not lock-free).
- Stop flag — only used when shutting down the entire thread pool.

And here's the code of the basic interface and private data members:

```
class ThreadPool {
    using TaskType = std::function<void()>; // Type alias
  private:
    std::vector<std::thread> threads_;  // Threads in pool
    std::queue<TaskType> qtasks_;  // Queue of tasks to run
    std::mutex mtx_;
    std::condition_variable cv_;
    bool stopflag_;  // Shutdown flag (set in destructor)
    ....
};
```

For safety, I've deleted some of the whole-thread-pool methods:

```
ThreadPool(const ThreadPool&) = delete;
ThreadPool(ThreadPool&&) = delete;
ThreadPool& operator=(const ThreadPool&) = delete;
ThreadPool& operator=(ThreadPool&) = delete;
```

Here's the basic constructor with the number of threads to create in the pool, by adding them to a vector of threads:

```
ThreadPool(size_t nthreads) : stopflag_(false) {
    for (int i = 0; i < nthreads; i++) {
        // Create new thread
        auto tobj = [this]() { worker_thread(); };
        threads_.emplace_back(tobj);
    }
}
```

Here's the worker that each thread runs, with an infinite loop waiting for tasks.

```
void worker_thread() {
    for (;;) { // forever
        std::unique_lock<std::mutex> lock(mtx_);
        cv_.wait(lock, [this] {
             return !qtasks_.empty() || stopflag_; });
        if (!qtasks_.empty()) {
            TaskType t = qtasks_.front();
            qtasks_.pop();
            lock.unlock();  // Unlock before running task!
            t();  // Run the task!
        }
        else { // Empty queue
            if (stopflag_) { return; } // Quit
        }
    }
}
```

Each thread will only exit if (a) the destructor sets the stop flag, and (b) there's no more tasks still on the work queue. This ensures the whole thread pool gracefully shuts down by first finishing all jobs.

And here's the destructor, which sets a global stop flag, notifies all the threads, and then waits for each one to stop.

```
~ThreadPool() {
    std::unique_lock<std::mutex> lock(mtx_);
    stopflag_ = true; // Set the shutdown flag
    lock.unlock();
    cv_.notify_all(); // Tell everyone to stop
    for (auto &t : threads_) {
        t.join(); // Wait for all threads
    }
    threads_.clear();
}
```

Here's the enqueue function to add a work task for a thread to run:

```
void enqueue_task(TaskType t) {
    std::unique_lock<std::mutex> lock(mtx_);
    qtasks_.emplace(t);
    lock.unlock();
    cv_.notify_one();  // Wake one worker
}
```

Here are some of the ways to call the enqueue function to submit work to run:

```
p.enqueue_task(my_test_task);  // Ptr-to-function
p.enqueue_task(std::function<void()>(my_test_task));
p.enqueue_task([]() { /*lambda function*/ });
auto functor = []() { /*lambda function*/ };
p.enqueue_task(functor);
```

The whole thread pool is far from perfect, and I'm sure you can see some areas needing work.

Problems to Avoid

There are a lot of little fiddly problems to overcome in the thread pool implementation, even with a wrapper around a standard queue object.

- Fiddly to get the scope right so that the worker function can access the queue object, but is also able to be put into a function object.
- Ensure that we must unlock before running any task (otherwise, all jobs are serialized!).
- Lambda function for the predicate function on the wait of the condition variable.

The above code needs some fixes:

- Enqueue should warn or throw if the thread pool is already stopped.
- Should use move semantics fully to avoid copying any task or thread objects.
- Call `joinable()` before `join()`, just in case.

Various fixes to move semantics are needed here.

- `enqueue_task()` should use `std::move()` to move a new task onto the queue.
- `worker_thread()` should use `std::move()` to pull a new task off the front of the queue.

Advanced Thread Pool Features

Some of the features that can be added to a more advanced thread pool implementation:

- Dynamically increase and decrease the number of workers.
- Priorities for the work jobs to run important work faster.
- Scheduling of jobs to run with a delay or at a specific time.
- Work stealing and thread-specific work queues.
- Support for a graph of interdependent jobs (i.e., a "compute graph" or "task graph").

The interface to the thread pool job submission could also need these capabilities:

- Arguments for tasks (e.g., via parameter packs and `std::forward`).
- Status results indicating success or failure (e.g., non-`void` functions).
- General capabilities to return answer objects to the work submitter.
- Interface for the work submitter to query job status.

Some additional devops infrastructure would be desirable for these thread pool classes:

- Monitoring support via logging, and instrumentation for production usage.
- Self-monitoring to detect straggler/hang jobs (e.g., never-finishing).
- Self-test capabilities for use while regression testing (non-production).
- Timing features for performance measurement (non-production).
- Statistics reporting for production or testing usage.

It's just a small matter of coding.

Task Graphs

Thread pools are mostly designed on the assumption that each piece of work is independent. Hence, the worker threads don't depend on each other in any way, but only on the producer thread that's adding work to the queue. This is the simplest and also the most common requirement.

However, work jobs that depend on each other are not uncommon. The overall network of dependencies between concurrent jobs can create a "graph" of work to be done, with additional synchronization required between the individual workers. An example of a more generalized thread pool that supports a work graph is listed in the references; see Puyda (2024).

References

1. Emily Dawson, April 2025, *Multithreading with C++: Parallel Programming Guide*, https://www.amazon.com/dp/B0F494Z76L/
2. Geeks for Geeks, 3 Jan, 2024, *Thread Pool in C++*, https://www.geeksforgeeks.org/thread-pool-in-cpp/
3. Ishan Tripathi, Dec 11, 2023, *Making a Thread Pool in C++ from scratch*, https://dev.to/ish4n10/making-a-thread-pool-in-c-from-scratch-bnm
4. Matheus Gomes, July 5, 2023, *Thread Pool In C++ – Writing a Concurrent Task Queue*, https://matgomes.com/thread-pools-cpp-with-queues/
5. Barak Shoshany, 27 Dec 2023 (v4), *A C++17 Thread Pool for High-Performance Scientific Computing*, https://arxiv.org/abs/2105.00613v2, Code: https://github.com/bshoshany/thread-pool
6. Dmytro Puyda, 23 Jul 2024 (v2), *A simple and fast C++ thread pool implementation capable of running task graphs*, https://arxiv.org/abs/2407.15805, Code: https://github.com/dpuyda/scheduling

33. Fine-Grained vs Coarse Locking

What is Coarse Locking?

Coarse-grained locking is a simple method of achieving synchronization with relatively few lock objects and not many calls to locking primitives. The locks are "coarse" because they control large chunks, such as a block of code in an entire member function, or access to an entire data structure. Some examples include:

- Long sequences of code with a lock request to start and release at the end.
- Single per-class mutex for your entire data structure.
- One global mutex for all the critical sections of code.

The effect of coarse-grained locking is to effectively limit all accesses to the code block or data structure to be one thread at a time (i.e., full serialization). By comparison, fine-grained locking has multiple locks and more frequent locking and unlocking requests, but over shorter blocks of code or controlling access to portions of a data structure. The fine-grained locking approach is more performant, but requires a lot more effort to code correctly.

Coarse-grained locking can be added to your code relatively quickly. Hence, the advantages of coarse locking include:

- Simplicity
- Thread-safety (it does work)
- Low lock overhead in some cases (fewer total calls to locking primitives)
- Lower risk of concurrency bugs (easy to implement)

The downsides of coarse locking are mainly about performance:

- Blocking other threads for longer (poor synchronization)
- Locking overhead required for read-only accesses
- Serialization of multiple concurrent readers (low parallelism)
- Increased lock contention (for the single mutex)

Adding Coarse Locking

A common requirement for locking is to create your own thread-safe containers, like stacks and queues, since the standard C++ containers are not actually thread-safe. If you have a class where you want its main data structure to be thread-safe, there's a surprisingly simple way to add coarse locking.

The steps are:

- Add a mutex as a data member.
- Add a mutex lock and unlock call to *every* member function.

Here's what the mutex data member to control synchronization in every object looks like:

```
#include <mutex>
#include <vector>

class MyVector {
  private:
    std::mutex mtx_;
    std::vector<int> vec_;
    // ...
  public:
    int get_count() { return vec_.size(); }
    // ...
};
```

Here's a `sum()` class member function, but without any thread synchronization:

```
int sum() {
    int isum = 0;
    for (int i = 0; i < vec_.size(); i++) {
        isum += vec_[i];
    }
    return isum;
}
```

The code to add a mutex with lock calls at the top, and unlock calls at the end of a function:

```
    int sum() {
        mtx_.lock();  // Acquire lock
        int isum = 0;
        for (int i = 0; i < vec_.size(); i++) {
            isum += vec_[i];
        }
        mtx_.unlock();  // Release lock
        return isum;
    }
```

Actually, this method of directly using `std::mutex` is not that good, because you have to make temporary copies of internal data, even in simple getters:

```
    int get_count() {
        mtx_.lock();  // Acquire lock
        int iret = vec_.size();
        mtx_.unlock();  // Release lock
        return iret;
    }
```

An even simpler approach is to use the special wrapper class, `std::lock_guard`, which means you only add one lock guard declaration statement to the top of every member function.

```
    std::lock_guard<std::mutex> lock(mtx_);
```

The mutex object is automatically unlocked at the end of the function, or whenever it returns, by the destructor of the lock guard wrapper object. This fixes the above problems with simple getters so that no temporary variable is needed, because the unlocking is automatically occurring after the return expression is calculated (i.e., effectively it's at the closing right brace of a member function, which is where the destructor runs).

The downside is that you actually have two objects:

- Mutex object (data member)
- Lock guard object (function-local scope)

Here's how it looks in the code:

```
int get_count() {
    std::lock_guard<std::mutex> lock(mtx_); // Acquire
    return vec_.size();
} // Release lock here!
```

Here's how the `sum()` member function looks:

```
int sum() {
    std::lock_guard<std::mutex> lock(mtx_); // Acquire
    int isum = 0;
    for (int i = 0; i < vec_.size(); i++) {
        isum += vec_[i];
    }
    return isum;
} // Implicit lock release here
```

Note that we can control the locking and release of a lock guard object by enclosing it in a narrower scope block. Here's the use of a dummy pair of braces to control the scope for a marginal efficiency gain:

```
int sum() {
    int isum = 0;
    {
        std::lock_guard<std::mutex> lock(mtx_); // Acquire
        for (int i = 0; i < vec_.size(); i++) {
            isum += vec_[i];
        }
    } // Implicit lock release here
    return isum;
}
```

As you can see, adding coarse locking to your whole class can be as simple was adding a single statement at the top of every member function. The main downsides include:

- Forgetting one of the member functions (concurrency bug).
- Performance issues from holding the lock too long.
- Read-only access to your object requires locking calls.

Note that `std::lock_guard` is not the only type of lock wrapper class to consider.

Other examples of standard classes that act as mutex wrappers:

- `std::unique_lock` — allows explicit unlock.
- `std::lock` — basic multi-mutex handling.
- `std::try_lock` — handling of unavailable locks.
- `std::scoped_lock` — handles multiple mutexes.
- `std::shared_lock` — flexible locking method.

Notably, there's the `std::unique_lock` wrapper, which has the advantage that it has an explicit `unlock()` method. This means you can more easily release the lock early if it's no longer needed, which reduces lock contention and delays in other threads. The unique lock wrapper still has the destructor as a backup to release the lock if the mutex hasn't already been unlocked before the end.

Disadvantages of Coarse Locking

This approach of using a coarse locking mechanism in literally every member function looks really inefficient, and it is! There are significant performance problems:

- Basic getter member functions become needlessly inefficient.
- Other `const` member functions need to lock just for read-only access.

Do we really need to lock the basic getters? For example, if a getter is just returning the current count of objects, does it need a lock? Probably not!

I mean, it returns an integer for the count, which is close to an atomic operation, and almost certainly atomic on many CPUs. And if another thread is modifying the data structure, changing the count, this is the caller's synchronization problem. It's really a "higher-level" multithreading problem than the issue here, where we're only trying to keep the data structure itself consistent across multiple calls to its member functions.

However, not all `const` member functions can avoid needing a lock. For example, a `sum()` function above that scans over all of the elements of the vector still needs synchronized access, to avoid some other thread modifying an element in the middle of a scan.

Overall, this coarse-grained locking approach works in terms of thread-safety, but is not ideal in terms of performance. We can probably avoid some of the locks in the simple getter functions.

However, several speed problems remain with this approach:

- Thread overhead even if the caller is only ever reading.
- Multiple concurrent readers are needlessly serialized.
- Not efficient for multiple readers and a single writer.

The cost overhead from coarse locking can be quite significant.

Coarse Locking Overhead

Let's see how much it costs to add coarse-grained locking via lock guards. I chose a basic standard queue container, with just integers. As a control, I declared a basic queue wrapper class without any synchronization.

```
template<typename T>
class QueueWrapNoSync {
private:
    std::queue<T> m_q;
public:
    int count() const { return m_q.size(); }
    T front() const { return m_q.front(); }
    void pop() { m_q.pop(); }
    void push(T t) { m_q.push(t); }
};
```

Next, I created another class with a mutex in the objects, and lock guard statements added to every member function.

```
template<typename T>
class QueueWrapLockGuard {
private:
    std::queue<T> m_q;
    std::mutex m_mutex;
public:
    int count() const {
        std::lock_guard<std::mutex> lock(m_mutex);
        return m_q.size();
    }
    T front() const {
        std::lock_guard<std::mutex> lock(m_mutex);
        return m_q.front();
    }
```

```
        void pop() {
            std::lock_guard<std::mutex> lock(m_mutex);
            m_q.pop();
        }
        void push(T t) {
            std::lock_guard<std::mutex> lock(m_mutex);
            m_q.push(t);
        }
    };
```

Here are the timing results:

```
2 Queues, Sequential (No synch): 64839 microsec
1 Queue, 1 Thread (No synch): 35528 microsec
2 Different Queues, 2 Threads (No synch): 38123 microsec
1 Same Queue, 2 Threads (No synch, Buggy!): 38097 microsec
1 Same Queue, 2 Threads (Lock Guard): 56024 microsec
```

This shows that two threads running with lock guard synchronization adds about 47% overhead versus without synchronization, although admittedly it removes the bugs. So, it is serializing approximately half of the second thread's execution, which is presumably the amount of time it is blocked waiting for a lock.

So, here we have coarse-grained locking significantly increasing the time cost, because the lock guards effectively serialized most of the interface to our queue. We can partially solve this by changing the getter to not use locks, because it's probably atomic, and tweaking the lock guard to release the lock slightly early.

The full solution to the cost overhead: fine-grained locking!

Fine-Grained Locking

Fine-grained locking is using locks over shorter sequence blocks of code or smaller parts of a data structure.

The general ideas are:

- More locks
- Smaller portions of data locked
- Shorter duration lock holds

Some of the goals of finer granularity locking include:

- Lock contention improved
- Reducing thread blocking delays (less waiting for a lock)
- Allowing multiple concurrent readers (shared read lock)

In the above example, it's difficult to insert granular locks into the queue data structure, because it's a builtin standard container, where we cannot easily modify the code. However, we can certainly apply fine-grained locking approaches to our own hand-coded containers. Some of the methods to get finer granularity of locking include:

- Lock durations — acquire locks late, release locks early.
- Granular locks — multiple locks on parts of data structures.
- Read-write locking — shared reader versus unique writer locks.
- Lock-free programming — using atomics instead of mutexes.

Some of these approaches are now discussed, and some are also covered in other chapters.

Granular Data Structure Locking

Whereas coarse-grained locking has one mutex for the entire data structure or container object, fine-grained locking uses many more mutexes or locks. The first point about implementing these strategies is to get used to the idea that mutexes and locks are just objects. Hence, we can use:

- Arrays of mutexes and locks
- Mutex or lock object data members

Hence, we can put mutexes or other locking objects inside our other objects, or part of containers, or whatever we want to do. Hence, we can use much more granular approaches that achieve the benefits of fine-grained locking: The idea is to use many locks such as:

- Locks for each individual node in a container.
- Locks for sub-parts of the data structure.

Locking each node in a data structure is as fine-grained as it is possible to go in a container. The idea is that a writer thread can modify other elements in the container, as long as it's not changing the one that you're using.

This can be effective at avoiding lock contention, as other threads would rarely be blocked, but does increase lock overhead for every object.

A less fine-grained approach is to use fewer locks, but still maintain multiple locks per container. Some examples of using fewer locks than per-node, but still having many locks for portions of a data structure include:

- Linked list sub-lists with locking from its sub-head node.
- Binary tree with locking used on a subtree.
- Hash table with locks for each bucket chain.

The idea with these methods is to avoid blocking other threads for every access to the entire container. For example, if your hash table has an array of mutexes, one per bucket, then readers and writers are only in contention for elements that map to the same hash bucket. This reduces lock contention, as it's a relatively rare event.

Lock Striping

Another variant of this approach is called "lock striping," and is a trade-off between the number of mutex objects and lock contention. The idea is to map all our data to a smaller number of mutexes. For example, in a hash table with a thousand buckets, rather than also using a thousand mutexes, we can use many fewer, and map the buckets to mutexes. The idea is like this in our container template:

```
    T key[NBUCKETS] hashtable_;
    std::mutex[NLOCKS] lockarr_;
    // ...
    size_t bucket = hash_function(key);
    size_t lockoffset = bucket % NLOCKS;
```

Here, we could have a hash table with 1,000 buckets in the hash table, but only 10 locks. This is a tenfold reduction in lock contention compared to the coarse locking approach of one lock per data structure.

Lock striping reduces the number of mutexes required, but will slightly increase lock contention compared to the granular approach of having one mutex per bucket. I'm not sure that I recommend lock striping in this example, because the advantage of using fewer mutexes for our hash table is mainly space reduction rather than speed, and don't we have plenty of that? On the other hand, there is extra cost per lock in terms of initialization (mutex constructors) and shutdown (mutex destructors), so lock striping can reduce this cost compared to fully fine-grained locking.

Lock Segmenting

Lock segmenting is another middle-of-the-road approach, with similar ideas to lock striping, in the sense that it uses fewer locks than data points. The idea is to have one lock per "segment" of the data structure being used. This has particular applicability to linear data structures, such as vectors and arrays, in areas such as linear algebra and AI engines.

If we have a vector of data, we often want algorithms to operate on "segments" of that data, so as to maintain cache locality advantages in a CPU architecture. Note that a GPU architecture prefers a striped approach, but that's in CUDA C++ with a totally different type of on-GPU threading model, not in C++ multithreading.

Doing data processing fast with cache-aware multithreading means that each thread operates on a segment of contiguous data, and we have a controller thread that's scheduling different threads to work on segments of the vector. Here's the idea of a hand-coded vector container that's segmenting the locks according to the data:

```
// Template: NARRAY = size, NLOCKS = lock granularity
float arrdata_[NARRAY];
const int NSEGMENTS = NARRAY / NLOCKS;
std::mutex lockarr_[NLOCKS];
static_assert(NARRAY % NLOCKS == 0);  // avoid extras

size_t map_offset_to_lock(size_t offset) {
    assert(offset < NARRAY);
    size_t lock_offset = offset / NSEGMENTS;
    assert(lock_offset < NLOCKS);
    return lock_offset;
}
```

Hence, any code that's working on a segment of the array, does this to figure out which mutex to acquire:

```
void process_segment(size_t offset) {
    size_t lock_offset = map_offset_to_lock(offset);
    lockarr_[lock_offset].lock();
    // ... etc
    lockarr_[lock_offset].unlock();
}
```

The idea is similar to lock striping, but this uses division rather than modulus.

Nearby offsets in lock segmenting will usually get the same lock, as they're in the same segment of contiguous data, whereas adjacent elements would get different mutexes in the lock striping approach. The advantage of lock segmenting over lock striping is that it allows contiguous data processing, whether reading or writing, and therefore has cache locality efficiency.

Higher-Level Concurrency Problems

Note that higher-level concurrency issues can occur with these approaches, such as lock striping or lock segmenting. The problems arise if you have whole-of-data algorithms, which limit the value of these middle-level locking ideas. For example, consider if you have two high-level methods that work on the entire vector of data:

- Sum vector — calculate the sum of the whole vector of data, or some other linear algebra metric like a dot product (reader).
- Scale vector — multiply the entire array by a factor (writer).

If both of these methods acquire locks one segment at a time (or striped), then the segment-level operations are going to get interleaved. For example, one of the segments being summed might get modified by the scaling method, before getting summed, so the sum returned has only calculated results properly on half the elements.

There's nothing wrong with the concurrency at the segment level, but the application-level logic is broken. The concurrency solutions at a higher-level are not pretty:

- Vector-level lock to serialize all whole-of-vector algorithms (a read-write lock), or
- Acquire all of the many locks for each segment or stripe (ugh!).

It's not quite that bad, since we'd use read-write locking, so that multiple reader algorithms could still run concurrently on the entire array. However, writer algorithms would get totally serialized on this approach, blocking all other readers and writers, which is not optimal.

More efficient would be to have a more complex scheduling algorithm, so that the "scale vector" method runs in a pipelined fashion, processing one segment behind the "sum vector" method, but that's tricky to do if each such segment is running in a different thread. However, if you don't do this, they're potentially going to interfere with each other at the higher-level, creating actual bugs in the application logic, despite being correctly synchronized at the segment level.

Read-Write Locking

An important improvement to lock contention is to allow multiple readers to access concurrently, but any "writer" must have unique access. This idea can be used for both coarse and fine-grained locking, and can be combined with moderate approaches like lock striping or lock segmenting. The goals of the read-write lock approach are:

- Multiple readers at the same time (but not any writers).
- Every writer needs exclusive control (no other readers or writers).

This is efficient in situations where there are lots of readers processing the data, and fewer writers. However, it can be less successful where readers and writers are accessing the data structure with approximately the same frequency, such as passing work on a queue in the producer-consumer model. Actually, in that model, both the producers and consumers are writers (not just readers), as they each push or pop the queue. You can make consumers into readers by using a delayed-pop idea, but eventually someone has to clean up the mess.

The standard C++ library has builtin support for achieving read-write locking. The way to achieve this is with a "`std::shared_mutex`" instead of a basic mutex. The changes to our code can be summarized:

- `std::shared_mutex` is now used in all of the class member functions (using `<shared_mutex>` header file).
- Readers request a `std::shared_lock` over the shared mutex (multiple concurrent readers).
- Writers request a `std::unique_lock` over the shared mutex (exclusive access).

Here's the modified code in full:

```
 #include <shared_mutex>

template<typename T>
class QueueWrapReadWrite {
  private:
    std::queue<T> m_q;
    std::shared_mutex m_mutex;  // Read-write
  public:
    int count() const {  // Reader
        std::shared_lock<std::shared_mutex> lock(m_mutex);
        return m_q.size();
    }
```

```
        T front() { // Reader
            std::shared_lock<std::shared_mutex> lock(m_mutex);
            return m_q.front();
        }
        void pop() { // Writer
            std::unique_lock<std::shared_mutex> lock(m_mutex);
            m_q.pop();
        }
        void push(T t) { // Writer
            std::unique_lock<std::shared_mutex> lock(m_mutex);
            m_q.push(t);
        }
    };
```

Here is the comparison of speed against the basic lock guard version with non-concurrent readers:

```
    1 Queue, 2 Threads (Lock Guard): 55214 microseconds
    1 Queue, 2 Threads (Read-Write): 51687 microseconds
```

There was about a 6.4% improvement by adding shared reader locking.

This makes sense, because most of the testing involves write operations of `push()` and `pop()`, but there is a small gain in concurrency from the read-only `count()` and `front()` operations.

But overall, it's still quite a lot of overhead, when you recall the lock guard version was 47% extra CPU time compared to without synchronization.

Maybe we should try a lock-free version.

References

1. Illinois Tech, May 2025 (accessed), Locks and locking strategies, https://moss.cs.iit.edu/cs450/slides/09-concurrency-c.pdf
2. Niklas Fors, December 5, 2013, Coarse-grained and fine-grained locking, https://fileadmin.cs.lth.se/cs/Education/EDA015F/2013/Herlihy4-5-presentation.pdf
3. Martin Fowler, 2003, Coarse-Grained Lock, https://martinfowler.com/eaaCatalog/coarseGrainedLock.html
4. Adam Belay, 2019, Locking, https://pdos.csail.mit.edu/6.828/2019/lec/l-locks.pdf

34. Core Pinning

What is Core Pinning?

Core pinning is a multithreading optimization where a single thread is "pinned" to one of the cores to give it higher priority. This means that important thread that runs the hotpath can have guaranteed CPU availability, rather than waiting for the default thread scheduling algorithms. Hence, core pinning can be a solution to avoid lock contention worries or excessive context switch in the main hotpath thread.

Core pinning is also called "thread affinity" and has multiple other names (e.g., "processor affinity" or "CPU affinity" or "CPU pinning"), but if you hear the words "pinning" or "affinity" in relation to threads, this is it.

Pinning has other meanings in related hardware architectures. There's a higher-level type of pinning whereby whole processes or applications are pinned to a CPU core by the operating system, rather than just a single thread, which isn't quite the same thing. Note also that CUDA C++ has another type of "pinned memory" for GPUs, but that's a memory upload optimization rather than a compute improvement.

The other side of core pinning is that you obviously don't pin the less important threads. All the lower-priority threads have fewer cores available, and are downgraded.

Pros and Cons

The use of core pinning is a very powerful type of hotpath optimization. The main pathways are super-optimized because of these factors:

- No context switches
- Fewer cache misses (no invalidated caches)
- Highest priority execution
- Guaranteed core availability (no delay)

The downsides are fairly obvious:

- That core isn't available for other work.
- Load balancing only available on the other cores.

And also, you can't do it too many times, because the CPU hardware only has a fixed number of cores.

Counting Cores

The code to set up core pinning is really a two-part procedure with these steps:

1. Determine how many CPU cores are available.

2. Pin a thread to one of them.

There are various non-standard ways to interrogate the system for its CPU settings. The standard method is to call the `hardware_concurrency()` function in the standard thread library, which tells you how many physical cores are in the CPU.

```
int number_of_cores()
{
    return std::thread::hardware_concurrency();
}
```

This has been a standard method since C++11, so it should be available to you. Alternatively, non-standard methods include:

- `sysconf()` — POSIX version in `<unistd.h>` for Linux.
- `GetSystemInfo()` — Win32 API in `<windows.h>`.
- `__cpuid()` — low-level intrinsic function in `<cpuid.h>` that wraps the `CPUID` machine instruction on x86 CPUs (Intel/AMD).

All of these functions offer a whole wealth of other hardware information about the CPU, rather than just the number of cores.

Setting Up Core Pinning

There's no language-supported standard way to set up core pinning using the C++11 `std::thread` library, nor does anything appear forthcoming in C++26 for this area. However, there are longstanding platform-specific functions to do this.

Sometimes, you don't need to code up core pinning in C++, but can use OS settings or commands. On Windows, you can set up a process-level CPU pinning for an application via the GUI. On Linux, there is a "`taskset`" command that allows running a program with core pinning.

Both Windows and Linux have non-standard C++ system calls that can set up core pinning for either a process or a thread. Linux uses the "pthreads" library to do core pinning, and Windows has some Win32 features. The sequence at a high-level:

1. Get a native thread id

2. Call the platform-specific core pinning API.

To implement core pinning in C++ on Linux you need to bypass `std::thread` to get to the underlying POSIX thread id, which has type `pthread_t` as defined in `<pthread.h>`. This is required because all the core pinning calls are POSIX functions on Linux. There are at least two ways to do this:

- `pthread_self()` — POSIX call to return the id of the current thread.
- `std::thread::native_handle()` — returns the "native" thread ID of a standard C++ thread object, which is a POSIX thread id on Linux.

Once you have a valid thread id, then you can set up core pinning for that thread. The programmatic C++ APIs on Linux are:

- Pin processes — `sched_setaffinity`
- Pin threads— `pthread_setaffinity_np`, `pthread_attr_setaffinity_np`

On Windows, these are the C++ APIs:

- Pinning processes — `SetProcessAffinityMask()`
- Pinning threads — `SetThreadAffinityMask()`

Now let's look at a full example on Linux.

Linux Core Pinning

Here's a native pthreads sequence to pin the current thread to a core:

```
#include <pthread.h>
#include <unistd.h>
#include <sched.h>

bool pin_me(int corenum)
{
    pthread_t tid = pthread_self(); // Get current thread id
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);          // Clear all core bit flags
    CPU_SET(corenum, &cpuset);  // Set one core bit flag
    // Pin the thread!
    int ret = pthread_setaffinity_np(tid,
                            sizeof(cpuset), &cpuset);
    return ret == 0;  // Zero return is success
}
```

Note that failures can occur when attempting to pin a thread to a core. The process needs adequate permissions to do so, and the core number needs to be valid for the given system.

This code uses "`cpu_set_t`" from `<sched.h>`, which is a bitmask (or other data structure) that represents a mask of one or more cores. There are various bit manipulation macros also defined in `<sched.h>` for use with this bitmask type:

- `CPU_ZERO()` — clears all the bits.
- `CPU_SET()` — sets one bit.
- `CPU_CLR()` — unsets one bit.
- `CPU_ISSET()` — tests one bit.
- `CPU_COUNT()` — counts how many bits are bit.

There are also some arithmetic operations on the CPU bit sets in `<sched.h>`:

- `CPU_EQUAL()` — test if two bitsets are equal.
- `CPU_AND()` — bitwise-and on all bits.
- `CPU_OR()` — bitwise-or on all bits.
- `CPU_XOR()` — bitwise-xor on all bits.

The CPU bitmask type `cpu_set_t` is not a C++ object, but a raw C-like structure, which means it can be copied or moved by bitwise copy using `memcpy`.

Note that `pthread_setaffinity_np()` can be passed a CPU set with more than one bit set, in which case the thread will be migrated to one of those cores. You can also examine the bitmasks via `pthread_getaffinity_np()`.

Isolating Linux Cores

To fully implement core pinning of a thread to a particular core on Linux, some further actions may be needed. Changes are required to Linux kernel settings to do things like:

- Isolating the core
- Disabling interrupts

Some of the Linux kernel parameters you may need to adjust include:

- `nohz` or `nohz_full`
- `isolcpus`
- `irqaffinity`
- `rcu_nocbs`

There is some industry wisdom to avoid core zero on Linux systems, because that's the CPU core that the kernel always tries to run system tasks on, as described in Bernhardt (2023). There's also a discussion of some odd issues with core 1 on Linux in Dawson (2023).

References

1. Machinet, March 13, 2024, *How to optimize C++ code for use in high-frequency trading algorithms?* https://www.machinet.net/tutorial-eng/optimize-cpp-code-high-frequency-trading-algorithms
2. Dung Le, Aug 13, 2020, *Optimizations for C++ multi-threaded programming*, https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
3. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
4. Eli Bendersky, January 17, 2016, *C++11 threads, affinity and hyperthreading*, https://eli.thegreenplace.net/2016/c11-threads-affinity-and-hyperthreading/

5. Bytefreaks, 23 November 2016, *C/C++: Set Affinity to process thread – Example Code 3*, https://bytefreaks.net/programming-2/c/cc-set-affinity-to-process-thread-example-code
6. Mark Dawson, Jr., February 12, 2023, *My Fear of Commitment to the 1st CPU Core*, https://www.jabperf.com/my-fear-of-commitment-to-the-1st-cpu-core/ (avoiding core 1 for CPU affinity).
7. Manuel Bernhardt, 16 Nov, 2023, *On pinning and isolating CPU cores*, https://manuel.bernhardt.io/posts/2023-11-16-core-pinning/ (examines costs of arithmetic operations versus cache mispredictions and context switches).
8. Davood Ghatreh Samani, Chavit Denninnart, Josef Bacik, Mohsen Amini Salehi, 3 Jun 2020, *The Art of CPU-Pinning: Evaluating and Improving the Performance of Virtualization and Containerization Platforms*, https://arxiv.org/abs/2006.02055
9. Kernel.org, May 2025 (accessed), *The kernel's command-line parameters*, https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html

35. False Sharing

False Sharing and Cache Line Sizes

False sharing is a slug in C++ multithreaded code preventing two threads from running as fast as they should. The idea of "false sharing" is that two threads can interfere with each other's memory caching. The sharing is "false" because it can occur with data that's not actually being intentionally shared between the threads, but is impeded simply because the memory addresses are too close together.

Why does it occur? The CPU's L1 and L2 caches don't just cache in single bytes, 16-bit words, or even 32-bit integers. Instead, they have caching in "chunks" in the hardware level, which are called "cache lines" (also "cache sectors" or "cache blocks" or "cache line sizes" or "bananas in pyjamas" if you prefer).

How big? Some examples of common sizes of these cache lines include:

- Intel CPUs — 64 bytes.
- Apple M2 — 128 bytes.
- Some AMD and other CPUs — 256 bytes.

Note that you can get this number for the L1 cache line size in bytes programmatically in C++17 via functions declared in the `<new>` header:

- `hardware_destructive_interference_size()`
- `hardware_constructive_interference_size()`

What this means is that, on an Intel CPU, the caches are updated 64 bytes at a time, because one "cache line" is read or written as the minimum size. This is good because:

- Cache loads are 64 bytes in parallel (in hardware).
- Cache writes (updates) store 64 bytes in parallel.

But this is bad because:

- Invalidating one cache byte also invalidates all 64 cache line bytes.

This is where we have a slowdown from false sharing. If one thread sets any value in a 64-byte cache line, then all of the other 63 bytes are also invalidated in the cache. If a second thread needs to use any of those other 63 bytes, then it needs a cache line refresh. Slowness ensues.

Example of False Sharing

A common example would be two integers, each 4 bytes in size, but close together so that they sit inside the same 64-byte cache line. The most common problems arise with atomics or mutexes close together, but they can affect any global variable.

Hence, first a simple example without any atomics, mutexes, or other thread synchronization. Let's just look at two threads that are updating their own global variable, with no overlap between the threads. In theory, these two threads should not affect each other at all. In reality, there are CPU cache lines.

Here are our two global counter variables:

```
int g_counter1 = 0;
int g_counter2 = 0;
```

In practice, false sharing is more likely to occur with two atomics declared close together. However, in this example we're just testing with two completely unrelated threads, with absolutely zero synchronization happening between them. They really shouldn't impact each other, if not for false sharing.

Here is the sequential code, which sets two global variables:

```
void runtest1_no_threads(int n)
{
   for (int i = 0; i < n; i++) {
      g_counter1++;
   }
   for (int i = 0; i < n; i++) {
      g_counter2++;
   }
}
```

Here are the two threads that aim to set those two global variables in parallel. Note that each thread only accesses one variable, without any "sharing" going on.

```
void thread1(int n)
{
   for (int i = 0; i < n; i++) {
      g_counter1++;
   }
}

void thread2(int n)
{
   for (int i = 0; i < n; i++) {
      g_counter2++;
   }
}
```

And here's the basic thread launching code:

```
void runtest1_threads(int n)
{
   std::thread t1(thread1, n);
   std::thread t2(thread2, n);
   t1.join();
   t2.join();
}
```

Finally, here is the timing code using `<chrono>`:

```
g_counter1 = g_counter2 = 0;
auto before = std::chrono::high_resolution_clock::now();
runtest1_no_threads(n);
auto now = std::chrono::high_resolution_clock::now();
auto diff = std::chrono::duration_cast
    <std::chrono::microseconds>(now - before).count();
std::cout << "Time (no threads): "
          << diff << " microseconds" << std::endl;
```

Here are the speed results from executing the sequential and threaded code for 100 million iterations using g++ on Linux.

```
Time (no threads): 256079 microseconds
Time (2 threads): 209341 microseconds
```

Note that the threaded code does not actually run twice as fast as the sequential code, despite having two threads that should run in parallel. In fact, it only improves on the sequential code by about 19%, rather than 50%. Why?

It's the magic of false sharing, whereby one thread writing to its variable slows down the other unrelated variable that's only being used by the other thread. The two threads are constantly writing to their own variable, which messes with the cached value of the other global variable used in the other thread. It's kind of like entanglement in quantum physics, if you like that kind of thing.

Detecting False Sharing

According to the documentation, Valgrind's DRD tool should be able to detect false sharing (and numerous other thread errors). However, I ran the command:

```
    valgrind --tool=drd ./test1
```

I did not get any warnings:

```
    ==8618== ERROR SUMMARY: 0 errors from 0 contexts
```

On closer reading of the DRD documentation, DRD seems to only detect a false sharing situation if the two threads are running on different cores, which may have been the reason.

Solutions for False Sharing

There are a few coding solutions to prevent false sharing. The basic idea is ensuring that the addresses of unrelated thread-shared global addresses are not too close. Options include:

- Putting global variables in random spots throughout your C++ code.
- Using `alignas` to enforce address spacing on alignment boundaries.

The first one is kind of a joke, although it would probably work in most cases. However, it's not technically guaranteed where the linker will put unrelated global variables in the address space.

A more elegant solution is to put variables, especially atomics, on address alignment boundaries. The idea is to ensure that each important global variable is alone in its 64-byte block.

The global variables in our declarations become:

```
alignas(64) int g_counter1 = 0;
alignas(64) int g_counter2 = 0;
```

By declaring them both as `alignas(64)`, it guarantees two things:

- The variables start on a 64-byte alignment boundary (we don't care about this here), and
- They are the only variable in that 64 bytes (this fixes false sharing).

The downside is that each 4-byte integer is stored in 64 bytes, so there's a total 60 bytes of unused padding added to global memory usage. But it's better to pad memory than to waste CPU cycles! (On the other hand, the CPU cache lines are also loading and storing 60 unused bytes, so we've somewhat undermined the efficiency advantages of the L1/L2 cache lines for this 64-byte block.)

Anyway, who cares, it works! Here are the faster speed measurements just from adding `alignas` statements:

```
Time (no threads): 260277 microseconds
Time (2 threads): 133947 microseconds
```

Wow! It's almost exactly half the time! The performance gain is about 49%, which is much better than 19% (due to false sharing slowdowns), and is close to the 50% gain we were aiming for with two threads. Maybe there's something to this multithreading stuff, after all.

Some Final Tweaks

As a finesse, you can assure that the addresses are far enough apart by simply checking in code. One possible method to make sure that some junior code jockey hasn't deleted your `alignas` statements:

```
assert( (char*)&var2 - (char*)&var1 >= 64);
```

Unfortunately, you can't do it faster at compile-time, since addresses of global variables are not "constant" enough for the compiler:

```
static_assert((char*)&var2-(char*)&var1>=64); // Fails
```

Note that some CPUs have cache line sizes up to 256 bytes. Hence, you might need `alignas(128)` or `alignas(256)` on those platforms.

Note also there are various other non-standard ways to achieve alignment, most of them having existed on platforms prior to the `alignas` specifier in the C++ standardization. For example, GCC has a whole set of old builtins. Feel free to use those old things and charge extra because you're writing antique C++ code.

Another point is that false sharing slowdowns can arise for non-global variables, such as dynamic allocated memory or stack addresses. It's not very likely for two threads to see contention over stack addresses inside their respective call frames, but it can occur with allocated memory blocks that are shared. There are various ways to get aligned addresses inside dynamic memory allocation, including aligned memory allocation primitives, so the same ideas can solve the problem.

Nevertheless, atomics declared as global variables are probably the most likely area where false sharing can occur. This suggests a general rule: all global atomics should be declared as `alignas`. I'm not sure I agree, and it does sound a bit drastic. This does avoid the performance slug of false sharing, but it will also waste significant memory with padding bytes.

References

1. Dung Le, Aug 13, 2020, *Optimizations for C++ multi-threaded programming*, https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
2. Paul J. Lucas Jul 13, 2023, *Advanced Thread Safety in C++*, https://dev.to/pauljlucas/advanced-thread-safety-in-c-3ap5
3. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
4. Valgrind, March 2025 (accessed), *DRD: a thread error detector*, https://valgrind.org/docs/manual/drd-manual.html#drd-manual.limitations

36. Lock Contention

What is Lock Contention?

Lock contention is a multithreading slowdown where threads are blocked waiting on locks held by other threads. If your code has a lot of busy threads, any of the synchronization code (e.g., using mutexes or condition variables) can lead to contention over accesses to shared data.

Note that lock contention is not the same thing as lock overhead. Lock contention is the extent to which threads get blocked waiting for a lock. Lock overhead is the extra cost of library calls that do lock-related stuff, such as the cost of requesting a lock, releasing a lock, creating a mutex, destroying a mutex, etc.

All multithreaded applications have some level of lock contention, otherwise why would it need locks at all? Hence, optimizing to reduce lock contention is something that you can't avoid. General points about lock contention include:

- More threads means more opportunities for lock contention.
- So does having more locks (all other things being equal).
- Unpopular shared data is unlikely to cause contention.
- Fine-grain locking is desirable for often-used data.

In the worst case, you get to a deadlock situation, which upgrades the lock contention problem from a slug to a bug.

Optimizing Lock Contention

General strategies for reducing lock contention include:

- Short critical sections
- Reduce total lock requirements
- Acquire locks late
- Release locks early

Here's the best one:

- No synchronization — don't use any locks at all!

Unfortunately, the "no locks" plan has its limitations, being mostly limited to read-only data used by multiple readers. Nevertheless, your first thought should be if there's a way to do this without needing to use a lock.

Some of the specific strategies for using fewer locks or otherwise reducing contention include:

- Consider using fewer threads (so less contention for locks).
- Maximize lock-friendly data handling (e.g., "immutable" read-only data).
- Review lock granularity (fine-grain vs coarse-grain vs a hybrid strategy).
- Tolerate lockless output (e.g., out-of-order debug messages aren't so bad).
- Limit block scope of `std::lock_guard` to release the lock early.
- Use `std::unique_lock` and other variants for more flexibility.
- Copy data to temporary variables to release locks before processing data.
- Use queues as the preferred method to transfer large amounts of data.
- Avoid false sharing (can impact lock contention issues).
- Release locks before blocking system calls, I/O waits, or network actions.

Some examples of other advanced strategies include:

- Reader-friendly containers (e.g., versioned data structures, copy-on-write).
- Kernel bypass (for I/O efficiency).
- Double lock check method (first check without lock, then acquire the lock).
- Exponential backoff when waiting (e.g., avoiding spinlock busy waits).
- Shard or partition data across multiple threads (avoids need for locks).
- Use message-passing via `std::promise` and `std::future` rather than shared memory.
- Thread-specific queues and "work stealing" design pattern.
- Lock-free algorithms with atomics not mutexes (very tricky to get right).

Avoid Lock Guard Delayed Unlocking

The `std::lock_guard` class is a wonderfully safe way to use mutexes, because it helps us avoid deadlocks and severe thread starvation if we forget to unlock our mutex (as if!). Unfortunately, it's too easy to use, and coders can forget to unlock.

The problem is that we can accidentally hold the lock for too long, which increases lock contention. Here's an example of the concept:

```
 std::mutex g_my_mutex;

void process_critical_data()
{
    // Step 1. Lock
    std::lock_guard<std::mutex> mylockguard(g_my_mutex);
    // Step 2. Get the data...
    // Step 3. Process the data ...
}
```

The problem is that we haven't really thought too much about where we should unlock. The above code doesn't release the mutex until after we've finished processing the data at Step 3, when the function returns, which is needlessly long.

One way to fix this would be to use some other more flexible locking wrappers that allow explicit control of the unlocking. Your basic choices are:

- `std::lock_guard` — can only unlock in its destructor (inflexible).
- `std::unique_lock` — allows an explicit `unlock` call (more flexible).

A simpler solution is to explicitly control the scoping that sets when the destructor of `std::lock_guard` triggers the release of the lock. Here's a better version:

```
void process_critical_data()
{
    {   // Step 1. Lock
        std::lock_guard<std::mutex> mylockguard(g_my_mutex);
        // Step 2. Get the data...
    }
    // Step 3. Process the data ...
}
```

This has added an extra pair of `{ }` braces around the first two steps. This triggers the scoping mechanism, so that the `std::lock_guard` destructor is called and the mutex is unlocked immediately after Step 2, at the inner right brace. Then Step 3 can process the data to its heart's content without blocking any other threads.

Fine-Grain vs Coarse-Grain Locking

Locking granularity has two basic strategies: go small or go big. Here's a summary:

- Coarse-grain — lock an entire data structure while updating it.
- Fine-grain — lock only in the exact critical code sequence that updates the data structure, deep in its internals.

The characteristics of these strategies can be summarized:

- Coarse-grain — longer duration, fewer locks overall.
- Fine-grain — shorter duration, more locks.

Fine-grain locking improves performance for data that is used often. By limiting the granularity of locking, each thread holds the lock for only a short period while performing a low-level update, so many threads can have the lock in turn.

However, fine-grain locking means frequently locks and unlocks, which involves some overhead. It also increases the overall complexity of the concurrency algorithms by needing multiple locks for small pieces of data, thereby creating greater risk of mistakes, such as an incorrect request order for multiple locks causing a deadlock.

Coarse-grain locking can reduce performance because it locks data for a longer period of time, when a broader update to a higher-level data structure is performed. The chance of lock contention for a long duration is higher than with fine-grain locking. Any thread seeking the lock is less likely to find a window to access it if the lock is frequently requested, so coarse grain locking is best for rarely-used data.

The advantage of fewer higher-level locks is simplicity. There is not only a lower risk of deadlocking errors, but also fewer chances to go wrong when ensuring concurrency is adhered to, and the access to the shared data is properly controlled. For example, when updating a large data structure with a single lock, this means that concurrency errors cannot occur at a lower level. Thus, it's easier for the thread to maintain a coherent state of the data structure, because there won't be any interleaved changes from other threads.

Hybrid locking strategy involves using a trade-off: using fine-grain locks for frequently-accessed critical sections, and coarse-grain locking for less popular data. This can be a pragmatic solution that balances speed with lower development complexity and risk mitigation.

Lock-Free Algorithms

Lock-free programming is a method of optimizing multithreaded code to avoid locks (i.e., mutexes). The advantages in speed arise from:

- Overhead of mutexes
- Lost performance from threads blocked awaiting a resource.

The main disadvantage of lock-free programming:

- Your brain will explode.

The internet is littered with articles about failed attempts to write lock-free algorithms, even by some of the best programmers. There are many ways to go wrong in the quest to get rid of mutexes.

Note that "lock-free" programming does not mean that you just search up "mutex" in vi, and then hit the "dd" button. No, lock-free programming is not just sequential programming. Instead, the idea is to switch to a faster concurrency method than mutexes, so this is the main idea:

- `std::mutex` — lock-based programming.
- `std::atomic` — lock-free programming.

The overall idea is to use an "atomic" operation instead of a mutex. To make this work, it's usually a quite complex atomic operation, such as a "Compare-And-Swap" (CAS) operation.

This is how a CAS operation works, with a number of steps all done atomically in one unbreakable sequence:

- Access a variable (that you want to set atomically).
- Compare it to the "old" or "expected" value.
- If it's equal to the old value, then successfully update to the new value (and done).
- If it's not equal to the old value, someone else has already updated it, so we fail (and then loop around and retry).

What a mouthful! Fortunately, C++ has the `std::atomic` class (since C++11) to take care of all that.

The main routines to use for a CAS instruction are:

```
    std::atomic::compare_exchange_weak
    std::atomic::compare_exchange_strong
```

Note that you will also need to know about "memory orders" around atomic primitives, as controlled via the `std::memory_order` library.

There are also a variety of non-standard methods to achieve lock-free programming with primitives in older code platforms, or in a platform-specific manner. Some of the primitives are:

- `InterlockedCompareExchange` — Win32 version in `<winnt.h>`.
- `OSAtomicCompareAndSwapInt` — iOS/Mac in `<OSAtomic.h>`
- `__atomic_compare_exchange` — older GCC version.

Note that the `std::atomic` class is not actually guaranteed to be a lock-free atomic operation on every platform. It's a good idea to test your platform using the "`is_lock_free`" primitive as part of your initialization or self-testing code:

```
    assert(std::atomic<int>::is_lock_free());
```

Thread Pools

Thread pools are a design pattern in C++ multithreading that avoids the cost of creating and destroying threads by using long-running threads. Instead of incurring this thread overhead, a "pool" of available threads have been pre-created, which sit there until work is available to be done. The main characteristics are:

- Idle threads wait for work (e.g., off a task queue).
- Threads are not destroyed after completing a chunk of work.

Thread pools are mostly used in a "producer-consumer" design pattern, although thread pools can also be used in other ways. There are effectively two thread pools in this design pattern:

- Producer thread pool
- Consumer thread pool

Typically, one or more producer threads adds work items to a queue, such as when it receives new data from a network source (e.g., exchange connection in HFT).

Another group of consumer threads is idle waiting to pull work off the queue. Consumers do the work, return the results, and then add themselves back to the group of idle consumer threads awaiting more work.

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Jeff Preshing, Jun 12, 2012, *An Introduction to Lock-Free Programming*, https://preshing.com/20120612/an-introduction-to-lock-free-programming/
3. Deb Haldar, August 17, 2017, *Top 20 C++ multithreading mistakes and how to avoid them*, https://acodersjourney.com/top-20-cplusplus-multithreading-mistakes/
4. Apple, March 2025 (accessed), *Mac OSX documentation*, https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/OSAtomicAdd32.3.html
5. Wikipedia, March 2025 (accessed), *Non-blocking algorithm*, https://en.wikipedia.org/wiki/Non-blocking_algorithm
6. Herb Sutter, September 08, 2008, *Lock-Free Code: A False Sense of Security*, Dr Dobbs Magazine (archived), https://web.archive.org/web/20150901211737/http://www.drdobbs.com/article/print?articleId=210600279&siteSectionName=cpp
7. Microsoft, 24 May, 2022, *InterlockedCompareExchange function (winnt.h)*, https://learn.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-interlockedcompareexchange
8. GNU Foundation, March 2025 (accessed), *6.26 Built-in Functions for Memory Model Aware Atomic Operations*, https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html
9. CPP Reference, March 2025 (accessed), *std::atomic::compare_exchange_weak, std::atomic::compare_exchange_strong*, https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange
10. CPP Reference, March 2025 (accessed), *std::memory_order*, https://en.cppreference.com/w/cpp/atomic/memory_order
11. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c

12. Alex McMurray, 12 February 2024, *The expert C++ programming technique you need to know for a HFT interview*, https://www.efinancialcareers.com/news/the-expert-c-programming-technique-you-will-need-to-know-for-a-hft-interview
13. Paul J. Lucas Jul 13, 2023, *Advanced Thread Safety in C++*, https://dev.to/pauljlucas/advanced-thread-safety-in-c-3ap5
14. Larry Jones, 27 Feb 2025, *Mastering Concurrency and Multithreading in C++: Unlock the Secrets of Expert-Level Skills*, https://www.amazon.com.au/Mastering-Concurrency-Multithreading-Secrets-Expert-Level-ebook/dp/B0DYSB519C/
15. Dung Le, Aug 13, 2020, *Optimizations for C++ multi-threaded programming*, https://medium.com/distributed-knowledge/optimizations-for-c-multi-threaded-programs-33284dee5e9c
16. Karthikeyan Akkapalli, Aug 25, 2024, *Multithreading in C++: Concepts, Challenges, Advanced Techniques, and Applications*, https://medium.com/@karthikeyan_akkapalli/multithreading-in-c-concepts-challenges-advanced-techniques-and-applications-b97cbdcf31c7

37. Atomics & Memory Orders

What are Atomics?

Atomic variables are a C++11 features whereby an operation on a variable can be done "atomically" and does not require any other cross-thread synchronization. The `std::atomic` library in the `<atomic>` header file exists to provide these capabilities across platforms in standard C++. Note that there's also a C version called `_Atomic`.

Atomics are mainly used to implement the "lock-free" versions of thread-safe data structures like concurrent stacks and queues. But that's the advanced stuff!

The first point is to note that atomics can implement thread-safe algorithms for much simpler requirements, such as:

- Counters
- Sums
- Maximum or minimum
- Boolean flags

Don't wrap a mutex or a lock check around a simple counter — use an atomic instead.

Standard Atomic Class

The atomic library is a templated class with pre-defined instantiations for several different types. Hence, you can use atomics with various types of variables:

```
    std::atomic<int> g_my_atomic_counter;
```

You can instantiate the atomic template with your own class types, but only if it satisfies various properties (e.g., trivially copyable). The main use of atomics is with scalar types such as integral types or pointers, which are almost always efficient.

Implicit atomics. Note that the performance of the atomic library can be very fast for simple scalar variables. On many platforms, this will just be a single machine code increment instruction on the underlying `int` variable, but on some obscure platforms it might be more complex. For example, on a lot of CPU platforms, the reading and writing of an `int` variable is implicitly atomic, because it runs in only a single CPU instruction. Hence, the members for `std::atomic<int>` might simply be a nothingburger that just accesses the integer variable underneath.

Emulated atomics. On the other hand, some platforms cannot really implement atomics properly for more complicated types, but has to use its own locking algorithms. Most C++ code using an atomic should still work either way, but this gives insight into its performance characteristics on different platforms.

To check on the status on this platform, there is the `is_lock_free()` and the C++17 `is_always_lock_free()` member function in `std::atomic` to test whether a particular instantiation is truly atomic, or whether the library has to emulate atomicity using hidden locks and mutexes. The first tests whether a particular variable is lock free, and the second is whether that type of atomic is always lock-free, which is a hair-splitting difference, but occasionally matters.

Atomic type aliases. If you get tired of typing the angle brackets for the template instantiation, there are some handy type aliases available since C++11, such as:

- `atomic_int`
- `atomic_short`
- `atomic_bool`
- `atomic_size_t`

There's a lot more, but I'm sure you get the idea.

Basic Atomic Operators

Integer types are particularly well-supported by the atomic library. In simple cases, you can use the atomic variable in a way that mimics its use for the underlying type. You can access the integer value of the above atomic just by using its name, and use various operator overloads that the atomic library provides for each type, such as assignment and increment.

For example, if you wanted to track a counter of things happening across multiple threads, you could just do this in every thread using a global-scope atomic variable:

```
    g_my_atomic_counter++; // incremented atomically
```

The unary operators defined for atomics on integer types include:

- Prefix and postfix ++ (increment)
- Prefix and postfix -- (decrement)

There are also various binary operators:

- Assignment (`operator=`)
- Extended assignment (e.g., `operator+=`)

Note that although there are not explicitly defined overloads for common binary operators (e.g., + or −), you can simply use the name of the atomic variable in such expressions, and it should get treated as an integer, via the overloaded type cast operator to the underlying type.

Don't move or copy atomics. Although you can do various operations on the variable wrapped by an atomic, you technically cannot copy or move the entire atomic object itself. It has deleted both copy and move versions of constructor and assignment operator.

Advanced Atomic Operations

An atomic variable is guaranteed by the C++ library to be performed as a single indivisible operation. However, there are cases where you want more control over the operation on the atomic, and also additional features that control reads and writes to the variable. Some of the more complex methods available for atomic variables include:

- `load()` — get the value (atomically).
- `store()` — write a value to the variable.
- `exchange()` — store a new value, and return old value.

These methods also have the ability to define a "memory order" for synchronization with other reads and writes to the variable.

This is a complicated issue in synchronizing atomics across multiple threads for lock-free programming.

There are more complicated arithmetic operations with similar features. Some of the useful operations that you can perform include:

- `fetch_add()` — addition
- `fetch_sub()` — subtraction
- `fetch_max()` — maximum (C++26)
- `fetch_min()` — minimum (C++26)

There are also the binary bitwise operations (since C++11) only for atomics of integral types:

- `fetch_and()` — bitwise-and operation
- `fetch_or()` — bitwise-or
- `fetch_xor()` — bitwise-xor

Atomic flags. The C++11 library also included a class of `std::atomic_flag`, which is useful for concurrency. This is a simple interface that mimics synchronization capabilities of mutexes and condition variables. It's simpler than defining your own versions using the basic `std::atomic` class with a scalar type.

C++20 Atomics

C++20 adds some extra member functions to `std::atomic` that give it new functionality that sounds a lot like a condition variable or a spinlock. The goal of adding these newer C++20 features was improved efficiency over similar synchronization methods. The members are:

- `wait()` — blocking call to wait until an atomic changes.
- `notify_one()` — notify one waiting thread.
- `notify_all()` — notify all the threads that are waiting.

These primitives allow a thread to wait for an atomic to change, which is a blocking call until its value changes (there are no spurious returns where the value has not changed). The notification methods allow for one or all threads to be signalled about a change to an atomic.

There's also some useful type aliases that can help pick the most efficient type of atomic on a platform. These types are declared in C++20:

- `atomic_signed_lock_free`
- `atomic_unsigned_lock_free`

Memory Orders

Memory orders are a feature of advanced atomics that is also defined in `<atomic>`. The goal is to help interleave atomic operations with other atomic or non-atomic arithmetic in a way that does not cause race conditions or other synchronization failures. The enumeration `std::memory_order` defines constants for a number of "memory orders" that can be used in atomic operations.

Simple atomics don't require any fancy memory orders. You don't really need to worry about memory orders for the very simple uses of atomics such as counters, which default to the safest and most restrictive memory order. But memory orders are critical for implementing advanced lock-free data structures with atomics.

The idea of memory orders is to block the compiler from doing some reordering optimizations that will break your code. If you don't set any particular memory order, then the default memory order is used, which is "sequential consistency" and has these properties:

- The most restrictive memory model — blocking the optimizer.
- The safest — least likely to cause concurrency bugs.
- The slowest — compiler reordering optimizations are blocked.

The definition is `std::memory_order_seq_cst` from `<atomic>`. It's not very readable, but I guess no-one on the standards committee wanted to type "sequential consistency" in their code.

There are a number of memory order constants that you can use. Here's a list to help confuse the matter:

- `std::memory_order_relaxed` — "relaxed" (the least restrictive, fastest, and riskiest).
- `std::memory_order_acquire` — "acquire" (restricts memory reads).
- `std::memory_order_release` — "release" (restricts memory writes).
- `std::memory_order_consume` — "consume" (affects dependent operations).
- `std::memory_order_acq_rel` — "acquire-release" (both reads and writes).
- `std::memory_order_seq_cst` — "sequential consistency" (default, most restrictive, safest).

What do they do? Umm, nobody really knows, so just use whatever AI suggests. Let's move on to the next chapter.

Using Memory Orders

If you're still here, here's the first point: you don't define an atomic variable with a specific memory order. Rather, the memory orders are passed as optional parameters for the major atomic operations:

- `load()` — get the value of an atomic variable.
- `store()` — set an atomic variable.

Every operation on an atomic can choose a memory order. Here's the overall sliding scale of options available to you:

- Relaxed — bugs.
- Sequential consistency — slugs.

Or you can choose something in the middle if you really know what you're doing. Pay your money and take your chances.

Relaxed Memory Order

The "relaxed" mode doesn't do much. It's pretty chill about whatever the compiler wants to do, and there are no constraints applied to the optimizer. Hence, it's the fastest and most unsafe, where the compiler is "relaxed" but "stressed" is the programmer's mode.

Using the relaxed mode is a significant optimization, so it pays to consider when you can get away with it. Some of the simpler uses of atomic variables for counters or flags don't need any memory synchronization at all. Let's declare some atomics:

```
std::atomic<int> g_atomic_counter;
std::atomic<bool> g_atomic_shutdown_flag;
```

The question is whether there are any other dependent variable reads or writes happening around your operation on the atomic variable. Examples where this is the case include:

- Basic atomic counter
- Global flag for all threads

If you're using an `atomic<int>` variable as a counter of something, it's quite possible that nothing depends on it.

You want every thread to be able to increment the counter (without losing one), but this is guaranteed by atomic semantics. The default is "sequential consistency" for this:

```
    g_atomic_counter++;
```

But it might actually be faster to do this in "relaxed" mode:

```
    g_atomic_counter.fetch_add(1,
std::memory_order_relaxed);
```

Another example is our global "shutdown" flag that tells all the threads to close up shop. As an atomic, we can directly assign it, which uses the "sequential consistency" memory order:

```
    g_atomic_shutdown_flag = true;
```

There aren't really any dependent operations on this flag, other than the threads occasionally check it. Note that an atomic flag like this doesn't do any signalling by default, so we're assuming that other threads are watching, or getting signalled another way. In any case, we can probably use "relaxed" mode to set our atomic flag:

```
    g_atomic_shutdown_flag.store(true,
                                 std::memory_order_relaxed);
```

We might also want to test `std::atomic_flag`, to see if it's any faster, since it's a pre-defined class with similar semantics.

Load and Store Memory Orders

The atomic load() and store() operations allow a memory order to be specified. Both of them default to "sequential consistency" (slow and safe), if no memory order argument is specified.

The alternative memory orders are quite limited for these primitives, because some memory orders cause undefined behavior. In addition to the default "sequential consistency" memory order, the options for a more efficient memory order are:

- `load()` — "consume" or "acquire" or "relaxed"
- `store()` — "release" or "relaxed"

Undefined Behavior

There are some memory orders that are simply incorrect, and lead to "undefined behavior" according to the C++ standard. Some examples include:

- `load()` — memory orders that are undefined:

  `memory_order_release` and `memory_order_acq_rel`

- `store()` —memory orders that are undefined:

  `memory_order_consume`, `memory_order_acquire` and `memory_order_acq_rel`

Note that "acquire-release" memory order cannot be used at all with these methods.

Data Hazards are not Memory Orders

You may have heard of an ordering issue called "data hazards" that includes problems such as:

- Read-After-Write (RAW)
- Write-After-Write (WAW)
- Write-After-Read (WAR)
- Read-After-Read (RAR) (harmless!)

However, data hazards are not actually related to memory ordering, nor even to multithreading. Instead, data hazards are a pipelining issue inside the CPU's instruction scheduler related to "instruction reordering" and "out-of-order" execution. There are many similar concepts in terms of the different orders that can cause problems, but memory orders are in multithreading of multiple threads, whereas data hazards are inside the CPU related to the instruction ordering within a single thread.

Hence, data hazards can be delegated to the hardware engineers, and us C++ programmers have one less thing to worry about!

Extensions

1. Explore the use of `std::atomic<bool>` versus the convenient alternative `std::atomic_flag` in modern C++.
2. Examine the performance of `std::atomic` for various types, examining the costs of primitives such as locking and unlocking, along with basic class operations such as construction, destruction, copying and moves.
3. Research the details of all the various memory orders.

References

1. Emily Dawson, April 2025, *Multithreading with C++: Parallel Programming Guide*, https://www.amazon.com/dp/B0F494Z76L/
2. Sourav Ghosh, July 2023, *Building Low Latency Applications with C++*, Packt Publishing, https://www.amazon.com/dp/1837639353
3. CPP Reference, May 2025 (accessed), *std::atomic*, https://en.cppreference.com/w/cpp/atomic/atomic
4. CPP Reference, May 2025 (accessed), *std::memory_order*, https://en.cppreference.com/w/cpp/atomic/memory_order

38. Lock-Free Data Structures

What are Lock-Free Data Structures?

Lock-free programming is a method of optimizing multithreaded code to avoid locks (i.e., mutexes) by using atomics instead. Mutexes have a significant overhead, whereas atomics are more efficient, but that's not the only benefit. The advantages in speed and lower latency arise from reducing:

- Overhead of mutexes and lock guards
- Lock contention overhead
- Lost performance from threads blocked awaiting a resource.
- Context switches (avoided)

Generally speaking, there should be a higher throughput with none of the threads blocked to wait, which also avoids context switching. Threads can execute an atomic operation and keep going, which is better for CPU utilization, assuming there is enough work needing to be done.

Lock-free algorithms also have some safety and resilience advantages. Since the threads no longer block waiting for locks, this avoids some common pitfalls in multithreading:

- Lower risk of deadlock or livelock
- Reduced chance of priority inversion

The main disadvantage of lock-free programming:

- Your brain will explode.

The internet is littered with articles about failed attempts to write lock-free algorithms, even by some of the best programmers. There are many ways to go wrong in the quest to get rid of mutexes.

There are actually some real downsides to lock-free programming, and it's not an automatic performance win.

Some issues include:

- Load balancing properties can change or worsen if no thread ever blocks.
- Lock-free primitives are not always faster than mutexes or other lock types.
- Low-contention applications may perform worse under lock-free methods.
- Weirdly, overall lock contention suffers if nobody ever gets swapped out.

And worst of all, errors in coding the complex lock-free algorithms can not only cause bugs, but can also introduce insidious slugs!

Implementing Lock-Free Methods

Lock-free programming is the hardest part of multithreading. If you can do this, you can do anything. But the reverse also applies: if you're still struggling to do other types of multithreading, don't try to do this yet. To do lock-free programming, you really need to understand:

- Overall locking strategies (mutexes, locks)
- Atomics (basic usage)
- Memory orders (in relation to atomics)

Note that "lock-free" programming does not mean that you just search up "mutex" in vi, and then hit the "dd" button. No, lock-free programming is not just sequential programming. Instead, the idea is to switch to a faster concurrency method than mutexes, so this is the main idea:

- `std::mutex` — lock-based programming.
- `std::atomic` — lock-free programming.

The overall idea is to use an "atomic" operation instead of a mutex. However, it is not adequate to use simple atomic operations, but you need to use the "compound" operations. Hence, to make this work, it's usually a quite complex atomic operation, such as a "Compare-And-Swap" (CAS) low-level operation or a "Fetch-and-Add" computation.

Let's examine the "Compare-And-Swap" approach in detail.

This is how a CAS operation works, with a number of steps all done atomically in one unbreakable sequence:

- Access a variable (that you want to set atomically).
- Compare it to the "old" or "expected" value.
- If it's equal to the old value, then successfully update to the new value (and done).
- If it's not equal to the old value, someone else has already updated it, so we fail (and then loop around and retry).

What a mouthful! Fortunately, C++ has the `std::atomic` class (since C++11) to take care of all that. The main routines to use for a CAS instruction are:

```
std::atomic::compare_exchange_weak
std::atomic::compare_exchange_strong
```

Note that you will also need to know about "memory orders" around atomic primitives, as controlled via the `std::memory_order` library.

Weak or Strong CAS?

Should you use the weak or strong version of the CAS primitive? The strong version is guaranteed to not fail for "spurious" reasons, but only if the atomic's value is not what you want. By comparison, the weak version can fail for two reasons:

- Wrong atomic value
- Spurious error failures

Thus, the strong version seems better, but even so, the most common idiom for using CAS in lock-free programming is the use of the weak version, but in a loop. This idea simply retries if the weak CAS primitive fails, whether due to the underlying atomic variable's value being wrong, or due to the obscure spurious failures.

Both the weak and strong CAS primitives are usually in a loop. The weak CAS can fail for two reasons, being spurious failures or another thread modified the value, and needs to retry. The strong CAS will not fail for spurious reasons, but can still fail for the second reason, and still often needs a loop. The weak version is more commonly used because it's somewhat more efficient, under the assumption that spurious failures are rare.

Example: Lock-Free Stack Array

A lock-free stack implemented in an array is a great example to use, because it has only one moving piece: the stack pointer. This is an integer index used to identify the level of usage in the array, and also doubles as a counter of how many items are on the stack.

Here's our basic interface for an array-based stack:

```
template<typename T, int N>
class LFStackArray {
private:
    std::atomic<int> sp_;   // Stack pointer
    T arr_[N];              // Fixed-size array
public:
    LFStackArray() : sp_{-1}, arr_{} { }
    ~LFStackArray() { }
    LFStackArray(const LFStackArray&) = delete;
    LFStackArray(LFStackArray&&) = delete;
    LFStackArray& operator=(const LFStackArray&) = delete;
    LFStackArray& operator=(LFStackArray&&) = delete;
};
```

And here are some basic member functions:

```
bool empty() const { return sp_ == -1; }
bool full() const { return sp_ == N - 1; }
int count() const { return sp_ + 1; }
```

And let's define the main member functions implementing the stack LIFO functionality. The `top()` function is a `const` member that does not pop the stack (like the standard C++ stack container):

```
T top() const {
    if (sp_ == -1) {
        throw std::exception("Stack underflow top");
    }
    else {
        return arr_[sp_];
    }
}
```

Here's the `pop()` function that decrements the stack pointer:

```
    void pop() {
        if (sp_ == -1) {
            throw std::exception("Stack underflow pop");
        }
        else {
            sp_--;
        }
    }
```

And here's the `push()` member function that increments the stack pointer:

```
    void push(const T& item) {
        if (full()) {
            throw std::exception("Stack overflow");
        }
        else {
            arr_[++sp_] = item;
        }
    }
```

See Any Bugs?

This code will run fine in many cases, but has several concurrency bugs if multiple threads are pushing and popping. The `sp_` variable is atomic, and all of the operations on this variable will be correctly serialized. Problems arise because each of the main member functions are accessing the atomic variable twice.

Any interleaving access in another thread that modifies the stack pointer between those two accesses can break the code. Since all the member functions are short, and the two accesses are within a few instructions, these bugs would be rare situations, but are still an insidious problem.

One way to fix these problems would be to just remove the exception-handling code. All of the extra reads on the stack pointer are to detect overflow and underflow conditions. But that's not a great design decision to make, based solely on our lack of expertise in lock-free programming.

CAS Versions

A better idea is to use the "Compare-And-Swap" (CAS) idiom, repeated in a loop, for proper lock-free versions. Here's an updated `pop()` method:

```
void pop() {
    int oldsp = sp_.load();
    if (oldsp == -1) {
        throw std::exception("Stack underflow pop");
    }
    while(!sp_.compare_exchange_weak(oldsp, oldsp - 1)) {
        // Nothing (try again)
    }
}
```

Note that in this version using `sp_.load()` is not really different from just using `sp_` by name (an implicit load), but the second part uses a different loop. The CAS call is used to check that the stack pointer still has the expected value (i.e., not modified by some other thread), and we loop around until it's true. Since this is a "weak" CAS call, it call also fail for spurious reasons, but that's not a problem because we just retry in that situation, too.

What's missing?

There are no memory orders specified anywhere, so the atomic calls are defaulting to "sequential access" in both the load and CAS loop. That's the safest memory model, but it's needlessly inefficient here.

There are three places where we can specify an alternative memory model. But which to choose? The best choices are:

- Initial `load()` call — "relaxed" memory model (fastest)
- Weak CAS success — "acquire" memory model
- Weak CAS failure (retry loop) — "relaxed" memory model (fastest)

We can get away with relaxed mode for the initial `load()` because it's not critical. We're testing for an error, and we also don't care too much about ordering of accesses leading up to the CAS test.

Similarly, we also don't much care about ordering whenever the weak CAS call fails, whether for spurious reasons or because the value has changed. Either way, we're just looping back to re-try, and the ordering up to the next CAS call doesn't matter.

However, we do care on a successful CAS result, which is when the stack pointer is actually being updated to a new value. Hence, we choose "acquire" rather than "relaxed" for that option.

Our new function looks like:

```
void pop() {
    int oldsp = sp_.load(std::memory_order_relaxed);
    if (oldsp == -1) {
        throw std::exception("Stack underflow pop");
    }
    while(!sp_.compare_exchange_weak(oldsp, oldsp - 1,
        std::memory_order_acquire,    // Success mode
        std::memory_order_relaxed)    // Failure mode
        ) {
        // Nothing (try again)
    }
}
```

Still Buggy!

There's an obscure problem in the above `pop()` function that indicates a misunderstanding of how the CAS primitives work. They don't just update the atomic, but also the passed-in parameter.

Let's consider a stack that currently contains one element, but two threads are both trying to pop the stack. Consider this sequence:

- Thread A: starts and calls `load()` inside `pop()`
- Context switch
- Thread B: runs a full `pop()` function to pop the stack (i.e., `load()` and then CAS success).
- Context switch
- Thread A: continues, but weak CAS fails (value of `sp_` was changed by Thread B).
- Thread A: loops around to retry.
- Thread A: weak CAS now succeeds using the `sp_` value updated by Thread B (yes, really)

The end result of all this is a major bug:

- The atomic is updated to the wrong stack pointer.
- The stack hasn't been popped twice (it should be, once by each thread).

The CAS primitive can change the variable passed as the expected value. Hence, the value of `sp_` in the above code can change before and after the loop, and also whenever there's a loop-around to retry.

Look at the official signature of the weak CAS function, and you'll notice that the first argument containing the "expected" or "old" value is a non-`const` reference. The second argument is not a reference.

Why aren't they the same?

The `compare_exchange_weak()` function can modify the expected value (used to test), but not the "new" value, used to store. This means that:

(a) If it succeeds immediately, the "old" variable will have the "new" value.

(b) If it fails and loops around, the "old" variable will have whatever value another thread changed it to.

When you examine lock-free versions with CAS primitives and loops, you'll notice a few things about the pattern:

1. The "old" value is retested every loop iteration (after CAS failure).

2. The "old" value is also retested after the loop (after CAS success).

3. The "expected" value parameter to weak CAS may also need to be recomputed each iteration, based on the "old" value (which can change), rather than using an unchanging separate variable to contain the expected value.

This is getting complicated! Well, yes, that's the fun of lock-free coding.

Anyway, here's the final version with the corrected weak CAS calls. This defers the underflow test until after the loop, where it catches both cases of underflow: initial underflow or an underflow caused by some other thread popping the stack out from under us.

The final code is:

```
void pop() {
    int oldsp = sp_.load(std::memory_order_relaxed);
    while (oldsp != -1
        && !sp_.compare_exchange_weak(oldsp, oldsp - 1,
            std::memory_order_acquire,   // Success mode
            std::memory_order_relaxed)   // Failure mode
        ) {
        // Nothing (try again)
    }
    if (oldsp == -1) {
        throw std::exception("Stack underflow pop");
    }
}
```

Hopefully, this new version now has all the various concurrent execution sequences covered:

- Success: valid pop on the stack without any changes by another thread.
- Success: valid pop on the stack but another thread removes one (or more) stack elements, but doesn't fully empty the stack.
- Underflow: Pop on an already-empty stack.
- Underflow: Pop on a non-empty stack, but another thread empties the stack before us.

Difficulties with Lock-Free Coding

What's so hard about coding a lock-free algorithm? Well, it's a totally different way of thinking about concurrency compared to the use of standard locking mechanisms.

Some of the problems include:

- Catering for all possible instruction ordering sequences.
- Choosing the right memory order to guarantee correctness.
- Higher-level concurrency problems with the interface.
- Handling the "ABA" problem where updates are missed.

We've already discussed instruction ordering and memory orders in Chapter 7 on atomics, so let's look at the other issues now.

High-Level Race Conditions

Even when we've correctly implemented the member functions with atomics and memory orders, this lock-free stack is still problematic to use. The stack itself will stay consistent no matter what member functions are called in what order, but there are higher-level concurrency problems with any paired usage of multiple member functions, such as sequences like:

- Top and then pop
- Test empty before calling pop
- Test full before calling push

What we'd need to do is define some more composite member functions using lock-free methods. Ideas for new methods to add in the interface for better usage in multithreaded applications include:

- Top-and-pop
- Pop-if-not-empty
- Push-if-not-full

ABA Problems

The ABA problem is a more general concurrency issue that can be particularly applicable to lock-free sequences. The ABA problem occurs where a shared variable undergoes this uncommon sequence with activity in one thread:

- Initial value — A
- Update to value — B
- Second update — A

The problem occurs in a second thread, and it's not really obvious why it's tricky. After the ABA sequence, the second thread sees the value as A, which is unchanged from the prior value it would have seen. Hence, the second thread doesn't know about the intervening value B. Depending on context, this is sometimes no problem, or sometimes a major concurrency error whereby the second thread wrongly assumes that nothing has changed, and the data structure hasn't been updated.

A good example is an array implementation of a lock-free stack. The index value is incremented by one on a push, and then decremented back to its prior value by a thread doing a pop. This is an ABA sequence on an integer index, whereby another thread might assume that the stack index has not changed.

If that thread accesses the "top" element, then an ABA sequence occurs in the first thread, the second thread may see that the stack index is unchanged and assume the same element is still on top of the stack, when in fact, there's an entirely different value. Remember, it's the atomic integer representing the stack index that's undergone the ABA sequence, not the actual object on the stack.

This problematic sequence can occur in any synchronization style including both locking and lock-free algorithms. It's quite an insidious bug, because the ABA sequence doesn't occur that often. In particular, a lot of the lock-free methods for using a CAS operation in a loop are checking for the old value, and can be vulnerable to an ABA sequence.

C++20 Atomics and Lock-Free

The C++20 standard introduced some extra primitives to the atomic class, which were similar in nature to condition variables and spinlocks (or a hybrid thereof). The primitives included:

- `wait()` — blocking call to wait until an atomic changes.
- `notify_one()` — notify one waiting thread.
- `notify_all()` — notify all the threads that are waiting.

An important point to note is that if you're using these C++20 primitives to implement thread synchronization, it's more like using locks than a lock-free algorithm. Where threads are blocked and waiting on an update to an atomic variable, that's a great feature of C++20, but it's no longer really a lock-free data structure. The main hallmark of lock-free programming, where every thread keeps going, is missing in that style.

Freestanding Atomic Functions

Standard C++ provides a number of "free-standing" atomic functions that mirror the member functions. For example, there is:

- `atomic_fetch_add()` — similar to `fetch_add()`.
- `atomic_fetch_sub()` — matches `fetch_sub()`.

There are also "explicit" versions of many of these functions:

- `atomic_fetch_add()` — default memory order.
- `atomic_fetch_add_explicit()` — extra argument for the memory order.

The differences in all these free-standing versions of the functions compared with the main C++ ones are:

- Not member functions (free-standing).
- Accept a pointer to an atomic, not a reference.
- Memory orders can be specified in the "explicit" versions.
- Consistent with the C-language versions defined in the C11 standard.

The reason for pointer arguments is that these functions are consistent with the C language versions declared in `<stdatomic.h>` in C11 (not C++11).

Always Faster?

Note that lock-free algorithms are not always a speed improvement. There is a significant case to be made that lock-free algorithms can increase thread contention. Hence, it is important to time your before-and-after if you're switching from a lock implementation to a lock-free version of your thread-safe data structure. Since the concern is related to lock contention when there are multiple threads, it is important to time performance of your overall application across multiple threads under a realistic load, rather than just benchmarking the low-level lock-free queue primitives. See the references section for various Stack Overflow conversations involving quite animated discussions on when and whether a lock-free algorithm is better or worse than locking methods.

Portability issues

There are also a variety of non-standard methods to achieve lock-free programming with primitives in older code platforms, or in a platform-specific manner. Some of the primitives are:

- `InterlockedCompareExchange` — Win32 version in `<winnt.h>`.
- `OSAtomicCompareAndSwapInt` — Mac variant in `<OSAtomic.h>`
- `__atomic_compare_exchange` — older GCC version.

Note that the `std::atomic` class is not actually guaranteed to be a lock-free atomic operation on every platform. It's a good idea to test your platform using the "`is_lock_free`" primitive as part of your initialization or self-testing code:

```
    assert(std::atomic<int>::is_lock_free());
```

Extensions

1. Implement a lock-free version of fine-grained locking using lock striping on a vector data structure with an array of atomics instead of mutexes (see discussion of "lock striping" in Chapter 4).
2. Implement a lock segmenting version of lock-free fine-grained locking on a vector data structure using atomic arrays, not mutexes (see the discussion of "lock segmenting" in Chapter 4).

References

1. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259, Code: https://github.com/0burak/imperial_hft
2. Jeff Preshing, Jun 12, 2012, *An Introduction to Lock-Free Programming*, https://preshing.com/20120612/an-introduction-to-lock-free-programming/
3. Deb Haldar, August 17, 2017, *Top 20 C++ multithreading mistakes and how to avoid them*, https://acodersjourney.com/top-20-cplusplus-multithreading-mistakes/
4. Apple, March 2025 (accessed), *Mac OSX documentation*, https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/OSAtomicAdd32.3.html
5. Wikipedia, March 2025 (accessed), *Non-blocking algorithm*, https://en.wikipedia.org/wiki/Non-blocking_algorithm
6. Herb Sutter, September 08, 2008, *Lock-Free Code: A False Sense of Security*, Dr Dobbs Magazine (archived), https://web.archive.org/web/20150901211737/http://www.drdobbs.com/article/print?articleId=210600279&siteSectionName=cpp
7. Microsoft, 24 May, 2022, *InterlockedCompareExchange function (winnt.h)*, https://learn.microsoft.com/en-us/windows/win32/api/winnt/nf-winnt-interlockedcompareexchange
8. GNU Foundation, March 2025 (accessed), *6.26 Built-in Functions for Memory Model Aware Atomic Operations*, https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html
9. CPP Reference, March 2025 (accessed), *std::atomic<T>::compare_exchange_weak, std::atomic<T>::compare_exchange_strong*, https://en.cppreference.com/w/cpp/atomic/atomic/compare_exchange

10. CPP Reference, March 2025 (accessed), *std::memory_order*, https://en.cppreference.com/w/cpp/atomic/memory_order
11. Sourav Ghosh, July 2023, *Building Low Latency Applications with C++*, Packt Publishing, https://www.amazon.com/dp/1837639353
12. Tim Blechmann, 2011, *Chapter 17. Boost.Lockfree,*, https://www.boost.org/doc/libs/1_53_0/doc/html/lockfree.html, https://www.boost.org/doc/libs/1_53_0/doc/html/boost/lockfree/queue.html
13. Stack Overflow, 2011, *Do lock-free algorithms really perform better than their lock-full counterparts?* https://stackoverflow.com/questions/5680869/do-lock-free-algorithms-really-perform-better-than-their-lock-full-counterparts
14. Stack Overflow, 2017, *Using Boost.Lockfree queue is slower than using mutexes*, https://stackoverflow.com/questions/43540943/using-boost-lockfree-queue-is-slower-than-using-mutexes

Part VI: Sequential C++ Optimizations

39. Timing and Benchmarking

Timing C++ Code

There are a number of reasons why it can be useful to time the execution of a program. Timing C++ code can be useful in determining which statements should be optimized whereas profilers may only indicate which functions are consuming time. Timing code can also determine the relative efficiency of various operations and give you valuable information about writing code for your machine (e.g., is shifting faster than integer multiplication?).

There are several ways to time your C++ code, some of which have existed for decades, and some that are newer and standardized. Here's a list of some options:

- `time` shell command
- `time` C++ function
- `clock` C++ function
- `<chrono>` standard C++ class

Another way to examine the efficiency of a C++ operation is to look at the assembly code. This is examined later in the chapter.

If the full execution time for a program is all that is needed, the Linux `time` command can be used to calculate the time required by a program. There are two versions — a stand-alone utility in `/bin` and a command built into `csh`. The command to run is usually:

```
time a.out
```

A different executable name could also be used and command line arguments can also be specified.

The Chrono Class

The `std::chrono` library is an awesome piece of work, and has many features. It's been part of the C++ standard since C++11. I'm only going to touch on a handful of basic measurements here.

Here's an example of how to measure the duration between two events:

```
auto before = std::chrono::high_resolution_clock::now();
// ... Do something
auto now = std::chrono::high_resolution_clock::now();
auto diff = std::chrono::duration_cast
     <std::chrono::microseconds>(now - before).count();
std::cout << "Time: " << diff
          << " microseconds" << std::endl;
```

There are other ways to do this, as the library is very flexible, with many capabilities. Reading the documentation for this class is enough to make my head spin. Someone had a lot of time to spend on time! Kudos to them. But one way is good enough for timing our C++ code, so let's move on and leave the rest as an exercise for the reader (LOL!).

The Clock Function

If a more detailed speed analysis is needed, it is possible to add C++ self-instrumentation code to your program to monitor its own performance. The basic idea is to use the standard library functions to monitor the time before and after an action. The advantages of the `clock` function over the new-fangled `std::chrono` library:

- Measures CPU clock ticks, not wall clock time.
- Works in C, if you need it, not only C++.
- Only have to remember one function name!

The oldest useful function is the "`clock`" function which has existed since the C programming language. The `clock` function counts the number of clock ticks since the program began executing. The "`time`" function, which keeps track of the real calendar time could also be used, but it is not a true indication of processor time on a large multi-user system. The `clock` function is correct for both single user and multi-user systems.

The `clock` function returns a value of type `clock_t` (typically `long` or `int`) that counts the number of clock ticks. This value can be converted to seconds by dividing by the constant `CLOCKS_PER_SEC`, also declared in `<time.h>`.

The basic idea of timing C++ code blocks is to call the clock function before and after an operation and examine the difference between the number of clicks. The code below examines the relative speed of shift and multiplication operations on int operands.

```
void profile_shifts()
{
    const int MILLION = 1000000;
    const int ITERATIONS = 100 * MILLION;

    int x = 1, y = 2, z = 3;

    clock_t before = clock();
    for (int i = 0; i < ITERATIONS; i++)
        x = y << z;
    printf("%d Shifts took %f seconds\n", ITERATIONS,
        (double)(clock() - before) / CLOCKS_PER_SEC);

    before = clock();
    for (int i = 0; i < ITERATIONS; i++)
        x = y * z;
    printf("%d Multiply took %f seconds\n", ITERATIONS,
        (double)(clock() - before) / CLOCKS_PER_SEC);
}
```

Clock Problems

clock Portability Pitfall. Note that some implementations on older Unix versions don't conform to the C++ standard and return the number of clock ticks since the *first call* to the `clock` function. This means that a single call to `clock` at the end of the program would always return zero. Hence, it is more portable to measure the number of clock ticks between two calls to clock, one at the start and one at the end. Obviously, you can also put the first call to "`clock`" at the start of the "main" function to avoid this rare glitch. Note that on implementations that are correct, a call at the start of "`main`" may be non-zero due to the overhead of global and static C++ object instantiations (i.e., constructors for global objects), which occurs before entering `main`.

Clock Tick Integer Division Pitfall. Note that the standardized `clock_t` type and `CLOCKS_PER_SEC` constant are both integers. Hence, here's a bug:

```
    clock_t diff = clock() - before;
    double seconds = diff / CLOCKS_PER_SEC; // Bug!
```

The problem is that it's integer division, so it inaccurately truncates to an integer. You need a typecast to `float` or `double` on either side of the division operator.

```
    clock_t diff = clock() - before;
    double seconds = diff / (double)CLOCKS_PER_SEC; // OK
```

Clock Tick Overflow Pitfall. The `clock` function also has a problem with wraparound on some implementations. Because of its high resolution, the total number of clock ticks can quickly overflow the maximum value that can be stored by the type `clock_t`. On one system the `clock` function will wrap around after only 36 minutes. If the program being timed runs for longer than this period, the use of `clock` can be misleading. One solution is to use the "`time`" function rather than "`clock`" when executions are longer, but this usually only has resolution to the nearest second.

Benchmarking

Benchmarking is a slightly different concept to tuning, and refers to testing the efficiency of certain operations, such as low-level operators, to find a more efficient way to do an operation. For example, if you want to compare multiplication versus addition, you write a program to run these operations a few million times. When changing a program to increase efficiency, you shouldn't assume that a certain operation is clearly faster, but you should benchmark whether the changes have noticeably increased the operation's efficiency (or even decreased it!).

Techniques for measuring program efficiency range from the stop-watch method to the use of sophisticated profiler software tools. If no profiler is adequate, the programmer can gain timing information by adding instrumentation statements to the program, although there are many pitfalls in attempting to determine the time taken by a sequence of statements.

The measurement of the memory usage and space-efficiency of a C++ program is a slightly more difficult problem. There are several types of memory: instruction code, static memory, read-only string literals, initialization data, global/static variables, the stack, and the heap. Measuring the memory usage of the stack and heap is somewhat difficult because of their dynamic nature.

However, various tools exist to measure the different types of memory, and clever use of C++ programming constructs can also yield reasonable data.

Benchmark programs attempt to examine how quickly your machine executes certain instructions, which is more useful for examining a single multiplication operation. You mainly use benchmarking for code that's running in low-level kernels, such as CPU speedups (e.g., AVX intrinsics) or examining the possible use of different GPU primitives.

Consider benchmarking for timing of low-level arithmetic operations on your platform. For example, how would you determine whether the integer multiplication operation `x*2` could be more efficiently replaced by `x<<1`?

How can you time these instructions? You obviously cannot just time a single operation of each with the "`clock`" function, because a single click tick contains many CPU cycles. So, you have to time thousands or even millions of such operations.

```
    for (int i = 0; i < 100 * MILLION; i++) {
        x << 1;
    }
```

We've already noted one problem: there's all this extra loop overhead time for the for loop conditional test (the "<" operator) and its incrementer (`i++`). The loop actually has three operations that are all about the same order-of-magnitude cost (i.e., <, ++, <<). To get at the operator cost, we'd need to subtract out the loop overhead. We could, for example, try to time an empty loop without any loop body, and subtract that from our final cost.

Benchmarking Problems

Null effect problems. Another problem is that we cannot easily time the operators with these statements in the loop body:

```
    x << 1;
    x * 2;
```

The compiler is clever enough to notice that the `x<<1` and `x*2` statements have no effect in the program above (and gives "null effect" warnings). The built-in optimizer may even remove them completely. So, they won't get timed properly, or at all, even in a loop.

Add volatility? One possible solution is that maybe the compiler can be forced to avoid this optimization on the original expressions by declaring x as a "volatile" variable.

```
    volatile int x = 0;
```

The volatile qualifier tells the compiler that all accesses to x are important, and that it should not remove any. The intended purpose of volatile is to allow the declaration of addresses for memory-mapped I/O, debugger-modified variables, or for variables modified by other programs (e.g., a semaphore modified by another program running concurrently).

However, we can use it here to force all accesses to x to occur even if they appear pointless.

On the other hand, by doing this, we've lost the ability to see the "real" time cost for these operations when they're running in normal code. Most variables aren't volatile.

Anyway, it doesn't even work properly. Unfortunately, the computations of the << and * operators in x<<1 and x*2 are not being assigned anywhere, so the computations themselves could be optimized out, even though the actual read operations on x must occur because x is volatile.

To force the << and * operations to occur, it is necessary to use their result somehow, such as by assigning it to the (volatile) variable x:

```
    x = x << 1;
```

Although all of the above improvements will enhance the previous version, a far better method of improvement is to time a loop that performs a huge number of the operations,.

Hence, we have to use something like these assignment expressions inside a loop:

```
    x <<= 1;
    x *= 2;
```

The code given here examines the relative speed of 10,000 shift and multiplication operations on int operands:

```
 volatile int x = 0; // volatile to prevent optimizations
clock_t before  = clock();
for (int i = 0; i < ITERATIONS; i++)
    x = x << 1;
printf("%d Shifts took %f seconds\n", ITERATIONS,
    (double)(clock() - before) / CLOCKS_PER_SEC);
before = clock();
for (int i = 0; i < ITERATIONS; i++)
    x = x * 2;
printf("%d Multiplications took %f seconds\n", ITERATIONS,
    (double)(clock() - before) / CLOCKS_PER_SEC);
```

Loop Unrolling

Unfortunately, the above method of measuring the speed of operations is not completely accurate, because it also includes the loop overhead (incrementing i from 1 to 10,000) and the cost of the assignment of the result to x. The loop overhead can be minimized by placing many operations within the loop, as below:

```
volatile int x = 0; // volatile to prevent optimizer
clock_t before = clock();
for (int i = 0; i < ITERATIONS; i++) {
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
    x = x << 1; x = x << 1; x = x << 1; x = x << 1;
}
printf("%d Shifts took %f seconds\n", ITERATIONS*20,
    (double)(clock() - before) / CLOCKS_PER_SEC);
before = clock();
for (int i = 0; i < ITERATIONS; i++) {
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
    x = x * 2; x = x * 2; x = x * 2; x = x * 2;
}
printf("%d Mult took %f seconds\n", ITERATIONS * 20,
    (double)(clock() - before) / CLOCKS_PER_SEC);
```

Unfortunately, the assignment operations are needed to prevent the optimizer removing the computations, as discussed above.

The only truly effective method of removing the cost of the assignment from the measurement is to time another separate loop, and subtract its time from that of the other loops, as below. This method also automatically accounts for the loop overhead cost, so the multiple operations inside each loop are not needed (and in fact would be incorrect). Our final version of the benchmark program is also made more sophisticated to output the relative magnitude of the two operations:

```
void profile_shifts4()
{
    const int MILLION = 1000000;
    const int ITERATIONS = 1000 * MILLION;
    volatile int x = 0; // volatile to prevent optimizations
    double time1, time2;
    // Time the loop overhead
    clock_t before = clock();
    for (int i = 0; i < ITERATIONS; i++)
        x = 1;
    clock_t loop_cost = clock() - before; // overhead
    double ovtime = (double)(loop_cost) / CLOCKS_PER_SEC;
    printf("%d overhead: %f seconds\n", ITERATIONS, ovtime);

    // Shifts
    before = clock();
    for (int i = 0; i < ITERATIONS; i++) {
        x = x << 1;
    }
    time1 = (double)(clock() - before - loop_cost)
            / CLOCKS_PER_SEC;
    printf("%d Shifts took %f secs\n", ITERATIONS, time1);

    // Multiplications
    before = clock();
    for (int i = 0; i < ITERATIONS; i++) {
        x = x * 2;
    }
    time2 = (double)(clock() - before - loop_cost)
            / CLOCKS_PER_SEC;
    printf("%d Mult took %f seconds\n", ITERATIONS, time2);

    // Compare both times, and print percentage difference
    const float ACCURACY = 0.00001f; // maximum error
    if (fabs(time1 - time2) < ACCURACY) // (almost) equal?
        printf("Shift and multiplications: same time\n");
    else if (time1 < time2) {
        printf("Shifts faster by %5.2f percent\n",
                (time2 - time1) / time2 * 100.0);
    }
    else {
        printf("Multiplications faster by %5.2f percent\n",
            (time1 - time2) / time1 * 100.0);
    }
}
```

Limitations of Benchmarking

Benchmarking of C++ using these timing methods is not perfect, but I've always found it useful. There are various reasons why this type of benchmarking timing results may not be fully correct.

- Hard to account for parallelism (e.g., GPU throughput)
- Single-threaded code is not always a true representation.
- Pipelining speedups often differ in production code (even for sequential CPU code, such as AVX intrinsics).
- Loop overhead is hard to separate from the raw operations (as seen above!)
- Compiler optimizations might modify or even remove the operations being benchmarked.
- Memory cache hit rates are too high because you're running tight code accessing only a few addresses.
- Optimization levels in test mode might not match your production version.
- Debug modes might not match production (e.g., if running in a debugger).
- Pipelining by the CPU of many instructions makes it appear better than reality.
- Unrealistic non-production conditions are being tested.

Compiler optimizations. In this day and age of amazing optimization algorithms, note that on some platforms the benchmarking code above may indicate that shifts and multiplications cost exactly the same. This is most likely an indication that the compiler automatically optimizes any multiplications by powers of two into left shifts. To get the true cost of a multiplication, the expression should be:

```
    x = x * x;
```

But even this might be optimized algebraically by a compiler. The only way to know for sure what's actually being benchmarked is to examine the assembly language.

Examining Assembly Output

Another way of examining the relative costs of particular operations for a particular compiler is to examine the assembly language produced by the compiler. Many compilers have an option to produce assembly language output. For example, under Linux the command may be:

```
    gcc -S main.cpp
```

This will produce the assembly language listing for the C++ source file and store it in a new file "main.s" as a human-readable text file. Without the -S option, the assembly output would have been passed to the assembler to create the machine code executable. GCC also has a "-masm" option that controls the different "dialects" of assembly language (e.g., "intel" or "att"). GCC also has a verbosity control on assembly output via "-fverbose-asm" and "-fno-verbose-asm" options.

Another way to generate assembly with GCC is the "-save-temps" option. This option tells GCC to save the temporary assembly language file that it used for the real compilation. Hence, this option can be used with the normal compilation mode to both build the code as normal and also output a ".s" assembly file. The advantage of this GCC "-save-temps" option over "-S" is that you don't need to create a separate build path for generating assembly text files.

Reviewing assembly code. Examining assembly language instructions produced for C++ operations can be very enlightening. For example, you can determine whether the compiler uses a special increment instruction for the ++ operator. Whether or not the compiler is performing various optimizations can also be examined.

Counting the number of assembly instructions is a simple measure and gives a reasonable indication of how efficiently an operation will be performed. A better method is to determine the number of cycles used by each instruction, but this requires a rather more intimate knowledge of the assembly language being used.

Many useful things can be discovered by examining assembly output. For example:

- Does the expression x*2 generate a multiply instruction or a shift instruction (or an addition instruction to do "x+x")?
- Does the compiler notice that x=x+1 can be replaced by x++?
- Is the integer % remainder operator implemented by a sequence of instructions?

Consider the use of the relational operators (e.g., >, <) in expressions such as:

```
    flag = x > y;
```

This will often produce a sequence of instructions because of the need to assign flag the value either 0 or 1.

The instructions may well look like the following pseudo-assembly language:

```
LOAD 10($sp) # Load x (from stack)
CMP 12($sp) # Compare with y (on stack)
BGT $1 # Branch if greater than
LOAD 0 # Result of > operation is 0
JUMP $2
$1:
LOAD 1 # Result of > operation is 1
$2:
STORE 14($sp) # Store in flag (on stack)
```

However, review the assembler for the similar test in `if` statements, such as:

```
if (x > y) ...
```

For an `if` statement, the instructions need not be as complex, because there is no need to store the value 0 or 1 anywhere. The assembly language could be similar to branches without computations:

```
LOAD 10($sp) # Load x (from stack)
CMP 12($sp) # Compare with y (on stack)
BLE $1 # Branch if NOT greater than
... # Code for if statement body
$1:
... # Statements after if statement
```

Examining Object Files

The `objdump` command is another useful tool on Linux for analyzing binary object files. `DUMPBIN` is the comparable tool on Windows for MSVS (or you can use the `LINK` command with the "`/DUMP`" option). These tools can get to the assembly language text in reverse, by disassembling the binary instructions that are in the object file, in combination with the various symbolic information.

`objdump` can be used to examine object files in various ways and there are various useful options. The "`-d`" and "`-D`" options provide disassembly where you can examine a full dump of the assembly code in printable form (as an alternative path to the "`-S`" option). The "`-h`" option shows the headers of the object file and "`-g`" shows debugging information in the file. There are numerous other options and the "`--help`" option can be used to list all options. The `objdump` command is part of Gnu Binutils, which also includes other useful binary file tools such as `nm`, `size`, `strip`, and `strings` utilities.

DUMPBIN also has various options that can be used on the DOS command-line. The default is "/SUMMARY" for a summary of the information about the object file. The "/DISASM" command shows the disassembly of the object file, which is in assembly language. Also useful is "/SYMBOLS" to show the symbolic names.

Performance Tuning Practices

How should the huge number of methods of improving program efficiency be applied to a program? The code transformations that improve the program by a significant amount should be tried first, and the smaller optimizations used only when it is important to squeeze out that last bit of extra speed in bottlenecks. Hence, I suggest the following steps for improving the efficiency of a program:

1. Time your program to get a baseline (i.e., run a full inference query).

2. Invoke the C++ compiler's built-in optimizer.

3. Profile the code and find the "hot spots."

4. Consider a better data structure or algorithm.

5. Use the major code transformations.

6. Use smaller code transformations, if speed is crucial.

The first step is to measure your code's time cost. Otherwise, how will you know whether anything made it better?

The next step is easy: turn on your optimizer. All modern C++ compilers have an option to invoke an optimizer on the code. The optimizer, although it may not always yield a major increase in speed, has one very important advantage — the programmer need not change the code. Hence, if a small improvement is desired, the optimizer can often provide it without much effort.

Software tuning. Assuming you're done with all the non-code changes to the system (e.g., hardware, networking), it's time to examine the C++. You can either start high by looking at the data structures, or start low by optimizing the busiest low-level kernels.

The choice of a better algorithm (usually with different data structures) for a program is not an easy method of program improvement. Simply identifying what would be a better algorithm is a difficult problem!

And once identified, the new algorithm must be implemented by the programmer, costing precious man hours. However, this is the best method to achieve an order-of-magnitude increase in the program's performance.

The next step is to profile in detail the C++ code to determine which functions (or statements) are accounting for most of the program's time; these are the "hot spots" of the program. This identification of costly statements is best achieved by a profiler, although if I had to take a guess, I'd say look at your vector dot product code. Identifying frequently called functions and deeply nested loops is often adequate. Once the hot spots are identified, all efficiency measures, large and small, should be applied to this code. Any improvement to the efficiency of a statement, no matter how small, will improve the overall efficiency greatly if that statement is executed often.

Once the most costly functions and loops have been optimized, other statements can also be optimized, although the increase in speed will not be as noticeable. Some of the better code transformations to apply are parallelization, loop optimizations (vectorizations), using pass-by-reference for passing structures or objects to functions, and replacing small functions with macros or `inline` functions.

Make it right first? The speed improvement techniques in C++ can be applied either as the programmer is writing the code, or after the development and debugging of the program. The second approach is often referred to as the "make it right first" rule. However, I believe that the first method is preferable simply because optimizing your program once it is working is a dangerous practice, and often introduces new bugs. Deferring efficiency improvement to the final development stage can also waste programmer time in improving the basic algorithms used in a program. Using efficiency techniques during the development of the program is a much sounder method of improving efficiency.

Tuning Trade-offs

Tuning a program is not always a clear-cut gain. There are numerous other quantities that efficiency may affect:

- Space versus time-efficiency.
- Robustness of a program.
- Readability and maintainability of a program.
- Portability of a program.

There is almost always a trade-off between time and space when making programs run faster. Many of the algorithm improvements sacrifice space for extra speed, such as caching and precalculation. An often overlooked trade-off is between program efficiency and a programmer's time in making the changes.

Changing a program for efficiency can introduce extra bugs into a program (although you could argue that it might remove bugs, too). If a piece of code has already been debugged, improving its efficiency may not be worth the risk to the robustness of a program.

Many of the program transformations used for efficiency can reduce the readability for a program. Naturally, this also makes it more difficult for a program to be maintained, and since the major cost in a program's development cycle is usually maintenance, improving efficiency may not be worth it in the long run.

Perhaps surprisingly, the efficiency of a program can usually be increased significantly without affecting portability. There are some efficiency techniques in this book, but there are many generic methods that work across all C++ code.

Almost all of the dangers of improving efficiency are dangers for the programmer. On the other hand, the users of a program will be well pleased by extra responsiveness, and this alone makes efficiency improvement a worthwhile exercise.

References

1. Linux Code, December 27, 2023, *Measuring Execution Time with Microsecond Resolution in C++*, https://thelinuxcode.com/cpp-microseconds/

40. Bitwise Operations

C++ Bitwise Operators

Here's a refresher on the C++ bitwise operators:

`x & y` — binary bitwise-AND

`x | y` — binary bitwise-OR

`x ^ y` — binary bitwise-XOR

`x << y` — binary left bitshift

`x >> y` — binary right bitshift

`~x` — unary bitwise-complement

Binary literals. Also, a reminder that C++ also supports binary literal constants with a "`0b`" prefix, similar to the hexadecimal "`0x`" prefix.

For example, to represent the constant `10` (ten), your C++ code can use:

```
    const int ten = 10;     // decimal
    const int ten = 0xA;    // hexadecimal
    const int ten = 012;    // octal
    const int ten = 0b1010; // binary
```

Bitwise badness: A few pitfalls in coding C++ bitwise operators should be mentioned:

- Integer-only: the C++ bitwise operators do not work on floating-point data types.
- Quiet overflow: if you do anything to overflow an integer type, nobody's going to tell you. For example, shifting the sign bit too far left with "`1<<32`" instead of "`1<<31`" will simply lose it. You might get a compiler warning, though.

- Two is not better than one. The `&` operator is bitwise, but `&&` is logical. Similarly, `|` and `||`. It's the reverse for < and << or > and >>. Choose the wrong one and you might get a compiler warning, if the stars are aligned and the wind is blowing easterly.
- Operator precedence is tricky and not what you'd expect (it's arguably broken, but rather too late to fix), so use lots of parentheses in bitwise expressions, and don't ignore C++ compilation warnings.
- Bitwise operators are not always well-defined on negative values (e.g., bitwise right shift is officially "undefined behavior" on a negative), so it's best to use "`unsigned`" types as operands to bitwise operators. Note also that it's often useful to add the suffix letter "`u`" to integer constants (e.g., `10u`, `0xAu` or `0b1010u`), when dealing with bitwise operations. This makes the constant of type "`unsigned`" and avoids various bitwise operator problems with signed numbers.

Bitwise operation algebraic properties: The interaction with zero is an important difference between the main operations:

- Bitwise-AND with zero equals zero: `a & 0 == 0`
- Bitwise-OR with zero equals the other value: `a | 0 == a`

The following inequalities for bitwise operators on non-negative integers can also be useful to know:

- Bitwise-AND only clears bits and is <= each operand: `a & b <= a`
- Bitwise-OR only sets bits and is >= each operand: `a | b >= a`
- Bitwise-AND equals the larger value only for equal numbers.
- Bitwise-OR equals the larger value only for subset bit patterns.

Addition versus bitwise operations: The relationship between the bitwise operators and the integer "+" operator can be useful to understand:

- Bitwise-AND is <= the sum of its operands: `a & b <= a + b`
- Bitwise-AND equals addition only if both numbers are zero.
- Bitwise-OR is >= the sum of its operands: `a | b >= a + b`
- Bitwise-OR equals addition only for disjoint bit sets or zeros.

Note that these relationships are for positive integer values. Bitwise operators need positivity in their daily lives, whereas addition is fine with lots of negativity.

Bit Flag Basics

The main use of C++ bitwise operators is to use bit flags in integer variables, which is very efficient in both storage space and execution time. A vanilla "`int`" can store 32 bit flags, and a "`long`" can store 64 bits. The basic bit operations in C++ use these bitwise operators:

- Check a bit — bitwise-AND (`&`)
- Set a bit — bitwise-OR (`|`)
- Toggle a bit — bitwise-XOR (`^`)
- Clear a bit — bitwise-AND with complement (`&` with `~`)

Here are some example macros for examining the bits in a 32-bit integer, which should be of "`unsigned int`" type:

```
// Bit Flags in Integers
#define AUSSIE_ONE_BIT_SET(x, b)    \
  (( ((unsigned)(x)) & ((unsigned)(b))) != 0 )
#define AUSSIE_ANY_BITS_SET(x, b) \
  (( ((unsigned)(x)) & ((unsigned)(b))) != 0 )
#define AUSSIE_ALL_BITS_SET(x, b) \
  ((((unsigned)(x))&((unsigned)(b))) == ((unsigned)(b)))
#define AUSSIE_NO_BITS_SET(x, b)   \
  (( ((unsigned)(x)) & ((unsigned)(b))) == 0 )
```

The corresponding macros to set and clear these bit flags are:

```
#define AUSSIE_SET_BITS(x, b)     \
  (( ((unsigned)(x)) | ((unsigned)(b))))
#define AUSSIE_CLEAR_BITS(x, b)   \
  (( ((unsigned)(x)) & (~((unsigned)(b)))))
#define AUSSIE_TOGGLE_BITS(x, b) \
  (( ((unsigned)(x)) ^ ((unsigned)(b))))
```

Yikes! What a mess! But all those parentheses are necessary to avoid precedence issues with preprocessor macros.

Bit Sets

You can consider a 32-bit integer to be a "bit set" of 32 distinct bit flags, where all 1s represent a bit flag that is in the set. A bit set is an inherently parallel architecture, even in ordinary sequential C++ code. The basic idea is that a 32-bit unsigned int stores 32 bit flags. Certain actions on the integer as a whole effectively process 32 bits in parallel. For example, it is very fast to check if any bits are set at all by testing whether the whole integer is zero.

In regards to bit sets stored in an integer, the basic set operations can be implemented very efficiently with C++ bitwise operators:

- Bitwise-AND (`&`) — intersection
- Bitwise-OR (`|`) — union
- Bitwise-complement (`~`) — set complement (negated set)
- Bitwise-and-complement ("`A&~B`") — set difference (set minus)

In addition, there are a number of fast operations that can be useful for bit sets:

- Integer zero — null set of bits.
- Integer negative-one — full set of all 1s.
- Bitwise "popcount" — set cardinality or number of elements.

Example code with these ideas for 32-bit sets implemented as unsigned integers:

```
    u != 0          // Test if any bit is set
    u3 = u2 & u1;   // Intersection of sets (Bitwise-AND)
    u3 = u2 | u1;   // Union of sets (Bitwise-OR)
    u3 = u2 ^ u1;   // Toggle bits in sets (Bitwise-XOR)
    u3 = ~u1;       // Set complement or inverse
```

The total number of bits set out of 32 can be computed fast as a "popcount" operation using intrinsic functions, such as "`__popcnt`" in Microsoft Visual Studio and "`__builtin_popcount`" for GCC (there are also versions for 64-bit longs). In x86 architectures, popcount is a single CPU instruction (`POPCNT`) implemented in hardware, and is therefore very fast.

Note that these C++ macros assume type "`unsigned int`" with 32 bits, and therefore 32 distinct bit flags in a single integer variable. For more bits, the "`unsigned long`" type could be used (64-bit), and there is also the "`long long`" type (128-bit).

The above macros would need to be changed to use type casts to "`unsigned long`" rather than just "`unsigned`" for a 64-bit version. For even more bits, a data structure called a "bit vector" can be implemented as an array of unsigned integers, which generalizes the bit set idea.

Bitwise Intrinsic Functions

Intrinsic functions, or "builtin" functions, are special C++ functions that are specific to the compiler environment. For example, Microsoft Visual Studio and GCC have different builtins. Intrinsics are usually implemented in very efficient ways, often directly mapping to CPU instructions, so they can be very powerful optimizations.

Some of the useful builtin functions for integer bitwise arithmetic are listed below. Most of these functions are for "`int`" or "`unsigned int`" (32-bit), but have other versions for `long` 64-bit or `unsigned long` 128-bit types. There isn't usually a version for "`short`" 16-bit integers.

Count Leading Zeros (CLZ): Various functions count the leading zeros, or similarly, the offset of the first set bit. This is scanning the bits from left-to-right and finding the most significant bit. One application of the CLZ intrinsic is a fast way to compute a truncated log2 of an integer, or similarly, computing the highest power-of-two in a number.

- `_BitScanReverse` (Microsoft intrinsic `<intrin.h>`): Finds the most-significant bit in a 32-bit integer. There's also `_BitScanReverse64`.
- `clz`: Count leading zeros (various versions); also sometimes called "`nlz`" for "number leading zeros".
- `__lzcnt`: Leading zeros count in Microsoft Windows intrinsics, use `<intrin.h>` for Microsoft Visual Studio C++.
- `__builtin_clz` (count leading zeros): GCC function to count the number of leading prefix zeros in an unsigned integer.
- `_CountLeadingZeros`: Microsoft `<intrin.h>` ARM intrinsics.

For all you silicon addicts, here's the CPU hardware instructions are underpin these intrinsics:

- `BSR`: Bit Scan Reverse x86 assembler instruction.
- `LZCNT`: x86 instruction for leading-zero count, similar to BSR.

Count Trailing Zeros (CTZ): Contrasting to the leading zero functions, these functions find the zeros on the right-hand-side of an integer. This is the least-significant bit.

- `_BitScanForward` (Microsoft intrinsic `<intrin.h>`): Finds the least-significant bit set. Long int version is `_BitScanForward64`.
- `__builtin_ctz` (count trailing zeros): GCC function counts zero bits on the right (least-significant bits).
- `ffs/ffsl`: Find first set (least-significant bit).
- `__builtin_ffs` (find first set): GCC function: find first set bit from the least significant bits (from the right bits).

The related x86 CPU hardware instructions are:

- `BSF`: Bit Scan Forward x86 assembler instruction.
- `TZCNT`: x86 instruction for trailing-zero count, similar to `BSF`.

If you'd rather code it yourself, there's Brian Kernighan's bit trick for LSB: bitwise-and of `n` and `n-1` (i.e., in C++ `n&(n-1)` finds the lowest set bit). But using the intrinsics should be faster.

Popcount (Set Bits Count): The count of 1s in a number is known as the "popcount" (which is short for population count) and there are various intrinsics:

- `__builtin_popcount`: GCC function to count the number of 1s in an unsigned integer.
- `BitOperations.PopCount`: Microsoft intrinsic function for bitwise popcount.
- `__popcnt`: AMD x86 popcount intrinsic using `POPCNT` x86 instruction (Microsoft platform)
- `_mm_popcnt_u32`: Intel x86 popcount intrinsic using `POPCNT` x86 instruction (Microsoft platform); use `<intrin.h>` on MSVS C++.
- `__builtin_parity`: GCC function tracking bitwise binary parity (whether the number of 1s is odd or even).

The x86 CPU hardware instruction is `POPCNT`, which computes the popcount faster than a hummingbird's wings.

Example: Integer Popcount

The “popcount” is short for “population count” of a binary number, and is the number of binary 1s in an integer number. This has applications such as quickly counting the number of elements in a bit set or bit vector.

Bitwise arithmetic can be used to check for a '1' value in each bit of an integer. Usually an unsigned type is used (as below), but bit twiddling of signed integers is also possible. This is the slow version in C++ that simply loops through each bit, checking if it is set:

```
int aussie_popcount_basic(unsigned int x)
{
     // Count number of 1s
     const int bitcount = 8 * sizeof(x);
     int ct = 0;
     for (int i = 0; i < bitcount; i++) {
         if (AUSSIE_ONE_BIT_SET(x, 1u << i)) ct++;
     }
     return ct;
}
```

Kernighan Popcount Algorithm: A faster version is to use a bit trick found by Brian Kernighan, author of *The C Programming Language*. For all values of `n`, the previous number `n-1` has one less bit set. So, if you do bitwise-AND of `n` and `n-1`, it removes the rightmost bit that is 1 (i.e., least significant bit). Hence, you can use this to optimize popcount by only looping as many times as there are 1s in the number (rather than always doing 32 iterations). Here’s the new C++ code:

```
int aussie_popcount_kernighan_algorithm(unsigned int x)
{
     // Count number of 1s with Kernighan bit trick
     int ct = 0;
     while (x != 0) {
         x = x & (x - 1);  // Remove rightmost 1 bit
         ct++;
     }
     return ct;
}
```

Intrinsic Popcount Functions: The Kernighan method is faster, but far from optimal. To do it super-fast, we have to look at existing builtin function primitives.

For example, Microsoft intrinsics include "__popcnt" or "_mm_popcnt_u32" (in header file <intrin.h>), whereas GCC has a "__builtin_popcount" function, which count the number of 1s in an unsigned integer. On x86 CPUs, the underlying intrinsics should be using the x86 assembler instruction named POPCNT.

Here is some example C++ code that works for Microsoft Visual Studio:

```
int aussie_popcount_intrinsics2(unsigned int x)
{
    return __popcnt(x);  // Microsoft intrinsics
}
```

Obviously, a faster version is to declare this one-line function as "inline" in a header file, or to convert to a C++ preprocessor macro, such as:

```
#define AUSSIE_POPCOUNT(x) (__popcnt((unsigned)(x)))
```

Example: Bitwise Log2 on Integers

Calculating the base-two logarithm of integers can be quite useful. There are various algorithms that use logarithms in AI.

Let's calculate the integer logarithm of an integer. This means we aren't doing the proper fractional logarithm of a number, but we are truncating it down to the nearest integer. For example, log2(7) will be truncated to 2, rather than 2.807. Note that we're assuming the input is unsigned numbers, since logarithms of negatives are undefined. Also, we have to decide how to handle zero, because log2(0) is undefined (or negative infinity if you prefer).

A simple way to implement a truncated integer log2 function is to use floating-point functions and type casts back to int:

```
int aussie_log2_integer_slow(unsigned int u)
{
    // Slow float-to-int version
    return (int)log2f(u);
}
```

This works, but it's inefficient to use floating-point arithmetic on integers. Surely there's a faster way?

After some thoughts about binary bits, we notice that log2 of an integer is just the index position of the highest bit in a number. The log2 of 1 is 0, because the '1' is in position 0. The log2 of 2 (binary 10) is 1 because the leftmost 1 is in position 1. The log2 of 4 (binary 100) is 2, where the 1 is in index 2. The number 7 is binary 111, so log2 is the position of the leftmost 1, which is position 2. So, log2(7) is the same as log2(4), but log2(8) is 3.

There are numerous builtin bitwise functions that can find the leftmost set bit. With sudden insight, we note that we can use "CLZ" (count leading zeros) to compute how many prefix zeros there are before the leftmost 1 bit (i.e., counts the zeros up to the most-significant bit from the left). We can then compute the bit index position from the right in a 32-bit integer as "32-CLZ". It's on the right track, and a bit of testing shows that the formula to use is "32-CLZ-1".

Here's some example code that uses this CLZ method to compute log2 of an integer. This works on Microsoft Visual Studio using the <intrin.h> header file to declare intrinsics.

```
int aussie_log2_integer_clz_intrinsic(unsigned int u)
{
    // LOG2 using CLZ
    int clz = __lzcnt(u);  // Count leading zeros
    const int bits = 8 * sizeof(u);
    return bits - clz - 1;
}
```

And here's the macro version for those who don't trust compilers to inline properly:

```
#define AUSSIE_LOG2_LZCNT(u) \
 ((8*sizeof(unsigned)) - (__lzcnt((unsigned)(u))) - 1)
```

And this is actually not optimal. We really should help the C++ optimizer by reordering this to move the "-1" subtraction operation next to the other constant, noting that "sizeof" is a compile-time constant expression in C++. Putting them together would make sure that the compiler correctly merges these operations using constant folding. On the x86 implementations, the CLZ builtin functions are presumably using the x86 LZCNT or BSR assembler instructions, which are both similar and fast.

Bug alert! Note that you can't use "ffs" (find first set bit) for this log2 method, because it gives you the offset of the least-significant set bit (i.e., the rightmost bit rather than the leftmost bit). The other x86 instructions of TZCNT (Trailing Zeros Count) and BSF (Bit Scan Forward) are also incorrect.

Example: Highest Integer Power-of-Two

Another simple trick related to the log2 calculation is to truncate a number to its largest power-of-2. This is equivalent to the value of its leftmost bit in binary representation.

For example, 8 (binary 1000) stays as 8, because it's 2^3, but 7 (binary 111) reduces down to 4 (binary 100), which is 2^2. As with the truncated integer log2 calculation, this method focuses on computing the leftmost 1 bit, which is known as the Most-Significant Bit (MSB).

Whereas the log2 calculation found the index position of that MSB, this power-of-two calculation requires the *value* of the MSB. In other words, we need to find the bit that is the MSB, and then keep only that bit. A simple way to do this is to compute the log2 of the integer efficiently, and then left-shift a 1 by that many places (using unsigned type). The basic idea is:

```
   int bitoffset = log2_integer_fast(i);
   int highestpowerof2 = 1u << bitoffset;
```

Note that this doesn't handle cases like zero, so it still needs a bit of extra code polishing work.

Integer Overflow and Underflow

Integer arithmetic overflow and underflow have traditionally been ignored in C++ programs, mostly by assuming that operations won't exceed the range of 32-bit integers. Most platforms don't fail on integer overflow, and quietly continue, without even triggering a signal like SIGFPE (floating-point error).

The absence of runtime warnings can potentially leave insidious bugs in your code, and is also an undefended attack vector for security. Also, perhaps ignoring overflow isn't the best strategy.

Integers have a fixed range of numbers that they can represent. For example, a signed 16-bit integer represents the relatively small range of -32,768 to +32,767, and an unsigned 16-bit number can be from 0 to 65,535. A 32-bit signed integer has a much bigger range from about negative 2 billion (-2,147,483,648) to about positive 2 billion (+2,147,483,647). For an unsigned 32-bit integer, there's no negatives, and the range is from zero up to about 4 billion (+4,294,967,295).

Feel free to memorize those numbers, as you'll be needing them at least once a decade. The ranges for 64-bit integers are massive numbers around `2^64`, which is approximately decimal `10^19`.

If integer arithmetic on a data type falls outside the range supported by that integer type, then an overflow or underflow occurs. There are symbolic constants for the minimum and maximum numbers for many types are in the `<limits.h>` standard header file.

- `int` — `INT_MAX` and `INT_MIN`
- `unsigned int` — `UINT_MAX` and `UINT_MIN`

The effect of integer overflow or underflow is platform-specific, but on most platforms, it is usually: *nothing!* It's a silent insidious bug in many cases. For a signed integer, overflow quietly wraps around from positive to negative, and underflow does the reverse.

Here's an example of overflow of an `int` type:

```
    int x = INT_MAX;
    assert(x >= 0);
    ++x;  // Overflow!
    assert(x < 0);
```

And this is underflow of `int`:

```
    int x = INT_MIN;
    assert(x < 0);
    --x;  // Underflow!
    assert(x > 0);
```

Floating-point types can represent much larger magnitude numbers than integers. Hence, another way for an integer to overflow is in a conversion from floating-point numbers.

```
    float f = (float)INT_MAX * (float)INT_MAX;  // Fine!
    int x = (float)f;  // Overflow!
```

For an unsigned integer, the results are a little different, since negatives are not possible. Instead, overflow wraps around from a large number to zero, and underflow (going below zero) wraps around to the largest unsigned number.

Preventing Integer Arithmetic Overflow. There's not really a good way to detect arithmetic overflow or underflow before it happens. Post-testing is easier.

For example, GCC and Clang have some intrinsics, such as "`__builtin_add_overflow`" for addition, which use post-testing of the x86 CPU overflow or carry flags for detecting integer overflow, and return a Boolean flag which you can use. The GCC documentation say it uses "conditional jump on overflow after addition" and "conditional jump on carry" for unsigned overflow. Here's an example:

```
if (__builtin_add_overflow(x, y, &z)) {
    // Overflow!
}
```

The mainstream prevention strategy is simply to choose a big integer type (at least 32-bit) and then hope that no outliers occur in your input data. Most programmers let the overflow occur and then check. Or rather, just between you and me, most programmers simply don't even check at all!

Technically, integer overflow is "undefined behavior" on C++, and it's certainly non-portable, so you really should check. But most platforms handle it the same way, by quietly wrapping the integers around in two's complement form.

Increment overflow. For incrementing integers, you can do a pre-test like:

```
if (INT_MAX == x) {
    // Overflow!
}
else {
    x++;  // Safe increment
}
```

Addition overflow. And here's a version to pre-test addition of two positive integers for overflow:

```
if (x > INT_MAX - y ) {  // x + y > INT_MAX
    // Overflow!
}
else {
    x += y;  // Add safely
}
```

Multiplication overflow. The test for multiplication overflow is even worse because it uses division:

```
if (x > INT_MAX / y ) {  // x * y > INT_MAX
    // Overflow!
}
else {
    x *= y;  // Multiply safely
}
```

Head in the sand approach. Unfortunately, pre-testing for overflow is massively inefficient, as shown above. Do you really want to do this for every addition or increment? Even post-testing for overflow isn't much better. Overall, there's good reason why most C++ programmers just skip it, and hope for the best.

Overflow management. The alternative to ignoring the problem is to consider various different risk mitigation strategies for integer overflow:

- Larger data types (e.g., `long`) for a larger range.
- Use floating-point types instead.
- Use `unsigned` type for non-negative variables (e.g., sizes, counts).
- Use `size_t` for the `unsigned` variable type (it's standardized).
- Enable compiler runtime checks (when debugging/testing)
- Range checking input numbers (e.g., model weights).
- Post-testing the sign of arithmetic results.
- GCC and Clang intrinsic functions with overflow testing.
- The `<stdckdint.h>` header file in C23 (that's the C standard, not C++23).
- Safe integer class wrappers.

Runtime overflow detection. Some C++ compilers provide limited support for runtime error checking of arithmetic. The x86 CPU has builtin overflow detection, with a quietly-set overflow flag and a carry flag, which some C++ compiler-writers have made use of.

GCC has an "`-ftrapv`" option which elevates overflow errors (presumably by using post-checking). GCC has defined a number of C++ intrinsic functions which you can use to perform overflow-safe integer arithmetic, such as:

- `__builtin_add_overflow` — addition
- `__builtin_mul_overflow` — multiplication

Microsoft Visual Studio C++ provides the "/RTC" option, which stands for "Run-Time Checks", or there's "Basic Runtime Checks" in the MSVS IDE Project Settings. However, these MSVS features don't check much for arithmetic overflow, with a focus on stack frame checking and uninitialized variables. The closest is "/RTCc" to detect data type truncations at runtime.

There's also a runtime debugging tool that focuses on integer overflow and other oddities. It's named "Undefined Behavior Sanitizer" or UBSAN for short. It works like Valgrind, by adding runtime instrumentation code.

Safe integer classes. Currently there's no standard safe integer types in C++, but adding them was unsuccessfully proposed in 2016. If you like a busy CPU, and what programmer doesn't, you can replace all `int` variables with "safe integer" class objects, with many examples of such classes available on the Internet. They're probably not as bad as I've implied, since C++ inlining should make the critical path quite short.

Missing Operators: NAND, NOR, XNOR

Note that there's no simple operator for NOR, NAND or XNOR in C++. And you might need them, since neural networks uses these uncommon bitwise operations more than normal C++ coding. For example, XNOR is needed as the vector dot product operator for binarized bit vectors, such as in binary quantization and also XNOR neural networks.

These missing operators can be easily simulated using two C++ bitwise operations, with a binary bitwise operation and the "~" bitwise two's complement unary operator afterwards.

```
NAND(x,y) = ~(x & y)
NOR(x,y)  = ~(x | y)
XNOR(x,y) = ~(x ^ y)
```

So, you can just code this as fast C++ macros, right?

```
#define NAND(x,y) ~(x & y)  // Bug alert!
#define NOR(x,y)  ~(x | y)
#define XNOR(x,y) ~(x ^ y)
```

No, this is broken in about half a dozen ways.

To write macros correctly, you need to ensure there's parentheses around the whole expression, and also around each parameter name, to avoid getting bitten by C++ macro expansion operator precedence problems. And these macros also don't work correctly if you pass in a non-unsigned integer.

Here's some example C++ macros that work for 32-bits:

```
#define AUSSIE_BITWISE_NAND(x,y) \
  (~(((unsigned)(x)) & ((unsigned)(y))))
#define AUSSIE_BITWISE_NOR(x,y)  \
  (~(((unsigned)(x)) | ((unsigned)(y))))
#define AUSSIE_BITWISE_XNOR(x,y) \
  (~(((unsigned)(x)) ^ ((unsigned)(y))))
```

You could also declare these macros as "`inline`" functions if you prefer. Note that these macros have a lot of parentheses to avoid various insidious precedence errors, and they also are limited to 32-bit operations. For 64-bit, you'd need to create alternative "`unsigned long`" versions.

These NAND/NOR/XNOR macros are convenient, but not very efficient since they perform two arithmetic operations. Single-operation versions are available in assembler if you really need them, accessible via C++ builtin intrinsic functions such as:

- `_kxnor` — x86 intrinsic for XNOR bitwise operation.
- `KXNORW/KXNORB/KXNORQ/KXNORD` — x86 assembler bitwise XNOR operations.
- `VPTESTNMB/VPTESTNMW/VPTESTNMD/VPTESTNMQ` — x86 assembler bitwise NAND operations.

Note for the sake of completeness that there are more weird bitwise operators that do different things on a pair of bits. There are four input combinations and therefore 16 possible binary operator functions. There are three C++ bitwise operators (AND/OR/XOR), plus the three extra ones coded above (NAND/NOR/XNOR), two trivial always-zero and always-one operations, two copy-operand functions, and six other ones that are equivalent to variations with negated operands (e.g., "`x&~y`" is one).

I'm not sure why you needed to know that.

Bitwise AI Applications

Bitwise operations are a well-known coding trick that has been applied to neural network optimization. Bitwise-shifts can be equivalent to multiplication and division, but faster. Other bitwise operators can also be used in various ways in inference algorithms. Some of the common uses of bitwise operators in AI engines include:

- **Arithmetic computation speedups:** Bit tricks are used in optimizations of multiplication operations with bitshifts, and also faster approximate arithmetic methods.
- **Sign bit manipulation:** Various optimizations are possible by direct bitwise operations on the sign bit of integers or floating-point numbers. For example, the RELU activation function tests for negatives, which are changed to zero, but positive values are unchanged. This can be implemented efficiently as a sign bit test.
- **Floating-point bit operations**: The bits of the numeric representations of IEEE 754 floating-point numbers, or the Google `bfloat16` type, include a sign bit, an exponent, and a mantissa. Normal bitwise arithmetic operators cannot be applied to floating-point numbers, because the C++ bitwise and bitshift operators only work on integer types. However, floating-point numbers are really just integers underneath, so there are various tricky ways that bitwise operators can be used on the underlying IEEE standard bit representations that are used by floating-point numbers. This is discussed in the next chapter on floating-point optimizations.
- **Look-up Tables**: Algorithms that use table lookups for speed improvement typically involve bitwise shifts in computing the table offset.
- **Data structures:** Some data structures used in optimization of neural networks that involve bits include hashing and Bloom filters.

Bits of AI Research: Some of the advanced areas where bitwise optimizations have been used in neural network research include:

- **Power-of-two quantization (bitshift quantization):** By quantizing weights to the nearest integer power-of-two, bitwise shifts can replace multiplication.
- **Bitserial Operations**: Bitserial operations are bitwise operations on all of the bits of an integer or bit vector. For example, the "popcount" operation counts how many 1s are set in the bits of an unsigned integer. The bitserial operations can be useful in neural network inference for computing the vector dot products in binary quantization or 2-bit quantization.

- **Advanced number system division:** See dyadic numbers and dyadic quantization for an obscure number system involving power-of-two division, which can be implemented as bitwise right-shifting.
- **Low-bit integer quantization:** When quantized to only a few bits, inference can use bitwise arithmetic and bitserial operations to replace multiply-accumulate. The main examples are binary quantization and ternary quantization, both of which avoid multiplication operations in favor of bitwise operations (or addition) and sign bit handling.
- **Shift-add networks:** Multiply-and-add (or "multiply-accumulate") can be replaced with bitshift-and-add.
- **Bit arithmetic neural networks**. These are neural networks where the neurons operate as bitwise operations. For example, see Weightless Neural Networks (WNNs).
- **XNOR Networks:** XNOR neural networks are similar to binarized networks. Their internal operations rely on the bitwise XNOR operation. The idea is that XNOR is actually an implementation of the multiplication operation on binary values. XNOR is an uncommonly used bitwise operation, and there's no builtin C++ operator for binary XNOR. However, there is always hardware XNOR support, such as a 64-bit XNOR instruction in the x86 CPU instruction set.

References on Bitwise Operations

If I've whetted your appetite for bit fiddling magic, there's plenty more:

1. Sean Eron Anderson (2005), *Bit Twiddling Hacks*, Stanford University, https://graphics.stanford.edu/~seander/bithacks.html
2. Ian Brayoni (2020), https://github.com/ianbrayoni/bithacks (Python code inspired by Sean Eron Anderson's Bit Twiddling Hacks.)
3. Henry S Warren (2012), *Hacker's Delight, 2nd Edition*, Addison-Wesley Professional, https://www.amazon.com/Hackers-Delight-2nd-Henry-Warren/dp/0321842685 Code: https://github.com/hcs0/Hackers-Delight
4. Antonio Gulli (2014), *A Collection of Bit Programming Interview Questions solved in C++ Kindle Edition*, https://www.amazon.com.au/Collection-Programming-Interview-Questions-solved-ebook/dp/B00KIIDPUG/
5. Jörg Arndt (2010), *Matters Computational: Ideas, Algorithms, Source Code*, https://dl.acm.org/doi/10.5555/1941953, https://www.jjj.de/fxt/fxtpage.html#fxtbook,
   Code: https://www.jjj.de/bitwizardry/bitwizardrypage.html
6. Sigrid/Jasper Neuman (2023), Programming pages, http://programming.sirrida.de/

7. Harold (2023), *Bits, Math and Performance*, Sep 2023, http://bitmath.blogspot.com/
8. Stephan Brumme (2023), *The bit twiddler*, https://bits.stephan-brumme.com/
9. Gurmeet Manku (2008), *Fast Bit Counting*, 5 Aug 2008, https://gurmeet.net/puzzles/fast-bit-counting-routines/

41. Floating-Point Computations

What are Floating-Point Numbers?

Floating-point numbers are typically stored in 32 bits for single-precision C++ "`float`" types, and it's actually a 32-bit integer behind the scenes. The main floating-point types that you already know from C++ programming are:

- Single-precision floating-point — 32-bit `float` (FP32)
- Double-precision floating-point — 64-bit `double` (FP64)

The smaller 16-bit floating-point numbers that are never used in everyday C++ coding, but are important for AI, include:

- Half-precision IEEE type — 16-bit "`short float`" (FP16)
- Half-precision Bfloat16 type — 16-bit "Brain float" (BF16)

If only there was really a "`short float`" type in C++. The BF16 type is the non-IEEE 16-bit float version from Google Brain. Note that there is new standardized support for these 16-bit types in C++23.

Which type of floating-point number should you use? That's when things get tricky, because there are many wrinkles in the choice between 32-bit and 16-bit floating-point. It's not always clear which floating-point size is the best to use. FP32 is the most common size used in basic Transformer inference, but FP16 is a good choice for quantization of models, because they are compressed to half the size and retain good accuracy. And BF16 has been very effective in terms of GPU-accelerated algorithms.

Some hardware accelerators support different formats and sizes for their parallel operations. And there are various software problems with portably coding 16-bit floating-point data types in C++, along with variable hardware support for 16-bit operations across platforms.

Less importantly, there are also some other floating-point sizes, both bigger and smaller:

- Quarter-precision type — 8-bit floating-point (FP8)
- Quadruple-precision type — 128-bit "quad" floating-point (FP128)

FP8 is mainly seen in research papers, and hasn't really caught on for quantization (8-bit integers are typically used instead). The bigger sizes FP64 and FP128 aren't really needed to make your model work accurately, so their significant extra cost in speed and size isn't worthwhile for only a small perplexity gain in most use cases.

Bit Representations of Floating-Point

Standardized bit patterns are used to represent floating-point numbers in a kind of scientific notation. There are three types of bits:

- Sign bit
- Exponent bits
- Mantissa bits

Firstly, there's one bit for the sign, indicating whether the whole number is positive or negative. Then the remaining bits are split up between the "exponent" (i.e., the "power"), and the "mantissa" (also called the "digits" or the "significand" or the "fraction"). In a standard 32-bit "`float`" type used in AI, there is:

- 1 sign bit
- 8 exponent bits
- 23 mantissa bits

How does that even make a number? Well, it's like scientific notation, if you are familiar with that. The exponent is the power and the mantissa is the digits.

Let's pretend computers use decimal digits. If it were in base 10 storage, the decimal number `1234` would be stored as:

- "`0`" for the sign bit — because non-negative.
- "`3`" in the exponent — the power is `10^3=1000`.
- "`1234`" as the mantissa — the digits make the fraction "`1.234`".

This would represent `+1.234x10^3` (which hopefully equals `1234`). That's how it would work for a decimal version.

But, as you know, silicon beasts are not decimal. A floating-point number is actually stored in binary, in a kind of base-two "binary scientific notation" numbering scheme. So, conceptually, `1234` would be stored as a power-of-two exponent that represents the largest power-of-two, which would be `1024`, because `2^10=1024`, so the exponent has to store power "`10`" (ten), which is `1010` in binary. And the `1234` would be converted to whatever the heck `1234/1024` is when you represent that in binary 0's and 1's, and remove the decimal point (which is implicitly "floating," you see?).

It's more complicated than this, of course. That's what standards are for! The exponent bits are actually stored with an "offset" number (also called a "bias"), which differs by the size of the exponent bits. And there also some special bit patterns for particular numbers, such as zero or "`NaN`" (not-a-number).

Clear as mud? Don't you wish someone could go back in time and invent a base-10 computer?

Standardized Bit Representations

There's nothing magical about the choices of how many exponent versus mantissa bits. In the early days, there were many variations, but then they were mostly standardized by the IEEE 754 standard.

32-bit Floating-Point Numbers: The most common type of floating-point is 32-bits, such as the C++ "`float`" type. Other than the sign bit, there are usually 31 bits to split between the two other types, and the standard method is:

- Standard FP32 (IEEE754). Usually a "`float`" in C++, or "single precision" number. Standard 32-bit floating-point is represented in binary as: 1 sign bit, 8 exponent bits, and 23 mantissa bits (plus an implied prefix '1' mantissa bit that isn't actually stored, so it's really 24 bits of mantissa values). The exponent is stored with offset 127.

16-bit floating-point Numbers: With the "half" float types, there are 16 bits. There are a few common representations of floating-point numbers in different numbers of bits.

The main ones are:

- Half-precision (FP16). This is the standard 16-bit floating-point number, also sometimes called "`float16`". Annoyingly, there no standard "`short float`" or other widely used predefined type in C++, although the C++23 standard adds one, so this may be changing soon. The most common IEEE754-standardized version of FP16 type uses 1 sign bit, 5 exponent bits, and 10 stored mantissa bits (plus implicit mantissa bit makes 11 bits). The exponent is stored with offset 15.

- `Bfloat16` (brain float 16 or BF16): This is a different 16-bit floating-point numeric format, originally proposed by the Google Brain division, specifically for use in AI applications. `Bfloat16` has 1 sign bit, 8 exponent bits and offset 127 (like FP32), and 8 mantissa bits (7 stored, 1 implicit). It is like FP32 but with the two lowermost bytes just thrown away, so conversion between `bfloat16` and FP32 is simpler than converting from FP32 to FP16.

8-bit Floating-Point (FP8). The use of FP8 mainly appears in quantization research papers, but its usage is increasing within industry. There is usually 1 sign bit, 4 exponent bits, and 3 mantissa bits (which makes 4 bits with an implied extra mantissa bit). The other type of FP8 is 1 sign bit, 5 exponent bits, and 2 stored mantissa bits (3 bits total). Interestingly, the NVIDIA H100 GPU supports both of these FP8 formats.

FP16 Problems in C++

I already mentioned how there's not a standard half-precision type in C++, although that is fixable in the future, once compilers have implemented the C++23 standard. Here are some of the attempts at a 16-bit type:

- `__fp16` — only supported by ARM architecture.
- `_Float16` — not portably supported.
- `short float` — doesn't seem to exist (I'm just wishful-thinking!).
- `std::float16_t` — defined in the C++23 standard.
- `std::bfloat16_t` — defined in the C++23 standard.

So, as of writing, if you want to code a 16-bit float in a portable way with C++, there's an ugly hack: `short int`.

A less fixable obstacle is that converting between FP32 and FP16 is not easy because their exponent bit sizes are different. So, it's fiddly to code, and not very efficient.

The alternative idea is to use "`bfloat16`" (BF16), which is the upper-most two bytes of FP32. Converting is just a bitshift 16 places or playing with bytes, so it's faster than FP16.

However, BF16 isn't high precision. With 8 mantissa bits (7 stored, 1 implicit), that's only about 3 decimal digits, because `8/3.3=3`, and `3.3` is `log2(10)`, in case you were wondering. But it's not much worse than FP16, which is only about 4 decimal digits using 11 binary mantissa bits.

Representing Zero

The sign bit, exponent, and mantissa can represent a lot of numbers, but not zero. We cannot just set all the mantissa bits to zero, because that's not zero, which is rather strange.

There's an implicit extra "1" bit so all the mantissa bits clear isn't `0.0000`, it's `1.0000`. It always starts with a "1" digit and there's literally no way to represent `0.0000`.

Also, the exponent can represent `-127` to `+128`, but setting the exponent to `0` also isn't zero, because `2^0` is `1`. And `2^-127` is very small and does get us very close to zero, but it's also not zero. With sudden horrifying insight, we realize:

There's no way to represent zero!

The solution is that the IEEE 754 standard designers decided to treat all bits zero as being really zero. All bits zero in the exponent is `0`, but then subtracting the `127` offset, means that it is `-127` (the smallest number). So, if we clear all the exponent and mantissa bits to zeros, the number should be `1.0x2^-127`, but we can all pretend it's actually zero. Then we can do some pretend coding, ahem, I mean *microcoding*, so that all our Floating-Point Units (FPUs) pretend it's zero, too.

Negative zero. Weirdly, there are two zeros: normal zero and negative zero. The IEEE 754 standard allows two different bit patterns to mean zero, depending on the sign bit. If we clear all the exponent and mantissa to zero, then the sign bit zero means zero, but the sign bit set to "1" means "negative zero".

I'm not really sure what negative zero even means! But sometimes when you work with floats, a `0.000` number will get printed with a "`-`" in front of it. Maybe it's negative zero, or maybe a tiny negative with hidden digits at the 15th decimal place.

Fortunately, most of the arithmetic operations treat negative zero the same as zero. The C++ compiler handles it automatically. Adding negative zero does nothing, and multiplying by negative zero is also zero. But one of the gotcha's if you're being tricky with the bits of a 32-bit floating-point number, by pretending it's a 32-bit integer: testing for zero isn't one integer comparison, it's two!

Representing Special Numbers

We've already discussed how zero is handled specially, and has a wonderful dichotomy. The full list of special floating-point numbers is:

- Zero
- Negative zero
- `+Inf` (positive infinity)
- `-Inf` (negative infinity)
- `NaN` (Not a Number)
- Denormalized numbers (subnormal numbers)

Whereas zero is represented by the exponent being all 0s, the special numbers `Inf` and `NaN` are represented by the exponent with all 1s. So, this means that the huge number `2^+128` is not actually represented, but reserved for these special values. And honestly, that's fine, because if `2^+128` isn't infinity, then I don't know what it is.

Infinity: `Inf` is represented by all 1s in the exponent, but all 0s in the mantissa. And if the sign bit is 1, then it's `-Inf` (negative infinity).

Not-a-Number: `NaN` also has all 1s for the exponent, but any other pattern of the mantissa bits means `NaN`. This means that there are many versions of `NaN`, for all variations of the mantissa bits, except when all mantissa bits are 0 (which means `Inf`). Also, if the sign bit is set, then the same patterns are also `NaN` (another kind of "negative `NaN`", but that distinction is rarely used).

Denormalized numbers: Apparently, the designers of the floating-point standards think there's a "huge" difference between `2^-127` and zero. So, they decided to "smooth" it out a little by using some special numbers called "denormalized numbers" (also called "subnormal numbers").

The standard does this by getting rid of the "implicit" mantissa bit. For one special exponent value, all 0s, the standard changes the meaning to consider the implicit hidden mantissa bit to be a leading 0, rather than a leading 1.

Hence, the mantissa can represent fractions less than `1.0`, such as `0.1101` rather than only `1.1101` (in binary). The special exponent with all 0s therefore never represents `-127`, but represents the special value zero (or negative zero) if all the mantissa bits are 0s, or a tiny denormalized number if any of the mantissa bits are set.

And even though the exponent with all 0s should represent `-127`, we pretend that it is `-126`, one less, for the denormalized numbers, for "smoothness" reasons that I leave as an exercise to the reader, mainly because I don't understand it. Note that denormalized numbers can also be tiny negatives if the sign bit is set.

Denormalized numbers are all very, very tiny, being less than `2^-126`, so this feature of floating-point standardization is more useful for high-precision scientific calculations at NASA or SpaceX, rather than for most applications. In fact, here's the news about denormalized numbers in most coding:

We don't use denormalized numbers.

In fact, we hate them, because they make our FPU run slow. So, really, the slowness of our floating-point code is the fault of the FPU hardware engineers, as we've long suspected. Fortunately, there's a way to turn denormalized numbers off and run faster, which is discussed below.

To summarize and/or to further confuse things, the exponent has two special cases: all 0s and all 1s. If the exponent bits are all 0s, the number is either zero (or negative zero) or a denormalized number (a tiny positive or negative). If the exponent bits are all 1s, then the number is `Inf` or `NaN` (or negative `Inf/NaN`).

Testing for Special Values: The C++ standard has a number of fast routines to test a floating-point number. Some of the useful ones in `<cmath>` include:

- `std::isinf()`
- `std::isnan()`
- `std::isnormal()`
- `std::isfinite()`

For more general analysis of floats, `std::fpclassify()` in `<cmath>` returns a code that matches special enum values:

`FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`

Unfortunately, it's hard to distinguish positive and negative infinity, or to detect negative zero using these functions. You'll need to add a call to the "`std::signbit`" function (since C++11 for `float` arguments or C++23 for `double`), which returns `true` if a floating-point number has the sign bit on. There also a "`std::copysign`" function to copy the sign from one `float` to another, which can be used for sign bit manipulations. Alternatively, define your own bitwise macro tricks for these inspections.

Underflow and Overflow

Underflow is when a tiny floating-point number becomes so small that we can only represent it as zero. This can be a very tiny positive or negative number. Note that a negative number with a huge magnitude (near negative infinity) isn't underflow; that's actually negative overflow. Underflow refers to tiny fractions.

Generally, underflow isn't a problem for most code, because a number that low isn't going to affect the results. Similarly, I don't think we need to worry much about subnormal/denormalized tiny numbers either. If a probability is `2^-127` (or `2^-126` for denormalized), well, it might as well be zero anyway.

If we're using `Bfloat16` for 16-bit processing, it still has 8 bit exponents, so the lowest value is almost the same number (about `2^-127`). If we've quantized the network to FP16 (also 16-bit but with a 5-bit exponent), then the lowest probability we can represent is `2^-31`, which is also a tiny probability.

Generally speaking, applications don't tend to worry about underflow in floating-point. If a floating-point calculation underflows, it should just go harmlessly to zero. More concerning would be integer underflow, which is a different issue of large negatives wrapping around to positives. Floating-point underflow is better behaved.

Overflow is when a number gets so large that it cannot be represented in floating-point. Note that there are two types of overflow: positive overflow and negative overflow.

The exponent is the problem for overflow. When the number is larger than the highest exponent power, then it's either a very large positive or a very large-magnitude negative number.

For an 8-bit exponent, that means `2^+127` (because `+128` is reserved for the special `Inf/NaN` numbers). For a 5-bit exponent in FP16, this means `2^+31`, which is, coincidentally, also a good salary to request at your next performance review.

Overflow can be a problem, but usually only in the low-bit processing code where arithmetic computations can sometimes go too high. When overflow occurs, it could become a special floating-point number (NaN or Inf), or an integer number might toggle over to negative (e.g., if integer-only-arithmetic quantized).

FTZ and DAZ CPU Modes

In many CPUs, the need to handle overflow, underflow and denormalized values is a cause of inefficiency. The CPU can do floating-point computations faster if it can ignore those situations. This would be in violation of the IEEE 754 standard, but sometimes you have to sacrifice greatness for speed.

There are two commonly used modifications to CPUs that speed up floating-point arithmetic, by ignoring underflow and tiny numbers:

> **Flush-To-Zero (FTZ)**. This mode means that when the results are "subnormal" they are "flushed" to zero instead of calculating the correct "denormalized" result. Since these denormalized numbers are tiny, this isn't a concern in most code.
>
> **Denormalized-Are-Zero (DAZ)**. This is similar to FTZ, but allows treating inputs that are some type of denormalized floating-point as a zero input.

Both these modes, FTZ and DAZ, are only relevant to very tiny numbers, well below the resolution that most applications need to worry about, so you can totally enable them, provided we can figure out how to do so. CPUs with support for the FTZ and DAZ modes include x86 CPUs and ARM Cortex cores, and likely other processors. Google TPU doesn't support FTZ/DAZ because it operates on `bfloat16` floating-point numbers.

Enabling FTZ and DAZ. Finding details on how to enable FTZ and DAZ is quite hard! For command-line options, it seems to be "`-ftz`" on Linux/Mac or "`/Qftz`" on Windows. To control these modes dynamically in C++ code, you need to modify the MXCSR x86-64 CPU control register at runtime to set (or clear) the bits corresponding to FTZ and DAZ. Some of the primitives available to do so via GCC intrinsics include:

- `__builtin_ia32_ldmxcsr`
- `__builtin_ia32_stmxcsr`
- `_mm_getcsr`
- `_mm_setcsr`

In MSVS, there are preprocessor macros for FTZ in `<xmmintrin.h>` and for DAZ in `<pmmintrin.h>` header files. These control the FTZ and DAZ bits in the MXCSR, which is a CPU register with flags to control the CPU and the FPU.

The C++ snippet to enable these modes looks like:

```
 #include <xmmintrin.h>
#include <pmmintrin.h>

void aussie_float_enable_FTZ_DAZ(bool ftz, bool daz)
{
  if (ftz) {    // FTZ mode
    _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
  }
  else {
    _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_OFF);
  }

  if (daz) {    // DAZ mode
    _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
  }
  else {
    _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_OFF);
  }
}
```

These intrinsics for FTZ and DAZ are dynamic C++ calls. You can also disable these modes in C++, or switch back-and-forth between them dynamically. The MXCSR values are per-thread, so these modes must be set at the start of every new thread.

Negative Zero

Floating-point representations have two zeros: positive zero (the usual "`0.0f`" one) and negative zero ("`-0.0f`"). Note that there's no negative zero in integers, but only in floating-point types, because integers use two's complement in C++.

Usually, you don't have to worry about negative zero float values, because all of the floating-point operations treat zero and negative zero as equal. Negative zero is not less than positive zero, but is equal instead. For example, the "`==`" and "`!=`" operators should correctly handle both zeros as the same, and testing "`f==0.0f`" will succeed for zero and negative zero.

Normal C++ operations on `float` types will automatically handle negative zero for you, such as "<" will treat the two zeros are equal, not less-than. This happens at the cost of some inefficiency.

Detecting Negative Zero. Testing for negative zero is not easy. Unfortunately, you cannot use the `std::fpclassify` function because it returns `FP_ZERO` for both positive and negative zero. Here are some fast macros for 32-bit floats that look at the bits by pretending it's an `unsigned` 32-bit integer:

```
    #define AUSSIE_FLOAT_TO_UINT(f)  (*(unsigned int*)&f)
    #define AUSSIE_FLOAT_IS_POSITIVE_ZERO(f) \
        (((AUSSIE_FLOAT_TO_UINT(f) )) == 0)  // All 0s
    #define AUSSIE_FLOAT_IS_NEGATIVE_ZERO(f)  \
        (((AUSSIE_FLOAT_TO_UINT(f) )) == (1u<<31)) // Sign bit
```

Note that these macros only work for `float` variables, not constants, because the address-of "`&`" operator gets a compilation error for floating-point constants (e.g., `0.0f` or `-0.0f`). Also, these only work for 32-bit `float` types, and comparable macros are needed for 64-bit `double` or 128-bit `long double` types.

Pitfall: Bitwise tricks on negative zero. There are some pitfalls with negative zero if you are trying to subvert the normal floating-point number representations and do bitwise operations on them (as I just did above!).

For example, if you're doing bitwise tests on a `float`, you may still need to test for two values of zero, such as using one or both of the above zero testing macros.

For magnitude comparisons of `float` types via their underlying bits, there's also a problem. Whereas positive zero is all-bits-zero and will equal integer zero or unsigned integer zero, negative zero has the uppermost bit set (the sign bit), so it will be a negative integer or a very large unsigned number. Hence, negative zero will sort as less than positive zero if using signed integer tests, or will sort as massively greater than many numbers if using unsigned integers for testing.

The problem with negative zero also means that doing any bitwise comparisons will fail. You cannot just compare the underlying integers for equality against each other, nor can you use byte-wise testing. For example, using `memcmp` for equality testing a `float` vector will occasionally fail for `float` values where positive zero compares against negative zero, leading to insidious bugs.

Optimization by Suppressing Negative Zero. Since negative zero introduces an inefficiency into basic `float` operations (e.g., == or `!=` with `0.0`), can we block it for a speedup?

Are there any settings that fix the CPU or the compiler to ignore negative zero?

The FTZ and DAZ modes are mainly for subnormal numbers, not negative zero. I'm not aware of any hardware CPU modes specifically for disallowing skipping negative zeros, and I wonder whether they would actually be a de-optimization anyway, by forcing the FPU to explicitly check for negative zeros. Apparently, FTZ might help avoid negative zero in computations, but I'm not sure it's 100% of cases. There is a GCC flag "`-ffast-math`" which disables the production of negative zero in software.

Negative Zero. Can we speed up the floating-point computations of our code by blocking all floating-point negative zeros? Then the FPU or GPU can assume there's only one type of zero, and run faster. We could either run in a negative-zero-disabled mode, or use our own bitwise test for floating point zero as all-bits-zero (i.e., using the unsigned integer trick).

What about zero values at runtime? Can we guarantee that it never contains a negative zero, and thereby speed up analysis?

Getting to the Bits in C++

The basic 32-bit floating-point number in C++ is a `float` with a size of 4 bytes. How can you manipulate the bits in a floating-point value, using the 32-bit `float` type? You cannot use any of the C++ bitwise operators on floating-point numbers, as they only work for integers.

The trick is to convert it to an unsigned integer (32-bit) with the same bits, and then use the integer bitwise operations. The obvious way to convert a `float` to `unsigned` is casting:

```
    float f = 3.14f;
    unsigned int u = (unsigned)f;  // Fail!
```

Nope. That doesn't get to the bits, because it does a proper conversion between floating-point numbers and integers, which is usually what you want when you aren't thinking about bits (i.e., all normal people).

To get to the bits in C++, we have to trick the compiler into thinking that it's already got an unsigned integer with pointer type casts:

```
    unsigned int u = *(unsigned int*)(&f);  // Tricky!
```

That's a bit old-school casting. Here's the modern way with `reinterpret_cast`:

```
    unsigned int u = *reinterpret_cast<unsigned int*>(&f);
```

Once we have the bits, then we can twiddle the bits of our unsigned integer to our heart's content. When we're finished, we can do the same trick in reverse to re-create a floating-point number:

```
    f = *(float *)(&u);    // Floating again...
    f = *reinterpret_cast<float*> (&u); // Trendy version
```

And here's a timely reminder that it's important to use an "`unsigned`" type in C++ for the bit faking code, because the ">>" right-shift operator has undefined behavior on negatives.

Other Methods: Type casts aren't the only way in C++. There's also a trick involving "`union`" structures, and you can also directly copy the bits to a differently typed variable using "`memcpy`" or "`bcopy`".

It seems to me that this type cast trick should be the fastest way, because a good compiler should convert the address-of, `reinterpret_cast` and indirection sequence into a simple variable copy, especially with the "`reinterpret_cast`" hint. However, I haven't actually benchmarked the speed of the different methods.

Pitfalls and Portability

Bitwise manipulation of float data is not the most portable code in the world. Let's examine some of the possible pitfalls in using these techniques.

Bitwise zero testing: If you've gone to the trouble to access the bits of a floating-point number, you might as well use them. Obviously, testing for "`0.0`" is a common requirement, so let's make it faster:

```
    #define FLOAT_IS_ZERO(f) \
     ((*reinterpret_cast<unsigned int*>(&f)) == 0u) // Bug!
```

Oops! We forgot about negative zero. There are two zeros in floating-point, depending on the sign bit, and it's hard to test it efficiently with bitwise operations (e.g., mask the sign bit or shift left first).

Strict anti-aliasing rule. An important point about all this is that most of it is platform-dependent, and officially "undefined behavior". Some of it is standardized by IEEE 754, but many variations are possible. Another issue is that there's a "*strict anti-aliasing rule*" that specifies that many of these tricks are officially non-standard methods. Accessing a floating-point number as if it's an unsigned number is a technical violation of this rule. The "`reinterpret_cast`" method is probably less likely to run afoul of this problem, but it's still not guaranteed.

Anyway, the `union` trick and the use of `memcpy` don't really strike me as being particularly more portable, although `memcpy` might be less likely to be optimized wrongly by a compiler making wrong assumptions. Some additional risk mitigations are warranted, such as adding a lot of unit tests of even the most basic arithmetic operations. However, you're still not officially covered against an over-zealous optimizer that might rely on there being no aliases allowed.

Byte sizes. Another much simpler portability issue is checking the byte sizes of data types, which can vary across platforms. Most of this bit-fiddling stuff relies on particular 16-bit and 32-bit layouts. It doesn't hurt to add some self-tests to your code so you don't get bitten on a different platform, or even by a different set of compiler options:

```
   aussie_assert(sizeof(int) == 4);
   aussie_assert(sizeof(short int) == 2);
   aussie_assert(sizeof(float) == 4);
   aussie_assert(sizeof(unsigned int) == 4);
```

Also note that for this to work well, both types must be the same size. So, this would be a useful code portability check if it worked:

```
   #if sizeof(float) != sizeof(unsigned int) // Fails!
   #error Big blue bug
   #endif
```

This macro preprocessor trick doesn't work because `sizeof` isn't allowed in a preprocessor expression, because the preprocessing phase precedes the syntax analysis. A better version uses a "`static_assert`" statement, which does compile-time checking in a more powerful way.

```
  static_assert(sizeof(float)==sizeof(unsigned), "Bug!");
```

Floating-Point Builtin Functions

The alternative to directly accessing the bits as an unsigned integer is to use the existing C++ functions. There are various existing functions for bitwise manipulation of floating-point numbers, in two categories: standard C++ library functions and compiler-specific intrinsics.

C++ has standard functions for the manipulation of floating-point numbers, and their bitwise representations.

- `std::signbit` — Portably test the sign bit of a floating-point number.
- `std::copysign` — Portably copies the sign bit from one `float`, merging it with the value of another (i.e., another's exponent and mantissa).

There are also various compiler-specific "intrinsics" or "builtins" to manipulate floating-point numbers. For the Microsoft Visual Studio C++ platform, these are in `<intrin.h>` and there are also versions for GCC and other compilers.

- `frexp` — Get the mantissa and exponent.
- `ldexp` — Bitshifting by an integer shift-count.
- `scalbn` — Also integer bitshift on a `float`.
- `logb` — Extracts the exponent.
- `ilogb` — Extracts the exponent to integer.
- `modf` — Splits into whole and fractional parts.
- `fma` — Fused multiply add on `float` (Microsoft intrinsic)
- `remainder` — Get fractional part of floating-point (Microsoft intrinsic)
- `_fcvt` — Low-level convert `float` to string (Microsoft intrinsic)

For many of the listed functions, there are additional versions for different floating-point data types, such as `float`, `double` and `long double`. For example, "`frexp`" will split a `double` type into its significand (fractional part) and exponent integer, but there's also "`frexpf`" for 32-bit `float` types, and "`frexpl`" for long double types.

Floating-Point Bit Tricks for AI

Once you've got the bits into an unsigned integer, what can you do?

Assuming you're willing to throw the standards documents to the curb, you can do quite a lot. The bits can be directly manipulated in non-obvious ways to speed up some types of floating-point arithmetic with integer bitwise arithmetic on the underlying bits. Examples of floating-point bit manipulations used to optimize neural networks include:

- Sign bit flipping: this can be used for fast non-multiplication binarized networks with floating-point computations.
- Exponent bit manipulations: bitshifting `float` values in logarithmic quantization can be implemented as integer addition on the exponent bits of a float.
- Add-as-integer networks: This method simply adds the underlying bit representations together as integers, to create a type of multiplication-free neural network. Weirdly, this simple trick implements an approximate multiplication algorithm known as Mitchell's algorithm.
- Fast `log2` computation on `float` types using the exponent bits directly.

The first step is to extract the bit patterns. Let's assume it's a standard 32-bit float type with 1 sign bit, 8 exponent bits, and 23 stored mantissa bits. You can get the different bits:

```
int signbit = (u >> 31);
int exponent = ( (u >> 23) & 255 );  // Fail!
int mantissa = ( u & ((1 << 23) - 1 ));
```

Nice try, but that's only 2 out of 3. The exponent is wrong here! The bits are correct, but it's not the right number. We have to subtract the "offset" (or "bias") of the exponent, which is 127 for an 8-bit exponent. This is correct:

```
int exponent = ( (u >> 23) & 255 ) - 127; // Correct!
```

Note that the sign bit and mantissa can be stored as `unsigned` (i.e., positive or zero), but the exponent must be a signed integer, even though it is extracted from the bits of an unsigned int. For a fraction like decimal `0.25` (i.e., a quarter), this is equal to `2^-2`, so the exponent is `-2`. In an 8-bit exponent, the range of the exponent is `-128` to `+127`. Note that the sign bit in a `float` specifies the overall sign of the whole number, and is not the sign of the exponent.

Here are some macro versions of the above bit extractions:

```
#define AUSSIE_FLOAT_SIGN(f) \
  ((*(unsigned *)&(f)) >> 31u)   // Leftmost bit
#define AUSSIE_FLOAT_EXPONENT(f) \
  ((int)(((((*(unsigned*)&(f)))>> 23u) & 255) - 127))
#define AUSSIE_FLOAT_MANTISSA(f) \
  ((*(unsigned*)&(f)) & 0x007fffffu) // Right 23 bits
```

Note that these macros don't work for constants, but give a compilation error such as "l-value required". This is because of the "&" address-of operator trick being used needs a variable, not a constant. I don't see an easy way around it for bitwise trickery.

If you dislike bits for some strange reason, here's a simple way to define the sign bit macro using the "<" operator, which also works on constants:

```
    #define AUSSIE_FLOAT_SIGN(f) ((f) < 0.0f) // Sign test
```

Example: Add-as-int Approximate Multiply

The add-as-integer method suggested by Mogami (2020) simply adds the integer bit representation of two floating-point variables, as if they are integers. It's quite surprising that this has any useful meaning, but it's actually a type of approximate multiplication called Mitchell's algorithm. Here's what the C++ code looks like on 32-bit `float` types:

```
    float aussie_add_as_int_mogami(float f1, float f2)
    {
        // Add as integer Mogami(2020)
        int c = *(int*)&(f1)+*(int*)&(f2)-0x3f800000;
        return *(float*)&c;
    }
```

The magic number `0x3f800000` is (obviously) equal to "`127<<23`" and its purpose is to fix up the offset of the exponent. Otherwise, there are two offsets with value 127 combined. (Is there a faster way? It's annoying to waste a whole addition operation on what's just an adjustment.)

Note that this algorithm is one exceptional case where we don't want to use `unsigned` integer types when tweaking bit representations. This trick needs the temporary variable of type "`int`" and the pointers to be "`int*`" so that it can correctly handle the sign bits of the two floating-point numbers.

This add-as-integer algorithm is not restricted to 32-bit `float` data. It should also work for 16-bit floating-point numbers in both `float16` and `bfloat16` formats, provided the magic number is changed to a different bitshift count and an added offset of 15 (not 127) for 5-bit exponents.

Example: Float Bitshift via Integer Addition

This is another surprising bitwise trick on floating-point numbers. You cannot perform the standard bitshift operators on `float` types in C++, so you cannot easily speed up floating-point multiplication via bitshifts in the same way as for integers.

Bitshifts are a fast way of doing an integer multiplication by a power-of-two (e.g., "`x<<1`" is the same as "`x*2`"). Note that it also doesn't work to convert the `float` to its `unsigned int` bit version and shift it using integer bitshift operators.

On some platforms, there are some builtin special functions such as `ldexp` and `scalbn` for doing bitshifting on `float` data. The `ldexp` function accepts an integer power, and then bitshifts a floating-point number by this many places. The `ldexp` function is for `double` types, `ldexpf` is for `float`, and `ldexpl` is for `long double` types. The `scalbn` set of functions appears to be almost identical to `ldexp` functions. There is also a reverse function "`frexp`" which extracts the significant (fraction) and the power-of-two for a floating-point argument.

Although we can't bitshift floating-pointer values, there is an intriguing alternative optimization using integer arithmetic directly: *addition*. The suggestion in the DenseShift paper (Li et al., 2023) is to simply add the shift count to the exponent bits using integer addition.

Here's some example C++ code that works for 32-bit floating-point numbers:

```
    float aussie_float_bitshift_add_int(float f1, int bits)
    {
        // Bitshift float by adding int to exponent bits
        // FP32 = 1 sign bit, 8 exponent, 23 mantissa
        unsigned int u = *(unsigned int*)&f1; // Get the bits
        if (u == 0) return f1;  // special case, don't change
        u += (bits << 23);  // Add shift count to exponent
        return *(float*)&u; // Convert back to float
    }
```

How does it work? Well, it makes a certain kind of sense. The exponent in a floating-point representation is a power-of-two, and we are bitshifting, which is increasing the number by a power-of-two. Hence, we can increase the power-of-two by adding 1 to the exponent, and it also works for adding by more than 1.

Note that this code also works for bitshift of a negative count (e.g., bitshift of -1 subtracts from the exponent and thereby halves the number) or zero (unchanged). However, this exponent-addition trick can overflow if the resulting number overflows or underflows the exponent range (e.g., `-128` to `+127`).

This method has thereby improved the runtime performance of floating-point multiplication by changing it to integer addition. The idea works provided we are multiplying by a power-of-two, which is done in logarithmic quantization. However, it's a little tricky in that special formats like zero (and `NaN`) are problematic for this algorithm. I had to add the test "`u==0`" which slows things down (maybe there's a better way?). Also, this approach can theoretically overflow the exponent bits, messing up the sign bit, but that's only if the `float` is very big or very tiny. Checking for all these wrinkles will slow down the code.

Example: Log2 Floating-Point is Exponent

The `log2` function for float types is a non-linear function that is quite expensive to compute. We already computed `log2` of an integer with low-level bit fiddling methods based on a count-leading-zeros algorithm in the bitwise operations chapter. There's also a different bitwise trick for `log2` of floating-point numbers. This method computes the truncated integer version of the `log2` algorithm (e.g., for use in logarithmic power-of-two quantization). There's a very easy way:

The base-2 logarithm is the exponent!

It's sitting right there, already calculated, hidden in plain sight amongst the 32 bits of your friendly `float` variables. Here's some C++ code to extract it:

```
int ilog2_exponent(float f)  // Log2 for 32-bit float
{
    unsigned int u = *(unsigned int*)&f;
    int iexp = ((u >> 23) & 255);  // 8-bit exponent
    iexp -= 127;  // Remove the "offset"
    return iexp;
}
```

Alternatively, for greater portability and probably extra speed, too, there are some standardized builtin C++ functions available across various platforms (including Linux and Microsoft) that can extract the exponent: `frexp`, `ldexp`, `ilogb`, and `scalbn`, are some that come to mind.

References on Floating-Point

1. Eric Sakk (2018), *Understanding Floating-Point Numbers*, Concepts in Computer Systems (Volume 2), 7 June 2018, https://www.amazon.com/dp/1983093025/
2. Sean Eron Anderson (2005), *Bit Twiddling Hacks*, Stanford University, https://graphics.stanford.edu/~seander/bithacks.html
3. T. Mogami (2020), *Deep neural network training without multiplications*, In Beyond BackPropagation WS at 34th Conference on Neural Information Processing Systems, 2020, https://arxiv.org/abs/2012.03458 (Uses integer addition of the bits of an IEEE 754 floating-point representation to perform approximate floating-point multiplication.)
4. Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lerevre, Guillaume Melquiond, Nathalie Revol, Damien Stehle, Serge Tones (2018), *Handbook of Floating-Point Arithmetic*, Birkhauser, 2018, https://link.springer.com/book/10.1007/978-3-319-76526-6, Contents: https://cds.cern.ch/record/1315760/files/9780817647049_TOC.pdf
5. Wonyeol Lee, Rahul Sharma, Alex Aiken (2016), *Verifying Bit-Manipulations of Floating-Point*, Stanford University, USA, https://theory.stanford.edu/~aiken/publications/papers/pldi16b.pdf
6. Xinlin Li, Bang Liu, Rui Heng Yang, Vanessa Courville, Chao Xing, Vahid Partovi Nia (2023), *DenseShift: Towards Accurate and Efficient Low-Bit Power-of-Two Quantization*, Oct 2023, https://arxiv.org/abs/2208.09708 (Uses integer addition on the sign and exponent bits of IEEE 754 floating-point to perform bitshifts on floats to perform power-of-two number quantization on 32-bit floats.)
7. Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, Joey Yiwei Li (2021), *DeepShift: Towards Multiplication-Less Neural Networks*, July 2021, https://arxiv.org/abs/1905.13298 (Bitwise shifting and sign bit manipulation.)

42. Arithmetic Optimizations

Types of Arithmetic Optimizations

There are two basic ways that arithmetic computations can be sped up whilst retaining the same results:

- Single operator improvements
- Expression-level optimizations (multiple operators)

As an example of single operator optimizations, consider replacing the multiplication operator. Alternative forms of arithmetic include bitwise shifting or addition. The ways to do fewer multiplications tend to involve higher-level algorithmic changes to the model, such as pruning or quantization.

Some of the methods of speeding up arithmetic come from the theory of compiler optimization (e.g., strength reduction, sub-expression elimination). Hence, the compiler will often automatically perform these types of optimizations (when the optimizer is invoked). To some extent, this makes these transformations redundant.

Even so, good programming practice is to avoid situations where these optimizations are needed on a large scale. The compiler does not look at the program as a whole and can miss some "obvious" optimizations.

Operator Strength Reduction

Individual operations in C++ can be optimized in several ways. The general term is "strength reduction" because a stronger operator with high computation complexity is "reduced" to an equivalent operator that is simpler and faster.

Strength reduction is a technique used in automatic optimization by compilers, but can also be used by programmers to improve algorithms.

The main "strong" operations that we're trying to avoid are:

- Floating-point arithmetic (even addition)
- Multiplication
- Division
- Remainder (`%` operator)
- Math functions (e.g., `sqrtf` or `expf`)

Strength reduction has particular relevance to AI engines because the main bottleneck is floating-point multiplication. Many of the research papers on speedups are about replacing the floating-point multiplication operation with something simpler, like addition or integer arithmetic.

Some of the general approaches in regard to strength reduction include:

- Bitwise operations (e.g., bitshifts can replace multiplication)
- Multiplication is slower than addition.
- Avoid division and modulo/remainder operators (they're the worst!)
- Use integer arithmetic rather than floating-point (where possible)
- Use `float` single-precision arithmetic, not double-precision.
- Approximate arithmetic (e.g., for math functions)

Bitshift for multiplication: The canonical example that everybody knows is that shift operators can replace multiplications by a power of two. But it's only for integers, not for floating-point numbers. Here's a dummy example of integer multiplication;

```
    y = x * 4;
```

This can be more efficiently coded as a left bitshift:

```
    y = x << 2;
```

Bug alert! If you're making this code change, you're likely to introduce some bugs. The "<<" and "*" operators have different precedence levels, so make sure you add more parentheses. Also, consider whether you need to use "`unsigned`" type when switching to a bitwise operator.

Right shift for division: The use of bitshifting works for division, too (but only for unsigned):

```
   y = x / 4;
   y = x >> 2u;  // faster
```

Bitwise remainder calculations: The arithmetic modulus operator (remainder) can also be optimized for power-of-two operands (but only on integers):

```
   y = x % 512;    // Remainder (mod)
   y = x & 511u;   // Bitwise-AND
```

And here's another one with integer relative comparisons versus bitwise-and, although this one might not necessarily be faster:

```
   if (x >= 512)
   if (x & ~511u)  // Bitwise-AND of complement (unsigned)
```

Avoiding multiplication: There are some simple cases even with the most basic operators that have multiple options:

```
    y = x * 2;
    y = x + x;   // Addition
    y = x << 1;  // Shift
```

Automatic Strength Reduction: In theory, C++ compilers could know what will be faster on its platform, and perform all these optimizations automatically when compiling the program. The optimizers probably do some of them, but they cannot do them all.

Intrinsic Functions: Other more advanced types of strength reduction involve avoiding costly primitives, such as mathematical functions. For example, there are bitwise arithmetic tricks to quickly compute the integer `log2` function.

GPU Strength Reduction: One final note is that when doing AI coding work, we aren't as concerned about which C++ operator works the best. The more important concern is which operation is most efficient in the GPU or other non-GPU hardware acceleration (e.g., AVX-512 on CPU).

Finally, note that these optimizations are local optimizations, and the same ideas apply globally to the entire AI engine architecture. There's been a lot of research trying to change *all* of the arithmetic in model inference from multiplication to bitshifting, such as using addition or bitshifts.

Avoid % Remainder Operations

One common use of the remainder operator is the use of modulo arithmetic, such as the wraparound array implementation of a queue abstract data type, where the value of a variable is cyclically counted from `0` up to `N-1`, and then back to 0. The most common idiom for coding this is:

```
x = (x + 1) % N;
```

However, the `%` operator is expensive, and in this case it is not really needed. The following code sequence performs the same task more efficiently:

```
if (x == N - 1)
    x = 0;
else
    x++;
```

This can also be written more concisely, but not necessarily more efficiently, as an expression with the "`?:`" ternary operator:

```
(x == N - 1) ? (x = 0) : (x++);
```

Another example of a clever avoidance of `%` is when the operand is similar to the usual byte or word size. For example, consider this remainder:

```
x % 256
```

This can be more efficiently coded with bitwise-and using:

```
x & 255
```

But this can be even more efficiently coded as a type cast:

```
(unsigned char) x
```

The conversion to this "`unsigned char`" type will be efficiently implemented by grabbing a byte out of a word. Unfortunately, this method is not portable to all obscure systems, as it relies on an "overflow" being handled harmlessly, and on "`unsigned char`" always containing 8 bits.

Reciprocal Multiplication

Division is a slow operation, whether in a CPU or a GPU. Multiplication is often significantly faster than division, and in some cases a division can be replaced by a multiplication using the reciprocal. A case in point is floating-point division by a constant. For example, consider the division:

```
    f = g / 100.0;
```

This can be replaced by the multiplication:

```
    f = g * 0.01;  // Reciprocal
```

If the divisor is a symbolic constant, it is possible to replace the symbolic constant with a hard-coded constant (or another symbolic constant). However, it is more convenient to replace the constant with an explicit reciprocal calculation. For example, consider the code:

```
    f = g / DIVISOR;
```

This can be rewritten as:

```
    f = g * (1.0 / DIVISOR);
```

The compiler should calculate the reciprocal using "constant folding" at compile-time. Note that the brackets around the division expression are probably not strictly necessary because optimizers know about associativity, but are certainly helpful to make life easier for the optimizer (and these poor critters need a break every now and then).

If the divisor is a complex expression, the compiler might not automate the efficient use of a reciprocal. Here's the slow version of division by a scale factor:

```
    v[i] /= sqrtf(3.14159f);
```

Here's the faster way using the reciprocal of the constant:

```
    v[i] *= 1.0f / sqrtf(3.14159f);
```

And we really should pre-calculate this constant using constant folding and a `static` variable:

```
    static const float scalefactor = 1.0f / sqrtf(3.14159f);
    v[i] *= scalefactor;
```

Integer Arithmetic

Real arithmetic is slow compared to integer arithmetic. Hence, it is favorable to replace real arithmetic by equivalent integer arithmetic. Real arithmetic can be replaced by integer arithmetic when only limited precision is required (e.g., 1-3 decimal places). To do this, work in integer units that are 10, 100 or 1000 times larger (for 1, 2 and 3 decimal places) so that the decimal places appear as the lower digits of the integers.

To convert the integer into its true integer and fractional parts is quite simple. To get at the fractional part, calculate the number modulo 10, 100 or 1000 (using the `%` operator). To get the true integer part, divide by 10 or 100 or 1000 — remember that integer division truncates the fractional part.

A good example is: when working in dollars and cents, do calculations in terms of cents (an integer). Then when printing it out, convert to dollars and cents using:

```
    cents = value % 100;
    dollars = value / 100;
```

However, note that this is now using two of the worst integer operators: remainder and division. The hierarchy of cost for integer operations is similar to floating-point: integer addition and subtraction are faster than multiplication, but division is still the worst, even for integers.

There appears little to be done to replace integer division with multiplication. Multiplying by the reciprocal will change an integer operation to a floating-point operation and will probably increase execution time. A power-of-two integer division could be done via the ">>" right bitshift operator, provided that it cannot be negative and uses an `unsigned` type.

Expression Transformations

Expression-level types of arithmetic improvements on an expression with multiple operations include:

- Constant folding (compile-time precomputation of constant expressions)
- Common subexpression elimination (only computing things once in expressions)
- Algebraic identities in computations
- Type consistency (avoid conversions)

Common Subexpression Elimination

Common subexpression elimination (CSE) is avoiding the recomputation of the same expression twice. There are many cases where the same computation appears multiple times in a single expression, or across the control flow of a program. Compiler optimizers attempt to automatically detect such cases and reuse the first computation.

In a complicated expression, there are often repeated sub-expressions. These are inefficient as they require the computer to calculate the same value twice or more. To save time, calculate the sub-expression first and store it in a temporary variable. Then replace the sub-expression with the temporary variable. For example:

```
x = (i * i) + (i * i);
```

With a temporary variable, this becomes:

```
temp = i * i;
x = temp + temp;
```

Note that this attempt to be concise is incorrect:

```
x = (temp = i * i) + temp; // Bug
```

This may fail because of its reliance on the order of evaluation of the + operator. It is not actually guaranteed in C++ that the + operator is evaluated left-to-right.

Common sub-expressions do not occur only in single expressions. It often happens that a program computes the same thing in subsequent statements.

For example, consider the code sequence:

```
    if (x > y && x > 10) {
        // ...
    }
    if (x > y && y > 10) {
        // ...
    }
```

The Boolean condition “x>y” need be calculated only once:

```
    temp = (x > y);
    if (temp && x>10) {
        // ...
    }
    if (temp && y>10) {
        // ...
    }
```

Algebraic Identities

The calculations in some complicated expressions can be reduced by transforming the expression into another equivalent form. The aim when using algebraic identities is to group the operations differently, to reduce the total number of arithmetic operations. Care must be taken to ensure that the new expression has equivalent meaning. For example, the short-circuiting of the logical operators can cause differences. Some useful algebraic identities are:

```
    2 * x == x + x == x << 1
    a * x + a * y == a * (x + y)
    -x + -y == -(x + y)
```

There are also Boolean algebraic identities that can be used to perform fewer logical operations:

```
    (a && b) || (a && c) == a && (b || c)
    (a || b) && (a || c) == a || (b && c)
    !a && !b == !(a || b)
    !a || !b == !(a && b)
```

Float Type Conversions

Hidden unnecessary C++ type conversions are a common source of extra inefficiency. The main type in a Transformer is usually "`float`" (32-bit), rather than "`double`" (64-bit). Avoid unnecessary type conversion code in two ways:

- Don't mix float and double
- Don't mix float and int

The use of `float` and `int` tends to be something professional C++ programmers are aware of, after having been burned a few times, and doesn't occur that often by accident.

However, inadvertently mixing `float` and `double` is difficult to avoid, and sneaks into your code all the time. For example, here's some C++ code that looks perfectly correct:

```
    float scalefactor = sqrt(2.0) * 3.14159;
```

You know this isn't AI code because it doesn't have 27 decimal places for pi, which we've memorized by rote. AI engines don't really need anywhere near that much precision, but it looks good for the boss.

The above code is also a small slug, because it may be unnecessarily using "`double`" size arithmetic, although the compiler might fix it with constant folding (but emit a warning anyway). Here's the corrected code:

```
    float scalefactor = sqrtf(2.0f) * 3.14159f;
```

Note that this example shows there are two places where an "f" suffix is needed to signify that `float` arithmetic is required:

- Numeric constants (i.e., "`2.0f`" specifying a 32-bit `float`, rather than "`2.0`", which is a 64-bit `double` constant).
- Standard C++ functions (i.e., the "`sqrtf`" function returns `float` rather than "`sqrt`" returning `double`).

Without the suffix "`f`", in both cases the default is `double` type constants and `double` arithmetic functions. A lot of C++ compilers will warn about these type conversions losing precision, so if you aim for warning-free compilation as a quality goal, you'll also fix most of these wasteful hidden type conversions.

43. Compile-Time Optimizations

C++ Compile-time Techniques

Compile-time processing is the optimal way to run a program. All the work is done by the compiler and none by your program. There are literally zero instructions executed on the CPU at runtime, whether it's doing training or inference. It will be blindingly fast for your users.

If only all code could be like that!

The reality is that programmers are still needed and that code still needs to run (sigh!). But to make it faster, there are lots of ways to have more computation done by the compiler, long before it ever goes near a user.

The C++ programming language has numerous features that help perform work at compile-time. These include ways to explicitly control what goes to the compiler, or to give more information to the compiler so that its optimizer can do good work on your behalf. Some of the various C++ language features to consider include:

- Conditional compilation — `#if/#ifdef` statements
- `inline` functions
- Templates — these expand at compile-time
- Symbolic constants — `const` or `#define`
- Function-like macros — `#define` with parameters
- Constant hints — `constexpr`, `if constexpr`, etc.
- Global and `static` variable initializations
- `static` data members — fixed data in C++ classes
- Type traits — compile-time type testing
- Restricted pointers — ignore aliasing risks

But when we're doing AI, there's another compile-time data structure to consider: the whole LLM model itself.

C++ Optimizers

Every C++ compiler has optimization built into the code generation phase. Typically, there are ways to specify that a higher degree of code optimization should be performed. Methods to control the settings include:

- Command-line arguments (e.g., "`-O1`" or "`/O1`")
- Configuration settings (e.g., Project Settings in the MSVS IDE)
- `#pragma` preprocessor directives

Take note of the meaning of the optimizer settings. For example, on MSVS the setting "`/O1`" optimizes for memory, not speed! Also, don't be like me and assume that the defaults are going to be what you want.

Looking at the MSVS IDE optimizer settings in my AUSSIE project file, I found:

- "Optimization" was "disabled" by default.
- "Enable Intrinsic Functions" was "No" by default. Why not?
- "Favor Size or Speed" was "neither" by default. Come on, why is there no "both" option?
- "Inline Function Expansion" was "default" at least.

When to enable the optimizer? Should you run the optimizer at every build? At what level?

Note that your policy should *not* be to turn up the optimization to maximum level just before you ship your code to users, because your code can change in a very bad way.

Don't assume that turning the optimizer mode up to super-crunch is always an easy win, as optimization can trigger latent glitches in your code by reorganizing memory or reordering instructions.

What does the optimizer do? In order to optimize code, it's important to know what sorts of optimizations your compiler is doing automatically. Compilers have been doing optimizations for literally 50 years, and the state-of-the-art is quite amazing, with an extensive body of research theory.

Some of the main automated compiler optimizations include:

- Constant folding/propagation
- Constant expression evaluation
- Common subexpression elimination
- Redundant assignment removal
- Strength reduction
- Algebraic optimizations
- Register allocation
- Loop optimizations (e.g., unrolling)
- Auto-vectorization

If you make simple changes to your code with some of the obvious things above, it's not going to give you a speedup. The compiler has already done it for you.

However, there's a limit to what compilers can do. They certainly can't make architectural changes, and there's also many mid-level algorithmic changes that cannot be automated.

Function calls inside expressions are a good example of code changes that might need to be manually optimized. When the compiler sees a function call used in arithmetic, it isn't always able to know what that function is going to do, and has to be conservative by avoiding possibly incorrect optimizations.

Floating-Point Optimizer Options

Some C++ compilers have optimizations that you can use to speed up your Floating-Point Unit (FPU). Some of the options for GCC include:

- "`-ffast-math`" option — This option is a broad enabler of multiple floating-point speedups, such as `-fno-math-errno` and `-ffinite-math-only`. It also disables negative zero.
- "`-fno-math-errno`" option — This allows the standard library math functions such as `sqrt` to run faster and also be more amenable to parallelization, simply by allowing them to never set the global "`errno`" variable. The use of `errno` was once a great way to track error codes, but it's also a blocker for thread-safety and parallelization. And let's be frank: you weren't ever checking `errno` anyway, so turn it off!
- "`-ffinite-math-only`" — This mode allows GCC math library functions to skip any checks for `Inf` or `NaN`, which can make them marginally faster.

Microsoft Visual Studio C++ also has its own set of FPU options:

- "Floating-Point Model" settings in a Project's Property Pages under "C++" for "Code Generation" has options "`/fp:precise`", "`/fp:strict`", or "`/fp:fast`"
- "Enable Floating-Point Exceptions" can be turned off if you like.

People Helping Parsers

The humble C++ compiler needs your attention. Hat in hand, the compiler is sitting there saying "I am but a poor, helpless lexer, without even a single neural network. Please help me." Hence, please consider donating your time to help a poor struggling compiler in your neighborhood.

There is a long history of the C++ compiler needing "hints" about optimization from the programmer. The early C++ language in the 1990s had a "`register`" specifier that hinted to the compiler that a variable was going to be highly used, and the compiler should optimize it by putting the variable in a CPU register. The "`register`" keyword has since been deprecated in C++17, which indicates that compiler register allocation algorithms no longer benefit from human help.

Some of the other longstanding C++ keywords that can be used for efficiency-related purposes include:

- `inline`
- `const`
- `static`

And with the evolving C++ standards, there's a whole new set of directives that are hints to the compiler about how to optimize:

- `constexpr`
- `constinit`
- `consteval`
- `reinterpret_cast`
- restricted pointers ("`restrict`")
- `[[likely]]` and `[[unlikely]]` path attributes

The `constexpr` and related directives help the compiler do "constant folding" and "constant propagation" to compute as much as possible at compile-time, thereby avoiding any runtime cost for lots of code.

In fact, the idea is extended to its logical asymptote, whereby you can declare an entire function as "`constexpr`" and then expect the poor compiler to interpret the whole mess at compile-time. Pity the overworked compiler designers.

The "`restrict`" pointer declarations help the compiler with advanced optimizations like loop unrolling and vectorization by telling the compiler to ignore potential "aliasing" of pointer variables, allowing much more powerful code transformations on loops. The restricted pointer optimizations have now been formalized in C++23, but non-standard versions have long existed. The possible benefit is that restricted pointer specifications might help the compiler do auto-vectorization of loops into parallel hardware-assisted code.

How much do these help? It's rather unclear, and the compiler is free to simply ignore these hints. Compilers already did a lot of constant propagation optimizations before the "`constexpr`" directives came along, so presumably compiler designers have upped their game even further now.

Inline Functions

Placing the keyword "`inline`" before any function declarations makes that function instantly disappear in a puff of smoke. Well, sort of. It gives your C++ compiler the hint to optimize the code by putting the function's body there instead of the function call. This is faster, but means there are many copies of the function's statements, so it increases code size.

Which functions should you inline? General wisdom is to do inlining for these types of C++ functions:

- Short functions (esp. single-statement functions)
- Getters and setters in a class
- Frequently called functions at the bottom of the call hierarchy.

The `inline` specifier is just a hint. Your compiler is free to completely ignore you. In fact, this choice will probably disappear in a few years, as compilers become better than humans at choosing which functions to inline.

If you want to force the compiler to inline, use preprocessor macros. However, there's a whole minefield of problems in function-like macros. For example, you need to add parentheses around the whole expression and also around each parameter's appearance in the replacement text. Hence, `inline` functions are much safer than macros.

The value of `inline` functions is not only from avoiding function call overhead. The merging of the statements into the caller's code also allows many other optimizations to be applied there as follow-up transformations. Constants can be propagated further through the inlined statements, which is similar to `constexpr`, but the range of optimizations is much larger with `inline`.

GCC has some additional C++ language features related to inlining. There is the "`always_inline`" function attribute which says to always inline this function, and the "`flatten`" attribute which says to inline every call to other functions inside this function. There is also the "`gnu_inline`" attribute that prevents creation of a non-inlined function body.

inline function limitations

The `inline` specifier is wonderful when it works. A very important point to note about `inline` functions is that the `inline` specifier, by itself, is not enough to guarantee that inline code will be generated. The other requirement is that the compiler must know the function body code, where the function is called.

Hence, an `inline` keyword in a function prototype declaration is not enough. The executable statements inside the function's definition (i.e., the function body) must be available to the C++ compiler. Otherwise, how is the compiler to know what inline code to expand a function call into? I guess in theory the C++ compiler could maintain a huge database of all the functions in your source code, or scan through all the CPP files to find it, and that would be amazing, but we're not there yet. In practice, the compiler will only inline functions where it has seen the function body within the current C++ source file or an included header file.

This requirement imposes two restrictions on the use of `inline` functions:

1. Member functions declared as `inline` should include the function body inside the same header file as the class declaration. This can be achieved by placing the function body of a member function inside the class declaration. For a more readable style when there are many `inline` member functions, the class declaration can declare the function prototypes, and then provide the `inline` function definitions immediately after it, in the same header file. This restriction ensures that whenever the class declaration is included as a header file, the member function body is available for inlining.
2.

2. Non-member `inline` functions must be defined before they are used within a source file, preferably by placing the `inline` functions in a header file. Placing `inline` functions at the top of a source file allows the inlining of any function calls later in the same source file, but calls to the functions from a different source file cannot be inlined by the compiler unless the `inline` function definition is placed in a header file.

Non-inlined functions

Some functions declared as `inline` will not be expanded into inline code by the compiler, simply because they are too complicated for the compiler to handle. In this case, the `inline` specifier is ignored and the function is treated like any other function. The sophistication of the inline code generation depends on the compiler implementor.

Even if a compiler could theoretically inline a function, the compiler is sometimes still forced to generate a "real" function. There are various possible reasons for this:

1. The name of an `inline` function is used as a pointer-to-function constant.

2. A call to the `inline` function from within another source file.

3. `virtual` member functions.

When an `inline` function is called from a source file, where the function body has not been made available, the compiler generates a real function call (simply because it cannot inline the function). Hence, the real function must exist and be linked like any other function. Fortunately, the placement of `inline` functions in header files as discussed above will avoid this for any function the compiler decides to inline.

Inline Variables

Since C++17 you can define a *variable* as "`inline`". What does this do?

Basically, it's not really much of a speedup, but makes it easier to manage global constants, global variables, or `static` data members in C++ classes. You can declare these variables as "`inline`" in a header file, with an initializer:

```
    inline int g_x = 3;
```

Then you can with wild abandon include that header file all over the place without any problems whatsoever. The C++ linker is required to:

- Merge all of them into one variable at link-time.
- Guarantee that it's initialized as specified.
- Have the same address for that variable everywhere.

I find this addition to C++ somewhat humorous because it fixes up a huge mess that's existed since old K&R C code, and I've battled against it many times trying to get my program linked. I'm not going to irritate myself by repeating all the quirks, but it was always messy whether you had a global variable that was `extern` or non-`extern`, initialized or non-initialized, in a header file or a non-header file. So, if you ask me, the way that "`extern`" variable declarations "worked" was always broken, and now it's fixed in C++17. Hooray! (A bit late for me.)

Overall, allowing "`inline`" for variables is helpful to efficiency because you can be guaranteed about constants, `static` members, or global variables at compile-time. And it's always nice to get your program to link.

Constant Specifiers

The "`const`" keyword means that something is constant, and cannot be modified. It is helpful for efficiency, but its role is also to help detect programming errors, where code accidentally attempts to modify a constant variable or object. There are multiple places where "`const`" can be used.

- Symbolic constants
- `const` variables
- `const` objects
- `const` function parameters (i.e., "`const&`" idiom)
- `const` member functions (read-only)

But don't get me started on "`const` correctness." I've seen too many dawns fighting with compilers about `const`. Anyway, let's move on, and assume *we love* `const`.

Basic const symbols. Symbolic constants can be declared as a representation of a numeric value or other type data (instead of using `#define` symbols):

```
    const float pi = 3.14;
```

Set-once variables with const. Variables can be made constant via "`const`", which is effectively the same as a symbolic constant, except that the initializer need not be a compile-time constant. It is a "set-only-once" variable. The C++ compiler ensures that `const` variables cannot be modified, once they are initialized.

```
    const int scale_factor = get_config("scale");
    const int primes[] = { 2, 3, 5, 7, 11, 13, 17 };
```

Function parameters and const. The `const` specifier can ensure that function parameters are not modified, especially for arrays passed by reference. `const` on a scalar parameter type such as `int` is not as useful, only ensuring that the code inside the function doesn't modify the parameter (which isn't really a problem anyway). However, the idiom of "`const&`" to specify a `const` reference as a function parameter allows constant pass-by-reference of object parameters, which is extremely important for C++ efficiency.

Instantiate-only objects with `const`. Class objects can be declared as `const` variables. When the variable is a `const` object, it can be instantiated via a constructor, but cannot be modified thereafter.

```
    const Complex cfactor(3.14, 1.0);
```

Member functions declared `const`. Class member functions can be declared by adding the keyword "`const`" immediately after the function parameter list:

```
    int MyVector::count() const;
```

The C++ compiler blocks a `const` member function from modifying data members, although it can still change "`static`" data members. For `const` object variables, the C++ compiler ensures that any calls to non-`const` member functions are disallowed.

Non-member functions. Note that a non-member function cannot be `const`. The actions of a `friend` function or other non-class function are controlled by using `const` on the parameters, rather than the whole function itself.

Beyond `const`. Newer C++ features have generalized and improved some of the uses of `const`. The "`constexpr`" specifier is more powerful in terms of allowing compile-time optimizations, as are its trickier derivatives "`constinit`" and "`consteval`." The newer use of "`inline`" on a variable (yes, a variable, not a function, supported since C++17), can be helpful for safely sharing constants across multiple files.

Constant Expressions Specifier

The constexpr keyword is an optimization hint for the compiler that's more powerful than "const." Whereas const only guarantees that something won't change, constexpr is a guarantee by the human that something can be evaluated at compile-time.

The compiler should use the constexpr hint to try to propagate constant values throughout the evaluation of expressions and function calls, producing an overall speedup. However, if the compiler doesn't have the capability to do the level of compile-time optimization required, or if the human has told the machine a bald-faced lie, there's no penalty and the code just runs like it never had a constexpr specifier.

There's not a whole lot of difference between const and constexpr if you use it only for named constants:

```
const float PI = 3.14f;
constexpr float PI = 3.14f;  // Same same
```

constexpr functions

The real power is when you use constexpr for functions.

```
const float SQRTPI = sqrtf(3.14f);   // Works?
constexpr float SQRTPI = sqrtf(3.14f); // Works?
```

Oh, dear! I just tested this code snippet, and the const version works, whereas the constexpr version fails to compile, which is the opposite of what I was expecting. According to an informed source that was trained on Internet scrapings, sqrtf is not going to be declared as a "constexpr" function until C++26. Alas, by then all C++ programmers will have been replaced by robots, so feel free to skip this section.

The apparently futuristic idea is that sqrtf should have a "constexpr" keyword in its declaration, because the function return value can be computed at compile-time if you pass it a constant argument. In other words, the compiler can evaluate "sqrtf(3.14f)" at compile-time. Hence, the whole function should be declared "constexpr" in the standard library header file.

The `const` version is also probably not evaluating the `sqrtf` function at compile-time, but just calling it dynamically whenever the `const` variable is first initialized (this non-compile-time initialization is allowed for `const` variables, provided you don't later attempt to change its value).

Anyway, you can already declare your own function with the "`constexpr`" specifier.

```
constexpr int twice(int x)
{
    return x + x;
}
```

`constexpr` functions vs `inline` functions

A lot of the same value in terms of optimization can be had by making a function just `inline` rather than `constexpr`. Note that you can use both, but officially `constexpr` for functions implies `inline` on the function as well.

Is `constexpr` any better than just `inline`? If you pass a constant argument to a small `inline` function, then the expansion of the function body will trigger various constant propagation optimizations, effectively evaluating most of it at compile-time, which is almost the same as `constexpr`.

`constexpr` is supposed to be more formal in guaranteeing that the result of a function is a compile-time constant, and the compiler is honor-bound to do "compile-time function evaluation" to get the constant return value. Also, a `constexpr` function is more officially usable as a compile-time constant, so that you can use an expression with a `constexpr` function's return value in various places where C++ needs a constant (e.g., an array size declaration, some `template` situations, etc.).

An `inline` function is also supposed to be optimized at run-time for non-constant arguments, and `constexpr` functions are implicitly `inline` functions. The code generation requirements of dynamic inlining are often more advanced that constant expression evaluation.

Also, the limitations on how a `constexpr` function can be structured are a lot easier to code than the unrestricted nature of an `inline` function body. However, as a practical matter, the compile-time evaluation of expressions and the code generation for inlined expressions have a lot of overlap, so I expect C++ compilers will mostly try to do both on every type of function.

The `inline` keyword also serves a weird secondary purpose, by guaranteeing that there's only one copy of the function. This means we can include header files with the full definition of that `inline` function anywhere we like, without getting a compiler error at link-time about multiple definitions. But this isn't a performance optimization, and the linker feature of `inline` is almost the opposite of what we want in making a function `inline`, because we don't want a real function to be called at all.

`if constexpr` statements

There is an alternative usage of `constexpr` in terms of "`if`" statement conditions (since C++17):

```
    if constexpr(cond)
```

This new syntax tags the condition as being amenable to computation at compile-time. Hence, the compiler should optimize the `if` statement to a constant value, and it can then determine at compile-time which branch should be executed. So, there is a double speedup from:

(a) the condition computation is removed at run-time, and

(b) code size reduction from unexecuted "dead code" being removed.

In fact, this determines at compile-time which code block will be *parsed*, so there are cases where you can avoid a compile-time error in templates by wrapping it inside an "`if constexpr`" check. This can be useful in compile-time situations such as `template` expansion, where you can prevent some expressions from being compiled, and also code bloat can be reduced.

`constinit` variables

The `constinit` specifier is like a hybrid between:

- `consteval` and
- `static`

The `constinit` specifier declares a variable that is `static`, with lifetime scope, that is initialized at compile-time.

A variable declared as `constinit` must be initialized, and cannot be modified (like "`const`"). However, the initializer needn't be a "constant expression" although it must be able to be calculated at compile-time.

Huh? That makes no sense. Sure, it does in the world of C++ standards. A "constant expression" with only constant arithmetic is a special subset of the full set of expressions that can be calculated at compile-time.

The best example is a call to a function that has one path where it's constant, and another path where it's not. The definition of "`somefunc`" has two paths:

```
int somefunc()
{
    if (something) return 27;
    else return some_random_number();
}
```

The "`somefunc`" function cannot be declared "`const`" or "`constexpr`" because it isn't always a constant on all paths.

However, if we're using "`somefunc`" at program startup initialization, we can try:

```
constinit int s_myconst = somefunc();
```

Here, if we know that it will use the constant path for some reason, the initialization of "`s_myconst`" will go through the fixed path to get the compile-time constant value of 27, we can tell the compiler that by declaring the variable as `constinit`.

Anyway, now that you've been forced to learn all that, just forget it. You'll be rarely if ever needing `constinit`.

`consteval` functions

Use `consteval` for functions that are always constant. A `consteval` function is strictly declared so that every invocation of the function *must* return a compile-time constant.

The `consteval` keyword is a subset of the `constexpr` functions (and it also implies `inline` on a function). Although a `constexpr` function is constant if its arguments are constant, it can also return a dynamic return value for non-constant arguments.

When would you use consteval versus constexpr functions? I mean, when you ask your boss to make you a cup of coffee, do you like to ask politely or do you issue commands? Supposedly constexpr is optional for the C++ compiler, whereas consteval is mandating compile-time evaluation.

Personally, I can't see much difference in general usage, since the compiler will probably optimize a constexpr function at compile-time if it's capable enough. Hence, for most regular functions I don't see very much benefit to using the consteval specifier over constexpr. There are some complicated places in C++ where it helps to guarantee a compile-time constant, such as reflexive types and other tricks in compile-time template usage.

Templates

C++ templates can be used for compile-time optimizations, rather than merely as a programming convenience for algorithm generality and interface improvement. By specializing templated code for a particular type or constant parameter, the effect is that the resulting code is more specific, giving the compiler an opportunity for better optimizations.

For example, if we have vector and matrix classes, then rather than having our code dynamically check whether our precision is 32-bit float, or 8-bit integers, or some other low-level type, we can use templated versions of the vector and matrix classes. This generates different functions for each type of data. At the cost of some extra code space, we've given the compiler the chance to do a much better job of optimizing the code for the specific low-level data types.

Going beyond just using template code to write the same algorithm for different types, there are ways to optimize code that is templated to do more at compile-time:

- Template class and function specializations
- Constant template parameters
- Compile-time conditional tests on types (e.g., sizeof, type traits, etc.)
- if constexpr syntax
- Variadic templates
- Template Metaprogramming (TMP) techniques
- SFINAE techniques

Constants can be used to instantiate template code in a way that helps the compiler to optimize by evaluating constant expressions. Template parameters don't need to be types, but can also be constant variables or numbers, such as the size of an array.

Using a template in this way is as efficient as hard-coding the array size, which helps the compiler to know exactly what it can optimize, such as if the array size is used in any computations.

If you think you can do better than the compiler's optimizer, remember that you can also override the generic template code. For example, you can instantiate your own specific version of a template class for a particular type. Similarly, you can provide a generic function declaration that instantiates a templated function with your explicit version.

An alternative to specializing a version of a template class or function is to use compile-time tests inside the generic template code. For example, you can use conditional tests involving compile-time operations:

- `sizeof`
- `typeid`
- `std::is_same_v`
- `if constexpr` conditional test syntax

Next level templating

C++ templates are a very powerful programming mechanism. In fact, you can define entire projects as templates inside header files. To get the most speedup out of template optimizations at compile-time, consider these methods:

- Type traits
- Variadic templates
- SFINAE
- Template Meta-Programming (TMP)

Type traits are a generic feature of C++ (since C++11) that you can use to interrogate the type of a variable. They are declared in the `<type_traits>` header file and there are numerous ways that you can test the type of a variable. The above example `std::is_same_v` is one example. As another example, there is `std::is_signed` and `std::is_unsigned` to test whether it's a signed or unsigned type. There's also `std::is_pointer` and `std::is_array` and various others. Combining type traits with "`if constexpr`" gives a powerful way to ensure templated code gets evaluated at compile-time, and to specialize blocks of code for particular types.

Variadic templates are another way to level up your code and have been supported since C++11. These are variable-argument templates via the use of the ellipsis "`...`" operator in a `template` declaration. This allows templates to accept a variable number of parameters for instantiation.

SFINAE. Another optimization for advanced templating is to rely on SFINAE semantics. This refers to "Substitution Failure Is Not An Error" and means that `template` instantiation that fails should not itself trigger a compilation error that prevents execution. More specifically, if the compiler tries and fails to instantiate a template, but there's another way to run it, such as a different overloaded function available, then the code should execute via the non-templated method. Relying on this capability in C++ not only avoids having compilation errors that block some advanced template usages, but can also be used to ensure compile-time calculations. However, although there are some good uses cases in making templates faster, SFINAE is an obscure programming technique that isn't widely used in everyday C++ programming.

Template Meta-Programming. Further optimization of templated code at compile-time is possible via the technique called "Template Meta-Programming" (TMP). Note that this refers to an unusual usage of templates in C++, where the idea goes beyond just using templates of code for different types (i.e., normal templating of classes). TMP is an advanced coding method that uses (misuses, perhaps) instantiation semantics of templates as a way of generating compile-time code, even for some conditional branches. However, this is an obscure method that is rarely needed, because most of the effects can be achieved via preprocessor macros, function inlining, and using "`constexpr`" in modern C++.

References

1. Bjorn Andrist, Viktor Sehr (2020), *C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition*, Packt Publishing, Dec 2020, https://www.amazon.com/dp/1839216549,
Code: https://github.com/PacktPublishing/Cpp-High-Performance-Second-Edition (Chapter 8 is on compile-time optimizations.)
2. Gnu.org (2023), *GCC Command Options*, GNU Compiler Collection, https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html
3. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259,
Code: https://github.com/0burak/imperial_hft
4. Sarah Butcher & Alex McMurray, 2 January 2025, *The C++ techniques you need for $600k hedge fund jobs*, https://www.efinancialcareers.com/news/low-latency-c

44. Zero Runtime Cost Operations

You want free CPU cycles? You got it! There are plenty of "freebies" in C++!

We've already talked about compile-time operations in C++, but here's a summary of some of the "hints" you can give to the compiler for a free gain, usually via helping the optimizer to do fancier optimizations:

- `inline`
- `template`
- `const`
- `constexpr` (also `consteval` and `constinit`)
- `noexcept`
- `static_assert`
- Restricted pointers (e.g., `__restrict`)
- `likely/unlikely` or `__builtin_expect` (expressions)
- `[[likely]]` and `[[unlikely]]` path attributes

I've missed a bunch of them, so you should re-read those chapters. Those are well-known optimizations via programmer hints.

Here are some other ones that are useful. If you see these keywords, these are free or compile-time operations:

- `auto` types (type deduction)
- `decltype`
- `final`
- `override`
- `explicit`
- `[[nodiscard]]` (function attribute)
- `= delete`

But there's always more.

Here are some advanced C++ language features that you might think cost real CPU juice, but are free for various language design reasons:

- Type traits — compile-time type operators (not RTTI).
- Concepts (C++20) — compile-time guarantees.
- Static reflection (C++26) — fixing RTTI inefficiencies.
- Profiles — safety with compile-time validation.
- Curious Recurring Template Pattern (CRTP) — useful for devirtualization.
- Structured bindings — grouped assignments are compile-time processed.

Type traits are a form of Compile-Time Type Information (CTTI) and work at compile-time.

Some examples are operations like `std::is_trivial` or `std::is_same`. However, note that you have to be careful not to move across into the much darker side of RTTI, which is `dynamic_cast` and `typeid`.

Free Type Cast Operations

There are various arithmetic operations that can look real, but actually disappear in a puff of compiler smoke. The first item on the list is type casts, which have many freebies:

- `reinterpret_cast`
- `static_cast`
- `const_cast`
- `std::move` (move semantics)
- `std::forward` (perfect forwarding)

Note that `std::move` is effectively a compile-time type cast, which turns an l-value into an r-value (I'm simplifying the idea here).

However, there are also overloaded versions of `std::move` with two or more arguments that really do move bytes at runtime (effectively doing `memcpy`), so be aware of the distinction between free uses of `std::move` for move semantics versus real byte movers.

Arithmetic type casts between similarly represented numbers can often be optimized away.

For example, these are usually free, or at least very fast:

- Downsizing integer type casts (e.g., `int` to `char`).
- Upsizing integer type casts (e.g., `char` to `int`)
- Floating-point type conversions (e.g., `float` to `double`)

Differently sized integer types seem like they would cost real instructions to convert between them. If a `char` is one byte and an `int` is four bytes, you'd think there's an operation that adds or removes three bytes. However, the compiler has many tricks up its sleeves here, such as:

- Copy propagation
- Register allocation
- Peephole optimizations

This is often true of the conversions between any of the many and varied integer types, from a 1-byte `char` to a 16-byte `long  long`. In the cases where the compiler cannot find a way to do it freely, the operation is very inexpensive anyway.

But note that not all type casts are free. In particular, converting between integers and floating-point types is expensive, in both directions, because the way these two types of values are represented is very different. Be careful with explicit type casts, but also any expressions that mix integer and floating-point types may have implicit type casts.

Optimized Away

Here's a somewhat random list of stuff that should get optimized away by the compiler. We can be reasonably sure these are free:

- Constant expressions (via "constant folding" and `constexpr` features)
- Small getter member functions (via inlining)
- Null-effect expressions (useful for compiling-out assertions)
- Unnecessary temporary variables (removed by copy propagation, peephole optimizations, and register allocation)
- Wrongly typed constants (e.g., using `1` or `1U` or `1.0` or `1.0f` should be implicitly type-converted at compile-time).
- Double negation (using "`!!(x)`" is a common trick).
- Algebraic simplifications (e.g., plus zero, subtract zero, times one, and many more).

- Explicit zero conditional tests (e.g., `if (x != 0)` or `if (ptr != nullptr)` equates to `if(x)` or `if(ptr)` at runtime).
- First data member in an object or structure (it's offset is zero, so there's a "plus zero" in the address calculation that is optimized away).
- Assertions and `#if DEBUG` (if compiled-out for production).

The compiler optimization of "dead code elimination" will make these control flow features free:

- `while(1)` — using `for(;;)` isn't faster!
- `if(true)` or `if(1)` or `if(0)` or whatever
- `do...while(0)` — a common macro trick.
- Short-circuited constants in `||` or `&&` operators
- Tested constants in the ?: ternary operator

You can always check the assembly code with "`gcc -S`" or the MSVS assembly debug window.

Standard Container Operations

A lot of the standard containers have many optimized specializations for builtin types. Hence, if you're using `std::vector<int>`, you can expect operations like `push_back` are inlined and very fast.

All of the contiguous containers and the non-contiguous linked containers would maintain incremental variables, making `begin()` and `end()` calls very fast.

Similarly, most of the containers maintain an incrementer counter of objects inside, so all calls to `std::size` are as fast as a getter accessing an integer data member (inlined, of course).

There are some relatively simple standard C++ data types where operations can often be inlined or optimized away by the compiler:

- `std::pair`
- `std::tuple`
- `std::optional`
- `std::expected`
- `std::variant` (modern C++ unions)

Finally, note that some calls to containers can lead to memory allocations, which is a slowdown. And various containers when used on your own non-scalar objects can trigger many calls to constructors or assignment operators, which is slow regardless of whether it calls copy or move versions.

I mean, moving is better than copying an object, but the optimizer can only do so much.

The Opposite of Free

There are also features of C++ that look like they should be free, but are actually costly. Perhaps we should call them "costlies"?

Elegance and the beauty of short code sequences is not the same thing as fast. Here are some examples of beautiful things that can be slow:

- Calls to `virtual` functions
- RTTI (i.e., `dynamic_cast` and `typeid`)
- Lambdas, functors and other function objects
- `std::function`
- Comparators (except maybe standard ones like `std::less`)
- Fold expressions
- Exception handling

The issue with lambdas and function objects is not clear-cut. If you use a lambda with a simple capture and an immediate assignment to a functor variable, which is then called, the optimizer probably can handle this and inline the function call. However, if you declare your own complex lambda as a comparator that is sent to a function (e.g., to `std::sort`), all of the calls to that lambda are probably not inlined, leading to a performance bottleneck.

Also, if you use a standard builtin comparator object like `std::greater` and pass it to `std::sort` or other library functions, it's likely that the operation has a pre-coded template specialization for that comparator, meaning it won't really be using it as a function call.

However, you might want to benchmark this or look at the standard library source to confirm there is such a specialization!

And here are some more slugs that are less obvious, because the code is concise and looks like it should be fast:

- Operator overloading (looks like a single instruction, but it's a function call, even if it's inlined).
- Initializer lists (can call lots of copy constructors).
- Pointer-to-function types (cannot be inlined).
- Implicit type conversions (especially via overloaded type cast operators).
- Temporary object creation (accidental)
- Type casts between `int` and `float` (explicit or implicit)
- Container `resize()` calls

Modern C++ is becoming such a complex language with conflicting goals of elegance and performance, so it's hard to know which things are freebies or costlies.

45. String Optimizations

Efficient Strings

The C++ `std::string` class is a beautiful and elegant class that has been well-designed and near-optimally implemented.

Its main advantages include:

- High-level abstraction of string coding
- Automates management of memory buffer allocation
- Safety (e.g., no buffer overflows when appending or concatenating)
- Moderately efficient

Note that I only said efficiency was "moderate"! As classes go, it's one of the most efficient, with lots of inline member functions and implementations super-optimized by compiler engineers. Some of the fast parts of the standard string class include:

- Small String Optimization (SSO)
- Fast to copy
- Fast move semantics

But it's still not as efficient as bypassing the string interfaces and doing low-level string processing directly with `char*` pointers and arrays.

So, here we have a perfect example of the maxim: *don't optimize prematurely!* I'm not advocating to replace all strings with C-style string operations, but if your profiler finds a hot-spot in a C++ string operation, you can do better.

Furthermore, if you're doing a very string-intensive application, such as text processing, the lowest level kernels that spin through the document probably shouldn't use the string class.

Common String Operations

If you have a string, and you want to do some work on that string, the `std::string` class is often very fast. In the situations where it's not, you can also revert to old-style efficient coding on `char*` pointers by using the interface-bypassing `data()` or `c_str()` methods to get to the raw character array.

String length. The `length()` method is extremely fast, and always so. The comparison goes like this:

- `length()` — always blazingly fast.
- `strlen()` — slow on very long strings.

Since the string class maintains the string length incrementally as a data member, it's already been precalculated. Hence, it's an inlined access to an already-computed integer.

In comparison, C-style null-terminated strings must scan for the null byte. Hence, `strlen()` is slow on very long strings, whereas `length()` is still fast.

String Equality Comparisons. Which method is faster is unclear, depending on the implementation of `operator==`, but my money's on the string class. In particular, it can compare the lengths quickly, since it has that precomputed for both strings. The full list of ways to compare strings:

- `operator==()` — fast version.
- `compare()` — explicit method version.
- `strcmp()` — old-style string comparisons.

Case-Ignoring String Equality Comparisons. There's not a standard case-ignoring version of the `compare()` method. However, there are non-standard implementations:

- `stricmp()` — Windows (MSVS)
- `strcasecmp()` — Linux (GCC)

String Search. This is a very simple and long-standing requirement. Your options are pretty obvious:

- `find()` — simple and fast!
- `strstr()` — the old C function.

Case-Ignoring String Search. There's not a standard method function named "`ifind`" or "`stristr`", but there are ways to get there:

- `strcasestr()` — Linux
- `StrStrIA()` on Windows in `shlwapi.h`

Reverse String Search. There the string class method `rfind()` for reverse string searching. There's not really a good alternative in the older C-style libraries.

Character Search. Searching a string for the first occurrence of a string characters. The options include:

- `find(char)` — string class overload.
- `strchr()` — old-style C function.

Reverse Character Search. The options here are:

- `rfind(char)` — another class overload.
- `strrchr()` — reverse long-standing C function.

Note that the `rfind()` version is likely faster than the older function on very long strings, because it has the string length precalculated in the string object and can jump straight to the end, whereas `strrchr()` has to scan inefficiently from the beginning of the string.

Multi-Character Search. If you want to search for a prefix or suffix of several characters, rather than just one, then the C++ string class has what you need:

- `find_first_of()` — first character from a set.
- `find_first_not_of()` — first character not in the set.

The suffix versions are:

- `find_last_of()`
- `find_last_not_of()`

Prefix and Suffix Tests. The standard C++ methods on the string class are:

- `starts_with()` (C++20)
- `ends_with()` (C++20)

Other options include:

- `string::find()` — search forwards
- `string::rfind()` — reverse search
- `LastIndexOf` — Win32 version

There's also some other options:

- `remove_prefix()` in `string_view` (C++17)
- `remove_suffix()` in `string_view` (C++17)

You can always code your own versions:

```
inline bool STRPREFIX(const char *s, const char *prefix)
{
    return strncmp(s, prefix, strlen(prefix)) == 0;
}
```

Here's a modern C++ style version:

```
inline bool string_prefix(
  const std::string& str, const std::string& prefix)
{
    return str.find(prefix) == 0;
}
```

And here's the same idea for suffix, using the "reverse find" method:

```
inline bool string_suffix(
    const std::string& str, const std::string& suffix)
{
    return str.rfind(suffix) + suffix.length()
       == str.length(); // Buggy!
}
```

Actually, that's a bit careless of the failure return `-1` from `rfind()`.

Here's a fixed version:

```
    inline bool string_suffix(
        const std::string& str, const std::string& suffix)
    {
        int offset = str.rfind(suffix);
        if (offset == -1) return false;  // not found
        return offset + suffix.length() == str.length();
    }
```

Note that `rfind` is needlessly inefficient here if the string is very long and the suffix is not present. It keeps on scanning all the way to the start of the string, rather than quitting early. There's certainly a faster way to do it, such as comparing the two lengths, using them to compute the address of where the suffix would be, and then use basic string equality testing.

Case-Ignoring Prefix and Suffix Tests. There's not much help with this in the standard libraries, so you'll have to roll your own with `strnicmp` (Windows) or `strncasecmp` (Linux):

```
    inline bool STRIPREFIX(
      const char *s, const char *prefix)
    {
      return strncasecmp(s, prefix, strlen(prefix)) == 0;
    }
```

Here's my attempt at a fast suffix version, which mixes C++ and C coding, but won't be slow on a long string:

```
    inline bool string_strisuffix(
        const std::string& str, const std::string& suffix)
    {
        int strlen = str.length();
        int suffixlen = suffix.length();
        if (suffixlen > strlen) return false;
        int offset = strlen - suffixlen;
        const char* raw = str.c_str();
        raw += offset;
        const char* suffixraw = suffix.c_str();
        return stricmp(raw, suffixraw) == 0;
    }
```

I'm sure that you could do better!

String Class Inefficiencies

What's so bad about the standard string class? Nothing, unless you want to do a whole lot of processing of strings. Here's a list of some of its problems:

1. It's a large object (e.g., 40 bytes).

2. Sequences of binary + operators.

3. Too many calls to `new` and `delete`.

4. No way to use a larger non-allocated buffer.

5. Cannot use reference counting and copy-on-write.

A lot of these concerns can be summarized: *it's too easy to use!*

Programmers tend to get comfortable with the very convenient ways that `std::string` can be used in C++ programs. In comparison, doing C-style string processing with low-level character buffers is painful!

Hence, there's a tendency to forget that C++ strings are significant objects that invoke memory allocation on all but the smallest of text strings.

String Memory Layout

The `std::string` class creates objects of a reasonable size, unlike C-style `char*` The string class is quite complicated, although great compiler engineers have made it look easy.

Some of the main points about string efficiency are:

- Small String Optimization (SSO) is standard (with a small internal buffer).
- Reference counting is not enabled (and nor is Copy-On-Write).

The use of SSO makes sense because otherwise even just declaring an empty string object would cause a memory allocation call to the `new` operator:

```
    std::string s1;   // No memory allocation!
```

We can interrogate the string objects about their features using standard member functions such as `data()`. If the pointer to the data is inside the object itself, then we're using SSO. And if two objects created from each other (via copy constructor and/or assignment operator) have the same data buffer address, then reference counting is enabled.

Here is some code that uses standard string member calls to determine some details about the layout of a string object.

```
void print_string_details()
{
    std::string str;
    cout << "Sizeof std::string = " << sizeof(std::string)
        << " bytes" << endl;
    int bytes = str.capacity() + 1;
    int header = (sizeof(str) - bytes);
    cout << "Capacity std::string = " << str.capacity()
         << " characters ("
         << bytes << " bytes)" << endl;
    const char* datastr = str.data();
    char* saddr = reinterpret_cast<char*>(& str);
    bool is_sso = datastr >= saddr
                  && datastr < saddr + sizeof(std::string);
    cout << "Short String Optimization (SSO): "
         << (is_sso ? "yes" : "no") << endl;
    cout << "Reference counting: "
         << (string_is_reference_counted(bytes*100) ?
             "yes" : "no") << endl;
    int offset = (int)(datastr - saddr);
    if (offset == 0) {
        cout << "Buffer start of object (offset=0)" << endl;
    }
    else if (offset + bytes == sizeof(std::string)) {
        cout << "Buffer at end string (offset = "
             << offset << ")" << endl;
    }
    else {
        cout << "Buffer middle of string (offset = "
             << offset << ")" << endl;
    }
    cout << "Header block bytes = " << header << " ("
         << offset << " before buffer, "
         << (header - offset) << " after buffer)" << endl;
}
```

And here are the results in MSVS on my Windows laptop:

```
Sizeof std::string = 40 bytes
Capacity std::string = 15 characters (16 bytes)
Short String Optimization (SSO): yes
Reference counting: no
Buffer in middle of string (offset = 8)
Header block bytes = 24 (8 before buffer, 16 after buffer)
```

As to the 24 header bytes here, that could be 3 pointers (8 bytes or 64-bits each), or maybe it's 1 pointer to the buffer and 2 different 64-bit integers for length and capacity. We can go exploring in the memory layout of the header block inside a string object to try to answer that question. It's non-standard coding that is implementation-specific, but plenty of people have done it!

46. Pointer Arithmetic

What is Pointer Arithmetic?

Pointer arithmetic is a tricky C++ optimization that can often be used to remove incremented variables in loops. Instead, a pointer can be incremented each loop iteration. This changes an array access "`arr[i]`" into a pointer access "`*ptr`" and is usually faster.

What is pointer arithmetic? Arrays and pointers are buddies in C++ and there's a way that mathematical arithmetic operators can work on both. Consider the declarations:

```
    int arr[10];
    int *ptr;
```

To start with, we can set the pointer at the array, and C++ allows us to use index notation on a pointer:

```
    ptr = arr;
    x = ptr[3];
```

Here, `x` will get the value of `arr[3]` via `ptr[3]`. The pointer and array are equivalent. Note that the "`&`" address-of operator can be optionally used here. We could have written "`ptr=&arr`" to copy the address, but it's optional.

C++ allows array index accesses on pointers with "`ptr[3]`" as above. We can also do this using "pointer arithmetic" with the "+" operator and the "`*`" pointer de-reference operator:

```
    x = *(ptr + 3);  // Same as ptr[3]
```

The expression "`ptr+3`" is the address of the third element in the array (i.e., `&arr[3]`), and the "`*`" dereference operator gets the value pointed to by the pointer (i.e., `arr[3]`).

Why does this work? If ptr is pointing to the start of an integer, shouldn't "ptr+3" be a weird address in the middle of an integer?

No, because C++ does "pointer arithmetic" on pointers. Because "ptr" is an "int*" type pointer, the compiler knows to work on "int" data. With pointer arithmetic, the "+" operation adds a multiple of the bytes of the size of int types. So "ptr+1" is not the address 1 more than ptr, it's actually 4 more than ptr for a 4-byte int (assuming 32-bit integers). And "ptr+3" is actually the address "ptr+12" in terms of bytes.

Which Operators Do Pointer Arithmetic? Pointer arithmetic works with a number of arithmetic operators:

- Increment — ptr++ adds 1*size bytes to ptr.
- Decrement — ptr-- subtracts 1*size bytes from ptr.
- Addition — ptr + n adds n*size bytes.
- Subtraction — ptr-n subtracts n*size bytes.
- Assign-Add — ptr += n adds n*size bytes to ptr.
- Assign-Subtract — ptr -=n subtracts n*size bytes from ptr.

Note that there's no pointer arithmetic multiplication or division. Actually, I was told that C++37 was going to have a C++ pointer multiplication operator that scanned down an array doing paired multiplications, adding them up as it went, and all in one CPU cycle, but then someone woke me up.

Pointer Comparisons: You can also compare pointers, which isn't really doing any special pointer arithmetic, but works as normal comparisons on their addresses:

- Equality tests — ptr1 == ptr2 or ptr1 != ptr2
- Less than — ptr1 < ptr2 or ptr1 <= ptr2
- Greater than — ptr2 > ptr2 or ptr1 >= ptr2

Segmented Memory Model Pointer Comparisons: Note that there's a weird portability gotcha in relative pointer comparisons (i.e., less-than or greater-than). They're only guaranteed to work in very limited scenarios by the C++ standard, such as when the pointers are both operating over the same array data.

Programmers tend to think of the address space as one huge contiguous range of addresses, where you can compare all of the pointers in the program against each other, and make some coding assumptions based on that.

However, there are architectures where pointer addressing is more complicated, such as where pointers are a multi-part number pointing into different memory banks with a more convoluted segmented addressing scheme. For example, pointers to allocated heap memory might be separate from the pointers to global static data, and not easily comparable.

Pointer Differences: You can subtract two pointers using the normal "`-`" subtraction operator. The result is not the number of bytes between them, but the number of objects. Hence, the two pointers must be of the same type (i.e., pointing to the same type of object). Consider this code:

```
int arr[10];
int *ptr1 = &arr[1];
int *ptr2 = &arr[2];
int diff = ptr2 - ptr1;
```

The value of "`diff`" should be `1` in C++ (rather than 4 bytes), because the two pointers are one element apart (i.e., 1 integer difference). Note that "`diff`" is a signed integer here, and the value of subtracting two pointers can be negative (e.g., "`ptr1-ptr2`" above would be "`-1`" instead). Technically, the official type of the difference between two pointers is "`std::ptrdiff_t`" which is an implementation-specific integral signed type that you can use if you're also the sort of person who alphabetizes their pantry.

Adding Pointers Fails: Note that adding two pointers with "`ptr1 + ptr2`" is meaningless and usually a compilation error. Also invalid are weird things like the "+=" or "-=" operators on two pointers. Even though "-" is valid on two pointers, "`ptr1-=ptr2`" fails to compile because the result of "`ptr1-ptr2`" is a non-pointer type.

Char Star Pointers (Size 1 Byte): Note that if you want to avoid pointer arithmetic, and see the actual numeric value of addresses, you can use a "`char*`" type pointer (or "`unsigned char*`"). Since `sizeof(char)` is 1 byte, then all of the pointer arithmetic will just add the expected number of bytes (e.g., `ptr++` on a `char*` pointer adds 1 to the address). If you want to know the actual total number of bytes between two pointers, then cast them to "`char*`" type before doing the pointer subtraction.

```
int diffbytes = (char*)ptr2 - (char*)ptr1;
```

Stride of an Array. A useful piece of terminology when processing lots of data in memory is the "stride" of an array. This means the number of bytes between adjacent array elements.

We can try to compute it as follows:

```
    int arr[100];
    int stride = &arr[2] - &arr[1];  // Wrong
```

Nope, that's a fail. This isn't the stride, because it did pointer arithmetic. The addresses of array elements are really pointers, so the stride variable above is always 1 (the adjacent elements are 1 apart in pointer arithmetic). We need to convert to `char` pointers to get the stride in bytes.

```
    int arr[100];
    int stride = (char*)&arr[2] - (char*)&arr[1];
```

Can't we just use `sizeof` to get the stride? Isn't the stride above going to equal `4`, which is `sizeof(int)`? Yes, in the example above the use of `sizeof` is correct, but no, that is not true in general. The stride will often equal the element size, but may be larger. For a simply packed array of integers or other simple types, the stride is almost certainly the size of the array element type. But this is not always true, such as if it's an array of a larger object with an awkward size that requires padding bytes for address alignment considerations.

Loop Unrolling Stride. The term "stride" also has a secondary meaning when talking about array processing with loop unrolling. The stride of an unrolled loop is how long of a segment is being processed in each section of loop unrolling code. For example, if a loop is unrolled with AVX-2's 256-bit registers (equals 8 32-bit `floats`), then the stride when discussed in the literature is either 8 `floats` or 8x4=32 bytes.

Void Pointer Arithmetic Fails: Note also that pointer arithmetic on a generic "`void*`" pointer should be a compile error, because it points to unknown size objects. Some C++ compilers will allow pointer arithmetic on void pointers with a warning, and pretend it's a "`char*`" pointer instead.

Finally, I don't think you can increment a "function pointer" in valid pointer arithmetic, but you're welcome to try.

Pointers and Arrays

There is a close relationship in C++ between arrays and pointers. Array names are, in many ways, just pointers to the first element in the array. The array indexing operation is identical to a pointer expression involving address arithmetic.

The following algebraic identities hold:

```
    array[exp] == *(array + exp)
    &array[exp] == array + exp
```

These relationships have a number of consequences. First, the commutativity of + means that `exp1[exp2]` is equivalent to `exp2[exp1]`, which leads to weird syntax tricks like "`n[ptr]`" instead of "`ptr[n]`".

Another consequence is that, in many situations, pointer variables can be used instead of arrays. For example, it is legal to apply the array indexing operator (i.e., square brackets) to a pointer. For example:

```
    x = ptr[3];
```

Just like `arr[3]`, this sets `x` to equal the third element away from `ptr`, where `ptr` is pointing into an array.

Array Function Parameters: The array and function relationship is complicated when an array is a function parameter. When an array is passed to a function, the address of the first element of the array is passed. An array formal parameter is implemented as a pointer variable (i.e., a pointer pointing to the start of the array).

This explains why arrays are passed by reference, not by value. A local copy of the array is not used inside the function. Instead, a pointer to the original array is used. Hence, any change to an element of the local array variable is actually changing the original array (i.e., pass-by-reference instead of pass-by-value).

The differences between pointers and arrays are few. The main one is that an array name is not a variable, whereas a pointer is. Hence, an ordinary array name declared as a local variable cannot be assigned to, or incremented, whereas a local pointer variable can be. An array is similar to a constant pointer (e.g., `int *const ptr`). Note that this is untrue when the array is a function parameter, when it can be incremented or modified.

There are also the differences between pointers and arrays in relation to initializations. Consider the two initializations:

```
char *p = "hello";
char arr[100] = "hello";
```

For the pointer `p`, the string "`hello`" is stored in separate memory. Only the required number of bytes are allocated (six, because of the extra character zero added by the compiler to terminate the string). For the character array "`arr`", 100 bytes are allocated, but only the first six are filled.

Pointer Arithmetic Loop Optimizations

The main way that we use pointer arithmetic for optimization is to change a loop over an array into loop pointer arithmetic. Note that this is primarily a sequential code optimization, and does not change anything in terms of vectorization for parallel execution.

Pointer arithmetic is mainly used to get rid of an incrementer variable in sequential code. Here's a vector dot product with basic incremented loop variable `i++` and array index syntax `v1[i]` used inside the loop:

```
float aussie_vecdot_basic(float v1[],float v2[], int n)
{
    // Basic vector dot product
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

And here's the same code when converted to pointer arithmetic:

```
float aussie_vecdot_ptr(float v1[], float v2[], int n)
{
    // Pointer arithmetic vector dot product
    float sum = 0.0f;
    float* endv1 = v1 + n;  // v1 plus n*4 bytes
    for (; v1 < endv1; v1++,v2++) {
            sum += (*v1) * (*v2);
    }
    return sum;
}
```

How does this work? We got rid of the temporary variable "i" by using pointer arithmetic "*v1" instead of array indices "v1[i]". We are also using the function parameters "v1" and "v2" as temporary local variables, as permitted in C++, so we don't need an extra temporary pointer variable.

The way this works with pointer arithmetic is v1 and v2 are treated as pointers, which works due to the near-equivalence of pointers and arrays in C++. Rather than using an array index "i" we increment both these pointer-array variables:

```
    v1++,v2++
```

These for loop incrementers "v1++" and "v2++" are both adding 4 bytes (the size of a 32-bit float) to the pointers. Also note these two increment statements are separated by the C++ comma operator, not by a semicolon.

The "endv1" end marker is calculated as the address of "v1[0]" plus "n*4" bytes, because the "+" operator in "v1+n" is pointer arithmetic addition, which is auto-scaled by the size of the pointed-to object (i.e., 4 bytes for 32-bit float here), rather than normal integer addition.

Note that a further micro-optimization is possible. We can change the less-than test ("v1 < endv1") to an inequality test ("v1 != endv1"), because equality tests are slightly faster than less-than tests. Since this test is effectively inside the loop and done every iteration, this might be worth doing.

The trade-off is safety: it'll become an infinite loop if you get the pointer math slightly wrong, but hey, your code has no bugs, right?

Smart Pointers

Smart pointers are a programming idiom to make C++ pointers safer. They are not a speed optimization, and in fact, they are a wrapper that adds extra logic around the use of a raw pointer, and will be marginally slower. However, they avoid many C++ pointer pitfalls, thereby improving reliability, and will reduce total allocated memory usage by avoiding memory leaks. There may even be an indirect benefit to execution speed if overall memory management is improved.

Programmers have been defining their own smart pointer wrapper classes for decades, but there is now standard support for the idea in the C++ library. In the typical idiom, a smart pointer tracks the creation and destruction of the object it points to, which ensures that the destructor is called.

This helps avoid "memory leaks" in standard C++ pointers where an object is allocated with "`new`", but is never deallocated by "`delete`".

The C++ standard libraries have various templates to support smart pointers, mostly since C++11, so they are longstanding features.

- `std::shared_ptr`
- `std::unique_ptr`
- `std::weak_ptr`

`std::shared_ptr` is a reference-counted shared pointer implementation. The idea is that it tracks the total number of pointers to an object, and then automatically destroys the object whenever there's no more pointers to it. This occurs when the last of the "`shared_ptr`" objects is itself destroyed, and then the reference count for the underlying object is zero.

`std::unique_ptr` is a one-to-one mapping of a smart pointer to an object. Whenever the `unique_ptr` object is destroyed (e.g., goes out of scope as a local variable), then both the smart pointer and its underlying object are destroyed or otherwise cleaned up. The `unique_ptr` object can refer to a single object allocated by "`new`" or a single array-of-objects allocated by the "`new[]`" operator.

`std::weak_ptr` is a less commonly used type that has relevance to `std::shared_ptr` in some complicated scenarios. Usually, you should choose either of `std::unique_ptr` or `std::shared_ptr`, depending on how many pointers will point to the underlying object.

Pointers vs References

Overall, pointers are a good and bad feature of C++. They are low-level variables that allow efficient processing of memory addresses, so we can code some very fast methods with pointers. They allow us to get very close to the machine.

On the downside, there are pointer pitfalls. Pointers trip up novices and experienced programmers alike. There is an immense list of common faults with pointer manipulation, and coding problems with pointers and memory management are probably half of the causes of bugs in C++ (at least). There are some tools that mitigate against pointer problems (e.g., Linux Valgrind) but it is a never-ending battle against them.

Pointers and arrays were implemented very similarly, and came from the earliest designs of the original C language.

Basically, arrays are treated as a specific type of pointer, with various differences depending on whether they are variables or function parameters.

Then came C++ to the rescue. References arrived with the new-fangled programming language (cleverly named as "C++") and were thoughtfully designed as a type of safe pointer that cannot be null, but is just as efficient as a pointer because the constraints on references are enforced at compile-time.

C++ allows two ways to indirectly refer to an object without needing to create a whole new copy: pointers and references. The syntax is either "`*`" or "`&`" for their declarations.

```
    MyVector *myptr = &mv;  // Pointer to mv object
    MyVector &myref = mv;   // Reference to mv object
```

Pointers and references are more efficient than spinning up a new copy of the object, especially when the underlying object is a complicated object. And when you have a function call, you should definitely avoid sending in a whole object.

```
    void processit(MyVector v)  // Slow
    {
        // ....
    }
```

This is inefficient because the whole MyVector object will get copied, via whatever copy constructor you have defined, which is slow. And if you haven't defined a copy constructor, then the compiler uses default bitwise copy of a structure, which is not only slow, but also rarely what you want, and often a bug.

The faster reference version is to use a "`const`" reference (or non-`const` if you're modifying it inside the function):

```
    void processit(const MyVector & v) // Reference argument
    {
        // ....
    }
```

The pointer version is:

```
    void processit(MyVector * v)  // Pointer argument
    {
        // ....
    }
```

Which is faster in C++ — pointers or references? The short answer of "not any difference" is the general view, because references are implemented as pointers by the compiler behind the scenes. The two functions above are not going to be significantly different in terms of speed.

The slightly longer answer is that references can be faster because there's no null case. A reference must always be referring to an object for the duration of its scope. The C++ compiler ensures that references cannot occur without an object:

```
    MyVector &v;          // Cannot do this
    MyVector &v = NULL;   // Nor this
    MyVector &v = 0;      // Nor this
```

A reference must be initialized from an object, and you cannot set references equal to pointers, because you actually have to de-reference the pointer with the "*" operator, which crashes if it's a null pointer:

```
    MyVector &v = myptr;  // Disallowed
    MyVector &v = *myptr; // Works if non-null
```

There's no way in C++ to get a zero value into a reference variable (we hope). For example, the address-of operator (`&`) applied to a reference variable returns the address of the referenced object, not the memory location of the reference itself. Hence, references are always referring to something and they cannot be equivalent to the null pointer.

References are slightly faster: The guarantee of an object for a reference fixes all those null pointer core dumps, and also relieves the programmer of the burden of testing for null pointers. The compiler does this guarantee for references at compile-time, so there's no hidden null check being done by the compiler at run-time, making it efficient. So, there's a minor speed improvement from using references, by not having to add safety checks for "`ptr!=NULL`" throughout the function call hierarchy.

Pointers can be better than references if you need a "null" situation to occur. For example, you're processing an object that may or may not exist, and you need the pointer to be allowed to be "NULL" if there's no object. This should occur rarely, and references should be preferred in many cases.

And finally, references aren't very useful when you're trying to scan through the data in vectors, matrices, or tensors in an AI engine. You can't do pointer arithmetic on a reference in C++.

47. Algorithm Speedups

Algorithm Optimization Techniques

This chapter presents some of the theory of the general techniques for optimizing algorithms. Changing the underlying algorithms used by the program is often the only real way to gain a large speed increase.

In particular, the algorithms and data structures used can often be modified to give a significant speed increase. Is there a better way to do what your program does? Is it doing too much unnecessary calculation? Although much depends on the programmer's ingenuity, there are some common techniques for improving performance of algorithms.

- Parallelization and vectorization
- Precomputation (save time by using space)
- Recomputation (save space by using time)
- Caching and computation reuse
- Greedy algorithms (immediate computation)
- Skipping algorithms
- Arithmetic strength reduction
- Integer arithmetic
- Change recursion to loops
- Incremental algorithms
- Choose a better data structure

The idea of "skipping" computations also has various sub-methods:

- Lazy algorithms (delay computation until needed)
- Common case first
- Simple case first
- Approximate tests first

Lookup Table Precomputation

Lookup tables are so widely used in AI engines that they're usually abbreviated as LUTs. The aim is to precompute results and replace frequently called costly function evaluations with table lookup (i.e., array references). Note that this use of precalculation is only worthwhile if some calculations are repeated and computing the same result.

As an example, we can replace a call to "`sqrtf`" with a precalculated table of square roots. In the subsequent calculations where square root is needed, a call to the `sqrtf` function is replaced by a table lookup.

The precalculation uses two separate functions: one to perform the precalculation, and another to access the values by table lookup. The precalculate function must be called once via a global initialization routine for the class. Alternatively, every call to the square_root function could self-check a `static` Boolean flag indicating whether the values have been precalculated yet, and call the precalculate function if not, but this is needlessly slower for every access.

Even more efficient is to use "offline precomputation" before your program even runs. This is a more efficient method whereby the data is not precalculated during initialization of the program, but is done earlier in an "offline" mode (e.g., as part of your build process). For example, the precomputed results are either stored to a data file, or converted to a C++ source file that is linked.

Another good example of precalculation is the Boolean functions on characters (e.g., `isupper`). To improve performance, it is possible to implemented these functions as a precomputed array of 256 `bool` values, or 256 bytes with 0 if `isupper` is false, and 1 if `isupper` is true. Then `isupper` is evaluated by indexing the character into the precomputed table:

```
    #define isupper(ch) ( precomputed_array[ch] )
```

In fact, many C++ compilers implement the `isupper` test and other functions in `<ctype.h>` as a table lookup over the 256 character values (plus an extra one for `EOF`), with a precalculated single bit flag per function — that is, one bit indicating `isupper`, another bit for `islower`, etc.

Lazy Evaluation

The idea of lazy evaluation is a slight amendment to precalculation or data structure augmentation. Full precomputation during program startup can be inefficient when only some of the values are needed.

Lazy evaluation works in a "lazy" manner, by only doing work when asked. Instead of precalculating every result, results are calculated only as needed. To use this method, some way is needed of indicating whether a result is already in the table. When seeking a result, it is necessary to check if the required value is already present. If so, table lookup is used to get the result. If not, the value must be calculated, stored in the table and that entry marked as present.

The precomputation of `sqrtf` can be modified to become lazy evaluation by adding another array of Boolean flags, indicating which of the square roots have been computed. When calculating a square root, the function checks if it has been computed, and calculates it if not.

```
float square_root_lazy_eval(int n)
{
    static float sqrt_table[NUM_PREC + 1]; // values
    static bool precalc[NUM_PREC + 1];     // flags

    if (!precalc[n]) { // precalculated?
        sqrt_table[n] = sqrtf((float)n); // real sqrt
        precalc[n] = true; // Mark as computed
    }
    return sqrt_table[n];
}
```

The use of lazy evaluation is slower than complete precalculation if all of the values are eventually calculated, because of the overhead of checking whether calculation is needed. Also, there's only an efficiency gain for values that are calculated twice or more. However, lazy evaluation can make the program faster overall if not all calculations are needed, but some are needed many times. Any unnecessary calculations are avoided. How lazy!

Source Code Precomputation

The examples of the precomputation of square roots in the previous two sections are not particularly efficient because they must still call the `sqrtf` function a number of times. A far more efficient alternative is to use C++'s compile-time initialization of arrays to set up the precomputed `sqrt_table` array inside the C++ source code. Hence, the square_root function becomes a simple lookup into an array variable as follows. Note that the array is declared as "`static`" so that the initialization occurs at compile-time.

```
float square_root_precalc(int n)
{
    const int NUM_PRECALC = 100; // Precalculate to 100
    static float sqrt_table[] = {
      0.000000f, 1.000000f, 1.414214f, 1.732051f,
      2.000000f, 2.236068f, 2.449490f, 2.645751f,
      2.828427f, 3.000000f, 3.162278f, 3.316625f,
      //... etc .....
    };
    if (n >= NUM_PRECALC) return sqrtf((float)n);
    return sqrt_table[n];
}
```

The simplest way to produce the values for the precomputed array is to write another program to produce them. Once the values are produced, this program could be discarded, or it could be left in the build process. The following program was used to produce the declaration of `sqrt_table` used in the square_root function given above. The output from the following program was copy-pasted into the source code for the program above.

```
void generate_sqrt_table()
{
    const int NUM = 100; // Precalculate to 100
    printf("static float sqrt_table[] = {\n");
    for (int i = 0; i < NUM; i++) {
        printf("%ff", sqrtf((float)i));
        if (i + 1 < NUM)
            printf(", "); // comma after all but last
        if (i % 4 == 3 && i + 1 < NUM)
            printf("\n"); // newline every 4 numbers
    }
    printf("\n};\n"); // finish off declaration
}
```

Source code precomputation should always be more efficient than lazy evaluation and run-time precomputation. However, source code precomputation is only applicable when the function can be computed at compile-time (e.g., any "`constexpr`" function). If the computation involves any variables whose values are known only at run-time, either lazy evaluation or run-time precomputation may be needed.

Incremental Algorithms

It is often easier to modify what has already been done than to start from scratch. This idea can be used to write faster algorithms. However, changing an existing algorithm to use incremental calculations will usually require a total redesign of the algorithm.

A simple example of an incremental algorithm is counting the number of symbols in a hash table. The non-incremental way to count them is to traverse the hash table, counting the number of entries along each hashed chain. The incremental method is to keeping a running count — increment it when a symbol is inserted; decrement it when a symbol is deleted. The incremental method is better if the count will be required many times. If the count is not required, there has been a small extra amount of unnecessary overhead.

Another good example appears in graphics animation when managing the buffers. When displaying a new screen, it is usually more efficient to change the existing screen buffer than to redraw the whole screen. The idea is to set only those pixels that need to be changed.

For another example, a chess-playing program uses a game tree and the minimax algorithm with a static evaluation function. This function usually analyses the material balance (i.e., how many pieces each side has), along with other chess strategy factors. A simple but inefficient method of computing the material value of a position is to add the values of each piece on the 64 squares.

The efficient incremental algorithm is to subtract the value of the piece from a running count whenever any piece is captured by the opponent.

Common Case First

When testing for a number of different conditions, it is best to test the most common case first. If it is true, the other tests are not executed. When using multiple `if-else-if` statements, place the common case first. For example, consider the binary search function:

```
    if (key > a[i]) {
        // ...
    }
    else if (key < a[i]) {
        // ...
    }
    else { // equality
        // ...
    }
```

Equality is least likely of all the three conditions, and hence it goes last. Greater-than and less-than are more common, so they go first.

The idea of common case first also appears in Boolean expressions using `&&` or `||`. The short-circuiting of these operators makes them very efficient when the common case is first. For `||`, the most likely condition should be placed first (i.e., most likely to be true). For `&&`, the most unlikely condition should be placed first (i.e., most likely to be false).

Simple Case First

This method is similar to common case first — the idea is to test the simplest condition first. More complicated and time-consuming computations can be avoided if the first test succeeds (or fails, depending on the context). This idea appears in two main situations:

- `if-if` construct (nested `if` statements), and
- logical operators (`&&` and `||`).

The simplest test should be the first of a pair of nested `if` statements and should also be the first operand of a `&&` or `||` operator. In the examples below, the sub-expression "`x!=0`" is evaluated first because it is the simplest and hence the least expensive to evaluate.

This is the nested-`if` example:

```
if (x != 0) {
    if (expensive_fn(x) != 0) {
        // ...
    }
}
```

This is the `&&` short-circuiting method:

```
if (x != 0 && expensive_fn(x) != 0) {
    // ...
}
```

Special Solution of Simple cases

In addition to putting a simple case first, it can also be efficient to solve simple cases differently to the general case. When solving a problem, simple cases can often be solved by specially designed fast functions. These "special solutions" can involve table lookup of precalculated values (e.g., storing the first ten factorials in an array) or just a fast algorithm for small cases (e.g., sorting less than five numbers quickly).

In general, the special solution of simple cases will give some speed increase if the simple cases are fairly common. The advantage of simple case precalculation over full precalculation is flexibility — it is not limited to those values that can be stored in a fixed size table.

The use of table lookup for simple cases for the factorial function is shown below. The use of the method here gives speed increase for all cases, not just the simple ones, because the recursive definition of factorial eventually breaks the problem down to a simple case.

```
int factorial_precalc(int n)
{
    const int NUM_PRECALC = 5; // How many
    static int s_precalc[NUM_PRECALC + 1] =
        { 1, 1, 2, 6, 24, 120 };

    if (n <= NUM_PRECALC)
        return s_precalc[n];
    else
        return n * factorial_precalc(n - 1);
}
```

Approximate Tests

Many algorithms can be improved by avoiding complex calculations with a fast preliminary test that is often successful. This is a special type of common and simple case optimization combined. This method is only worthwhile when avoiding the complicated test is highly probable; if avoiding it is unlikely, the extra simple test reduces efficiency because it adds (slightly) to the run-time cost.

Zero skipping. In an AI engine, a common example is "zero skipping." A low-cost test of a weight against zero can avoid the complexity of computing vector and matrix operations with that weight.

Bounding Sphere Tests in Ray Tracing. As an example in 3D graphics, to implement a ray tracing algorithm for graphical image rendering, it is necessary to determine whether a ray strikes an object. Since the objects are often complex and more often than not the ray will miss an object by a large amount of space, a simple test can be used to quickly identify rays that are close enough to the object to intersect with it. A good simple test is to determine if the ray intersects with the bounding sphere of an object, as it is relatively efficient to determine this. If the ray does intersect the sphere, the more expensive tests are applied to determine if the ray intersects with the object. If the ray does not intersect with the sphere, the cost of the more expensive tests has been avoided. Interestingly, the simplicity of testing the intersection of a ray with a sphere helps explain why there are so many ray-traced images of spherical objects.

Bounding-box 2D collision detection. The similar idea of a bounding rectangle is useful for collision detection in coding 2D arcade games. Collision detection usually involves testing many pairs of objects in a two-dimensional setting, and the tests are complicated because of the different shapes of the objects. The more complicated tests can be avoided by examining whether the bounding rectangles of each object are intersecting. If they do intersect, then a closer examination of whether the objects have pixels that overlap is carried out.

Rectangle Shapes. For yet another example of using a simple test to avoid complicated tests, consider the problem of a GUI-based drawing program. Typically, the user can select a vertex (e.g., the end of a line segment) by clicking "close" to the vertex. In other words, the user must click the mouse within a specified radius of the point. Hence, when the mouse is clicked, the program must compare the mouse location with all the currently active vertices. The obvious method is to use the distance formula for two points and apply the following test on the x and y coordinates of the mouse and all points.

Here's the code:

```
const float DISTANCE = 2.0f;
float diffx = xMouse - xPoint;
float diffy = yMouse - yPoint;
float distance = sqrtf( diffx * diffx + diffy * diffy);
if (distance <= DISTANCE) {
     // clicked! ...
}
```

Firstly, the efficiency of this test can be improved simply by avoiding the calculation of the square root. Squaring both sides of the equation gives the equivalent test:

```
float distance_squared = diffx * diffx + diffy * diffy;
if (distance_squared <= DISTANCE * DISTANCE) {
     // clicked! ...
}
```

However, the multiplications involved in computing the squares of the two sub-expressions on the left are quite expensive, although the square on the right-hand side will be a compile-time constant. A simple test can be used to avoid the expensive multiplications in most cases. If the difference between either the x or the y coordinates is greater than `DISTANCE`, then the points cannot be close enough. Although the cost of these tests is quite high because the absolute value for the difference must be found, it should still cost less than two multiplications, and will be more efficient if there are many widely spaced points to be tested. The code using this idea is:

```
bool check_point_clicked(int xm, int ym, int xp, int yp)
{
    const float DISTANCE = 2.0f;
    int xd = xp >= xm ? xp - xm : xm - xp;
    if (xd > DISTANCE)
        return false;
    int yd = yp >= ym ? yp - ym : ym - yp;
    if (yd > DISTANCE)
        return false;
    return xd * xd + yd * yd <= DISTANCE * DISTANCE;
}
```

Of course, algorithm improvements are even more effective. The best way for improving the efficiency of this program is to avoid the need for multiplications entirely, by changing the program specifications (!) so that the definition of clicking "close enough" to a vertex with a mouse refers to clicking within a *square* around the point, instead of a circle. Squares don't need multiplication.

Augmenting Data Structures

An interesting type of caching is where the data is stored inside the main data structure, rather than in a separate cache. Instead of recalculating derivative data every time you need it, a faster way is to store the data in the data structure. This is a form of caching that saves the time of recalculation, which need be done only once. If the data ever changes, the calculations must be redone and stored again. Hence, this method works best where data is unchanging, but can also tolerate modifications.

As an example of augmentation, consider a struct defined to represent a line segment (e.g., in a CAD drawing program). The struct contains four fields, for the x and y coordinates of the start and end points:

```
    struct line_segment {
        int x1, y1; // Start point
        int x2, y2; // End point
    };
```

Consider the computation of the length of the line segment, using:

```
    float flen = sqrtf((y2 - y1) * (y2 - y1)
                     + (x2 - x1) * (x2 - x1));
```

If the length is a common calculation, it can be beneficial to cache the length of the line segment as an extra field in the `struct`:

```
    struct line_segment {
        int x1, y1; // Start point
        int x2, y2; // End point
        float length; // Length of line segment
    };
```

Whenever this length is needed during calculation it is immediately available as a field member. However, it is important to be careful that there is no consistency problem (where the `length` field is not the true length of the line segment). The main danger is that the `length` field won't be recalculated every time one of the other fields change.

48. Memory Reduction Optimizations

Memory Reduction in C++

There are many general techniques for reducing the memory requirements of a C++ program. These techniques herein aim to reduce memory usage of a program so that:

(a) your C++ does not waste too much time on memory management activity, such as allocating too much memory, and

(b) your C++ code can execute on a low-memory platform, such as an IoT embedded device.

In these days of cheap gigabytes of memory in every PC, memory reduction techniques are perhaps not as important as those for increasing speed. However, there are certainly situations when reducing space requirements is far more important than increasing the speed of a program. This section discusses a number of general techniques for reducing C++ memory requirements.

Unfortunately, reducing space requirements can also lead to loss of speed. There is often a trade-off between space efficiency and time efficiency. Every C++ program uses memory for a number of different purposes, and each of these areas needs to be attacked separately. The memory usage of the program can be divided into the following memory sections:

- Executable instructions
- Static storage
- Stack storage
- Heap storage

The executable instructions for a program are usually stored in one contiguous block of memory. Static storage refers to memory used by global and local `static` variables, string constants and (possibly) floating-point constants.

Stack storage refers to the dynamic storage of non-`static` local variables. Heap storage refers to the memory space that is dynamically allocated using the `new` and `delete` operators and the `malloc/calloc/free` standard library functions.

The memory requirements for the executable instructions are largely independent of the other memory areas, whereas the techniques for reducing the memory required for the other three areas are often similar. However, care must be taken that applying a technique to reduce data space does not increase the amount of C++ code too greatly, thus increasing the executable size.

Compact Data Representation

Different algorithms may store data differently and thereby reduce memory requirements. There are many ways to represent data, and all have varying space usage. For example, storing all the primes less than 1000 can be done with a list of integers, a list of the incremental differences between successive primes, or a bit vector with one bit for each integer up to 1000.

Different data structures. The program should be examined to determine if a large space reduction can be achieved by changing to different data structures. For example, the program could use arrays instead of linked lists or binary trees to avoid the extra space due to pointer storage. However, this also wastes more space if the array is not full, and it is even better to use dynamic arrays, which do not waste any storage, as exactly the right amount of memory is allocated. Unfortunately, using different data structures can sometimes reduce the time-efficiency of programs.

Data compression. Compressing data can reduce space requirements when large amounts of data are involved. Hmm, let's pause for a moment and try to think of an example application with lots of data. Just jump in whenever you're ready.

Billions or trillions of weights in an LLM are a good candidate. Model compression is the theoretical term and involves either using smaller data sizes (e.g., 8-bit integer weights instead of 32-bit `float` data) or "pruning" of weights we don't need. More generally, data compression algorithms have been used in research on AI models, such as sparsity, run-length encoding and Huffman encoding.

Proceduralization. Another data representation technique is to use a function to represent data. Instead of a list of the first 1,000 primes, you could create an "`is_prime`" function that contains a big C++ `switch` statement, with all the primes as `case` values, which return true. You could also write a piece of code to create this source code automatically.

Recomputation. Another example of proceduralization, consider storage for several images generated by a fractal algorithm: the simplest method of storing the images is to store them as large image files. But a much more space-efficient method is simply to store the values of any arguments passed to the function creating the fractal images. This way, the images can be recreated by calling the fractal generation function with the correct arguments. The only space used is a few extra values containing the arguments and the code instructions for the function. However, the recalculation of an image by this method is extremely time-inefficient.

Reducing Data Size

There are many techniques for reducing the size of program data. These techniques apply to all three types of memory — static, stack and heap storage. In some cases, a method may increase the memory storage in one area to decrease the memory usage in another, which is valid only if the total storage requirements decrease.

Use `char` arrays not `std::string`. The use of `std::string` is very convenient, but if your program has many strings, the extra storage used by the `string` objects can add up. Consider managing your own raw `char` arrays as C-style strings if you really need the space.

Avoid max-size arrays or buffers. When using an array data structure or buffer, there is temptation to be lazy and just make it bigger than it will need to be. Avoid this temptation and optimize the memory usage properly. Change an oversize array into a dynamically allocated array, if size can be determined easily at runtime.

Smart buffers or smart array classes. An alternative to using an oversize array or buffer is to create "smart" classes that manage this, by automatically extending the array or buffer if more elements are needed. The `std::vector` class is a good way to do this.

Bit vectors. These can be used where information can be reduced to a single Boolean value, such as bit flags or masks. The use of bit vectors is very compact in terms of space, and there are standard C++ libraries to implement these efficiently.

Unions. When using a lot of structures, space can be reduced by overlaying the data fields. This can only be done if the fields to be overlayed are mutually exclusive (i.e., they never have active data in them at the same time). There is a special C++ data type for this purpose: the `union`.

Linearize multi-dimensional dynamic arrays. Use the simpler and smaller size of a one-dimensional array, with the two-dimensional structure mapped onto it with index calculations. This adds more runtime cost, but saves space over multiple levels of dynamic array allocations.

Reusing space. One way to conserve memory is to reuse the space used by a variable. The `union` data type is an example of this general idea, and another is reusing variables for different purposes. For example, rather than letting several functions each have a local temporary buffer, they could all use the same global variable (although this is a very dangerous practice). As another example, if a program uses two similar arrays, examine whether the two arrays can share the same storage (possibly as a `union`). Note that I don't recommend any of these approaches: too dangerous!

Small data types: `short`, `char`. Instead of using arrays of `int`, use arrays of `short`, `char` or `unsigned char`. There is no problem with this method, provided large integer values are not being stored (e.g., larger than 127 for `char`, or larger than 255 for `unsigned char`). This technique is also worthwhile when applied to `int` fields in objects although alignment restrictions may limit the improvement — use the `sizeof` operator to determine if the size of the object has been reduced. Smaller local variables could also be declared as a smaller type, but this may increase the executable size due to type conversions. Note that speed can be compromised by using smaller data types because of the type conversions that often result. Similarly, use `float` instead of `double`, where the greater precision of results is not important (e.g., an AI model).

Bit-fields in objects. When storing small integers in objects or structures, there is a way to specify exactly the number of bits required. These types are called "bit-fields" and can only be used for fields inside objects, structures or unions. You cannot declare a local variable with a bit-field type. When using bit-fields, small integers or Boolean flags are automatically packed into a `struct` or `union`. This reduces storage requirements significantly, but reduces speed because it is necessary to pack and unpack bits.

Parallel arrays versus arrays of objects or structures. Because of alignment restrictions, an object or structure may have unusable extra padding bytes. The number of padding bytes can be determined by using the `sizeof` operator, and subtracting the sizes of each individual field from the size of the object. If there are padding bytes, replacing an array of `struct` with a number of "parallel" arrays removes the need for this padding.

Packing. When dealing with large arrays of small integers, it can be more efficient to pack them together (i.e., more than one value per word), particularly when the information is binary (true or false), because only one bit per value is needed. The easiest way in C++ is to use `std::bitset`. Note that bit-fields are a method for packing provided by the compiler that can support more than one bit. They are also much easier to use than coding it yourself.

Packing object arrays with `#pragma pack`. Microsoft C++ compilers support the "`#pragma pack`" preprocessor directive, which can specify the packing and alignment characteristics of an object. This can allow arrays of these objects to be packed more closely into storage.

Reordering fields in objects and structures. Because of the word alignment on some machines, the order of fields in an object or structure can change the total size of the object. This only applies to objects containing different size fields. A general rule for minimizing the space is to order the fields from largest to smallest. This heuristic may not give the best ordering — examine the size of a few different orderings using the `sizeof` operator, if space is crucial. This is a machine-dependent optimization, and may not work well on some machines.

Store integer codes instead of string names. If you're storing a string to represent some particular type or a limited set of names, or something with a finite set, then you can use an enum instead. If you need to generate the actual string name, use an array lookup or a `switch` statement to return the equivalent string constant. For example, when dealing with AI word tokens, which are indeed fixed and finite, use the integer token code without storing the word as a string, while maintaining a single copy of the vocabulary strings (which you need anyway for the tokenizing algorithm).

Measuring Code Size and Static Storage

In general, it is more difficult to measure how much space a program is using than to measure how much time it is using. However, most environments provide some means of determining the size of instructions and static data in an executable program. If nothing else, the size of the executable file can be a reasonable guide.

The `size` command. Under Linux and UNIX, a useful command is the "`size`" command, which examines an executable program and reports the memory used by its instructions and its global or local `static` variables. However, it does not (and cannot) report the stack or heap usage because the amount of such memory used is dynamic, and hence cannot be found by analyzing the executable.

The command is simply:

```
size a.out
```

This produces output similar to the following:

```
text data bss dec hex
20480 8192 0 28672 7000
```

The "text" value refers to the machine code instructions for the program code. Both the "data" and "bss" areas refer to global and local `static` variables. The "data" area refers to variables which have been explicitly initialized with values (e.g., string literals or initialized global variables); the "bss" area refers to variables with implicit initialization which defaults to zero (e.g., global variables or arrays without non-zero initializers).

Function Code Sizes: If the code size is needed on a per-function basis, Linux and most other UNIX environments support the "`nm`" command. Windows also supports the `nm` command.

```
nm a.out
```

The `nm` command differs slightly across older UNIX variants, but will usually print out information including the start and end address of a function, from which the size of a function can be trivially computed.

Link Maps: Window users may be able to use a "link map" report. This allows to find out about executable size by examining the output produced by some C++ compilers at the link stage (although not all compilers will produce useful output).

For example, the DOS "`link`" command with the "`/map`" option can be used when linking the object files:

```
link /map *.obj
```

Code Bloat

The size of the executable depends on the size of your C++ source code. Hence, the obvious way to reduce executable size is to go to the beach.

Take a day off! Stop writing code, for goodness sake!

Remove unnecessary code. Methods to reduce the number of executable statements in your program could involve deleting non-crucial functions from the program, and eliminating any dead code or old redundant code that has been "left in" for various reasons. The use of compile-time initialization of global and `static` variables instead of assignment statements is another method for reducing code size. Turning off debug code such as assertions, debug tracing, and self-testing code can also work, but this loses the supportability benefit of shipping a fully testable version.

Compile-for-space options. Another possibility is that your compiler may support an option that causes the optimizer to focus on space reduction. This causes it to generate executable instructions that are as compact as possible, rather than being as fast as possible.

Avoid using large libraries. Pay attention to what code libraries you are linking with. Some of them are quite extensive, and may be much more than you need. Try to use the basic standard libraries as much as possible.

Template overuse. Templates are a common cause of "code bloat" and their usage should be reviewed. This is particularly true if you are using an integer-parameterized template in order to gain compile-time efficiency, or an approach such as Template Meta-Programming (TMP). If these templates are used with a large number of constant values, many copies of the template's executable code will be generated.

Avoid large `inline` functions. Overuse of `inline` functions has the potential to create more executable code. Try to limit your use of `inline` to small functions where the overhead of the function call is significant compared to the relatively low runtime cost of the function body. Don't inline large functions that are doing lots of processing each call.

Inline tiny functions. Although inlining large functions can cause code bloat, the reverse is usually true for very small functions. All of those getter and setter member functions have about one instruction. The code generated from an inlined call to these tiny functions may be much smaller than the instructions to call a real function.

`constexpr` is `inline`, too. Remember that `constexpr` functions are also effectively a type of `inline` function. Again, try to limit these to relatively small functions. If a `constexpr` function is called with non-constant values, or is beyond the compiler's ability to properly inline, then multiple copies of the executable code may result.

Library linkage. The size of the executable depends not only on the C++ code, but also on the extra library functions that are linked by the linker. Although it may seem that the programmer has no control over this, there are some techniques for reducing the amount of linked code. The techniques depend largely on how "smart" your linker is — that is, whether the linker links only the functions you need.

Use DLLs for common libraries. Dynamic link libraries (DLLs) are one way to reduce the size of the executable, because the library executable code is loaded at runtime. If the DLL is a commonly used library, such as the standard C++ runtime libraries, not only will your executable smaller, but it's also efficient at runtime because it will be loaded only once into memory, even if many programs are using the code. However, making your own special code into a DLL isn't likely to offer much memory benefit at runtime, since it will simply be loaded dynamically rather than immediately at load-time. However, if it's a library that isn't needed in many invocations of your program, you can save memory by deferring loading of the library until you can determine whether it will be required.

Remove executable debug information. Executable size can be reduced by avoiding generation of the "debug" information and symbol table information. For example, with GCC don't use the "`-g`" debugging information or "`-p`" profiling instrumentation options. Linux programmers can also use the "`strip`" utility which strips symbol table information from the executable after it has been created. However, the extra symbol table information is more relevant to the amount of disk space the executable file uses than to the amount of memory it uses during runtime execution.

Reducing Static Storage

Static storage refers to the memory for global and local `static` variables, string constants and floating-point constants. All of the general size-reduction above can reduce the size of the global and `static` variables.

String literal static memory. The space requirements for string constants can be reduced if the compiler has an option to merge identical string constants (which arise quite frequently).

If there is no such option, or the option does not merge string constants across object files (which is quite likely), merging string constants can be achieved by the programmer, although the method is far from elegant. For example, including this variable in a header file and using it in multiple source files may create multiple copies of the string literal:

```
    #define TITLE "A very long string ... "
```

Instead, a global variable can be declared to hold the string constant and the name of this char array is used instead of the string constant. In modern C++ you can use "`inline` variables" to avoid linker problems with multiple definitions.

```
inline const char TITLE[] = "A very long string ... ";
```

This change is unlikely to reduce the speed of the program, nor does it increase memory requirements even if `TITLE` is used only once (there may seem to be an extra 4 bytes to hold a pointer value pointing at where the string of characters is stored, but this is not so).

Large global variables. If there is a large global or `static` variable or array, the amount of static storage can be reduced by allocating it on the heap using `malloc` or the `new` operator, or by making it an automatic variable.

This is particularly useful if the object has a short "lifetime", in the sense that it is used only briefly (e.g., the array is used as temporary storage inside a function). When the variable is used all the time, this change doesn't reduce the overall space problem, but simply moves the problem to another area.

Stack Usage

Stack storage refers to memory storage used for function calls, and includes (non-static) local variables, function parameters and system information used to keep track of function calls. Hence, the basic methods of reducing stack storage are:

- Use fewer and smaller automatic local variables.
- Use fewer and smaller function parameters.
- Use "`const&`" to pass objects by reference.
- Use global or `static` local variables instead.
- Reduce the depth of function call nesting.
- Avoid recursion (always).

Data sizes. The size of parameters and local variables can be reduced using the general methods of using smaller data types. Another method is to avoid passing large objects and to only large objects by reference (which is faster anyway). Don't use large arrays or buffers as local variables, but prefer allocated buffers or global buffers, or declare them as local static variables.

Fewer parameters. The number of parameters can be reduced by using global variables, or by packing a number of parameters into an object and passing the whole object (which is often faster, too).

Fewer local variables. The number of local variables can be reduced by re-using local variables, although this can introduce bugs if not enough care is taken. Common examples of reusable variables are scratch variables, such as temporaries or `for` loop index variables. Another method of reducing the number of local variables is to use parameters as if they were local variables (this is safe because of call-by-value). Overall, most of these suggestions are minor improvements, unless you're using very large arrays or objects as local variables.

Flatten call hierarchies. Reducing the depth of function call nesting (especially by avoiding recursion) also reduces stack space requirements. This can be achieved by using preprocessor macros or `inline` functions (but this may increase code size). You can also refactor your code to avoid too many layers of wrapping functions in interfaces. Naturally, recursion should be avoided as much as possible by using iterative loop algorithms or tail recursion elimination.

Reducing Heap Usage

Your C++ IDE should support tools that track heap or stack usage dynamically. For example, MSVS has a "heap profiler" tool that you can enable. Linux tools such as Valgrind can be very usual to examine heap memory usage. The amount of heap storage depends on the size of blocks, the number of blocks and how quickly blocks are deallocated. The size can be reduced using the general techniques of reducing data sizes (e.g., small data types, packing, unions).

Fewer allocation calls. The number of heap blocks affects heap usage in the obvious way (more blocks means more memory) and because of the fixed space overhead of a few hidden bytes to store information about the block (so that `delete` or `free` can de-allocate it). When small blocks are used, it can be useful to pack more than one block together to avoid this fixed overhead.

Avoid small frequent allocations. If your frequently-used class allocates a small amount of memory in a constructor and then deallocates it in the destructor, consider alternatives. Small amounts of data could be stored in extra fields.

Memory leaks waste memory. Obviously, avoiding memory leaks which are never returned to the heap is important to reducing heap memory usage. There are many tools and debug libraries available to detect leaks, and ongoing use of these tools will reduce overall heap fragmentation.

Early deallocation of memory. It's a win if you have avoided leaking the memory, but that's not the end of the story. All allocated memory should be returned to the heap as early as possible. If memory is not deallocated, unused memory (called "garbage") can accumulate and reduce the available memory.

Avoid `realloc`. Measure and manage any calls to realloc, as they can be a significant cause of heap memory fragmentation. And they're also not time-efficient, so reducing them is a win-win.

Manage `std::vector` sizes via "`reserve`". The "`resize`" operations in the container `std::vector` can lead to lots of extra unnecessary allocation requests. Judicious use of the "`reserve`" function can avoid this.

Linearize multi-dimensional allocated arrays. One big allocation of a linear array is much more efficient on the heap than allocating separate blocks for rows or lower-dimensions of the array. An array of pointers into the linearized large block is only one more allocation, and has the same efficiency as having each pointer be a separate dynamically allocated subarray.

Smart buffers. Use objects that contain a limited amount of memory, which is used for the typical cases. If a longer string, or larger array is required, it needs to allocate memory. Overall, this can massively reduce the number of blocks.

Memory fragmentation. Reduce memory fragmentation by reducing both allocations and deallocations. It's also important to manage the different sizes of allocations, as varying block lengths cause more fragmentation.

Per-class allocators. In severe situations, take control of your class's dynamic objects by defining your own per-class allocators. Since the allocators knows that all block requests will be the same size, it can not only be faster, but also better at reusing memory blocks and avoiding memory fragmentation. But this method can also be a big fail if coded lazily to first allocate one huge chunk of memory. These allocators should dynamically manage requests for more storage, using a reasonable incremental size, rather than guessing their maximum requirements up front.

References

1. Ulrich Drepper (2007), *What Every Programmer Should Know About Memory*, November 21, 2007, http://people.redhat.com/drepper/cpumemory.pdf
2. Agner Fog (2023), *Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms*, PDF: https://www.agner.org/optimize/optimizing_cpp.pdf
3. Kurt Guntheroth (2016), *Optimized C++: Proven Techniques for Heightened Performance*, O'Reilly Media, https://www.amazon.com/dp/1491922060
4. Wikibooks (2023), *Optimizing C++/Writing efficient code/Performance improving features*, Wikibooks, https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Writing_efficient_code/Performance_improving_features
5. Bjorn Andrist, Viktor Sehr (2020), *C++ High Performance: Master the art of optimizing the functioning of your C++ code, 2nd Edition*, Packt Publishing, Dec 2020, https://www.amazon.com/dp/1839216549, Code: https://github.com/PacktPublishing/Cpp-High-Performance-Second-Edition (Chapter 7 is on memory management.)
6. Dung Le, Jul 30, 2020, *CUDA Memory Management & Use cases*, https://medium.com/distributed-knowledge/cuda-memory-management-use-cases-f9d340f7c704
7. Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H.S. Torr, Pushmeet Kohli, *Learning to superoptimize programs.* In International Conference on Learning Representations (ICLR) (2017). https://arxiv.org/abs/1611.01787
8. Z Guo, Z He, Y Zhang, 2023, *Mira: A Program-Behavior-Guided Far Memory System*, PDF: https://cseweb.ucsd.edu/~yiying/Mira-SOSP23.pdf (Interesting memory management methods.)

49. Loop Vectorization

Sequential vs Parallel Loop Optimizations

Loops are often sources of inefficiency and can be optimized in numerous ways. And the basic algorithms for neural networks are full of loops, with nesting to multiple levels in tensor operations. Increasing throughput of GPU data processing is one of the main goals achieved by loop optimizations.

Not all loop transformations are created equal. Some of them are best for sequential code optimizations, whereas other loop transformations are used to parallelize loops for vectorization. Loop transformations that are good for both sequential and parallel loop optimization include:

- Loop unrolling —reduce test overhead and parallelize the body.
- Loop peeling — unroll the first few iterations.
- Loop coalescing — flatten nested loops.
- Loop splitting — split out subportions of the iteration range.
- Loop collapsing — another way to flatten nested loops.
- Loop interchange — switch inner and outer loop iterators of nested loops.
- Loop reordering — change the ranges of inner and outer nested loops.

Some loop transformations are mainly for sequential improvements, and are not parallelization in themselves. However, these techniques can sometimes help with parallelization if they enable another followup loop parallelization optimization. Loop transformation optimizations which tend to be good for sequential code optimizations but not parallelization include:

- Loop fusion — combine or "fuse" the bodies of two loops.
- Duff's device — amusing but impractical coding trick for loop unrolling.
- Loop code motion — move or "hoist" loop-invariant calculations from the loop body to pre-loop initialization.
- Loop perforation — randomly skip some loop iterations; it's really a thing.
- Loop sentinel — fake it till you make it.
- Loop iterator strength reduction — change "*" to "+" if you can.
- Loop reversal — going backwards, and yet, still making progress!

Parallelizing loop optimizations with a main goal of vectorization of the loop body include:

- Loop fission — opposite of loop fusion; split a single loop body into two.
- Loop tiling — process sub-parts of contiguous data in separate loops.
- Loop distribution — split two sub-parts of a loop body into two simpler separate loops.

Loop Fusion

Loop fusion is a well-known code optimization where two separate loops are merged into a single loop. This does not change the amount of in-loop computation in either loop body, but reduces the loop overhead of the exit test by half. There is also often a benefit from data locality that reduces data movement and temporary data storage, which can also improve overall speed.

Note that loop fusion is not great at vectorization, because complicated loop bodies are actually harder to parallelize. Most of the benefits arise in traditional sequential code execution, which is why its theory dates back many decades. For modern parallel execution on GPUs, loop fusion is often a poor choice, and more benefits may arise from loop fission (the opposite of fusion) and loop vectorization.

Example: Loop Fusion: The general idea is to combine the body of two loops into a single loop. Here is a simplistic example with the (non-fused) loops for initializing two vectors using two sequential loops:

```
for (i = 0; i < n; i++) v1[i] = 0;
for (i = 0; i < n; i++) v2[i] = 0;
```

And here is the version with loop fusion:

```
for (i = 0; i < n; i++) {
    v1[i] = 0;
    v2[i] = 0;
}
```

Note that the loop fusion version incurs the same number of assignments for initialization, but only half of the loop overhead cost (i.e., half of the "`i < n`" and "`i++`" operators have been optimized away). And for the sake of argument, let's pretend that we don't know a better way to initialize a vector data structure in C++ like `memset` or `calloc` or load-time `static` variable initialization.

Loop Perforation

The intentional introduction of randomness to executable code is known as a "stochastic" algorithm. Personally, I'm more familiar with the unintentional introduction of randomness, otherwise known as a "bug," but now when it happens you can tell your boss that you were adding "stochastic functionality."

Code perforation is an optimization technique that trades accuracy for speed, by randomly (ahem, I mean, stochastically) skipping some computations. Essentially, using loop perforation is similar to an approximation with a random element, but in a generalized way for any iterative code. It's kind of like how teenage children randomly skip their homework.

Loop perforation skips iterations of a loop in a probabilistic manner. Randomly skipping some percentage of the loop bodies doesn't sound like a good plan, but it has its merits. In an AI inference computation, there's so much going on that no-one's going to notice a few missed beats. Apparently it can even be useful. Well, at least it's faster to do nothing.

Example: Loop Perforation: Here is an example of adding loop perforation to a vector dot product computation. This is an incredibly slow version, and is not recommended, but is just to give the idea of skipping a percentage of the iterations:

```
float aussie_vecdot_perf(float v1[],float v2[],int n,int pc)
{
    // Loop perforation -- vector dot product
    float sum = 0.0;
    for (int i = 0; i < n; i++) {
        if ( ( rand() % 100 ) + 1 <= pc) {
            // This iteration is perforated...
            continue; // Skip it...
        }
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

Loop Unrolling

Loop unrolling is a code optimization where the body of a loop is repeated in sequential code. This speeds up the algorithm because the overhead of both the incrementer and the loop iteration test is avoided.

In some cases, the entire loop can be unrolled, usually when the loop iterations are finite and known at compile-time. In other cases of partially unrolling, the loop body can be repeated multiple times, and thereby the loop test only occurs every few iterations.

For an AI engine, loop unrolling is used as an optimization in a few places. It is one of the optimizations used by kernel fusion, along with loop fusion and others. Since many meta-parameters of AI models are finite and fixed numbers (e.g., the "model dimension"), there are many cases where an entire loop can be unrolled and then vectorized into the GPU.

The logical extension of loop rolling is done by machine learning compilers, at least from a conceptual point of view. These ML compilers unroll the inference loop and the lower-level loops in matrix operations, thereby creating a finite graph representation of the entire inference sequence. If all is unrolled, there are no loops in the graph (an "acyclic" graph) and it is of finite size. The process of model inference is propagation of data through the graph. There are many "graph optimizations" that can be made on this graph representation of the AI model.

Example: C++ Loop Unrolling of Vector Dot Product. Here is the basic C++ non-unrolled vector dot product code:

```
float aussie_vecdot_basic(float v1[], float v2[], int n)
{
     // Basic vector dot product
     float sum = 0.0;
     for (int i = 0; i < n; i++) {
         sum += v1[i] * v2[i];
     }
     return sum;
}
```

If we know the value of *n*, e.g., that *n=5*, then we can completely unroll it:

```
return v1[0] * v2[0]
     + v1[1] * v2[1]
     + v1[2] * v2[2]
     + v1[3] * v2[3]
     + v1[4] * v2[4]
     ;
```

If we don't know the value of *n*, we can still unroll multiple iterations. Here's an example of 4-level loop unrolling of vector dot product in C++ by assuming that *n* is a multiple of 4:

```
float aussie_vecdot_unroll4(float v1[],float v2[],int n)
{    // Loop-unrolled Vector dot product
     if (n % 4 != 0) {
         aussie_assert(n % 4 == 0);
         return 0.0; // fail
     }
     float sum = 0.0;
     for (int i = 0; i < n; ) {
         sum += v1[i] * v2[i]; i++;
         sum += v1[i] * v2[i]; i++;
         sum += v1[i] * v2[i]; i++;
         sum += v1[i] * v2[i]; i++;
     }
     return sum;
}
```

And here's a generalization of that 4-level unrolling with extra code to handle the leftover cases if *n* is not a multiple of 4. Although the extra cases look messy, they are not actually the main performance bottleneck.

```
float aussie_vecdot_unroll4b(float v1[],float v2[],int n)
{
    // Better loop-unrolled Vector dot product
    int i = 0;
    float sum = 0.0;
    if (n % 4 != 0) {
        // Handle the extra cases...
        switch (n % 4) {
        case 1: sum += v1[i] * v2[i]; i++;
            break;
        case 2:
            sum += v1[i] * v2[i]; i++;
            sum += v1[i] * v2[i]; i++;
            break;
        case 3:
            sum += v1[i] * v2[i]; i++;
            sum += v1[i] * v2[i]; i++;
            sum += v1[i] * v2[i]; i++;
            break;
        default: aussie_assert_not_reached(); break;
        } // end switch
        // Keep going with rest of the vector
    }
    for (; i < n; ) {  // Unrolled 4 times...
        sum += v1[i] * v2[i]; i++;
        sum += v1[i] * v2[i]; i++;
        sum += v1[i] * v2[i]; i++;
        sum += v1[i] * v2[i]; i++;
    }
    return sum;
}
```

This code is just an example for explanation. There are various further code optimizations that can be done for production-level efficiency. For parallelization, the loop body should call an intrinsic function to vectorize the method. For an AI engine, we could choose our model dimension and other meta-parameters as multiples of the loop unrolling factor, and thereby avoid ever having any of the "leftover" cases.

For sequential code, we could change it to use pointer arithmetic rather than array indices, we might try replacing the four `i++` operators with `i+=4`, change the integer modulo operator (`%`) to a bitwise-and operator test (i.e., use "`n&3`" not "`n%4`", which works since 4 is a power-of-two), and it also might be better to use "+" rather than the "+=" operator. Finally, if we carefully code the leftover cases, the main loop could be unrolled to many more levels than just four.

Duff's Device for Loop Unrolling

There's a neat coding trick called "Duff's Device" for loop unrolling, which uses a `switch` with `case` fallthrough to mimic assembler coding style. However, it's not great for vectorization as it's likely to confuse the compiler, so may be mostly of theoretical interest.

```
float aussie_unroll4_duff(float v1[],float v2[], int n)
{
    // Unrolled dot product with Duff's Device
    int i = 0;
    float sum = 0.0;
    switch (n % 4) {
        for (; i < n; ) {
            case 0: sum += v1[i] * v2[i]; i++;
            case 3: sum += v1[i] * v2[i]; i++;
            case 2: sum += v1[i] * v2[i]; i++;
            case 1: sum += v1[i] * v2[i]; i++;
            default:;
        } // end for
    } // end switch
    return sum;
}
```

What's happening here? My brain hurts looking at this code! The trick is that the outside `switch` branches into a `case` that is inside the body of a `for` loop. This is not normal everyday coding, because there's a loop inside a `switch`, and the loop body crosses over several `case` statements.

Also, none of the case statements has a "break" statement and they instead rely on fallthrough semantics. Similarly, the "default" clause is mainly just to avoid getting a spurious compilation warning (i.e., "missing default"), and also has no "break" with only a lonely semicolon. Note also that the case labels are written in reverse order from top to bottom (3..2..1), except for 0 at the top.

How does this even work? The first point is that it *does*. This code performs the exactly correct number of iterations for any value of n (except n==0), and similar versions with an unrolling factor of more than 4 will also work (i.e., if you change "n%4" and add more case constants). The code looks like a hack, but actually uses standardized C++ semantics of case fallthrough and switch multi-way control flow and should work on all platforms. Branching into the middle of a loop with a switch is valid in C++ provided it doesn't bypass any local variable initialization (hence, don't put "sum" into the switch). Also, the case fallthrough semantics (i.e., without a "break" ending each "case") are standard for C and C++ since inception. Finally, note that this code is buggy for n==0, because it incorrectly does 4 iterations, so it ideally needs a parameter validation assertion at the start.

Bug alert! Note that you cannot tweak the "i++" instruction using the standard idiom:

```
   sum += v1[i] * v2[i++];  // Bug!
```

The obscure problem is that the "*" operator doesn't guarantee left-to-right evaluation of its operands. The code assumes a computation evaluation order of: v1[i], v2[i], *, i++, starting from the left. However, the C++ optimizer can legally do this order of operations: v2[i], i++, v1[i], *, which is not what you intended and gets the wrong array element for v1[i].

This code might be unreliable across platforms, or it might work in the debugger mode, but fall over once you turn on high levels of optimization. So, there is an "order of evaluation" pitfall if you put "++" in an operand of the "*" operator or many other binary arithmetic operators.

Is Duff's Device any faster? The short answer is "not really," although it looks very appealing (or appalling). Firstly, note that this trick is not actually very useful for vectorization, because a switch cannot branch into the middle of a vectorized intrinsic (i.e., if you replace the loop body with a SIMD instruction). Furthermore, although I haven't tested it, I doubt many optimizers will be able to auto-optimize that complex control flow with SIMD instructions. In sequential code, this method also isn't much faster, as it doesn't really have any fewer operations than a basic unrolled loop (i.e., with extra cases handled separately before or after the main loop).

The above example of Duff's Device can be further sped up using pointer arithmetic and "looping down to zero" optimizations, but so can the other unrolled versions. However, there is a minor speed advantage in terms of "instruction locality" because the above code is very concise.

The main advantage of Duff's Device is to bamboozle your colleagues. You can use Duff's Device with any unrolling factor, not just 4 as in the example shown above (e.g., change to 8 by using "`n%8`" and adding cases for 4, 5, 6, and 7, ordered from 7 down to 1, leaving 0 on top). Actually, the unrolling factor needn't be a power-of-two. Make it a prime number for extra bonus points. If you want more of this kind of coding trickery, also search up Jensen's device and Pigeon's device.

Loop Tiling or Blocking

When you hear about a "tiled MatMul" or a "blocked GEMM," this is the "tiling" or "blocking" optimization method it refers to. MatMul is matrix multiplication and GEMM is General Matrix Multiplication (i.e., the same thing). Tiling is the optimization that most applies to speeding up matrix or tensor multiplication in AI engines.

This optimization is for two-dimensional data (e.g., matrices). When you hear "tiles" or "blocks," think squares or rectangles of data. For example, if you have a 512x512 matrix, then a tiled algorithm might act on 16x16 sized chunks, one at a time. Loop tiling is an optimization of two-dimensional or three-dimensional data such as matrices or tensors. The one-dimensional equivalent of processing sub-parts of a one-dimensional array is called "strip mining", "loop sectioning" or often simply "vectorization."

In other words, tiling means operating on small subsections of a matrix. If you hear "tiled tensor" that could mean two-dimensional data (i.e., just a fancy name for a matrix), or alternatively it might refer to three-dimensional data, in which case, don't think anything or else your head will hurt.

Loop tiling is a method of executing sub-parts of nested loops in a way that maximizes data locality, increases cache utilization, and improves parallel execution. This is also called "loop blocking" because it processes the data a "block" at a time, although the term "tiling" is more widely used in research. The two-dimensional sub-partitions of the data that are square or rectangular are called "tiles" or "blocks".

The same number of arithmetic operations are performed in a tiled versus non-tiled algorithm. However, there should be fewer loads of the data into memory with tiling. The downside is that tiling introduces additional loop overhead. In fact, rather than flattening nested loops over a 2-D array (e.g., 512x512), tiling often introduces additional levels of nesting! The two small loops that spin through the 16x16 square shape of a single "tile" or "block" are often newly added inner loops. So, loop tiling often adds two new layers of nested loops inside your already-nested loops. It makes you wonder how it can even be faster!

Example: Tiled Matrix Clear: For these examples, there is a type "`ymatrix`" type:

```
typedef float ymatrix[ROWS][COLUMNS];
```

If we forget about `memset`, here is the simple code to clear a matrix one element at a time in a brute-force nested loop (non-tiled):

```
void aussie_clear_matrix(ymatrix m)
{
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLUMNS; j++) {
            m[i][j] = 0.0;
        }
    }
}
```

Now we decide to add a 4x4 square tile optimization to this code. The result is an extra two levels of nested loops. Here is the basic code which assumes that the row and column dimensions are exact multiples of the tile size, so there's no extra leftover cases to handle:

```
void aussie_clear_matrix_tiled(ymatrix m)
{
    const int TILEX = 4, TILEY = 4; // 4x4 tile size
    static_assert(ROWS % TILEX == 0, "Exact X");
    static_assert(COLUMNS % TILEY == 0, "Exact Y");
    for (int i = 0; i < ROWS; i += TILEX) {
      for (int j = 0; j < COLUMNS; j += TILEY) {
          // Do the 4x4 tile...
          for (int tx=i; tx < i+TILEX; tx++) {
              for (int ty=j; ty < j+TILEY; ty++) {
                  m[tx][tiley] = 0.0f;
              }
          }
      }
    }
}
```

Unrolled Tiles. One followup optimization trick with a tiled loop algorithm is to apply loop unrolling to the two inner loops. This avoids the extra overhead of the two extra inner loops, but retains the data locality benefits of tiling. This optimization results in a fully "unrolled tile" computation without any extra inner loops. In the above example, the two inner loops of a 4x4 tile would be replaced with 16 unrolled computations in sequence. Or for a vectorized version, a fully unrolled tile would be 4 sequential calls to vectorized intrinsics that each do 4 operations in parallel (e.g., AVX intrinsics each do 4 `float` operations in parallel).

Example: Tiled Matrix Multiplication: Tiling techniques are widely used inside neural network code to improve the efficiency of MatMul and thereby get better throughput of tensor calculations from a GPU. Matrix multiplication is a good candidate for this optimization because it has O(n^3) arithmetic calculations, but uses only O(n^2) data. Hence, a naive matrix multiplication algorithm that doesn't address locality will re-load the same data into memory many times, whereas a tiled algorithm can reuse the same data more efficiently.

A tiled version of MatMul processes "tiles" or "blocks" of each matrix one at a time (i.e., small square or rectangular sections), with the aim of keeping small parts of the matrix in the memory cache while they are processed. The algorithm progresses across the matrix a tile/block at a time, rather than scanning all the way down one dimension (row or column). The same number of multiplication operations are performed as a non-tiled MatMul, but data locality and cache freshness should improve the overall speed.

Loop Fission

Loop fission is an optimization that is the opposite of loop fusion. Instead of fusing two loops into one, we take one loop and split parts of it into two loops. Loop fission also been called other names such as "loop splitting" or "loop distribution."

Loop fission can be more efficient for parallel execution (e.g., vectorization for GPUs), but is often slower for sequential execution. Whereas loop fusion aims to remove the overhead of one of the loops, loop fission tolerates an increased loop overhead in return for simpler loop bodies that can be parallelized. The kernel optimization of "kernel fission" is based on loop fission, and loop fission is one technique used to achieve vectorization for GPUs.

The main reason to use loop fission is hardware acceleration via loop parallelization. A complicated single loop can often run faster if split into two simpler loops, if hardware acceleration can be accessed.

This is true even if the two resulting loops must run sequentially, because the iterations of each loop are parallelized, but there's a double benefit if the two whole loops can also run in parallel.

Example: Loop Fission in BatchNorm: A good example arises in part of the code for batch normalization. Each element of the vector needs to have two operations performed on it: subtract the mean (re-centering) and multiply by a variance factor (re-scaling). The naive implementation of the second half loop in BatchNorm looks like this:

```
float denom = sqrtf(varc + eps); // Scale factor
for (int i = 0; i < n; i++) {
    // Normalize: re-center and scale
    v[i] = (v[i] - fmean) / denom;
}
```

This is difficult to hardware accelerate because it's unlikely that there's a combined "subtract-and-then-divide" operation to apply to all elements of a vector in parallel. The first point is that maybe there's an "add-and-then-multiply," in which case we can use the negative of the additive factor and the reciprocal of the scaling factor. However, assuming there's not, loop fission can be used to split the single complicated loop into two sequential loops.

```
float negmean = -fmean;  // Use negative for addition
float denom = sqrtf(varc + eps); // std. deviation
float recip = 1.0f / denom;  // reciprocal multiply
// Loop 1: Re-center using mean
aussie_vector_add_scalar(v, n, negmean);
// Loop 2: Re-scale by factor
aussie_vector_multiply_scalar(v, n, recip);
```

Each of the two loops is now easy to hardware accelerate, because they are both very simple vector operations: "multiply-by-scalar" and "add-scalar." Every platform is likely to have hardware acceleration APIs for those simpler operations. So, to summarize, we got an explosive boost to hypersonic rocket speed using atomic operations with loop fission.

Isn't that just the bomb?

Loop Reversal

Loop reversal is the optimization of making the loops go backwards. It does the same number of arithmetic operations, but in reverse order, so there is no change in the total arithmetic operations.

This goal is a speedup by "looping down to zero" with a faster loop test, but it is often a de-optimization even for sequential execution. Typical CPU processors rely on ascending order of memory accesses for predictive cache pipelining, and reverse array access is a worst case for that.

Loop reversal is also not a useful parallelization method in itself. Vectorization for GPU computation doesn't really work in reverse. However, reversing a loop can sometimes be useful as an initial transformation on nested loops if reversing the inner loop's direction allows another followup loop vectorization technique.

Example: Reversed Vector Dot Product: Loop reversal can be used on vector dot product, as below, but it probably shouldn't be. Here's the basic idea:

```
float aussie_vecdot_rev(float v1[], float v2[], int n)
{
    float sum = 0.0;
    for (int i = n - 1; i >= 0; i--) {
        sum += v1[i] * v2[i];
    }
    return sum;
}
```

Note that there are several coding pitfalls to avoid. The loop variable "`i`" cannot be "`unsigned`" or "`size_t`" type, because the test "`i>=0`" would never fail, creating an infinite loop. Also, the reversed loop needs to start at "`n-1`" and must use "`i>=0`" (not "`i>0`") to avoid an off-by-one error. The above code also craters for "`n<=0`" and needs a safety test.

Loop Code Motion

Loop code motion is moving loop-invariant code from inside the loop body to the pre-initialization code for the loop. Any code that has the same value should not be performed inside the loop body. Instead, it should be pre-calculated before the loop, and stored in a temporary variable. This is sometimes called "hoisting" the code out of the loop.

Example: Loop Code Motion: One common example of unnecessary recalculation of loop-invariant values is in the loop test. The code in the Boolean test for the loop is actually part of the loop body.

An example of code that re-calculates the loop limit:

```
for (i = 0; i < vec.num_elements(); i++) {
   // ...
}
```

The "`num_elements`" call is probably loop-invariant, assuming the vector doesn't change size during processing. Maybe the "`num_elements`" function is declared "`inline`" and the C++ compiler will fix it anyway. Nevertheless, this is a candidate for loop code motion, using a temporary variable instead:

```
int n = vec.num_elements();  // Loop-invariant value
for (i = 0; i < n; i++) {
   // ...
}
```

Loop Distribution

Loop distribution is type of loop code motion that creates two loops from a single loop that contain an "`if`" statement. The hoisted code is a conditional test. Some early papers in the 1990s called it "loop unswitching." Some papers use the term "loop distribution" with the different meaning of splitting a loop into two loops, which we call "loop fission."

The goal of loop distribution is to move an "`if`" test out of the loop body, by creating two loops, and ends up creating two separate loops on two pathways. This sounds similar to loop fission, but loop distribution is a more general optimization that doesn't require parallelization to get a speed improvement (whereas loop fission does).

Instead, loop distribution gets a benefit in ordinary sequential execution because it moves the `if`-test computation out of the loop body to a once-only pre-initialization test (i.e., "hoisted"). Note that only one of the two loops is executed each time, and these two loops are never executed in parallel, so this technique is not really a type of loop fission.

Example: Loop Distribution: Here's a dummy example of implementing an "add-or-subtract" function using a passed-in Boolean flag.

```
    void aussie_vector_addition_slow(
        float v[], int n,
        bool do_add, float scalar)
    {
        for (int i = 0; i < n; i++) {
            if (do_add)
                v[i] += scalar; // Add
            else
                v[i] -= scalar; // Subtract
        }
    }
```

The problem is that the test "`if(do_add)`" is computed for every loop iteration, and yet "`do_add`" is a loop-invariant flag variable. The faster version is to use loop distribution to move the `if`-test into the loop initialization, and then split the two pathways inside the loop to instead have two separate loops. Here's the faster version:

```
    void aussie_vector_addition_loop_distribution(
        float v[], int n,
        bool do_add, float scalar)
    {
        if (do_add) { // Add scalar
            for (int i = 0; i < n; i++) {
                v[i] += scalar;  // Add
            }
        }
        else {  // Subtract scalar
            for (int i = 0; i < n; i++) {
                v[i] -= scalar; // Subtract
            }
        }
    }
```

This example is still far from optimal. For starters, it should be using pointer arithmetic rather than array indices.

Loop Reordering

In neural networks, there are many loops, and many ways of nesting them, or running them in sequence. The convolution layers in CNNs can have literally seven layers of nested loops. Hence, there are various research papers exploring different orders to perform the various computations.

Loop reordering is the general class of optimizations that involves reordering loops or their iterations. This can refer to changing the ordering of two sequential loops or two nested loops. The reordering optimization to reverse the inner and outer nested loops is more precisely called "loop interchange." A single loop can also be reordered with "loop reversal."

Loop reordering is an optimization that doesn't reduce the total computations, because it always executes the same number of iterations as the original version. However, loop reordering may have several benefits:

- Vectorization. Putting the loop in a different order may make it more vectorizable, or may allow other loop transformations to be applied before vectorization.
- Data locality. Reordering the loops may improve data locality and cache access speed by doing the operations in a different order. This reduces the cost of accessing the data into memory (or low-level caches), rather than the cost of the arithmetic. It is therefore related to memory/dataflow optimizations and pipelining optimizations.
- Reduced loop overhead. Both loop interchange and loop reversal can reduce the general overhead of loop testing. Loop interchange allows the shorter loop to be on the outside. Loop reversal allows "looping down to zero" which reduces overhead.

Loop Iterator Strength Reduction

Loop strength reduction is the arithmetic optimization of "strength reduction" applied to loop iteration variables. For example, strength reduction aims to replace multiplication with addition. Consider this loop:

```
for (int i = 0; i < n; i++) {
    a[i] = 10 * i;
}
```

This can be optimized to change the multiplication into an incremental addition:

```
for (int i = 0, x = 0; i < n; i++) {
    a[i] = x;
    x += 10;
}
```

Note that the loop strength reduction optimization isn't a good choice for loop parallelization. Although it would be desirable to change a vectorized multiplication to addition, this optimization has changed to an incremental algorithm. This makes each loop iteration dependent on the prior one, with the results dependent on the previous computation, so they cannot be done in parallel.

Loop Coalescing

Loop coalescing is a loop optimization that involves flattening two nested loops into one non-nested loop. Typically, loop coalescing will still operate on a 2-dimensional array, whereas flattening both the nested loops and the array is called "loop collapsing."

As a dummy example, consider a matrix initialization via nested loops:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        arr[i][j] = 0.0f;
    }
}
```

Loop coalescing involves changing to a single loop, but still using two indices i and j, which are calculated from the main linear index.

```
int maxx = n * m;
for (int x = 0; i < maxx; x++) {
    int i = x / n;
    int j = x % m;
    arr[i][j] = 0.0f;
}
```

The benefit in speed from loop coalescing can arise by simplifying the loop, which makes it easier to parallelize via hardware acceleration, and also maybe a different data access pattern which might improve data locality and cache freshness.

This optimization is not always possible, as nested loop logic is often quite complicated, and flattening a nested loop may actually worsen data locality in many instances. However, the linear nature of a simple loop can make the code to send off chunks to a GPU much easier.

Loop Collapsing

Loop collapsing is closely related to loop coalescing, since both aim to flatten nested loops, but loop collapsing is a special situation where the array is also flattened to one dimension.

Consider a matrix initialization via nested loops over a 2-dimensional array:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        arr[i][j] = 0.0f;
    }
}
```

The loop collapsed version has one big loop over a different one-dimensional array:

```
int maxx = n * m;
for (int x = 0; x < maxx; x++) {
    arr2[x] = 0.0f;
}
```

This loop transformation to a single loop is obviously more amenable to vectorization.

Loop Peeling

Loop peeling is a type of loop unrolling that involves unraveling only the first few iterations of a long loop. This is also similar to "loop splitting" with two sections, where the first section is over the early range, and the second range is the main section of all remaining iterations.

Loop peeling is beneficial to the overall loop efficiency if there is code in the loop body that is only required for one or two early iterations, which can then be removed from the main loop body. Similarly, there can be benefit in unraveling the last few iterations of a loop, which is a similar technique.

One common case of loop peeling is when the first iteration is different from the rest, so peeling off a single iteration is valuable.

```
    for (int i = 0; i < n; i++) {
        arr[i] = (i == 0) ? 0.0f : 1.0f;
    }
```

In this case, we can peel off the first "`i==0`" iteration into a single unrolled instruction, and change the main loop to start at 1. This is also a trivial form of "loop distribution," where we are hoisting an "`if`" conditional test out of the loop. The new code becomes:

```
    arr[0] = 0.0f;  // Peeled
    for (int i = 1 /*not 0*/ ; i < n; i++) {
        arr[i] = 1.0f;
    }
```

This peeled version is faster in terms of both sequential or parallel execution. The loop body has less computation and is also more amenable to vectorization.

Loop Splitting

Loop splitting refers to splitting the sequential iterations of a loop into two loops, which each perform part of the original loop's iterations. Loop splitting is closely related to "loop sectioning" ("strip mining"), but often relates to more complex arithmetic in the loop body. Note that "loop peeling" is a special case of loop splitting where the first section is a small range of a few initial iterations, but these few iterations are unrolled rather than looped.

Loop splitting takes a single loop and transforms it into at least two "split-out" loops, one for the early iterations, and one for the remainder. However, loops can also be split out into more than two loops.

In loop splitting, each split-out loop is shorter than the original loop. Unlike loop fission, the two loops operate over different subportions of the iterator variable range, executing the same number of total iterations, rather than double iterations as in loop fission.

Example: Loop Splitting: Here's some example code to "sqrtize" a vector, using a cached optimization for the numbers up to 100.

```
    void aussie_vector_do_sqrt(float v[], int n)
    {
        for (int i = 0; i < n; i++) {
            if (i < 100) { // Fast cases
                v[i] = aussie_sqrt_optimized(v[i]);
            }
            else {  // General case
                v[i] = sqrtf(v[i]);
            }
        }
    }
```

However, we can use loop splitting to split this big loop into two shorter disjoint ranges. Instead of 0..n-1, we do 0..99, and then 100..n-1. Each loop header is over part of the range, and has a simpler loop body. Note that this code fails with an array bounds violation for small values of `n` less than 100.

```
    void aussie_vector_do_sqrt_loop_splitting(
          float v[], int n)
    {
        for (int i = 0; i < 100; i++) { // Fast cases
            v[i] = aussie_sqrt_optimized(v[i]);
        }
        for (int i = 100; i < n; i++) { // General cases
            v[i] = sqrtf(v[i]);
        }
    }
```

The loop splitting optimization is beneficial if the loop body has different sections of code that only relate to a subset of the iterator range. Hence, the loop bodies for the two loops can be reduced to execute less code. Overall, there is still the same number of iterations performed in the two loops combined, but each loop performs only a proportion of the original iterations on a simpler loop body. This optimizes sequential execution and the simpler code in each loop body may make vectorization of one or both subloops easier. Furthermore, both subloops could run in parallel.

Loop Interchange

Loop interchange is an optimization of nested loops that switches the inner and outer loops. In a typical nested loop, the outer loop body and loop test is executed rarely, almost lazily, whereas the inner loop body is scrambling along in a frantic mess. Loop interchange simply switches them, reversing their roles.

Why is this an optimization? Although the same number of loop iterations still occur in total, and the newly-made inner loop body is also thrashed, various improvements can arise from reversing the iterator variables, usually to make the innermost loop the longest. Possible optimizations result from:

- Fewer outside computations. A shorter outside loop reduces the arithmetic operations of the outer loop, whereas the inner loop's number of computations is unchanged in either loop structure.
- Data locality. Another possible improvement is in data locality, which can reduce cache misses and speeds up the overall execution. Note that this benefit is not guaranteed just by switching loops, and sometimes loop interchange can worsen data locality; careful analysis is needed.
- Inner loop vectorization. Another important possibility is that reversing nested loops can create opportunities to apply other loop optimizations to the new inner loop, notably to vectorize the inner loop.

Shortest loop outside, longest innermost loop: One of the considerations of loop interchange is the optimization of putting the shortest loop on the outside, and making the innermost loop with the longest range of iterations. This is an optimization for both sequential or parallel execution. For sequential execution, there is less overhead from the outer loop, because it is shorter. For parallelization, there is improved vectorization of the inner loop, which now has a longer range.

Consider this example:

```
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 50; j++) {
            // ...
        }
    }
```

The current loop nesting has the longest loop (to 1000) on the outside, and the shorter loop (to 50) as the innermost loop.

Loop interchange simply makes it the reverse nesting:

```
    for (int j = 0; j < 50; j++) {
        for (int i = 0; i < 1000; i++) {
            // ...
        }
    }
```

Considering sequential execution, the inner loop body is executed the same number of times, so there's no difference. This also includes the inner loop's conditional test and incrementer, which are different variables in the two examples, but also execute the same number of times (50,000 times). However, consider the different outer loops. The first example is 1000 iterations, whereas the second example's outer loop is only 50 times. Hence, the loop reordering optimization of "shortest outer loop" and "longest innermost loop" has saved 950 of the outer loop's calculations (i.e., loop test and incrementer). Any extra code that's in the outer loop, either before or after the inner loop, would also be executed fewer times.

There is also an advantage for vectorization. In the first example, we could possibly have 1000 vectorized operations of data size 50. In the interchanged loops, there are 50 operations on vectors size 1000. Hence, there is more opportunity for much larger vectorization gains in the second format with the longest inner loop.

Loop Sentinel

Loop sentinels are an optimization that removes the overhead of checking an array index or pointer scanning an array or pointer chain. The technique does this by adding a pretend extra element onto the end of the array, in a way that we can pretend to succeed.

And since we're guaranteed to always succeed, we don't need to check for failure while scanning the loop.

This technique is not particularly useful for vectorization, but is quite powerful for long sequential scanning of arrays. It also has the downside of requiring at least one writeable array element, so it cannot run on read-only arrays.

Example: Check Vector Negatives: Here's the basic loop sentinel version that sets up a dummy success in `v[n]`:

```
bool aussie_vector_has_negative_sentinel(
    float v[], int n)
{
     v[n] = -99.0;  // Dummy negative (BUG!)
     int i = 0;
     for ( ; /*GONE!*/; i++) {
         if (v[i] < 0.0) break;  // Found negative
     }
     if (i == n) return false;  // Fake success
     return true;  // Found a negative (for real)
}
```

However, this is actually buggy, since "`v[n]`" is potentially an array overflow. A better version can manipulate the last valid element "`v[n-1]`" instead of modifying "`v[n]`". Then, we have to remember to fix it before we leave town. And we also have to remember to check the last vector element that we temporarily overwrote wasn't also a real success.

```
bool aussie_vector_has_negative_sentinel2(
    float v[], int n)
{
    float save = v[n - 1];  // Save it!
    v[n - 1] = -99.0;  // Dummy negative at end
    int i = 0;
    for ( ; /*GONE!*/; i++) {
        if (v[i] < 0.0) break;  // Found negative
    }
    v[n - 1] = save;  // Restore it!
    if (i == n - 1) {
        // At the dummy (fake success)
        if (save < 0.0) return true; // Must check
        return false;
    }
    return true;  // Found a negative (for real)
}
```

Loop Strip Mining (Loop Sectioning)

Loop strip mining is a loop optimization that scans or "mines" various "strips" in an array. It is related to "loop tiling" on arrays in two dimensions, but strip mining only applies to processing one-dimensional arrays. Loop strip mining is also called "loop sectioning" because it breaks an array up into sections that are operated on.

For a basic example, consider a simple array initialization:

```
for (int i = 0; i < n; i++) {
    arr[i] = 0.0f;
}
```

Let's assume we can parallelize this with 16 elements at a time (e.g., 512 bits total parallel processing, which is 16 separate 32-bit `float` variables). So, we want to process "strips" of length 16. For simplicity, let us assume that `n` is divisible exactly by 16, so there's no leftover work after the main loop.

```
for (int i = 0; i < n; i += 16) {
    // Initialize arr[i]...arr[i+15] in parallel
}
```

Obviously, this is a dummy example, where `memset` would do better for zeroing the array. Also, this really looks exactly like "vectorization" to me, where we are vectorizing 512 bits at a time (16 `floats`), and indeed the research mentions vectorization as one application. But loop strip mining and vectorization are not exactly the same techniques, because loop strip mining is a more general idea with other applications.

Loop Spreading

Loop spreading is an optimization of two non-nested sequential loops that have different iteration ranges. Typically, this refers to where the end ranges differ significantly. If the loop ranges only differ by an off-by-one issue, then only loop normalization is required.

Loop spreading modifies one of the loops, so that part of this loop fully overlaps with the other loop (i.e., ideally one loop "spreads out" further to match the other loop's end bounds). Hence, after loop spreading has occurred, this subloop can be fused with the other loop, and possibly parallelized. The remaining iterations that are not overlapping then have to be addressed in a followup partial loop (only for one of the loops).

Loop spreading mainly enables loop fusion as a followup optimization. For using loop fission on the two loops, it is not necessary to do loop spreading, since the two loops are already split apart, and each loop could already potentially be vectorized independently.

Loop Normalization

Loop normalization is not directly an optimization, but is a preliminary loop transformation that can make further loop optimizations easier. Followup optimizations might be to fuse the two loops with loop fusion, or to parallelize each loop, such as with loop fission or vectorization.

The goal of loop normalization is to make the loop iteration variables act across the same range. This applies to two sequential loops, rather than nested loops. Hence, loop normalization is needed when two loops in sequence are starting at different offsets (e.g., one is `i=1` and another starts at `i=0`), or are finished at different endpoints (e.g., `n` versus `n-1`).

If two loops have the same number of computations, but with different ranges, then one loop can be changed with an offset. For example, these loops differ with ranges `0..n-1` and `1..n`:

```
    for (int i = 0; i < n; i++) a[i] = 0;
    for (int j = 1; j <= n; j++) b[j] = 0;
```

These can be adjusted to the same ranges with a “`j+1`” index offset, as follows:

```
    for (int i = 0; i < n; i++) a[i] = 0;
    for (int j = 0; j < n; j++) b[j+1] = 0;
```

If the two loops have a different number of iterations, typically off by 1 or 2, then “loop peeling” can be used to unroll and split off one or two iterations and shorten the longer loop, so that both loops have the same number of iterations over the same range. For example, in this example, one loop is `0..n-1` and another is `0..n`:

```
    for (int i = 0; i < n; i++) a[i] = 0;
    for (int j = 0; j <= n; j++) b[j] = 0;
```

The way to normalize the loop ranges is to "peel" off the last iteration of the "j" loop:

```
    for (int i = 0; i < n; i++) a[i] = 0;
    for (int j = 0; j < n; j++) b[j] = 0;
    b[n] = 0;  // Peeled
```

This example has peeled the longer loop to make it shorter. An alternative would be "loop spreading" to lengthen the shorter loop, such as by adding an extra padding element into the array.

Normalizing two loops doesn't change the number of arithmetic computations. However, once two loops have normalized ranges, it becomes easier to see opportunities for further optimizations such as loop fusion or loop fission.

Loop Skewing

Loop skewing is a somewhat mind-bending method to change nested loops to make them more parallelizable. This technique applies when there are two nested loops, but the inner loop is difficult to parallelize because of a dependency on the outer loop variable. The performance advantage from loop skewing is not directly its usage, but because skewing changes then make possible other loop optimizations, especially loop interchange, which reorders the inner and outer loop.

The loop skewing solution is far from obvious. The range bounds of the inner loop are changed by "skewing" them by a factor based on the outer loop variable. And then, by some magical potion, this somehow breaks the dependence on the outer loop, and then the inner loop can run fast on a GPU. Who knew?

As a simplistic example, consider two nested loops:

```
    for (int i = 0; i < 1000; i++) {
        for (int j = 0; j < 50; j++) {
            arr[i][j] = something;
        }
    }
```

We can skew the inner loop by adding a skew factor based on the outer loop variable (e.g., "i" or "i+1" or something similar). Add this skew factor to the ranges of j, but then subtract the skew factor ("i") from any usages of the index "j" inside the inner loop's body.

```
for (int i = 0; i < 1000; i++) {
    for (int j = i; j < 50 + i; j++) {
        arr[i][j - i] = something;
    }
}
```

Hence, `j` has changed from the range (`0`...`50`) to the skewed range (`i`...`i+50`), by adding the skew factor "`i`" to the start and end. The use of "`j`" in the inner loop body has changed from "`j`" to "`j-i`" (i.e., subtracting the skew factor "`i`"). The result is a kind of skewed and "triangular" shape of `i` and `j` indices, but the actual arithmetic calculations are unchanged.

This newly skewed code isn't any faster, does exactly the same calculations on the 50,000 elements of array `arr`, and indeed is actually worse because of the extra "`50+i`" and "`j-i`" computations. However, in some cases, doing this weird skewing transformation then allows us to follow up with a loop interchange optimization, switching the inner and outer loops. And I'm not even going to pretend to understand this, but there are situations where the non-skewed inner loop cannot be vectorized or interchanged, but after we've skewed the loop, then we can interchange it, and then we get via hocus pocus a different inner loop that can then be vectorized. Hopefully, the GPU knows what's going on.

References

1. Allen, F. E., and Cocke, J. 1972. *A catalogue of optimizing transformations.* In Design and Optimization of Compilers, Prentice-Hall, Englewood Cliffs, N.J., pp. 1–30.
   PDF: https://www.clear.rice.edu/comp512/Lectures/Papers/1971-allen-catalog.pdf
2. D. F. Bacon, S. L. Graham, and O. J. Sharp. 1994. *Compiler transformations for high-performance computing* . ACM Computing Surveys 26, 4 (1994), 345–420. https://dl.acm.org/doi/10.1145/197405.197406,
   PDF: https://people.eecs.berkeley.edu/~fateman/264/papers/bacon.pdf (Paper with extensive coverage of numerous compiler auto-optimizations of program code.)
3. Paul Bilokon, Burak Gunduz, 8 Sep 2023, *C++ Design Patterns for Low-latency Applications Including High-frequency Trading*, https://arxiv.org/abs/2309.04259,
   Code: https://github.com/0burak/imperial_hft
4. Eric LaForest, March 19, 2010, *Survey of Loop Transformation Techniques*, ECE 1754, http://fpgacpu.ca/writings/SurveyLoopTransformations.pdf

5. B Qiao, O Reiche, F Hannig, 2019, *From loop fusion to kernel fusion: A domain-specific approach to locality optimization*, 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), https://ieeexplore.ieee.org/document/8661176 (Theory of loop fusion generalized to graph kernel fusion for image processing.)
6. Kathryn S. McKinley, Steve Carr, Chau-Wen Tseng, 1996, *Improving data locality with loop transformations*, ACM Transactions on Programming Languages and Systems, Volume 18, Issue 4, pp 424–453, https://dl.acm.org/doi/10.1145/233561.233564
7. B Blainey, C Barton, JN Amaral, 2002, *Removing impediments to loop fusion through code transformations*, International Workshop on Languages and Compilers for Parallel Computing, LCPC 2002: Languages and Compilers for Parallel Computing pp 309–328, https://link.springer.com/chapter/10.1007/11596110_21

50. Parallel Data Structures

Bit Vectors

Bit vectors are conceptually an array of N bits with 0 or 1 values. The term "bit set" is almost synonymous, but has a slightly different meaning. A bit vector maps a number at the index position to its binary bit value, whereas a bit set specifies whether a number is in a set of numbers. Both interpretations are valid, depending mostly on the application, and the underlying implementation of the data structure is almost identical.

In AI applications, a bit vector may represent a set of weights with 0 or 1 values, such as with binary quantization or XNOR neural networks. The computation for vector dot product on two bit vectors can be performed arithmetically using bitwise arithmetic.

Sparsity optimizations are another application of bit vectors. Pruning can often create "sparse" weight matrices, with lots of zeros and very few non-zero weights. A bit vector can then efficiently represent whether a weight in a vector has a non-zero value, which is then used to avoid doing any computations on zero values. An alternative to bit vectors for sparsity is to use permutation arrays of indices, as discussed further below.

Another application of bit vectors occurs in Bloom filter data structures, which are a probabilistic hybrid of hash tables and bit vectors. In this usage, a bit set represents whether an input number is found in the set of already-mapped numbers.

In practice, bit vectors or bit sets are often implemented as arrays of unsigned integers, with the bits packed into each integer. If the underlying unsigned type is 32-bits or 64-bits, then many bitwise operations on bit vectors can be performed 32 or 64 bits at a time, achieving significant parallelism without using any major form of hardware acceleration beyond basic CPU instructions. Use of AVX SIMD instructions can then further vectorize many operations without a GPU. But it absolutely flies if you use a GPU with bit vectors or bit sets, because that's two levels of parallelization.

There are several pre-built C++ bit set classes that can be considered:

- `std::bitset<N>` (in `<bitset>`)
- `std::vector<bool>`
- `boost::dynamic_bitset<>`

If the maximum size of the bit vector is known at compile-time, which is often the case with AI models, then `std::bitset` is a good choice. If not, then `std::vector<bool>` or `boost::dynamic_bitset<>` are good choices for dynamic-sized bit vectors. Alternatively, you can build your own bit vectors, if there is a particular need to hand-code them or if you just want some fun.

Permutation Arrays

Most of the vectors in AI engines are not just random lists of numbers. Rather, they are (conceptually) an array of the probabilities of output words, where the position in the vector indicates which word. So, if we have our `logits` array, then `logits[0]` is the probability of "`the`" whereas `logits[1]` is the probability for "`cat`", and so on, up to about 50,000, which is a common vocabulary size for LLMs.

Problems arise if we want to sort our probabilities in the logit array, and we need this for our decoding top-k algorithm. We can't just sort the vector of probability numbers, because we'll lose track of which probability maps to which token number.

Permutation arrays to the rescue! A permutation array is an array that is the same size as some other array, but maps to the *indices* of the other array. A permutation array for our vocabulary has 50,000 integers, each of which is the index into other arrays.

The downside of permutation arrays is that they introduce inefficiency in both space and time. Space usage is increased by having two vectors. The time cost to access a vector element increases, too. Rather than just looking up the probability for the nth word in the logits array (i.e., "`prob=logits[n]`"), we have a two-step procedure:

> 1. Look up the index in the nth element of the permutation array (i.e., "`i=permut[n]`"),
>
> 2. Use that index to look up the probabilities in the main logits array (i.e., "`prob=logits[i]`").

So, it's bigger and slower. *Some rescue.*

However, permutations can be valuable if it allows us to do much less arithmetic overall, which is the case with "sparse" arrays where most elements are zero. This is why permutation arrays are used for LLM sparsity optimizations, but not in normal practice.

Sorting with a Permutation Array: The way to sort another array, indirectly via a permutation array, is shown in detail for the top-k decoding algorithm. The basic idea is:

1. Set up the identity permutation.

2. Sort using an indirect procedure: (a) compare elements in the main array indirectly accessed via the permutation array, (b) swap the indices in the permutation array (not changing the main array).

So, the original array doesn't actually get sorted with only the permutation array changing. If we want to print out the main array in a sorted list, we have to do so via the permutation array. The original main array is still unsorted if we access it directly.

Sparsity with Permutation Arrays. Sparsity is an optimization where most of the weights have been "pruned" to zero, and only a small percentage remain non-zero. This saves a lot of storage space for the model, and can also run much faster. The basic vector dot product kernel only needs to calculate with non-zero weights, so we want a way to avoid processing all of the many zero weights. Again, permutation arrays are the solution!

Sparse vectors (or matrices or tensors) can be stored as parallel arrays of:

- Non-zero weights only
- Permuted integer index of that non-zero weight in the original vector

These two arrays are much shorter than the original vectors if there is high sparsity. If sparsity is 90%, then 10% of numbers are non-zero, and the permutation approach uses two arrays, so it is 20% of the original size. The cost of doing a sparse dot product has reduced from the full length of the original vectors, down to the average sparsity factor (i.e., how many non-zero values). In other words, the number of multiplication computations goes down to 10% FLOPs, although there's the extra permutation calculation, so it's might seem like it's 20%, but we can often hardware-accelerate the permutation array step in CPU or GPU architectures.

Hence, sparse vector dot products are fast. Calculation of the vector dot product for AI inference need only multiply using the much smaller number of non-zero weights.

Can we vectorize permuted arrays for hardware acceleration? Short answer: yes. Permutations can be vectorized with hardware acceleration in both CPU and GPU versions. The C++ AVX "gather" (load) and "scatter" (store) intrinsics work for x86 CPUs. Different GPU primitives are available for permuted arrays.

Sparsity doesn't really work without permutations. A raw full-size vector containing lots of zeros doesn't vectorize well, because it still sends all of those zeros for processing. A permuted index of sparse values works much better because it only considers non-zero values.

Vector Hashing

Vector hashing is needed in various parts of an AI engine as a speedup. There are various AI research papers on using hashing for various computations involving vectors and tensors of higher dimensions. Implementations of such algorithms are available in open source and commercial "vector database" products that you can use. Some of the applications for LLMs include inference caching, embeddings, and RAG architectures.

But how do you hash a full-length vector? Or a matrix? It's a complicated theoretical area. One of the main techniques is Locality-Sensitive Hashing (LSH), which is hashing to find vectors that are "close" in *n*-dimensional space.

One of the interesting research areas for vector hashing is total precomputation of vector dot products. Think about precomputation of vector dot products in AI inference. If you could hash the two vectors, then you could replace the main bottleneck in AI inference with two hash lookups. Is there a way to efficiently convert a vector dot product operation on two vectors into a hash lookup, thereby avoiding all those multiplications? What about speedup of matrix multiplication by hashing?

Remember that you can pre-compute anything about the weights before inference, because they don't change during inference. Hence, one of the vectors could potentially be pre-hashed offline. Maybe you could even use some type of "perfect hashing" for those vector hashes, if you've got a big enough compute budget. But you can't pre-hash both of the vectors or pre-compute the dot product, because the other vectors are dynamically calculated along the way, dependent on inputs. This is being examined by advanced researchers, and is still a work in progress.

Perfect Hashing

Perfect hashing aims to achieve collision-free O(1) hashing at runtime, by investing a lot of offline compute budget to find an optimal hash function for a set of static data. There are many possible hash functions, and some are better than others. Perfect hashing tries to find an optimal hash function within the search space for possible methods. Mostly, it's by trial-and-error. Searching for a perfect hash function typically uses a brute-force and computationally expensive method for simply trying multiple hash functions and testing them for collisions.

Perfect hashing only works in the situation where all of the possible keys are known in advance (i.e., static data). Interestingly, this is exactly the situation with AI model vocabularies!

Hence, the idea of perfect hashing can be used to improve the performance of a hash table in the tokenizer. The general concept is that different hash tables are tested with various different meta-parameters (e.g., the hash table size, and multipliers in the hashing function). So, you can test various different hash functions against the 50,000 known tokens in the vocabulary, until you find a "perfect" one where there are no clashes. Amusingly, this longstanding algorithmic method sounds exactly like doing Neural Architecture Search (NAS) to find the best AI model hyper-parameters.

Bloom Filters

Bloom filters are a probabilistic data structure based on a combination of hashing and bit vectors. Multiple hash functions are computed for each key, and this is used to set bitflags, as described in more detail below. Bloom filters are mentioned in various research papers on AI, but are not yet used much in industrial AI applications. Perhaps they should be, as they seem very efficient.

Like hashing, Bloom filters have been used as a data structure to speed up neural network inference. However, much of the research literature about Bloom filters is about a different topic: Weightless Neural Networks (WNNs). WNNs have a different type of neuron based on binary bits, rather than matrix multiplications. These bitflag neurons can be approximated using Bloom filters. As such, that part of the research is less relevant to optimization of Transformer inference, and has not been examined in detail below.

How do Bloom Filters work? Given a key, multiple hash functions are calculated for that key, and a binary flag is set in a bitflag table for each of those hash offsets. In this way, an input key maps to a pattern of multiple bits.

The Bloom filter lookup for a key value works as follows: To test whether a key is found, the multiple hash functions are computed, and then the bitflag table is analyzed to see if all those bits are set. If any of the bits are missing, the key is *not* in the Bloom filter. If all of the bits are found, the key is *probably* in the Bloom filter, but it may also be that other keys have coincidentally set all those bits (a "false positive"), so it is not 100% guaranteed to be present.

If a probabilistic speedup is good enough, then a Bloom filter is all you need. For a 100% accurate table lookup, adding a second different type of backup data structure needs to be queried to confirm. Hence, the Bloom filter is a fast test to see if a key is not in a set, but a slow test if the key is found. This makes it an example of "common case first", where fast computations skip more involved computations.

The computational complexity of Bloom filters is constant, but not as fast as hashing. A hash filter uses only a single hash function, so it has O(1) lookup. However, a Bloom filter uses multiple functions, k, with O(k) lookup complexity.

References

1. Thomas Dean, Mark A Ruzon, Mark Segal, Jonathon Shlens, Sudheendra Vijayanarasimhan, and Jay Yagnik. 2013. *Fast, accurate detection of 100,000 object classes on a single machine*. In Proc. CVPR. https://web.stanford.edu/class/cs231m/references/hashing-dpm.pdf
2. Braddock Gaskill, 18 Oct 2019, *The Bitwise Hashing Trick for Personalized Search*, https://arxiv.org/abs/1910.08646
3. Alexander Golynski, Alessio Orlandi, Rajeev Raman, S. Srinivasa Rao, 10 Aug 2011, *Optimal Indexes for Sparse Bit Vectors*, https://arxiv.org/abs/1108.2157
4. Seungmin Yu, Xiaodie Yi, Hayun Lee, Dongkun Shin, 30 Jul 2024, *Toward Efficient Permutation for Hierarchical N:M Sparsity on GPUs*, https://arxiv.org/abs/2407.20496
5. M. Mor, A. S. Fraenkel, 1982, *Permutation Generation on Vector Processors*, The Computer Journal, Volume 25, Issue 4, November 1982, Pages 423–428, https://doi.org/10.1093/comjnl/25.4.423 https://academic.oup.com/comjnl/article/25/4/423/366370
6. D Liang, M Hashimoto, H Awano, 2021, *Bloomca: A memory efficient reservoir computing hardware implementation using cellular automata and ensemble bloom filter*, 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), https://ieeexplore.ieee.org/abstract/document/9474047/
7. Wikipedia, *Bloom filter*, https://en.wikipedia.org/wiki/Bloom_filter
8. Sourin Chakrabarti, 18 Nov 2020 (v2), *Efficient image retrieval using multi neural hash codes and bloom filters*, https://arxiv.org/abs/2011.03234
9. Avi.im, 22 Dec 2024, *How bloom filters made SQLite 10x faster*, https://avi.im/blag/2024/sqlite-past-present-future/
10. Atsuki Sato, Yusuke Matsui, 6 Feb 2025. *Cascaded Learned Bloom Filter for Optimal Model-Filter Size Balance and Fast Rejection*, https://arxiv.org/abs/2502.03696

51. Lookup Tables & Precomputation

Precomputation with Lookup Tables

Look-up tables (LUTs) are a well-known simple data structure for optimizing code. They have been used to optimize neural networks in various ways. Some examples include:

- Precomputed activation functions
- Zero-multiplication networks
- Approximation of non-linear functions

Precalculation or precomputation is a code optimization where results are partially or fully calculated ahead of time. This method is similar to caching and computation reuse but refers to calculations being performed long before they are needed, often at program startup or compile-time, and stored in lookup tables. Like caching, this method trades extra space for time.

Vectorization of LUTs is possible with hardware acceleration primitives that support parallel memory accesses using integer indices. For example, the x86 CPU with AVX intrinsics has a set of "gather" instructions for doing indexed lookup that can be used to load from a LUT into the internal registers, and "scatter" instructions for storing the registers back to an indexed LUT.

Typical precalculations are those where the results are computed at program initialization or compile-time. The best methods generate the results at compile-time, and are simply loaded as data, such as numeric constants or pre-initialized data arrays. There are multiple ways to do this:

- Program startup initialization
- Lazy evaluation
- Binary data file
- Precompiled source code

One method for precomputation of larger amounts of data in an array or lookup table is to perform the initialization dynamically at program startup. A lookup table can be populated with the required results, before the main logic of the program begins. Or alternatively, the idea of "lazy evaluation" allows storing the precomputation into a lookup table only when the program first needs the data.

A faster alternative is to calculate all this data offline before program startup, and store the results in a binary data file. This data file can then be loaded into an array at program startup, without needing to perform any of the arithmetic computations. Whether this is beneficial depends on the cost of the computations versus the cost of file loading.

The logical extension of the precomputation method for a large number of numeric results is to write special C++ code that performs these calculations, but then outputs the results into a text file in the exact format of a C++ source code file (rather than a data file), that declares a global array name and the numeric values. This auto-created C++ code is then linked with your program.

Example: LUT Precomputation for sqrt

Let's say that you want to optimize a slow non-linear function like "`sqrtf`" (or "`expf`" or "`logf`"). These are good candidates for optimization because of their non-linearity.

The first point is that you'd better do a really good job, because there are actually hardware instructions for these common math functions, even in x86 architectures. So, you could easily optimize this into a table lookup, and find that your C++ code is still slower than the single CPU instruction that's called by the standard C++ library versions. Hence, investigate the C++ intrinsic functions for common math functions before you assume that you can do better than electrons zipping through silicon.

This example investigates precomputing "`sqrtf`" even though that may not be as fast as hardware-acceleration. However, the same ideas apply to precomputing more sophisticated derivative functions, such as Softmax and activation functions, which are not hardware-supported (or not yet, anyway). The same general ideas apply.

The basic method for table lookup optimization is:

- Declare a big array (the bigger the better).
- Run a loop sending every value to the real "`sqrtf`" function.
- Store each result in the big array.
- Now you have a precomputed table of all possible values.
- Later, use an array index lookup to compute the function fast.

How is than any faster? I mean, we've just called "`sqrtf`" a bazillion times with numbers that we probably won't ever need. Yes, there is extra cost, and we are running slower during program initialization. There are at least two ways to fix this:

1. Load the array values from a pre-built binary data file instead, or,

2. Precompile the array data into a C++ source code file.

However, this complaint underestimates just how many times the code may call these functions. Even with this startup cost, once that is all done and dusted, we have a big array of precomputed data that we can use to speed up the program execution, which is our main goal. And in a production environment, any extra startup cost is hopefully amortized over many executions.

Example: Precomputing sqrt of integer: For simplicity, we're going to first assume that we're computing a `float` square root of integers. The function we are precomputing is "`int-to-float`" type. This makes it easier, because the `int` can be used as an array index.

Here's my big array with about 65,000 entries:

```
#define AUSSIE_SQRT_PRECOMP_MAX (1u<<16)
float g_sqrt_precomp_table[AUSSIE_SQRT_PRECOMP_MAX];
```

Here's the unoptimized function "`int-to-float`" version of "`sqrtf`" that we are planning to precompute:

```
float aussie_sqrtf_basic_int(int x)
{
    return sqrtf((float)x);
}
```

Here's the initialization call to the precomputation routine, sending in the array, the size *N*, and the function pointer:

```
aussie_generic_precompute_int(
    g_sqrt_precomp_table,    // Big array
    AUSSIE_SQRT_PRECOMP_MAX,  // N
    aussie_sqrtf_basic_int    // Function pointer
);
```

And here's the code to run the big precomputation loop:

```
void aussie_generic_precompute_int(
  float arr[], unsigned int maxn, float (*fnptr)(int))
{
    for (unsigned int i = 0; i < maxn; i++) {
            arr[i] = fnptr(i);
    }
}
```

So, that's all there is to the startup initialization of the lookup table. Once this function returns, we now have a big array full of data. Here's what the new optimized "`sqrtf`" looks like:

```
float aussie_table_lookup_sqrt(int i)
{
    return g_sqrt_precomp_table[i];
}
```

And we can either make that function "`inline`" or use a C++ preprocessor macro:

```
#define AUSSIE_TABLE_LOOKUP_SQRT_BASIC(i) \
     ( g_sqrt_precomp_table[(i)] )
```

So, here are a few provisos about this code:

1. Might be slower than `sqrt` in hardware (needs benchmarking).

2. Unsafe array accesses (e.g., crashes on negatives or larger numbers).

3. `unsigned int` types might overflow and spin infinitely for precomputing tables of size "`1<<32`" (change to `unsigned long`).

4. The memory size of the precomputed table for `1<<16` is already about 262k (65k times 4 bytes).

Float-to-Float Precomputation

Using a precomputed table lookup for a float-to-float function is more complicated than integers. However, this is also the main approximation needed for non-linear functions, or even the basic math library functions like `sqrtf` or `expf` or `logf`.

Why is it tricky? The reason that `float` inputs are more difficult is that we need to convert a `float` into an array index in order to look it up. For example, we could try type casts:

```
    int offset = (int)f;
```

But that limits us to only precalculating values for 1.0, 2.0, 3.0, etc. Our approximation works poorly on any fractions, and we also haven't limited the array index to a fixed finite range, so it won't work for any negative values or very large positive values. And the type cast of a `float` is also slow!

Scaled Multiple: Another idea is that we could scale it upwards to get more decimals:

```
    int offset = (int) (f * 1000.0f);
```

This approach at least gives us 3 decimal places: e.g., 1.234 or 23.456, or similar. We will still have to check for negatives and large values to bound it. But again, this is even slower!

Bitwise Floating-Point Truncations: The above truncation via a floating-point scaled multiple is not very fast. Twiddling the bits is much faster. For example, when we have a standard 32-bit `float` type, it has 1 sign bit, 8 exponent bits, and 23 mantissa bits. This is from left-to-right, with the sign bit as the most significant bit, and the low-end mantissa bits are the least significant bits. Remember that this is like Scientific notation:

- Number = Mantissa x 2 ^ Exponent

Also, the sign bit makes it all negative, if set. Note that exponent in 8-bits encodes the numbers -128 to +127, so that ranges from very small 2^-128 near-zero values, to very huge 2^127 sized values.

If the mantissa was in decimal, and it was "1234567" and the exponent was "17" then we'd have:

- Number = 1.234567 x 10^17

If the mantissa was 23 bits, it's actually binary digits, with about 3 binary digits per decimal digit, so a 23-bit mantissa is about 7 or 8 decimal digits. Note that the mantissa is actually 24 bits, not 23, because there's an extra "implicit one" mantissa bit, not that it changes the above calculation, but you needed to know that for C++ trivia night.

So, if we think about it for a year or two, it becomes obvious that the rightmost bits of the mantissa are simply the rightmost digits in "1.234567", and if we truncate some of the rightmost bits, it's like truncating a very small fraction (e.g., "1.234567" becomes "1.2345" or whatever).

Hence, a first idea is just to cut off 2 of the 4 bytes of a 32-bit `float`. This leaves us with 1 sign bit, 8 exponent bits, and 7 mantissa bits (plus 1 implied bit makes 8 mantissa bits). In decimal, the 8-bit mantissa now encodes only about 2 or 3 decimal digits, as if we've truncated "1.234567" to "1.23".

Incidentally, congratulations, you've created "*bloat16*" type, which is what Google did with TPUs, making a 2-byte `float` format with 1 sign bit, 8 exponent bits, and 7 stored mantissa bits. So, now you can get into your blue telephone booth, time travel back a decade, file a patent, and retire on your royalties. If you're ever a contestant on *Wheel of Fortune* you probably won't need to know that the "b" in "bfloat16" stands for "brain float" and that is such a great name. But I digress.

Anyhow, this idea actually works for precomputation. A 2-byte integer in `bloat16` format is easy to extract from a 4-byte FP32 float (i.e., the uppermost two bytes). The trick for bitwise processing is to convert the `float` to `unsigned int`, because the bitwise shift operators don't work on `float` (it's planned for C++37, as I heard at my fungus collector's club trivia night).

```
float f32 = 3.14f;
unsigned u32 = *(unsigned int*)&f32;
```

Extracting the top-most 2 bytes (16 bits) is simply a right bitshift:

```
unsigned ubf16 = ( u32 >> 16 );
```

Note that here's a good reason that we had to use "`unsigned`" integer type. The right bitshift operator (>>) has undefined behavior on negatives, so "`int`" type wouldn't work predictably (or portably) if the floating-point sign bit was set.

The result is a 16-bit `unsigned` integer to use as the array index. Hence, there are only `1<<16=65,536` entries in our precomputation table. Assuming we store results as 4-byte `float` values, this makes the precomputation array's memory size about 262kb. What's more, it works for negative `float` numbers, because the sign bit is still part of that shemozzle, and we also don't need to check any minimum or maximum bounds, because it works for all 32-bit float numbers.

Precomputing with 24-Bit Lookup Tables: Interestingly, none of the above code is especially tied to 16-bit sizes. The `bfloat16` version truncates 32-bit float to 16-bit by truncating the rightmost 16 mantissa bits. But we can actually choose to keep however many mantissa bits we like. The trade-off is that more mantissa bits increase accuracy, but at the cost of needing a much bigger precomputation array (doubling the storage size for each extra bit).

Let's try only cutting the rightmost 8 mantissa bits, leaving us with 24 stored bits total (i.e., 1 sign bit, 8 exponent bits, and 15 stored mantissa bits). The mantissa bits reduce from 23 to 15 (plus one implied bit makes 16), so this now stores about 5 decimal digits (e.g., "1.2345"), giving quite good precision on our results. When I tested the 16-bit version, it had some reasonably large errors of almost 0.1 in computing `sqrt`, whereas this 24-bit version has much lower errors, as expected.

Code changes are minor. The bitshift operations simply change from 16 bits to 8 bits (i.e., `32-24=8` bits). This is the precomputation loop for 24 bits:

```
void aussie_generic_precompute_24bit_float(
    float farr[], unsigned int maxn,
    float (*fnptr)(float))
{
    for (unsigned int u = 0; u < maxn; u++) {
        unsigned int unum = (u << 8u); // 32-24=8 bits
        float f = *(float*)&unum;
        farr[u] = fnptr(f);
    }
}
```

And this is the call to the precomputation function in the startup phase:

```
aussie_generic_precompute_24bit_float(
    g_sqrt_float_24bit_precomp_table, // Bigger array
    (int)AUSSIE_SQRT_24bit_MAX,     // 1 << 24
    aussie_sqrtf_basic_float        // Function pointer
);
```

The table lookup routine also similarly shifts 8 bits, rather than 16, but is otherwise unchanged:

```
float aussie_table_lookup_sqrt_24bit_float(float f)
{
    unsigned u = *(unsigned int*)&f;
    u >>= 8;  // 32-24=8 bits
    return g_sqrt_float_24bit_precomp_table[u];
}
```

Note that this only works if we are sure that both "`float`" and "`unsigned int`" are 32-bits, so we should check that during startup with some assertions via `static_assert`. If we are sure of that fact, then not only will it work, but we don't also need to check the array bounds. It won't try a negative array index, and won't overflow no matter what bit pattern we send it in as a `float`.

But there is one problem. If we send the fast table lookup version the special `float` value of `NaN` ("not a number"), then the table lookup routine will actually return a valid numeric answer, which probably isn't what we want. Maybe we need to add a check for that special case, and this needs more testing.

The new size of the precomputation array is `2^24=16,777,216`, so we have about 16.7 million results If our results are 32-bit `float` values, our `bloat16` precomputed array above requires about 262kb, and the new size with 24-bits is a lookup table (array) of about 67 megabytes. It wouldn't have worked on my old TRS-80 CoCo in 1986, but it'll work nowadays.

Precalculating C++ Source Files

One way to improve on the precomputation of a big array is to skip it entirely during startup by writing a lot of code. It's like using an AI coding copilot, only it's not really. I mean, come on, the day an AI writes better code than me is the day that I retire to the hologram beach with my robot dog companions.

The idea here is to write a program to generate a C++ source file that contains the global precomputed lookup table. Yes, it's a C++ program that creates part of a C++ program, which is almost like your AI has become self-aware, only one step away from *Skynet*. Well, maybe not, it's just a dumb C++ program written by a dumb human creating some dumb data.

Anyway, this auto-generated C++ code can be compiled and linked into your C++ program, and used like a global array of data in other parts of the program. Zero calculations are required at runtime, and the data can be read-only.

The benefit is that this auto-generated code method does not even require the time cost of startup initialization for any precomputations. There's not even the cost from data file loading. Instead, the data is auto-loaded by the linker-loader during executable file instantiation (i.e., when the user starts the app). The only downsides for the user are that the size of the executable program increases, which means more disk space usage, and that application program startup may take longer and it will use more memory (regardless of whether it ever needs this precomputed data). Also, various offline tasks take longer for the software developers, such as compilation and linking for testing, which is why we bill per hour.

I tried this out for precalculating GELU with a 24-bit table. The C++ source file was size 514k for 24-bit precomputation table of size `1<<24`. This is what the auto-generated source code should look like:

```
// Precomputed table source code: GELU,
// "gelu_precomp_24bits.cpp"
float g_gelu_table_precompute_24bits[] = {
0f,
1.793662034335765850782373866611092648039e-43f,
3.587324068671531701564747733222185296077e-43f,
5.380986103007297552347121599833277944116e-43f,
7.174648137343063403129495466444370592155e-43f,
...
...
};
```

Here's the code to generate the code to generate the code to generate the code:

```
void aussie_generic_setup_table_FP32_24bits_PRINT_SOURCE(
     char* nickname,
     char* outfname,
     float (*fnptr)(float),  // e.g., GELU
     int maxn,  // e.g., 1<<24
     float arrout[]  // array to store (optional)
 )
 {
     // Print C++ of 24-bits GELU precomputed table
     if (!fnptr) {
         aussie_assert(fnptr);
         return;
     }
     // Generate C++ source code so we can pre-compile
     // the precomputed GELU table (24-bits)
     // There are 2^24 = 16.7 million numbers...
     FILE* fp = stdout;
     bool writingfile = false;
     bool add_commented_number = true;
     if (outfname && *outfname) {
         fp = fopen(outfname, "w");
         if (!fp) {
             aussie_assert(fp);  // file write failed
             return;  // fail
         }
         writingfile = true;
         add_commented_number = false;  // No extra comments
     }
     unsigned int u = 0;
     fprintf(fp, "// Precomputed source code: %s, \"%s\"\n",
          nickname, outfname);
     fprintf(fp, "float g_gelu_table_pre_24bits[] = { \n");
     char numbuf[5000] = "";
     for (; u < maxn /*1<<24*/ ; u++) {  // For 2^24=~16.7M
             unsigned int uval = u << 8; // zeros in lsb
             float f = AUSSIE_UINT_TO_FLOAT(uval);
             float g = fnptr(f);  // Call GELU or whatever
             if (arrout) arrout[u] = g; // Store precomp data

             // Format: %g means the smaller of %e or %f
             // ... %e is exponent format (scientific-like)
             char* buf = numbuf;
             // Format %g (Number) and suffix "f" (float)
             sprintf(buf, "%40.40gf", g);
             if (strchr(buf, 'n')) {
                     // Nan or "-nan" ... use dummy value
                     strcpy(buf, "0.0 /*nan*/");
             }
             // Remove prefix padding spaces...
             while (buf[0] == ' ') buf++;
```

```
            // Remove suffix zeros ...
            int len = (int)strlen(buf);
            if (buf[len - 1] == 'f') len--; // skip suffix f
            if (buf[len - 1] == '0') {
                while (len > 5) {
                    if (buf[len - 1] == '0'
                        && isdigit(buf[len - 2])) {
                        if (buf[len] == 'f') {
                            // remove it, but leave 'f'...
                            buf[len - 1] = 'f';
                            buf[len] = 0;
                        }
                        else {
                            buf[len - 1] = 0; // remove it
                            buf[len] = 0;
                        }
                        len--;
                    }
                    else break;
                }
            }

            if (add_commented_number) {
                    fprintf(fp, "%s // (%40.40f) [%u] \n",
                        buf, f, u);
            }
            else {  // No comments...
                    fprintf(fp, "%s,\n", buf);
            }

            // Progress update
            if (u % 100000 == 0 && u != 0) {
                if (writingfile) // Progress to stdout
                    fprintf(stdout, "%u -- %s\n", u, buf);
                // Comment occasionally
                fprintf(fp, "// U= [%u]\n", u);
            }
    }
    fprintf(fp, "}; \n");  // Close initializer...
    if (fp && fp != stdout) fclose(fp);
}
```

Conclusions on Source Code Generation: Does it work? Yes and no. It builds the output file quite quickly, zipping through `1<<24` computations and writing to disk. But I can't get this 24-bit version with its 500k CPP source file to actually compile in the Microsoft Visual Studio IDE.

Maybe it works on Windows command-line or Linux GCC, but I haven't tried.

Anyway, this self-generating code idea is certainly quite workable for table lookups of approximations for FP16 numbers (16-bit half-precision floating-point), because the lookup table needs to "only" contain 2^16=65,536 numbers. This is about a 200k C++ source file in plain text, and creates linked data of about 65k times 4 bytes equals about 256k space usage. This would use half that space if you also store the computation as 16-bit numbers rather than 32-bit floats or integers.

References

1. Nils Graef, 12 Mar 2024 (v3), *Transformer tricks: Precomputing the first layer*, https://arxiv.org/abs/2402.13388 Code: https://github.com/OpenMachine-ai/transformer-tricks (Because the first layer only depends on the embeddings, it can be precomputed.)
2. SZ Lin, YC Chen, YH Chang, TW Kuo, HP Li, 2024, *LUTIN: Efficient Neural Network Inference with Table Lookup*, ISLPED '24, August 5-7, 2024, Newport Beach, CA, USA, https://dl.acm.org/doi/pdf/10.1145/3665314.3670804
3. S Fanning, *Fixed Point Multiplication-Free Implementation of Deep Neural Networks for Embedded Systems*, Masters Thesis, School of Electrical and Electronic Engineering, University College Dublin 2018, https://seanfanning.eu/posts/projects/low-bitwidth-neural-networks/Thesis_SeanFanning_13360951.pdf
4. Mohammad Samragh Razlighi; Mohsen Imani; Farinaz Koushanfar; Tajana Rosing *LookNN: Neural network with no multiplication*, Design, Automation & Test in Europe Conference & Exhibition (DATE), 27-31 March 2017, https://ieeexplore.ieee.org/document/7927280 (Lookup-table based multiplication.)
5. Covell M, Marwood D, Baluja S, Johnston N., *Table-based neural units: Fully quantizing networks for multiply-free inference*, 2019, arXiv preprint arXiv:1906.04798, http://arxiv.org/abs/1906.04798
6. Joonsang Yu, Junki Park, Seongmin Park, Minsoo Kim, Sihwa Lee, Dong Hyun Lee, Jungwook Choi, Dec 2021, *NN-LUT: Neural Approximation of Non-Linear Operations for Efficient Transformer Inference*, https://arxiv.org/pdf/2112.02191
7. Neelesh Gupta, Narayanan Kannan, Pengmiao Zhang, Viktor Prasanna, 8 Apr 2024, *TabConv: Low-Computation CNN Inference via Table Lookups*, https://arxiv.org/abs/2404.05872
8. Darshan C. Ganji, Saad Ashfaq, Ehsan Saboori, Sudhakar Sah, Saptarshi Mitra, Mohammad Hossein Askari Hemmat, Alexander Hoffman, Ahmed Hassanien, Mathieu Léonardon, 18 Apr 2023, *DeepGEMM: Accelerated Ultra Low-Precision Inference on CPU Architectures using Lookup Tables*, https://arxiv.org/abs/2304.09049

9. Grigor Gatchev, Valentin Mollov, 4 Apr 2021, *Faster Convolution Inference Through Using Pre-Calculated Lookup Tables*, https://arxiv.org/abs/2104.01681
10. Han Guo, William Brandon, Radostin Cholakov, Jonathan Ragan-Kelley, Eric P. Xing, Yoon Kim, 15 Jul 2024, *Fast Matrix Multiplications for Lookup Table-Quantized LLMs*, https://arxiv.org/abs/2407.10960
11. Davis Blalock, John Guttag, 21 Jun 2021, *Multiplying Matrices Without Multiplying*, https://arxiv.org/abs/2106.10860
12. Gunho Park, Hyeokjun Kwon, Jiwoo Kim, Jeongin Bae, Baeseong Park, Dongsoo Lee, Youngjoo Lee, 10 Mar 2025, *FIGLUT: An Energy-Efficient Accelerator Design for FP-INT GEMM Using Look-Up Tables*, https://arxiv.org/abs/2503.06862

Appendix A: Long List of Low Latency Techniques

This is a compilation of coding efficiency and low latency C++ programming techniques from various books and articles:

- C++ Low Latency, David Spuler, March 2025.
- CUDA C++ Optimization, David Spuler, June 2024.
- Generative AI in C++, David Spuler, March 2024.
- 500+ LLM Inference Optimization Techniques (blog article)

Low Latency C++ General Software Approaches:

1. Cache warming
2. Core pinning ("affinity")
3. False sharing (avoiding)
4. Branch prediction optimizations
5. Hotpath optimizations
6. Slowpath removal
7. Kernel bypass
8. Lock contention (reducing)
9. Lock-free programming (with atomics and memory ordering issues)
10. Thread pools
11. SIMD CPU instructions
12. Inline assembly language ("asm" statements)
13. Intrinsic functions (often closely mapping to machine code instructions)
14. In-memory logging
15. Cache locality (for L1/L2/L3 memory caches and instruction caches)
16. Specialized data structures
17. Thread-Local Storage (TLS) ("`thread_local`" type in C++11)
18. Shared memory (e.g., `shmctl` which is the main "shared memory control" function, `shmget`, `shm_open`, `ftruncate`)
19. Memory mapped files/devices (e.g., `mmap`, `munmap`)
20. Asynchronous programming (`std::async`)

Concurrency-Friendly Data Structures:

21. Read-only data structures
22. Reader-friendly data structures (e.g., many readers, one writer)
23. Copy-on-write data structures (for readers)
24. Versioned data structures (for readers)
25. Partition data across threads (vertically: columns)
26. Shard data across threads (horizontally: rows)
27. Read-Copy-Update(RCU)—mostly the same as copy-on-write.
28. NUMA-aware data structures—reduce cross-node communications
29. Transactional memory (synchronization efficiency, reduces contention) — use atomic or isolated transactions (an emerging technology)

Hotpath Optimizations:

30. Optimize all steps in the hotpath (e.g., data ingestion, decision, trade execution, logging, risk management)
31. Profile the hotpath specifically (e.g., a test mode that always runs the hotpath)
32. Examine assembly code of the hotpath
33. Avoid any memory allocation calls on hotpath (e.g., memory pools, use preallocation)
34. Avoid free/deallocation of memory on hotpath
35. Use preallocated memory on hotpath
36. Review data de-serialization and serialization costs
37. Use in-memory databases for any significant amounts of incoming data
38. Keep the client network connection warm (method depends on the API)
39. Re-use objects to avoid constructor/destructor calls on hotpath

General Tuning Advice:

40. Avoid micro-optimization
41. Avoid optimizing error handling code (it's a slowpath)
42. Loop optimizations (see below)
43. Avoid nested loops
44. Tune inner loop for nested loops
45. Avoid excessive function wrapper overhead

Performance Profiling Tools:

46. `gprof`
47. `perf`
48. `prof` (older)
49. `pixie` (older)

Lock Contention Reduction:

50. Late lock acquisition
51. Early lock release
52. Short critical section of code
53. Generally reduce total numbers of locks used
54. Locking fine-grain vs coarse-grain
55. Use fine-grain locks for contested resources
56. Use a hybrid fine-grain/coarse-grain lock strategy
57. Release locks before significant computation
58. Copy data to temporary variables to unlock before computation
59. Release locks before blocking for I/O
60. Release locks before blocking for system calls
61. Release locks before blocking for networking
62. Tolerate lockless output overlaps
63. `std::shared_mutex` and `std::shared_lock` — for multiple reads, one writer.
64. Double lock check method (check first without a lock)
65. Use `std::promise` and `std::future` not shared memory.
66. Thread-specific queues and "work stealing" design pattern
67. Use a lock-free queue data structure
68. `thread_local` keyword (C++11)
69. `std::lock_guard` (C++11)
70. `std::lock_guard` early release by scope control
71. `std::unique_lock` (C++11) (this allows more granular control than `std::lock_guard`)
72. `std::scoped_lock` (C++17)
73. Locking with timeouts (try locks)
74. Avoid spinlock busy waiting
75. Exponential backoff to avoid spinlock costs
    See also "lock-free programming" and "concurrency-friendly structures"

Thread/lock overhead reduction (generally):

76. Reduce thread launch overhead
77. Reduce thread destruction overhead
78. Reduce lock acquisition/release overhead
79. Reduce lock contention overhead
80. `std::make_shared()` or `std::allocate_shared()` standard functions do only one allocation (combined shared pointer and control block), whereas `shared_ptr<type>` does two allocations (both the shared pointer and the control block are separate).
81. Weak pointer references (`std::weak_ptr`) can delay the deallocating a `shared_ptr` and its object even after the main reference count is zero.

System code optimizations (general ideas):

82. Avoid system calls to reduce context switches (in Linux)
83. Use C++ "intrinsics" functions (highly optimized assembly-level code)

Linux socket programming:

84. Non-blocking sockets versus using `select()` with a timeout—allows thread to do "other" useful work rather than just wait.
85. `poll()` or `epoll()` system call rather than waiting

Context Switching Reduction:

86. Thread counts (not too many threads)
87. Thread specialization
88. Thread specialization (producer-consumer thread model)
89. Use custom thread pools with only preallocated memory block pools.
90. spinlocks avoid context switches (good if spins for only a short time)
91. Avoid context switch cost by thread doing "other" work, not just blocking.

Cache Locality Optimizations:

92. Tiling/blocking algorithms
93. Tiling/blocking matrix multiplication (MatMul/GEMM)
94. Smaller data type sizes for increased locality
95. Choose a CPU with a larger L1 "cache line size" (64-256 bytes common)
96. `std::hardware_destructive_interference_size`, `std::hardware_constructive_interference_size` (C++17)
97. `std::initializer_list` (C++11) can be used as a standardized lightweight container with contiguous elements
    See also "cache warming (prefetch)" optimizations
    See also "false sharing (avoid)" optimizations

Instruction Cache Locality Optimizations:

98. Prefer shorter blocks of code in the hotpath
99. Consider not inlining function calls (for instruction cache locality)
    See also "branch prediction optimizations"

Branch Prediction Optimizations (General):

100. Branch elimination
101. Branch compiler hints
102. Branch prediction heuristics
103. Branch profiling (two-phase)
104. Branchless programming
105. Tools—measure branch prediction data (e.g., `perf`)

Branch Reductions Techniques:

106. Algorithm-level changes to reduce branches
107. Keep loop bodies short (shorter branches)
108. Reduce far branching (e.g., function calls)
109. Reduce overall use of function calls (see function call optimizations)
110. Reduce use of `if` statements
111. Reduce use of loops
112. Reduce use of `break` statements (in loops, not `switch`!)
113. Reduce use of `continue` statements
114. Reduce use of `switch` statements
115. Reduce short-circuiting in `&&`/`||` operators
116. Reduce short-circuiting of `?:` ternary operator
117. Avoid `virtual` function calls (hidden dynamic branches)
118. Avoid pointer-to-functions (hidden dynamic branches; blocks inlining)
119. Avoid function objects/functors (hidden dynamic branches)
120. Avoid lambda functions passed as arguments (depends on how well the optimizer can handle them)
121. Reduce long `if-else-if` sequences
122. Reduce nested `if-else` sequences
123. Avoid branches depending on anything unpredictable
124. Avoid branches depending on user inputs
125. Avoid branches depending on random numbers
126. Avoid branches depending on system clocks
127. Sort array data for efficient branch prediction, if scanning through the linear array comparing the data (e.g., before testing for error range)
    See also "compile-time optimizations" (remove branches at compile-time)
    See also "loop optimizations" (reduce loop iterations, e.g., loop unrolling)

Branch Prediction Heuristics:

128. Common case code in `if` block
129. Uncommon case code in `else` block
130. Error handling code in `else` block (uncommon code)
131. Avoid zero-iteration loops (never entered)
132. Avoid single-iteration loops (never loop back)

Branch Prediction Compiler Hints:

133. `[[likely]]` and `[[unlikely]]` path attributes (C++20)
134. `likely()` and `unlikely()` expressions (C++20)
135. `__builtin_expect` (GCC)
136. `LIKELY` and `UNLIKELY` macros via `__builtin_expect` (pre-C++20)
137. `[[noreturn]]` (C++11)
138. `[[assume(expression)]]` attribute (C++23)

139. hot (GCC function attribute)
140. GCC __builtin_unreachable
141. std::unreachable—helps branch prediction (C++23)
142. [[fallthrough]] — more for safety than speed (C++17)
143. -fdelayed-branch compiler flag
144. -fguess-branch-probability compiler flag
145. -fif-conversion and -fif-conversion2 compiler flags
146. Use "likely" and "unlikely" in custom assertion macros
147. Use "likely" and "unlikely" in error handling code macros

Branch Profiling:

148. -fprofile-arcs (GCC option)
149. -fprofile-generate (GCC command-line argument)
150. -fprofile-use (GCC command-line argument)
151. Branch profiling with 100% hotpath (test modes)

Branchless Programming Techniques:

152. Ternary operator preferred over if statements (if CMOV instruction)
153. Boolean variables as 0 or 1 in arithmetic
154. Logical operators (&&/||) as 0 or 1 in arithmetic
155. Bitwise operators (&/|) replace logical operators (&&/||)
156. Sign bit extension bit masks
157. Lookup tables for branchless programming
158. XOR trick to swap two integer variables without a temporary variable

Slowpath Removal:

159. Optimize error checking pathways
160. Remove error checking tests
161. Defer error checking tests to later
162. Combine error checking tests together (and do it later)
163. Avoid adding error checks deeper in the call hierarchy
164. Never-failing functions (cannot return an error)
165. Don't use memory allocation (avoids memory allocation failure)

Cache Warming Methods:

166. Prefetch memory primitives
167. __builtin_prefetch (GCC)
168. _mm_prefetch (GCC)
169. volatile on temporary variables
170. Dry-run execution mode
171. Branchless dry-run execution with arr[2] declarations
172. Use read-only cache warming pathways (avoids cache invalidation for other threads)
173. Use deep cache warming all the way down into the NIC

174. Optimize cache warming by fewer data reads (relies on cache line sizes)
175. Reduce cache warming code to the maximum size of the low-level memory cache (this avoids redundant warming when cache is already full).

False Sharing (Avoiding):

176. Using `alignas(64)` or 128 or 256 to avoid false sharing (C++11)
177. Use `alignas` on all shared memory or atomics (C++11)
178. Tools to automatically detect false sharing (DRD fails?)

Parallelism (General Categories):

179. Multithreading
180. Multiprocess
181. Vectorization
182. Pipelining
183. Parallel execution modes (C++17)
184. Coroutines (C++20)

Advanced C++ Concurrency Data Structures:

185. Read-only ("immutable") data structures
186. Lock-free algorithms and data structures
187. Linear search can be efficient for small sizes because of cache prefetching (e.g., rather than binary search; also doesn't need sorting maintained)

SIMD Instructions:

188. AVX (x86 CPUs)
189. ARM Neon
190. `std::simd` (experimental/C++26)
191. `<immintrin.h>`

Linux O/S Optimizations:

192. Process priority upgrades ("`nice`" command or system call)
193. Disable unimportant processes
194. Overclocking CPU
195. Overclocking GPU
196. Disable Security Enhanced (SE) Linux
197. Disable accounting mode in Linux (should be off anyway)

Linux Kernel Optimizations:

198. Scheduling algorithm kernel modifications
199. Tweak TCP/UDP network buffer settings (Linux kernel)
200. Turn off file "last access date" storage ("`noatime`" in `/etc/fstab`)

System Hardware Optimizations (Categories):

201. Processor hardware (CPU)
202. Network optimizations
203. Disk optimizations
204. RAM Memory optimizations

Processor Hardware Major Categories of Optimizations:

205. CPU
206. GPU
207. NPU
208. FPGA
209. ASIC

Networking Hardware Optimizations (Categories):

210. NIC
211. Switches
212. Load balancer devices
213. Size of the packet buffer of a switch (optimizing for)

Networking Transmission/Protocol Optimizations (Categories):

214. Physical proximity
215. Co-Lo
216. TCP
217. UDP (faster than TCP but unreliable)
218. Optical networking (optical fiber cables)
219. Microwave network transmission
220. Packet fragment manipulations (e.g., out-of-order)
221. Reduce packet fragment collation overhead
222. Reduce packet consistency checking (error safety overhead)

Networking Software Optimizations:

223. TcpDirect/Onload
224. SolarFlare/OpenOnload (kernel bypass)
225. Exablaze (NIC with kernel bypass support)
226. DMA
227. PCIe bus
228. Compress data sizes for your network transmissions
229. Sticky sessions (avoids sending user-specific caches between servers)
230. Shared storage rather than server-to-server networking (e.g., NAS/SAN)
231. Use custom wrappers for TCP and UDP network processing

GPU & Distributed Networking Optimizations:

232. RDMA
233. nvlink
234. Infiniband
235. RoCE
236. GPUDirect
237. PXN

Deployment Optimizations (Website backends):

238. DNS optimizations
239. Round-Robin DNS (RRDNS)
240. SSL time optimizations
241. etags (website server speedup)
242. Multiple identical servers architecture
243. Use subdomains for static files
244. CDN for static files
245. Compression modes enabled
246. Static files compressed
247. Minify static files (CSS, JavaScript)
248. Merge multiple small files together
249. Use smaller image files (low precision)
250. Merge multiple small icon images into one image file
251. Cache duration settings
252. Database optimizations (various, e.g., MySQL/MariaDB/MongoDB)
253. Database indexes
254. Application server optimizations (e.g., Tomcat)

Apache/Nginx Subprocess Optimizations:

255. Use FCGI not classic CGI integrations
256. Flush `stdout` of subprocesses (partial output earlier to Apache or Nginx)
257. Close `stdout` of subprocesses before shutdown sequence (sends finishes earlier to Apache or Nginx)
258. Early tests for violations and invalidity (fails quickly)

Algorithm Enhancements:

259. Precomputation (lookup tables)
260. Precomputation to data file
261. Precomputation of source code
262. Incremental algorithms
263. Data structure augmentation
264. Parallelization
265. Vectorization
266. Caching

267. Lazy evaluation
268. Common case first
269. Simple case first
270. Approximate tests first
271. Bounding box approximate tests
272. Bounding sphere approximate tests
273. Avoiding `sqrt` by using arithmetic on squares
274. Integer arithmetic on squares: avoid floating-point by working on squares
275. Use variance not standard-deviation (arithmetic on squares)
276. Approximations
277. Compute budget algorithms
278. Probabilistic/stochastic algorithms
279. Skipping algorithms
280. Heuristic algorithms
281. Greedy algorithms

Memory Reduction Strategies:

282. Take care as memory reduction as methods can reduce speed (trade-offs)
283. Reduce allocated memory
284. Smaller data sizes
285. Pack data into smaller integer sizes
286. Pack data into bits
287. Pack data using bit-fields
288. Pack data into unions
289. Use `std::bitvector`
290. Use `std::vector<bool>` (a special bit-packed template instantiation)
291. Structure packing (also for class data members): reorder different-sized data members for better packing and fewer padding bytes
292. Structure packing: biggest data types first (heuristic)
293. Structure packing: MSVS `/d1reportSingleClassLayout` compiler option to report on it
294. `#pragma pack` directive reduces padding to reduce size, but may worsen structure access costs
295. Stack data reductions
296. Avoid deallocation of heap memory when in shutting-down mode

Heap Allocated Memory Reduction Strategies:

297. Fewer allocated memory blocks
298. Avoid frequent small allocations
299. Preallocation of dynamic memory
300. Memory fragmentation avoidance
301. Memory leak avoidance
302. Merge memory allocations together
303. Memory pools (fixed-size allocations, often a type of preallocation)

304. Memory pool with O(1) deletion and O(1) insertion via permutation array
305. Merge fixed-size allocated objects into a large array
306. Custom memory allocators (generalized)
307. Class-specific memory allocator
308. Custom global memory allocator
309. Late allocation (allocate memory as late as possible)
310. Early `free` memory (deallocate as early as possible)
311. Early `delete` memory (deallocate early)
312. Avoid `realloc` (slow, memory fragmentation)
313. Smart dynamic buffers (hybrid of allocated and non-allocated memory)
314. `std::aligned_alloc` - memory alignment improvement (C++17)
315. `std::aligned_union` (C++11)

Static Memory Size Reductions:

316. Avoid large global arrays and buffers
317. Avoid large static arrays and buffers
318. Avoid large static C++ data members
319. String literal memory reductions

Stack Memory Size Reductions:

320. Avoid large local arrays and buffers
321. Avoid large function non-reference parameter arrays and buffers
322. Use pass-by-reference on large function parameters
323. Use integer parameters as local variables
324. Consider stack versus memory allocation
325. Flattening/reducing function call hierarchy
326. Inline small functions (compiler can disappear them)
327. Use `#define` macros for small functions (versus inlining)

See also: function call hierarchy flattening
See also: recursion avoidance

Code Size Reduction Strategies:

328. Code size reductions
329. DLLs versus static libraries
330. Remove executable debug information
331. Avoid the compiler "`-g`" debug option
332. Avoid the compiler "`-p`" profiler option
333. Unix `strip` command
334. Avoid large `inline` functions (instruction cache locality)
335. Don't overuse "always inline" or "force inline"
336. Template overuse
337. Google "`bloaty`" tool

Standard Library Optimizations (STL Optimizations):

338. String processing efficiency (e.g., "+" for `std::string` can be slow)
339. `std::vector` of non-trivial class objects calls constructor/destructors
340. Control array size for `std::vector` using "`reserve()`"
341. Use `std::sort` rather than `qsort`
342. `bsearch` is not your friend
343. Consider hard-coded arrays versus `std::array` versus `std::vector`
344. Compare the first letters of strings before calling `strcmp`
345. Consider type casts to `int` versus `round()`, `ceil()`, `floor()`
346. Avoid `printf` or `fprintf` format string processing with `putchar`, `putc`, `fputc`, `puts`, `fputs`
347. Hand-code faster versions of the `abs` and `fabs/fabsf` primitives that don't handle `Inf/NaN` numbers (but benchmark it).
348. Change `strlen("literal")` to `char arr[]="literal"` and use `sizeof(arr)-1`
349. Don't use `strlen(s)` in a `for` loop condition
350. Consider your own `atoi/itoa` versions that don't handle obscure cases.
351. Avoid `sprintf` and `snprintf` (both are slow)
352. `sync_with_stdio(false)`
353. `std::stringstream` is slow (hand-code text field processing instead)

Data Structures:

354. Hashing (basic)
355. Perfect hashing
356. Bit vectors
357. Bit sets
358. Bloom filters (bit vectors + hashing)
359. Binary tree
360. Sorted arrays
361. Unsorted arrays
362. Stacks
363. Queues
364. Dequeues
365. Vector hashing
366. Permutation arrays
367. Locality-sensitive hashing (LSH)
368. Bit signatures (vector algorithm)
369. K-means clustering (vector algorithm)
370. Hyper-cube (vector algorithm)
371. Approximate nearest neighbor (ANN) (vector algorithm)

Variable Optimizations:

372. Prefer `int` types to `char` or `short` (usually)
373. Prefer `int` types to `unsigned int` (usually)
374. Prefer `int` types to `size_t` (`unsigned long`; consider `uint32_t`)
375. Avoid unnecessary initializations
376. Re-use objects to avoid initializations/destruction
377. Avoid temporary variables
378. Use reference variables instead of full temporary variables
379. Avoid creating temporary objects
380. Put commonly used data fields first in struct/class
381. Declare variables as close as possible to usage
382. `if` initializer syntax (C++17)
383. `switch` initializer syntax (C++17)
384. Avoid bit-fields (smaller but slower to access or set)
385. Use memory alignment primitives to avoid slow-downs
386. Put the most-used data member first (it has a zero offset)
387. Order data members most used to least (small offsets are faster, in theory)
388. Array initializer lists as local variables (re-initialized each call)
389. Structure of arrays (SoA) data layout is often more vectorizable than Array of Structures (AoS).

Arithmetic Optimizations:

390. Operator strength reduction
391. Reciprocal multiplication
392. Integer arithmetic
393. Use `float` not `double`

Expression Optimizations:

394. Expression transformations
395. `const`
396. `mutable` keyword — bypasses `const` (C++98) (speedy but unsafe)
397. Common subexpression elimination (CSE)
398. Constant folding
399. Template fold expressions (C++17) are concise but lots of computation
400. Expression templates—avoids explicit temporary variables, compiler optimizes it better.
401. Constant propagation
402. Redundant assignment removal
403. Strength reduction
404. Algebraic identities
405. Implicit type conversions (avoiding; type consistency)
406. `explicit` keyword (prevent implicit type conversions) (C++98)
407. Brace initialization syntax `{}` (avoids implicit narrowing conversions)

408. auto variable declarations avoid accidental temporaries and implicit type conversions.
409. Don't mix float/double types (including their constants)
410. Don't mix integer types
411. Prefer signed integers over unsigned types
412. Short-circuiting of sub-expressions (using &&/||/?:)
413. Register allocation optimizations
414. mprotect page system call —optimization to make memory writeable
415. <algorithm> simple algorithms: min, max, etc.
416. Range check faster with "(unsigned)i<MAX" not "i>=0&& i < MAX"

Memory Block Operations:

417. Prefer contiguous memory blocks (locality, efficient block operations, etc.)
418. Different class types can allow block copying: POD (Plain Old Data), trivial types, standard memory layout types (e.g., check in a template using std::is_trivial)
419. Copy arrays by wrapping them in a dummy struct
420. Copy arrays with memcpy
421. Compare arrays with memcmp (very dangerous: padding bytes, negative zero, NaNs)
422. Use memcpy not memmove if arguments won't overlap.
423. Linearize multi-dimensional arrays (contiguous memory blocks)

Operator Strength Reduction Optimizations:

424. Replace * with bitshifts
425. Replace * with addition
426. Replace x*2 with x+x
427. Replace % with bitwise-and (&)
428. Replace % with increment and test
429. Replace % with type casts (if byte sizes)

Bitwise Optimizations:

430. Intrinsic bitwise functions
431. CLZ (count leading zeros) bitwise intrinsics
432. CTZ (count trailing zeros) bitwise intrinsics
433. Popcount bitwise intrinsics (set bit count)
434. Kernighan bit trick (find highest bit set)
435. Fast NOR/NAND/XNOR via assembly instructions
436. Fast LOG2 of integers
437. Fast largest power-of-two of integers

Floating-Point Optimizations:

438. Convert float to 32-bit integers (float bit manipulations)
439. FTZ (Flush to Zero) mode
440. DAZ (Denormals Are Zero) mode
441. LOG2 of floating-point is the exponent
442. Zero/negative zero bitwise tests
443. Disallow negative zero (to use faster zero comparisons)
444. NaN (Not-a-Number) bitwise tests
445. Inf/-Inf bitwise tests
446. Avoid denormalized numbers
447. Disable denormalized numbers (subnormals) (compiler/library modes)
448. Avoid underflow in floating-point (ignore it)
449. Avoid overflow in floating-point (ignore it)
450. `memcmp` float vector equality (disallow special values for fast `float` vector equality comparison)
451. Fast detection of special values in `float` vectors (bitwise operations)
452. Floating-point intrinsic functions (various)
453. Exponent addition: bitshift floating-point by addition of the exponent bits
454. Sign bit flipping/extraction/setting (bitwise tricks)

Compiler Settings for Floating-Point:

455. GCC `-ffast-math` option — faster math mode.
456. GCC `-fno-math-errno` — faster math by not setting `errno`.
457. GCC `-ffinite-math-only`
458. GCC `fno-trapping-math`
459. MSVS `/fp:precise`, `/fp:strict`, `/fp:fast`
460. Disable floating-point exceptions

Loop Optimizations:

461. Exit loops early (e.g., `break` or `return` statements)
462. Finish loop body early (i.e., `continue` statement)
463. Correct choice of loop
464. Loop unrolling
465. `#pragma unroll`
466. Loop fusion
467. Loop perforation (probabilistic)
468. Loop tiling/blocking
469. Loop fission
470. Loop reversal (don't use!)
471. Loop code motion ("hoisting")
472. Loop distribution
473. Loop iterator strength reduction
474. Loop coalescing

475. Loop collapsing
476. Loop peeling
477. Loop splitting
478. Loop interchange
479. Loop sentinel
480. Loop strip mining (loop sectioning)
481. Loop spreading
482. Loop normalization
483. Loop skewing
484. Loop interleaving

If Statement Optimizations:

485. Replace `if-else-if` sequences with `switch`.
486. Replace `if-else-if` sequences with lookup table loop.

Switch Statement Optimizations:

487. Use compact numeric ranges in `switch` (compiler can use a LUT)

Compile-Time Optimizations:

488. `inline` functions
489. `always_inline` specifier
490. GCC `flatten_inline` specifier
491. `gnu_inline` GCC specifier
492. Keep `inline` functions short (helps compiler to inline)
493. Keep `inline` functions in header files (source available to all its calls)
494. Avoid making `virtual` functions "`inline`"—compiles but a slug.
495. `sizeof`
496. Use `sizeof` with `static_assert` (e.g., portability checks)
497. Virtual functions cannot be inlined (although it compiles)
498. Pointer-to-function usages of functions cannot be inlined
499. Function objects (functors) cannot always be inlined
500. Lambda functions cannot always be inlined
501. `inline` variables (C++17) (helps with linking)
502. `static_assert` (compile-time assertions)
503. `const` is good
504. `constexpr` (C++11) is great
505. `constexpr` functions allow `if`, `switch`, loops, etc. (C++14)
506. `constexpr` lambda functions (C++17)
507. `constexpr` and placement `new` (C++26)
508. References to `constexpr` variables (C++26)
509. `if constexpr` statements
510. `constinit`
511. `consteval`
512. `if consteval` (C++23)

513. Type traits <type_traits> (C++11)
514. `typeid` is slow (RTTI)
515. `std::is_same_v` (type trait test)
516. Template specialization (for specific types)
517. Template specialization (for constant integers)
518. Variadic templates (C++11)
519. Template Meta Programming (TMP) still works, but prefer `constexpr`
520. Auto-vectorization (by compiler)
521. Auto-unrolling of loops (by compiler)
522. SFINAE tricks (mostly an issue for compiler engineers)

Pointer Aliasing:

523. Reorganize functions with awareness of pointer aliasing issues
524. Restricted pointers (to avoid pointer aliasing slowdowns)
525. `-fstrict-aliasing` compiler option (alternative to "`restrict`")

Pointer Arithmetic:

526. Loop pointer arithmetic
527. End pointer address tricks (Loop pointer arithmetic)
528. Use references not pointers (avoids null testing)
529. Prefer postfix operations with the `*ptr++` idiom (not prefix `++ptr`)
530. Pointer comparison tricks
531. Pointer difference tricks
532. Avoid safe pointer class wrappers (prefer raw pointers for speed)

Pointer Optimizations (Other):

533. `reinterpret_cast` (helps the optimizer and is effectively a free compile-time hint)
534. Avoid `dynamic_cast` (to downcast from a base to a derived class, which can be helpful for specializing member calls, but dynamic casts can be expensive at runtime because of RTTI)

Function Optimizations:

535. Return early from functions
536. Flatten function call hierarchies
537. Callbacks are an extra layer of function call
538. Lambda functions are convenient but are an extra function call layer (though often inlined)
539. Function objects (functors) are an extra function call
540. Avoid recursion (completely; we're not in High School anymore)
541. Replace simple recursion with a loop
542. Replace complex recursion with a stack
543. Tail recursion elimination
544. Recursion higher base level

545. Collapse recursion levels
546. Specialize functions with default arguments (use two versions)
547. Specialize functions with `void` and non-`void` versions (if the return value is often ignored)
548. Avoid function pointers (cannot be `inline` or `constexpr`)
549. Merge multiple Boolean function parameters into a "config" object with Boolean data fields.
550. `noexcept` attributes allow compiler to avoid adding extra code (C++11)
551. `std::initializer_list` can be used to return multiple values (benchmark against other methods)

C++ Class Optimizations:

552. `friend` functions (bypass interfaces)
553. `friend` classes (bypass interfaces)
554. Return references rather than objects
555. Avoid temporary class objects in expressions
556. Add extra member functions to avoid temporary object creation
557. Pass objects by reference to functions (i.e., "`&`" or "`const&`")
558. Disable copy constructors with "`private`" or "`= delete`"
559. Disable assignment operators with "`private`" and "`= delete`"
560. Declare overloaded assignment operators with `void` return type (except when defaulting)
561. Re-use objects to avoid constructor and destructor calls
562. Avoid calling the destructor when in shutting down mode
563. Uninitialized memory: `std::uninitialized_fill` (C++17)
564. CRTP (Curiously Recurring Template Pattern): derived class derives from base class which is itself a template involving a pointer to the derived class (optimizes polymorphism to be compile-time, avoiding virtual function calls; also, this allows more inlining of these calls.)
565. Move constructors
566. Move assignment operators
567. `std::move` (C++11, C++14) is usually a compile-time cast.
568. Return object reference types (not complicated objects)
569. Avoid virtual function calls with explicit calls to the specific function
570. Specialize inherited member functions (for the more restrictive type)
571. Avoid overloading the postfix increment/decrement operators
572. Block the overloaded postfix increment/decrement operators (`void` body or `=delete`)
573. Consider skipping destructor cleanup if program is shutting down
574. Avoid accidental double initialization of data members in constructors
575. Avoid redundant initialization of members in constructor and "setup"
576. Specialize member functions with default arguments (use two versions)
577. Default constructors/destructors with "`=default`" may be more efficient than hand-coded versions.

578. Trick for singleton pattern in multithreading — thread initialization of a function-local static variable, other threads block, once-only initialization guaranteed by C++ compiler.

Advanced C++ Compiler Optimizations:

579. Copy elision (compiler auto-optimization with avoidance of calls to copy constructor in certain cases)
580. Guaranteed copy elision (C++17)
581. Named return value elision (a type of copy elision)
582. Temporary return value elision (a type of copy elision)
583. Copy elision in exception handling (special case for copy elision)
584. Allocation elision (new operator) (C++14)
585. Use xvalue or "expiring value" optimizations (various)
586. Trick: disallow creating an object on the stack, make its destructor private.
587. Trick: to disallow creating an object on the heap: make declarations of the `new` and `new[]` operators private.

Byte Block Operations in C++ Classes: (Use with extreme care!)

588. `memset/bzero` to zero in a constructor — fast but dangerous, overwrites internal "vtable" data in object if class has any `virtual` functions, does not call constructors of its data members or base class members; also cannot use an initializer list as this overwrites with zero after any objects were set by the initializer list.
589. `memcpy` to bitwise copy in a copy constructor or assignment operator — fast but dangerous, improperly copies internal vtable data in object if class has any virtual functions, does not deeply copy any of its members or base class members nor call their constructors.
590. `memcpy` to bitwise copy in a move copy constructor or move assignment operator — fast but dangerous; improperly copies "vtable".
591. `memcmp` to bitwise compare for equality/inequality tests — fast but fails due to pitfalls: padding bytes, bit-field members, negative versus positive zero floating-point values, NaN floating-point values.
592. Virtual inheritance — usually for pure virtual base classes; avoids double objects if the same base class type is inherited in two different pathways.

Timing C++ Methods:

593. `std::chrono` C++ class (highly granular)
594. `clock()` C/C++ function
595. `time` command (Linux shell)
596. `time()` function (granularity is only in seconds)
597. `gettimeofday()`

Benchmarking C++ Methods:

598. Loop unrolling for accurate benchmarking
599. Use `volatile` specifier for accurate benchmarking
600. Loop overhead measurement for accurate benchmarking
601. Google Benchmark: Apache 2 license;
code: https://github.com/google/benchmark

Compiler Settings:

602. Optimizer settings
603. Optimizing for space/memory size (compiler flags)

General Build & Software Development Practices for Efficiency:

604. Maintain separate builds for slow testables versus production executables
605. Compile-out assertions
606. Compile-out self-testing code
607. Compile-out debug code or tracing code
608. Ensure test code not accidentally left in production (test a global flag based on these macros at startup)

CUDA C++ GPU Optimizations:

609. Coalesced memory accesses
610. Thread specialization (GPU)
611. GPU thread pools
612. Producer-consumer thread pools
613. GPU kernel optimizations
614. Striding (GPU kernels)
615. Overlapping GPU uploads and compute
616. Overlapping with recomputation/rematerialization
617. Offloading to CPU
618. Pinned memory blocks
619. Warp divergence (warp coherence)
620. Grid optimizations
621. Grid size optimizations

Core Utility Classes (Efficiency Helpers): (to build for overall efficiency practices)

622. Bitwise macro library (bitflag management)
623. Floating-point fast bitwise operations macro library
624. Benchmarking/timing library
625. Smart buffer library (reduce allocations by combining allocated/non-allocated memory management)
626. TCP/UDP wrapper library
627. Specialized data structures for small amounts of data (faster than STL)

628. Sorted array and binary search (small array size)
629. Lock-free queues
630. Perfect hashing library
631. Bit vector data structures (possibly based on STL)
632. Bit set data structures (possibly based on STL)
633. Bloom filter library
634. Vector hashing library
635. Caching utilities library
636. Source code precomputation library
637. Basic data and statistics on vectors (e.g., averages, std dev/variance, etc.)
638. Incremental vector algorithms (averages, min, max, etc.)
639. Branchless coding primitives library
640. Graph library for locking analysis
641. Data compression library
642. Approximate tests library
643. Math library (versus STL)
644. Memory pools library (fixed-size custom memory allocators)
645. Custom memory allocator library
646. Placement `new` operator versions
647. Placement `delete` operator (write your own)
648. Multi-dimensional array library (linearize your vectors, matrices, tables, or tensors)

AI Kernel Optimizations (using LLM Inference Optimizations for non-AI low latency applications): (subset of methods to consider) Reference: 500+ LLM Inference Optimization Techniques (blog article)

649. Kernel fusion
650. Kernel fission
651. Kernel tiling/blocking
652. Quantization (integer-based approximation of floating-point)
653. Low-bit quantization
654. Binary quantization (1-bit)
655. Integer-only arithmetic
656. Floating-point quantization (FP16/FP8/FP4)
657. Mixed precision quantization
658. Logarithmic quantization
659. Dyadic quantization
660. Low rank matrices
661. MatMul/GEMM optimizations (many)
662. MatMul data locality optimizations
663. Sparse MatMul
664. Approximate matrix multiplication
665. Contiguous memory block matrix multiplication

666. Cached transpose MatMul
667. Fused transpose MatMul
668. Tiled/blocked MatMul
669. Sparsification (Pruning/Sparsity)
670. Token pruning (input compression)
671. Token skipping
672. Token merging
673. Data compression algorithms
674. Early exiting (of layers)
675. Caching optimizations
676. Vector computation caching
677. Zero skipping
678. Negative skipping
679. Padding optimizations
680. Zero padding removal
681. Zero-multiplication arithmetic
682. Adder/addition (zero-multiply)
683. Bitshifts (zero-multiply)
684. Bitshift-add (zero-multiply)
685. Double bitshift-add (zero-multiply)
686. Add-as-integer (zero-multiply)
687. Logarithmic arithmetic (zero-multiply)
688. Hadamard element-wise matrix multiplication
689. End-to-end integer arithmetic
690. Table lookup matrix multiplication
691. Weight clustering (grouped quantization)
692. Vector quantization
693. Parameter sharing
694. Activation function optimizations (non-linear functions)
695. Precomputation of Activation functions
696. Approximation of Activation functions
697. Integer-only approximation of Activation functions
698. Fused activation functions
699. Normalization optimizations (non-linear vector data functions)
700. Fused normalization optimizations
701. FFN optimizations (double MatMul)
702. FFN approximations
703. FFN integer-only
704. Decoding algorithm optimizations
705. Speculative decoding
706. Multi-token decoding
707. Ensemble decoding
708. Consensus/majority-vote decoding
709. Easy-hard queries

710. Batching computations
711. Advanced number systems
712. Posit numbers
713. Dyadic numbers
714. Hybrid number systems
715. Fixed point numbers (integers not floating-point)
716. Block floating-point (BFP) hybrids
717. Logarithmic number system (LNS)
718. Disaggregation (prefill/decoding)
719. Computation re-use
720. Conditional computation
721. Approximate caching
722. Addition arithmetic optimizations
723. Approximate addition
724. Bitwise arithmetic optimizations
725. Fast multiplication arithmetic
726. Approximate multiplication
727. Logarithmic approximate multiplication
728. Approximate division
729. Bitserial arithmetic

Appendix B: C++ Slug Catalog

Slug Hunting Advice

This appendix is about speeding up your C++ programs through general improvements to sequential or parallel coding. Before we begin with anything that's actually useful, I have to introduce the obligatory wrist-slapping politically-correct deslugging advice for programmers. Hence, here are some general nuggets of advice when attempting to speed up your program:

- Profile twice, code once. Performance profiling tools exist for a reason.
- Don't micro-optimize. Unless you're into that kind of thing. But really, try to sit on your hands.
- Do macro-optimize. Think about your data structures and algorithms.
- Optimizing introduces new bugs. 100% guaranteed! Don't optimize the night before your release. Re-run your test suite.
- Don't optimize exception handling. Tweaking rarely-executed code is a poor use of your geniousness.
- Use open source third-party libraries that have already been optimized by others.

Or just ignore that advice and go crazy. It's just too much fun optimizing when the alternative is dreary debugging. Pro tip: it's even more fun writing a book on optimizing!

Where to hunt slugs? Some of the common large-scale issues with coding inefficiency in typical C++ programs include:

- Function call hierarchies
- Nested loops
- Overuse of memory allocation
- Constructor and destructor inefficiencies
- Inefficient algorithms (e.g., linear search of arrays)
- Unnecessary overhead or wrappers
- Recursion. After you've coded up your university assignments (remember Tower of Hanoi, anyone?), please forget recursion exists.

C++ Speedup Techniques: Some of the general ways to speed up C++ programs at the design structure or algorithmic level include:

- Faster data structures (e.g., hash tables).
- Faster algorithms (e.g., fix linear search to something faster like, you know, hashing again).
- Parallelize via multi-threading, multi-process, multi-core, multi-GPU, multi-something.
- Vectorization (parallelize your important loops)
- Precompute expensive functions into a lookup table at compile-time (e.g., activation functions).
- Cache any complex calculations to trade extra space for time savings (e.g., KV caching).
- Change floating-point to integer operations (quantization, anyone?)
- Replace recursion with iteration. Subtract ten bonus points if you need to do this.

Some of the high-level C++ coding optimizations include:

- Flatten function call hierarchies (stop wrapping everything so much, and inline the small functions at the bottom).
- Optimize loops, especially nested loops (e.g., move loop-invariant code out, loop unrolling, vectorization, etc.)
- Templates are effectively a compile-time optimization that improves speed at the cost of code space.
- Reduce memory allocation (use less memory overall or replace memory allocation with temporary stack buffers).
- Operator strength reduction (e.g., replace "*" with "+", a pipe dream of all AI engineers).
- Declare variables as close as possible to where they are used. This avoids instantiating objects that aren't needed on some paths.
- Use pointer arithmetic, especially for loops over arrays.
- Bitwise operations are fast, but the basic C++ integer operations are also fast too, nowadays. Benchmark, don't assume.
- Use short-circuiting of the `&&` and `||` operators, and also the ternary `?:` operator, to avoid expensive function calls.

And finally, some things you might forget (and some that are forgettable):

- Benchmark any important changes (e.g., operator strength reductions).
- Turn up your C++ optimizer. There are higher settings you could try.
- Add compile-time optimization hints (e.g., `constexpr` and `restrict`).
- Overclock your PC (like a gamer).
- Sell your car to buy a better GPU.
- Put every function in a header file and make them all `inline`.
- Reorder your `case` labels. Surely it helps.
- Change `i++` to `++i` in everyone else's code.

C++ Class Slugs

The C++ class features are designed to add encapsulation and modularity, while retaining speed, but there's still plenty of ways that slugs can crawl into your classes. C++ class optimizations include:

- Ensure small member functions are `inline`, especially those that do "get" and "set".
- Add `inline` to other `friend` or non-class functions (esp. if small or commonly used).
- Pass objects to functions using "`const&`" (pass-by-reference), rather than pass-by-value.
- Watch out for temporary objects. These can occur in simple assignments or function call expressions or in weird ways like accidentally making your overloaded assignment operator have the wrong type.
- Use reference variables instead of copying objects into temporary variables.
- Take care templating class objects (e.g., when using the `std::vector` class for a vector of your class objects). Lots of hidden calls to constructors and destructors may arise in resizing.
- Use the initializer list in the constructor for initializing data members.
- Use `friend` functions for faster accesses to internal object data.
- Block accidental calls to the copy constructor or class assignment operator (i.e., if you aren't defining them, make a dummy version that is "`private`" with a "`void`" function body).
- Avoid returning objects if you can. Return a reference if it's safe to do so.
- Take care with "wrapper" classes like "smart pointers", "smart integers" or "smart buffers". Usually, they're safer but slower. How smart is that?

Bypass interfaces with friend functions

Using `friend` functions may be faster because they can bypass class getter and setter member functions. If a `class` declaration has a good deal of private data, it is common C++ style to declare an interface of public member functions to access private data. Although the class interface can be quite efficient if member functions are declared as `inline`, the need to call a function to access a data value can still make it inefficient in some cases. The use of `friend` functions and `friend` classes can be efficient because this bypasses the class interface. For example, a member function to set a data member may perform some range checking on the value, but if we can be sure that a particular function will not use incorrect data, a `friend` function can be used to bypass this checking.

`friend` functions (or `friend` classes) should not be considered unless the function needs very fast access to data members, and the member functions to access the data perform other computations. Note that a member function, with its special privileges, also bypasses the class interface (because it is part of it), and `friend` functions should not be used where member functions would be more appropriate. Programming style is the consideration here, as they would both have similar efficiency.

A good example of `friend` function efficiency occurs when an operator function operates on two different classes, such as when we need an operator that multiplies a `Matrix` object by a `Vector` object to yield a new `Vector`. Assume that both classes have member functions to access individual elements of the `Vector` or `Matrix`. Consider the declaration of the multiply function as neither a `class` member nor a `friend` function, as in:

```
const int N = 10; // Number of elements in vector/matrix
class Vector {
    double data[N];
public:
    double get_element(int i) const { return data[i]; }
    void set_element(int i,double value) { data[i]= value; }
};

class Matrix {
   double data[N][N];
public:
   double get_element(int i, int j) const {
        return data[i][i]; }
};
```

```
    Vector operator * (const Matrix& m, const Vector& v)
    {
        Vector temp;
        // multiply matrix by vector
        for (int i = 0; i < N; i++) { // for each row
            double sum = 0.0; // sum of N multiplications
            for (int j = 0; j < N; j++) {
                sum += m.get_element(i, j) * v.get_element(j);
            }
            temp.set_element(i, sum); // store new element
        }
        return temp; // return new vector
    }
```

This will be horribly inefficient because the `operator*()` function must go through both class interfaces to access elements. Although it isn't necessarily any less efficient here, if range checking of the array index `i` were present in the member functions to set or access the elements, this would cause inefficiency.

Note that if the `Vector` class overloaded the `[]` operator instead of using a `get_element` member function, this would make no difference to efficiency—notational convenience is gained but the `operator[]` function has the same cost as any other function.

One alternative to consider is to make the `operator*` function another member of the `Vector` class, but this will still mean using the interface for the `Matrix` class. A more efficient solution is to make the `operator*` function a `friend` of both `Matrix` and `Vector` classes, thus allowing it direct access to their individual data elements, bypassing any range checking on array indices. The more efficient version, using a `friend` function, is:

```
    const int N = 10; // Number of elements in vector/matrix
    class Matrix;
    class Vector {
        double data[N];
    public:
      friend Vector operator*(const Matrix& m, const Vector& v);
    };

    class Matrix {
      double data[N][N];
    public:
      friend Vector operator * (const Matrix& m, const Vector&
v);
    };
```

```
Vector operator * (const Matrix& m, const Vector& v)
{
    Vector temp;
    // multiply matrix by vector
    for (int i = 0; i < N; i++) { // for each row
        double sum = 0.0; // sum of N multiplications
        for (int j = 0; j < N; j++) {
            sum += m.data[i][j] * v.data[j]; // access data
        }
        temp.data[i] = sum; // store new vector element
    }
    return temp; // return new vector
}
```

The disadvantage of using `friend` functions is the same as their advantage: they pierce class encapsulation. Because a `friend` function makes use of hidden private data members, and any change to the class may require a change to the definition of the `friend` function, whereas in the first version of the `operator*` function, the use of the "`get_element`" functions of both `Vector` and `Matrix` meant that it would need no changes, provided the "`get_element`" functions were correctly changed within the class.

Avoid Virtual Functions

Object-oriented programming purists will hate me for this section. C++ `virtual` functions are a wonderful incarnation of OOP and they can be beautiful and elegant. But you need to avoid them sometimes if speed is your goal.

They're also very fast function calls, even though done dynamically. Although `virtual` function calls seem like they're complicated and possibly slow, they're actually very carefully designed to be very fast to call in C++ class hierarchies. There's lots of painstaking work for compiler designers to get them to compile correctly, but their runtime efficiency is great for programmers. The implementation is effectively a small lookup table with function pointers. It's a couple more assembler statements before the function call, and the overhead of calling a function will dwarf that cost.

So, why do I say to review your use of `virtual` functions? Because they're an optimizer blocker. Since they're a dynamic runtime function call, there's much less opportunity for the C++ compile-time optimizations to remove these calls. Indeed, the compiler cannot always determine what function is being called and you can lose these speedups:

- `inline` functions
- `constexpr` function evaluation

Hence, I say you have to choose carefully in the use of virtual functions. Avoid them for speed-critical functions, and don't use them only for good OOP style when you don't really need them. But also, don't be afraid of using them in other instances because they're only marginally slower than a non-inlined function call. Kudos to the C++ language designers for that!

Avoid unnecessary virtual function calls

The use of virtual functions, when they are not needed, is obviously inefficient. virtual functions are needed only when dealing with pointers or references to objects of unknown type. If the program never uses pointers or references to objects, or if it does not have any derived classes, no function needs to be virtual and the use of virtual wastes space. In addition, because virtual functions relate only to the use of derived classes, declaring any functions as virtual in a class that has no derived classes is also unnecessarily inefficient.

One common situation where virtual may appear necessary, but need not be, occurs with redefining a member function in a derived class. This does not necessarily mean that the function must be defined as virtual in the base class (nor in the derived class — the virtual keyword is never needed in the derived class). Of course, if the program starts using pointers or references to these classes, the functions may need to be virtual, in which case it may be better style to declare the member function as virtual.

A call to a virtual function need not always be a "real" virtual call. For example, passing an object by reference (either as a reference or as a pointer type) can occur when changing functions to pass-by-reference for efficiency improvement.

Any calls to virtual functions inside that (not necessarily virtual) function will be such that the compiler cannot know that an ordinary function call to the member function would suffice. It does not perform any global analysis to determine that all arguments to the function are base objects, and not derived objects. For example, in the following code, it isn't clear that the call to the (virtual) print function could be replaced by an ordinary call:

```
void print_base_object( Base & object)
{
    object.print();
}
```

The overhead of virtual function calls can be removed whenever the programmer can be sure that only one type of pointer/reference to an object is being used. In particular, whenever a programmer can be sure that a pointer/reference to a base class object points to a particular object, the qualified member function name can be used. For example, the `virtual` call uses:

```
    p->print();
```

And the more efficient code that avoids a `virtual` function call is:

```
    p->Base::print();
```

An example of extra information making this change possible occurs when a program uses a number of different (homogeneous) linked lists, with each linked list containing the same type of object (one with base objects, one with derived objects). When implementing a `print_list` function to print out a linked list, you can write it generally to call a `virtual`-declared `print_object` function:

```
    void LinkedList::print_list()
    {
        for (Base *temp = head; temp != NULL; temp=temp->next())
            temp->print_object();
    }
```

This means that each call to `print_object` has the run-time overhead of a `virtual` function call. A more efficient alternative is to make use of the knowledge that each list must contain the same type of object, and have two different `print_list` functions (i.e., use a `virtual` function to do the dirty work of printing the objects).

```
    void Base::print_list_hidden()
    {
        for (Base *temp = this; temp != NULL; temp=temp->next())
        temp->Base::print_object();
    }

    void Derived::print_list_hidden()
    {
        for (Derived *temp = this; temp != NULL;
        temp = (Derived*)temp->next())
        temp->Derived::print_object();
    }
    void LinkedList::print_list()
    {
        if (head != NULL)
            head->print_list_hidden(); // call virtual function
    }
```

With this approach, all of the lower-level calls to `print_object` can be bound at compile-time and the only `virtual` call is the call to `print_list_hidden` at the very top. Hence, by using our knowledge about the linked lists, we have reduced the number of run-time `virtual` function calls.

Specialize inherited member functions

In an inheritance hierarchy, the derived class is a specialized version of the base class. This means that member functions inherited from the base class can often be rewritten more efficiently to make use of the known special features of the derived class objects.

Example: Triangular Matrix Algebra. As an example, consider a class "`UTMatrix`" (upper triangular matrix) which is derived from class "`Matrix`" and represents matrices where all elements below the main diagonal are zero.

The general matrix "`add`" function of the `Matrix` class is inherited by the `UTMatrix` class, and it will work correctly. However, this inherited function is inefficient and it is more efficient to add a new member function to the `UTMatrix` class to add two upper triangular matrices avoiding all additions involving elements below the diagonal (because they are known to be zero).

In fact, it is also more efficient to write special functions to add ordinary matrices to upper triangular matrices. The computation of the determinant of a triangular matrix is also more efficient than that for a general square matrix, so this member function should also be rewritten in the `UTMatrix` class.

Example: Complex Numbers. As another example, consider a class "`Imaginary`" (imaginary numbers) derived from another class "`Complex`" (complex numbers). For all operations involving `Imaginary` objects, it is certain that the real part of the complex number is zero. Hence, it is more efficient to rewrite all inherited operations that use the real part of a `Complex` object, such as: addition, multiplication, norm, etc.

The main disadvantage of specializing member functions is that the code reuse advantage of inheritance is negated; more programmer time must be spent on recoding the specialized member functions. Other disadvantages are the increased probability of error, most special cases to test, and an increase in executable code size.

Assignment Operator Return Type

The return type of the overloaded assignment operator should usually be a reference type or `void`. A common mistake is to make it return a class object. Consider the following class declaration:

```
class Integer {
    private: int val;
    public:
    Integer operator = (const Integer &x);
    // ...
};

Integer Integer::operator = (const Integer &x)
{
    val = x.val; // copy data
    return *this; // return left operand
}
```

This declaration of the assignment operator to return an object permits expressions using the result of assignment, such as:

```
Integer x, y, z;
x = x + (y = z); // embedded assignment
x = y = z; // multiple assignment
```

However, it needlessly calls the constructor and destructor for a temporary object, leading to inefficiency, and occasionally to error. The correct declaration of the assignment operator is to return a `const` reference to `Integer`. This simply requires an `&` in the return type declaration, as follows:

```
const Integer& Integer::operator = (const Integer &x)
{
    // ... same as above
}
```

Note that `const` is required because the use of a non-`const` reference return type is slightly undesirable because it allows the very strange (and probably incorrect) multiple assignment:

```
(x = y) = z;
```

Although the failure to declare the return type as a reference above was a slug, rather than a bug, it can be more dangerous.

For a MyString class with dynamic allocation, using an object return type of MyString instead of MyString& will cause a temporary object to be created at the return statement, using the copy constructor with "*this" as the argument. If the copy constructor is defined correctly, this is often just an instance of inefficiency, but it may also lead to fatal errors related to temporary objects. When the copy constructor isn't defined correctly, the programmer has an error with an increased level of complexity caused by temporary objects.

Return Type Void: Note that it may be far better simply to declare the return type of the assignment operator as void, rather than a reference type. Although this prohibits embedded assignments in expressions and also multiple assignments, these are poor style anyway and should probably be discouraged. Using return type void is also slightly more efficient because no value need be returned. However, returning the reference type is the more common C++ idiom.

Singleton Classes

In a one-instance class there will only ever be one object defined from it. There are called "singletons" in the "design patterns" parlance. In this situation the class can be defined very efficiently by making use of compile-time initialization with data members declared as "static" members.

An example is a hash table implementation of a symbol table (e.g., in a compiler keyword table or an AI vocabulary table used by the tokenizer), where only one symbol table will ever be used. The crucial fragment from this code is:

```
class SymbolTable {
  private:
    Node * table[TABLE_SIZE]; // Hash table - array of ptrs
  public:
    SymbolTable(); // constructor
};

//-----------------------------------------------------
// Constructor - initialize the hash table to empty
//-----------------------------------------------------
SymbolTable::SymbolTable()
{
    for (int i = 0; i < TABLE_SIZE; i++) // all ptrs NULL
    table[i] = NULL;
}
```

If there will only be one hash table, the constructor is needlessly inefficient. A more efficient version declares the hash table as a static data member and the implicit initialization to zero will set all the pointers to NULL at compile-time. The efficient code for a one-instance hash table is:

```
class SymbolTable { // ONE INSTANCE ONLY
  private:
    static Node *table[TABLE_SIZE]; // Compile-time init
  public:
    SymbolTable() { } // constructor does nothing
};
```

Temporary Objects and Destruction

Temporary objects are created automatically by the compiler in a number of situations. This is a similar idea to that of a C++ compiler generating temporary values for intermediate results of a computation. However, a temporary with class type will have its constructor and destructor activated, so temporary objects can be quite expensive.

For example, try the following class to demonstrate how a temporary object is defined for intermediate expression results, particularly that returned by the + operator:

```
#include <iostream.h>
class Integer {
private: int val;
public:
    Integer() { val = 0; cout << "Constructor\n"; }
    ~Integer() { cout << "Destructor\n"; }
    Integer(const Integer &x)
    {
        val = x.val;
        cout << "Copy Constructor\n";
    }
    void operator=(int x) { val = x; }
    void operator=(const Integer &x) { val = x.val; }
    friend Integer operator+(Integer &x, Integer &y);
};

Integer operator+(Integer &x, Integer &y)
{
    Integer temp; // user-defined temporary
    temp.val = x.val + y.val;
    return temp; // creates compiler temporary
}

int main()
{
    Integer i, j, k;
    k = i + j;
}
```

There are 4 calls to the ordinary constructor corresponding to `i`, `j`, `k`, and `temp`; there is a single call to the copy constructor that occurs when the `return` statement creates a temporary object for the object returned from operator +. This temporary object is the result of `i+j` and is then assigned to `k`.

In this case there are poor performance and no errors related to temporary objects and in most cases, temporary objects are transparent to the programmer for a correctly defined class (i.e., having both assignment operator and copy constructor). However, if the programmer unwittingly stores a reference or pointer to members of a temporary object, there may be errors in a later use of the reference or pointer. The problem is that temporary objects can be destroyed by the compiler as soon as they have been used in the computation, and so the reference or pointer is no longer valid. However, since the timing of the destruction of temporaries is undefined, some compilers will not exhibit an error for such code because they leave the destruction of temporaries till late; it depends on how aggressively a particular compiler performs its internal code optimization.

Overloaded Postfix Increment Operator

The postfix increment operator (`x++`) is a big slimy slug. I'm not talking about your `for` loop with "`i++`" versus "`++i`" for an integer, which is the same on any compiler since about the 1990s, despite the endless online arguments about it. I'm talking about overloaded increment and decrement operators for classes.

In C++ you can declare separate prefix and postfix increment overloaded operators for a class, by putting an extra dummy "`int`" parameter in the postfix version. You can also leave out a postfix version, and the prefix version will be called for both usages. The default call to prefix versions is not a slug, but a potential bug if you copy-paste code or use postfix ++ in template code. Also, returning the current object for the prefix increment operator is only a minor slug, because you're returning a reference to the current object (and a reference is really just a pointer).

Postfix operations are much worse. They are slower than airport queues at Thanksgiving. The semantics of the postfix increment operator (`x++`) in the C++ language are effectively:

1. Create a temporary copy of your object.

2. Increment the current object.

3. Return the temporary object.

If you actually do this big shemozzle for a class object, you've got a whole lot of processing happening on a temporary object that's probably not even used. Maybe the optimizer will cut a lot of it as dead code, or maybe not. With the horrors of that echoing in your mind, here's my first suggestion:

Don't even declare postfix overloaded operators for your class.

Don't overload the postfix increment operator. In fact, you can stop it being used by declaring a dummy version that is "`private`" (stops external usage) with a "`void`" function body (stops internal usages).

```
    private:
        void operator++(MyClass &x, int) void;   // Postfix denied!
        void operator--(MyClass &x, int) void;
```

Void Return Type: Note that attempts to call a postfix ++ operator on a class type may occur in template instantiation with your type. If it's your template, change the template code to use prefix operators. If you really must define an overloaded postfix increment or decrement operator, then here's my second suggestion:

Make the return type "void"

Hence, a basic usage of "`x++`" will compile and work correctly. Not only will it be efficient to not return anything, but the compiler will also ensure that nothing more fancy will run. A compilation error will block any use of postfix ++ that relies on the operator returning the old object. In other words, this will be fine:

```
    x++;
```

But this will get a compiler error alerting you to a problem:

```
    y = x++;    // Error
```

Standard Vector Object Resizing

The standard `vector` class is usually very efficient for basic data types, but you need to take care if you instantiate it with a class type. The risk is that you'll have hidden calls to this class type's constructors and destructors, potentially for every element of the `vector`, under various circumstances.

This slug is a type of "hidden copy constructor call" problem. If you don't manage the size of the standard C++ `vector` class objects in the initialization or via the "`reserve`" method, there can be a lot of hidden resizing happening behind the scenes whenever you are adding elements to the vector. This will at least be doing bitwise copies of the elements of each vector. But it's even worse if the vector contains complex objects with a defined copy constructor. When it's resizing the `vector`, it will call the copy constructor for each and every object that is an element of the `vector` because it needs to move them all.

Even for basic data types there can be some cost to copying the data when resizing. You can take control of this with the "`reserve`" function, so that the `vector` object doesn't need to keep resizing itself if you're adding to it.

Skipping Destructor Cleanup

It's really good OOP coding style for your destructor to carefully clean up every resource your object needed, and you know, beautiful coding idioms are just so very important. I certainly wouldn't want to be the person to tell you to do some ugly hack, even if it made everything a whole boatload faster. Umm, really, I wouldn't want to, but if you promise not to tell anyone you heard it from me...

Typically, destructor cleanup means calling "`delete`" on allocated memory used by the data members, and for complex objects, it may also mean closing files. And I often find that the cost of the destructor starts becoming significant in its own right. And one destructor call can trigger lots more, like roaches, only without the social skills. If you call "`delete`" on any member objects or worse, arrays-of-objects, then those destructors get called, and this triggers a whole blam of code that cascades down the object hierarchy.

Here's a thought: *don't cleanup!*

This is an optimization worth considering in some cases:

- Batch jobs
- Re-launching server daemons
- Program is shutting down anyway

If your program is a run-once batch job, and it's not going to be running again with a new request, or even if it's an AI inference server process that handles 1,000 user queries, after which another copy will launch in its place, then you can make like a teenager, and don't cleanup. Thumb your nose at Valgrind and comment out all those `delete` lines in your destructors.

Let the memory leak!

Program exit is a special case that you can detect. If your program is exiting "cleanly" then it does destructor calls to all of the global objects, and so on. And you usually know in the code when the program is shutting down, whether from a user choice, a timeout or limit exceeded, or something internal like an assertion failure. One idea is to use a global Boolean flag that says "I'm shutting down" and then check it inside all of the main destructors:

```
MyClass::~MyClass()
{
     if (g_aussie_im_shutting_down) return;  // Skip!
     ...
     // Lots of stylistically beautiful code
}
```

Is it safe? What happens if you just skip all the cleanup? Well, nothing bad in many cases. The operating system cleans up the allocated memory as part of reclaiming *all* of the memory. Files are a bit more of a complicated story. Standard C++ shutdown should also properly close any files opened for reading, although you might possibly lose some buffered output written to a log file, so maybe you should still flush buffers or close those files.

This idea of skipping destructors isn't always workable. It's not always clear that ending the process will properly save buffered output in closing files. As another more complex example, if there's an abnormal disconnect from a database session or a remote network connection hangup (e.g., socket session not ended properly), there might be some other consequences, like error messages in the logs locally or for the remote peer.

Initializer lists for member objects

When a class declaration contains a class object as one of its members it is important to use the correct method of initialization to retain efficiency. Consider the declaration of a class `B` containing a member object from class `A`:

```
class A {
  private:
    int val;
  public:
    A() { val = 0; }
    A(int x) { val = x; }
    void operator = (int i) { val = i; }
};
```

```
class B {
  private:
    A a; // member is itself an object
  public:
    B() { a = 1; } // INEFFICIENT
};
```

Declaring an object of type B will cause the default constructor for the member object of type A to be invoked immediately before the default constructor for B. Then the = operator for class A is used to set the member object, a. Hence, the constructor for B involves a call to A's default constructor and a call to the assignment operator. The call to A's default constructor is redundant and should be avoided. Fortunately, C++ provides a convenient syntax for passing arguments to constructors of member objects. The default constructor for B should be recoded to use the initializer list:

```
B() : a(1) { } // EFFICIENT
```

This initialization syntax causes the constant 1 to be passed to the constructor for the member object, a (the constructor accepting the int parameter is called, instead of the default constructor). Thus, instead of calling the default constructor and the assignment operator for A, only the int constructor for A is called.

This initialization method is efficient whenever calling the default constructor for a member object is not appropriate, for instance, when the member object is initialized by a call to the assignment operator within the main object's constructor (as above, where B's constructor assigned to its member of type A). This common form of initialization can be used for any type of data member (i.e., not only class objects), although it will be neither more nor less efficient than assignment for built-in types. The special initialization syntax should be used wherever it is applicable, since it can never be less efficient than assignment to the data members within the constructor, and will often be more efficient.

Initializer lists for base objects

Base objects. Similar efficiency considerations apply to constructors in derived classes, since the data member(s) in the base class act like an object member. The constructor for the base class is always called when a derived class object is constructed. When the default constructor for the base class is of no use to a derived class object, it is more efficient to pass arguments directly to a non-default base class constructor, using the special initialization syntax. The same syntax applies as for data member initialization, except that the type name of the base class is used instead of the name of a data member. A contrived example of this form in initialization is:

```
class Derived : public Base {
  public:
    Derived() : Base(0) { } // Call Base(int) constr
};
```

Avoid temporary objects

In the same way that temporary integer variables are used to compute an integer expression, so too are temporary objects used in non-trivial expressions involving class objects. For example, consider this code where the `Complex` class has defined the + and = operators:

```
Complex c1,c2,c3;
c1 = c2 + c3;
```

This is likely to create a temporary Complex object as the result of the addition, and this temporary object is then passed as an operand to the = operator. In other words, the expression is actually evaluated as:

```
operator=( c1, operator+(c2, c3) );
```

A temporary object must be created to store the "+" sub-expression computed for the second argument, and then passed to the "=" operator. Whether the operands to `operator=` are passed by reference or by value has no effect on whether a temporary is created in this situation (it will only affect the creation of new objects inside the `operator=` function).

One (rather inelegant) method of avoiding this creation of temporaries is to create a specialized function to handle it:

```
void AssignThree(Complex&c1, Complex&c2, Complex&c3);
...
AssignThree(c1,c2,c3); // c1 = c2 + c3;
```

The function should probably be a `friend` function to allow efficient access to the data members of the three `Complex` objects.

The problems with this solution are its very poor style (because the neatness of the use of overloaded operators is lost), and also its non-general character. More complicated expressions will still generate temporaries, unless more special functions are added as `friend` functions, leading to even worse style. This "cure" is perhaps worse than the disease.

Avoid temporaries via extra member functions

There are situations where the removal of temporaries does not lead to poor style. Consider the following definition of a minimal `Complex` class:

```
class complex {
  private:
    double re; // real part
    double im; // imaginary part
  public:
    // Constructors
    complex() { re = 0.0; im = 0.0; }
    complex(double r) { re = r; im = 0.0; }
    complex(double r, double i) { re = r; im = i; }
    // Copy constructor
    complex(complex &c) { re = c.re; im = c.im; }
    // Overloaded assignment operator
    void operator = (complex & d) {
           re = d.re; im = d.im; }
    // Overloaded + operator
    friend complex operator+(complex &c1, complex &c2);
};

inline complex operator + (complex &c1, complex &c2)
{
    return complex(c1.re + c2.re, c1.im + c2.im);
}
```

Consider this class definition when used in the following code sequence:

```
complex c1, c2;
c1 = 2.0;
c2 = c1 + 3.0;
```

The effect is identical to:

```
c1 = complex(2.0); // invoke "double" constructor for 2.0
c2 = c1 + complex(3.0); // invoke "double" constr for 3.0
```

The C++ compiler automatically creates two temporary objects from the `double` constants, and calls the `double` constructor to do so. The inefficiency of the creation of a temporary object and the call to the constructor can be avoided by adding a few more functions to the class declaration:

```
 void operator = (double d) { re = d; im = 0.0; }
friend complex operator + (double d, complex &c2);
friend complex operator + (complex &c1, double d);
```

If these functions are present, then the double constants are passed directly to the double parameters of these functions. No temporary object is created, and hence the constructor is not called. Note that two symmetric versions of operator+ are required because the C++ compiler cannot assume that the commutativity of + holds for user-defined class objects.

By making the "interface" efficient for mixing complex and double variables, the creation of temporaries has been reduced. This can be generalized: it is better to provide member or friend functions to class X for a specific parameter type Y, than to provide only a constructor to create new X's from Y's.

Declare objects close to use

The C++ language allows variable declarations to appear almost anywhere within a program. Although the placement of variable declarations may seem unrelated to efficiency, it can have some effect when objects with non-trivial constructors are declared. For efficiency reasons, an object must be declared as close to its first use as possible. In particular, the C style of declaring all variables at the top of a function is often inefficient. Consider the C++ code below:

```
    void dummy(...)
    {
        complex c; // create object
        if (... ) {
            .... // use c
        }
    }
```

The complex object is not used if the condition in the if statement is false — the constructor and destructor for the unused object are called needlessly.

Declare Objects with Full Initialization

Another consideration is that objects should not be declared until there is enough information to construct them fully. For example, given a user-defined class "complex", consider the following code:

```
    complex c; // construct c
    // ....
    c = 1.0; // initialize c
```

This is less efficient than calling the correct constructor directly by using:

```
    complex c(1.0); // construct and initialize c
```

The first code sequence involves a call to the default constructor and the overloaded operator=, whereas the second declaration calls only the (double) constructor for the complex class.

Unfortunately, there are practical limits to the extent to which objects can be declared near their first use. If the first use of an object is inside a compound statement, and the object must also be used outside the compound statement, the scope resolution rules prevent the declaration from being placed inside the compound statement. For example, consider the code below:

```
double d;
complex c;
while(....) {
    cin >> d; // get double value from user
    c=d; // set complex number
}
cout << c; // print the complex number
```

In this sequence, it would be more efficient to declare "c" inside the loop block using the direct call to a double constructor:

```
complex c(d);
```

However, this would prevent the use of c outside the scope of the braces. This limitation is an unfortunate consequence of the programming language design choice to make braces both the method of grouping statements and the scoping mechanism in C++ (but there are many more important advantages supporting this decision). Unfortunately, it is not even possible to remove the braces in the above example, using the comma operator as by:

```
while(....)
    cin >> d, complex c(d); // FAILS: compilation error
```

C++ syntax prevents a declaration from being an operand of the comma operator.

Nothing Constructors. What we really want is a way to declare a class type variable, but *not* run its constructor. I'm not aware of a good way to do this. One way would be to use pointers and dynamically allocated "complex" objects, which is successful and standardized, but this adds extra memory management overhead.

Here's a thought. Maybe something like this works? Declare a dummy constructor with a dummy parameter type:

```
    class Banana { };
    complex(Banana b) {   } // nothing!
```

Then your call to the dummy constructor is hopefully optimized to nothing:

```
    Banana b;
    complex c(b);  // Nothing!
```

Data Member Optimizations

These optimizations apply to C++ objects or structures. There are various ways to speed up the data accesses and writes to a data member in an object.

Avoid bit-fields. Bit-fields are a special C++ feature designed to reduce space in an object or structure.

```
    struct node {
        unsigned int visited :1; // bit-field
    };
```

Avoid bit-fields if you want runtime speedup. They are great at reducing memory size, but often at the cost of extra run-time overhead on any accesses to these fields. Hence, for improved efficiency, at the cost of space wastage, remove the "`:1`" qualification and change to a small data type such as `bool`, `char`, or `unsigned char`.

Memory alignment: If there are mixed size data members, or there are some with "`alignas`" alignment settings, then memory alignment issues can needlessly create an oversize object. This is more of a problem in terms of unnecessary space usage, but adds inefficiencies in the need to initialize or copy the extra padding bytes for large arrays of objects. The general rules for minimizing size are to: (a) order members from large to small, and (b) group like-sized data types together.

Most used data member first. The machine code for an access to a structure or object's data fields usually involve a base address of the object, to which is added an offset that is specific to each field. References to the first field of a structure can often be more efficient because there is no need to add an offset (i.e., the offset is zero). Hence, the most used class data member or structure field should be placed first in the declarations.

Order data members by usage. It's not just the first data member whose order matters. Memory access issues such as data locality, predictive caching and memory access pipelining mean that all of the most-used data members should be close together in an object. In very large objects, there are some platforms where smaller offsets are more quickly calculated, such as data members with less than 128 or 256 as their offset. Hence, a simple optimization is to order the data member declarations according to their usage.

Function Slugs

Functions are an important building block of your code. Some ways to get the slugs out of functions include:

- Declare small functions `inline`.
- Avoid recursion.
- Pass objects by reference.
- Avoid function pointers.
- Specialize functions with default arguments.

Avoid Function Pointers

C++ allows a data type called a "function pointer" or a "pointer to a function" as part of its standard language. These are carefully type controlled, so they are reasonably efficient. However, they are not any faster than regular function calls, just because they're a fancy pointer construct, and there's a simple reason that they're not super-efficient: *they're function calls!*

A function pointer is a call to a function, so it has the whole sequence to implement. It's not much worse than a standard function call, but there's another problem. Function pointers make it difficult for the C++ compiler to get rid of the function call. The use of a function pointer will obscure much of the normal compile-time optimization logic. Hence, function pointers can be less efficient for:

- `inline` functions
- `constexpr` functions
- Intrinsic functions

In summary, they're a neat feature of C++, but not an efficiency gain. Use function pointers if they are convenient, but not as a speedup.

Change recursion to iteration

Recursion is an elegant method of problem solution, but often incurs unnecessary function call overhead. Where possible, recursion should be replaced with an iterative algorithm. For example, the famous example of a recursive "factorial" function would always be coded in a loop by professional programmers.

Fibonacci numbers. With a little insight, many recursive algorithms can be coded without recursion. For example, the Fibonacci number sequence (1,1,2,3,5,8,13,...) is defined by having the next number as the sum of the previous two numbers, with the following recursive rules:

```
Fib(0) = 1
Fib(1) = 1
Fib(n) = Fib(n-1) + Fib(n-2)
```

This has the obvious and very elegant recursive implementation:

```
int fibonacci(int n)
{
    if (n <= 1 )
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

However, there is no need to use recursion here, and a short loop is adequate. A non-recursive computation of the Fibonacci numbers is shown below:

```
int fibonacci(int n)
{
    int small = 1, large = 1;  // F0 = F1 = 1
    while (n > 1) {
        int temp = small + large; // Fn = Fn-1 + Fn-2
        small = large;
        large = temp;
        n--;
    }
    return large;
}
```

Binary Trees. There are many examples of common algorithms that are unnecessarily coded using recursion. Almost all linked list algorithms can be coded without recursion, as can the most common binary search tree operations: search, insertion and deletion. For example, the recursive implementation of tree insertion is:

```
void insert(Tree *root, Tree new_node)
{
    if (*root == NULL) // Found bottom of tree
        *root = new_node; // insert here
    else {
        if (new_node->data <= (*root)->data)
            insert(&(*root)->left, new_node);
        else
            insert(&(*root)->right, new_node);
    }
}
```

The non-recursive version of binary tree insertion is given below. It is somewhat less elegant, uses a few more variables, but should be more efficient.

```
void insert(Tree *root, Tree new_node)
{
    Tree temp = *root;
    if (temp == NULL) // empty tree special case
        *root = new_node;
    else {
        for (;;) {
            if (new_node->data <= temp->data) { // go left?
                if (temp->left == NULL) { // leaf?
                    temp->left = new_node; // insert it
                    return; // finished
                }
                else
                    temp = temp->left; // go left
            }
            else { // going right
                if (temp->right == NULL) { // leaf?
                    temp->right = new_node; // insert it
                    return; // finished
                }
                else
                    temp = temp->right; // go right
            }
        }
    }
}
```

I'm sorry, Professor! Your recursive code is short and beautifully elegant, but mine is longer, uglier, and faster! Maybe I shouldn't tell my Professor that I've never coded a binary tree since finishing my degree?

Hash tables are the name of the game.

Eliminating tail recursion

Recursion is rarely a good solution, but some types of recursive algorithms are not easy to change to loops, because they would require a stack data structure to do so. If a stack is needed, there may be little gain in removing recursion fully — it depends on how efficiently recursion is implemented by the compiler on the builtin C++ function call stack, versus your skill in hand-coding a stack data structure.

In these situations, a simpler optimization is still possible without a stack. Partial recursion elimination without the need for a stack is possible via the elimination of "tail recursion." Tail recursion occurs when the last action of the recursive procedure is to call itself.

A simple modification changes this last recursive call to become a loop back to the top of the current invocation. For example, consider the preorder traversal of a binary tree. The simplest recursive algorithm is:

```
void preorder(node_ptr root)
{
    if (root != NULL) {
        visit(root);
        preorder(root->left);
        preorder(root->right); // Tail recursion here
    }
}
```

Tail recursion can be eliminated by replacing the `if` statement with a `while` loop. The transformation effectively reduces recursion by half, as the second recursive call is eliminated. This reduction in recursion is achieved with virtually no extra overhead!

```
void preorder(node_ptr root)
{
    while (root != NULL) { // while loop replaces if
        visit(root);
        preorder(root->left);
        root = root->right; // Move to right subtree
    }
}
```

Replacing recursion with a stack

Some recursive algorithms cannot be easily replaced by iterative loop equivalents. For example, in the preorder binary tree traversal above, we were unable to remove both of the recursive calls. In these situations, recursion can be replaced with an algorithm using a stack data structure.

All recursive algorithms can be replaced by a stack because recursive algorithms are actually using an implicit stack (the program stack of function calls). Whether use of a stack will be more efficient than recursion depends on a number of factors. The choice of a stack over recursion is machine-dependent. In particular, it is quite likely that the program stack is supported by efficient low-level instructions and that (recursive) function calls are executed very efficiently. Can you do better?

On the other hand, recursion requires that much information be stored on the stack (i.e., parameters, automatic local variables, machine registers), whereas an algorithm making use of an explicit stack will usually only need to store a few items, making it potentially faster than the function call stack. If the maximum size of the required stack is known beforehand, a stack can be quite efficiently implemented as an array, whereas a dynamic stack as a linked list will usually be more costly because of the cost of memory allocation.

The following shows the preorder traversal with tail recursion elimination removing one recursive call and an explicit stack replacing the other. In this case, the explicit stack need only store pointers.

```
void preorder(node_ptr root)
{
    stack_type S;
    init_stack(S); // set to empty stack
    while (root != NULL || !is_empty_stack(S)) {
        if (root != NULL) {
            visit(root); // visit a tree node
            push(S, root->right); // save right subtree
            root = root->left; // go to left subtree
        }
        else
            root = pop(S); // get node from stack
    }
}
```

Collapsing recursive calls. If you can't be bothered changing a recursive algorithm to a loop or stack, here's a smaller optimization to consider. By channeling the spirit of loop unrolling, we can "collapse" one or more levels of recursion into sequential code. The method of "function call collapsing" can be applied to recursive functions in this limited sense. Obviously, it isn't possible to collapse a recursive function call completely into inline code, but it is possible to collapse a few levels of recursive calls at a time, reducing the total number of recursive calls by a constant factor.

Moving the recursive base case higher. The simplest method is to test the base case one level higher up. In the simple implementation of the preorder traversal , the recursive base case is "`root==NULL`". If this occurs, the function call does nothing. One simple method of avoiding these unnecessary function calls is to test for the base case *before* the recursive call. The new function becomes:

```
void preorder(node_ptr root)
{
    while (root != NULL) {
        visit(root);
        if (root->left != NULL) // Test moved up
            preorder(root->left);
        }
        root = root->right;
    }
}
```

Collapsing multiple levels of recursion. By converting multiple levels of recursive calls into sequential code, the function does much more work each time, but makes recursive calls less frequently, thereby reducing function call overhead. For example, the preorder traversal can be rewritten so that the current node and its two children are handled by the function, and then recursive calls are made for any of the children's children:

```
void preorder(node_ptr root)
{
    if (root != NULL) {
        visit(root);
        if (root->left != NULL) { // do left child
            visit(root->left);
            preorder(root->left->left);
            preorder(root->left->right);
        }
        if (root->right != NULL) { // do right child
            visit(root->right);
            preorder(root->right->left);
            preorder(root->right->right);
        }
    }}
```

But alas, we've reverted here to a fully recursive version again, just to show function call collapsing. The above method should also be combined with (a) tail recursion elimination, and (b) a stack data structure. This is left as an exercise for the reader (thankfully), and as a project scope estimate, I suggest two weeks!

Use Parameters as local variables

Parameters to functions can be used as if they were local variables. Because of C++ call-by-value parameter passing of basic types (not arrays), the modification of a parameter inside the function does not change the values of any variables not local to the function. This method saves on initialization time, and on stack space. In the example below, to zero an array, the size is counted down, rather than having a local variable counting up.

```
void zero_array(int arr[], int n)
{
    while (n > 0)
        arr[--n] = 0;
}
```

This code also has the optimization of "looping down to zero". Note that we have to be careful that this code doesn't access `arr[n]`, but does correctly clear `arr[0]`. I think it works correctly, but my brain is on fire trying to check it.

Pass function parameters by reference

Passing objects or large parameters by value is an inefficiency. The C++ language provides a very convenient method of achieving pass-by-reference, by simply using `&` in the parameter declaration. One method of improving efficiency is to pass objects to functions as reference parameters.

Behind the scenes, pass-by-reference is like passing a single pointer as the parameter. This avoids not only the cost of copying a large object onto the stack, but also the cost of the copy constructor and destructor for the object within the function (i.e., the parameter is a separate object when passed by value).

A function parameter can be changed to use pass-by-reference parameters only if it does not change the object. Fortunately, modifications to parameters can be detected simply by qualifying the parameter declaration with `const`, thus forcing the compiler to warn about any modifications to the object within the function. An example of the use of reference parameters in the definition of a `Complex` object is shown below:

```
class Complex {
    double r, i;
  public:
    Complex & operator += (const Complex & c);
    // c is passed by reference for efficiency
    // The return type is also a reference
};

Complex & Complex::operator += (const Complex & c)
{
    r += c.r; // add to both data fields
    i += c.i;
    return *this; // return reference to updated object
}
```

Const reference parameters. Passing the argument by reference improves efficiency by avoiding big objects. Note that the parameter is declared "`const`" as well as "`&`" indicating a reference. This "`const&`" pattern is the common C++ idiom for simulating a non-modified pass-by-value object send into a function as a faster reference type.

Returning References. This code also has a second optimization: reference return types. Making the return value a reference is also efficient, because the `return` statement does not invoke the copy constructor. Note that a returned reference is necessary only if the user of the `Complex` class uses complicated expressions such as `x+=y+=z`. If such expressions are not required, efficiency can be improved by making the return type `void`.

Objects Only. The use of references is best limited to class objects, and also to structures and unions. Arrays are already passed by reference in C++ and hence there is no need to change them. The use of references for scalar types (integers, `float`, `double`, and pointers) is unlikely to give much improvement, if any, and might even be slower for some.

Pitfall: Temporary Objects. Another disadvantage of using reference parameters for scalar types like "`int`" is the inefficiency caused if a constant value is passed as an argument (i.e., a number not a variable). Paradoxically, passing a constant argument to a reference parameter is not an error in C++, but instead a new temporary object with this type is created automatically by the compiler and its address passed.

Implicit "this" object. Note that the object to which a member function is applied is already passed by reference in a certain sense, because it is using the implicit "`this`" parameter. Hence, the simple types of member function calls are already efficiently using a hidden type of pass-by-reference of the object itself. Consider this code:

```
int MyClass::fn() // member function
{
    return x;
}
```

It is not faster with a non-member `friend` function call that uses an explicit reference parameter. This code will not be more efficient (and is probably less efficient):

```
int fn(MyClass & object) // friend function
{
    return object.x;
}
```

Specialize functions with default arguments

Every default function argument is a place where you can optimize. Default arguments to functions are not a source of inefficiency in themselves, and cost no more than using a fixed-argument function and passing some constants explicitly. However, the use of default arguments indicates the possibility of improving efficiency by replacing a single function with a number of specialized functions.

How to do this? Instead of one function with a default argument, create two functions using function overloading. The specialization of the function into two separate functions will often make other optimization techniques possible, thus improving overall efficiency at the cost of some duplication of executable code. As an example of the possibilities that can exist, consider the function with default arguments:

```
void indent(int n = 4) // default argument n=4
{
    for (int i = 0; i < n; i++)
        cout.put(' ');
}
```

Rewriting this single function as one general function and one specialized function leads to opportunities for optimization in the specialized function.

In this case, loop unrolling can be employed:

```
void indent() // Specialized function (n=4)
{
    cout.put(' '); // Loop completely unrolled
    cout.put(' ');
    cout.put(' ');
    cout.put(' ');
}

void indent(int n) // General function
{
    for (int i = 0; i < n; i++)
        cout.put(' ');
}
```

Note that this optimization is also limited in scope, as there any need to change any other code that calls the functions. The C++ compiler will automatically make the correct choice of which overloaded function to call. Another thought for improved readability is to name the specialized function differently (e.g., `indent4`), which requires calls to the function to be changed. However, default arguments are certainly convenient and the slight increase in efficiency should be balanced against the loss of good programming style.

Medium-Sized Slugs

There are a lot more examples of possible inefficiencies in C++ coding. Some of the types of errors that are "medium-sized" slugs include:

- Automatic array initializations with constant data.
- Loop test function calls (i.e., expensive loop conditional tests).
- Member initializations in the constructor body (they should be in the initializer lists).
- Program startup hidden initializations (global or `static` object constructors).
- Small non-`inline` functions called frequently.
- Busy wait loops.
- Unnecessary code inside loops.
- C++ classes wrapping simple data types (e.g., overuse of "smart pointers" or "smart integer" classes).
- Overuse of standard string concatenation operations.
- Recursion is almost always a slug.

Automatic Array Repeated Initialization

A simple example of unnecessary double initializations is any type of large local variable, such as an automatic array. When a function makes use of a large array variable with constant data, or even a large constant object, the variable should probably be declared as both "`const`" and "`static`", even if it need not retain its value between calls. Consider the following code example:

```
char *convert(int day)
{
    char *days[] = { "Monday", "Tuesday", "Wednesday",
                     "Thursday", "Friday",
                     "Saturday", "Sunday" };
    return days[day];
}
```

The initialization of array "`days`" illustrates an inefficiency. The initialization for "`days`" occurs every time the convert function is entered. It would be much more efficient to declare "`days`" as a `static` variable to avoid it being re-initialized, and also "`const`" to help the compiler optimize.

Data Structure Double Initialization

If you have an initialization routine that does a lot of work, it sometimes becomes a slug by accident. I'm not talking about a single variable initialization, but the initialization of a large program data structure at startup, like a precomputed lookup-table or a perfect hashing algorithm. In the design patterns vocabulary, such a situation is a "singleton" data structure, where only a single object ever exists in the program. It's easy to lose track of whether its initialization routine has been called, and then it gets called twice (or more!).

An example would be some of the precomputation methods whereby a large lookup-table is initialized at program startup. For example, a 24-bit lookup table has been used elsewhere in this book to optimize AI activation functions such as GELU.

The way to avoid the slug of double-initialization is simply to track calls to the initialization routine.

The idiom that I use is a local `static` variable of type `bool` at the start of the initialization function:

```
    static bool s_once = false;
    if (s_once) {
        aussie_assert(!s_once);  // Should be once only
        return;  // Avoid double intialization!
    }
    s_once = true;
```

Another way is to actually count the calls with an integer, which is a generalization that works for additional scenarios:

```
    static int s_calls = 0;
    ++s_calls;
    if (s_calls > 1) {
        aussie_assert(s_calls <= 1);
        return;  // Avoid double intialization!
    }
```

You can wrap these multiple lines of source code up into a single "`aussie_assert_once`" macro, if you want a simpler method.

Singleton global objects. If you've done the hard yards to declare a big data structure like this as its own class, then you can simply instantiate only one object (i.e., as a global). The C++ class infrastructure does well in ensuring that a constructor is only called once. Even so, it may be worthwhile to declare a `static` data member and use similar logic to ensure that initialization on this object isn't ever done twice.

In any of these situations, it's a worthwhile investment of a couple of CPU instructions, an increment and a test, to avoid accidentally running the whole routine again. Since the code is virtually identical for all cases, to avoid copy-paste typos, you could even hide these few statements behind a standard C++ preprocessor macro with a name of your choosing Or you could even use an `inline` function with the "`return`" statement changed to throwing an exception.

Busy waiting for input

Humans are very slow compared to computers. In particular, a computer can do much work in the background, even when handling the (slow) interactive input of a human.

Hence, one method of improving efficiency is to perform background processing while awaiting input, instead of using blocking input that waits for a keypress before doing anything. In other words, you can't use std::cin or scanf for non-blocking keypress polling.

A common example of this idea is chess-playing programs that "think" during their opponent's time. The computer can continue its game-tree analysis while waiting for the player to press a key or click a mouse. The C++ standard provides no simple standardized function for non-blocking input. In general, there are two ways:

- Keyboard polling API calls (non-portable).
- Multi-threading with input on one thread and processing on another.

There are various non-portable ways to poll for key presses. For example, on Windows there's the "`_getch`" or "`kbhit`" functions (also "`_kbhit`"), which are all deprecated. Assuming you've found a workable polling API call, at some regular interval, perhaps before each node of the game tree is analyzed, the chess program checks if a key has been pressed. If a key has been pressed, the chess program stores information about its current analysis, and processes the user's keystroke. Unless the key press completes the user's move, the background analysis can continue after processing the key.

Overall, there's no simple and standardized way to do non-blocking input in C++. This is probably because of C's ancestry, where it was difficult to poll the keyboard on a traditional UNIX line terminal. Multi-threading can be used in C++ to achieve the result instead.

Slow disk I/O

The cost of performing I/O on disk files can make up a large proportion of the run-time cost of some programs. For reducing the amount of data to be read from or written to the disk, the main methods are:

- Use smaller records.
- Cache frequently used records.
- Buffer multiple reads or writes.
- Compress data.
- Use better data structures.

A very simple method of reducing disk I/O is to reduce the size of records being read or written. This can be achieved using many of the methods to create smaller objects. There are various methods in C++ to reduce a class object's byte size: unions, bit-fields, packing, smaller data types, or reordering data members.

Caching is useful if some records are being read more often than others. It is a very general idea and there are many possible implementations. You can even create your own caching mechanism.

It may be possible to keep all of the most frequently used records in main memory, writing them to disk only at the end of the program (even caching records in memory and writing them to disk for every modification will still avoid the cost of multiple disk reads).

If this method cannot be used, try using several memory locations for record I/O, and whenever a read operation is required, examine these in-memory records first. If any of them is the required record, the cost of a disk read is avoided. Caching always has a slight overhead, and may increase run-time slightly if the desired records are rarely in memory; however, it will never increase the amount of disk I/O and the computational overhead is likely to be small compared to the cost of reading a record from disk.

When reading or writing multiple contiguous records, disk I/O can be speeded up by reading in a number of records each time. The advantage is that buffering multiple operations reduces the number of disk seek operations. For example, when using `<stdio.h>`, the buffering can be changed using the `setbuf` and `setvbuf` functions.

Another alternative is to use other low-level I/O functions, such as the Linux `open`, `read` and `write` functions. However, this method reduces portability of the code.

When the amounts of data being read are quite massive, the level of disk I/O can be reduced by compressing the data in the file. Read and write operations then have the overhead of uncompressing or compressing the data, but the cost of this computation may well be less than that of the disk I/O (or it might also be more; be careful!). However, methods of compressing data are beyond the scope of this book.

The use of a different data structure for data in disk files is often worthwhile. In particular, if the disk file is being searched, then many search algorithms are applicable. For example, binary search can be performed on a direct access file if the data is sorted.

However, even binary search is inefficient for large disk files, and data structures specifically intended for disk data should be used. The B-tree is a commonly used data structure, and hashing is another possibility. Unfortunately, these algorithms are highly advanced and again beyond the scope of this book.

Incorrect choice of loop

Although the choice of loop is largely a matter of style, there is an important difference between the post-tested "`do`" loop, and the pre-tested "`for`" and "`while`" loops. The loop condition of a `do-while` loop is not evaluated on the first iteration and the loop body is always executed at least once. However, a `for` or `while` loop condition is evaluated before the first iteration and the loop body need not be executed at all. A common form of minor inefficiency is declaring loops that are always executed the first time, such as:

```
    bool done = false;
    while(!done) {
        // ....
    }
```

It is more efficient to use the `do` loop, which avoids a single evaluation of the loop condition:

```
    bool done = false;
    do {
        // ....
    } while(!done);
```

The use of the correct type of loop is also helpful to the optimizer. It is valuable to know that a code segment is always executed once.

Infinite loops are control flow structures that can also be detected and used by the optimizer. Hence, you should code an infinite loop explicitly by using one of the common idioms:

```
    for(;;)          // Forever
    while(1)         // Common
    do..while(1)     // Not commonly used
```

This allows the compiler to generate efficient code, because you've made it easy for the compiler to recognize the loop as infinite.

Exit loops and functions early

Control structures should be exited as soon as possible, including function paths and loops. This means judicious use of return for functions and break or continue for loops.

Using "return" as early as possible in a function is efficient. It prevents unnecessary code being executed. Testing for edge cases at the start of a function is an example of using the return statement to do "easy cases first" or "simple cases first" optimizations.

Exit loops early. Similarly, both break and continue are efficient, as no more of a loop is executed than is necessary. For example, consider the code using a Boolean variable "done" to indicate the end of the loop, as in:

```
done = false;
while (!done) {
    ch = get_user_choice();
    if (ch == 'q')
        done = false;
    else
        ... // rest of loop
}
```

The faster code has a break statement used to exit the loop immediately:

```
while (1) { // Infinite loop
    ch = get_user_choice();
    if (ch == 'q')
        break; // EXIT EARLY!
    else
        ... // rest of loop
}
```

Unfortunately, the overuse of jump statements such as break and continue can make the control flow of a program less clear, but professional C++ programmers are used to these statements being used often.

More Slug Repellent

There's plenty of other optimizations in the other chapters on compile-time optimizations, code transformations, loop optimizations, and AVX vectorization. Well, actually most of the book! Nevertheless, here's a list of some more C++ code optimization techniques for you to consider. Some of the bigger ideas:

- Use "move constructors" instead of copy constructors where appropriate (since C++11).
- Use `static` data members where appropriate, so they are initialized once only.
- Use `std::sort` rather than `qsort`.
- Don't put `try`..`catch` inside an inner loop that's a bottleneck.
- Use `std::bitset` for bit sets or bit vectors.
- Use the "iterators" design pattern rather than returning a full scan of a data structure all at once (saves memory and allows early exit).
- Consider basic C++ arrays instead of `std::vector` if it has a fixed size (known at compile-time) or its maximum size is small enough.
- Consider C++20 coroutines where appropriate for the architecture.
- Structure of arrays (SoA) data layout is more vectorizable than the Array of Structures (AoS).

And some of the smaller optimizations:

- Commonly used object or struct fields should be first. On some platforms, smaller offsets from the start of an object are accessed faster. Also, the very first field has offset zero, which is optimized away, so put the most used field first.
- Avoid long `else-if` sequences. You are effectively doing linear search on the problem space in a long block of `if-else-if` statements. The best alternative is to use a `switch` statement, if the conditions are constants. For non-constant conditions or string comparisons, consider tabularizing the options and/or using heuristics to bifurcate the search space (e.g., start with a `switch` on the first letter of a string).
- Use compact numeric ranges for `switch`. If the case numbers are close together, the compiler will probably use a lookup-table in assembler. If the cases are sparse, it can be forced to do an `if-else-if` equivalent in machine code.
- Correct choice of loop. If the condition is true at the first iteration, use `do-while` loops.
- Instead of range checking a signed integer with "`i>=0 && i < MAX`" use a typecast with "`(unsigned)i<MAX`" because negatives become large

unsigned positives, and a cast from `int` to `unsigned int` isn't a real instruction at run-time.

- Enable the FTZ ("flush-to-zero") and/or DAZ ("denormals-are-zero") floating-point modes on your CPU, even though they violate the IEEE 754 standard. You probably don't care about tiny floating-point numbers in your weight or probability calculations.
- Enable GCC's floating-point arithmetic speedup options: `-ffast-math`, `-fno-math-errno`, `-fno-trapping-math`, and `-ffinite-math-only`.
- `bsearch` is slow. Choose a better method.
- Use `static_assert` rather than `assert` (e.g., to check data type sizes).
- Copy arrays by wrapping them in a dummy `struct` and using C++ `struct` bitwise assignment. It might be faster than `memcpy`.
- Use `memcpy` rather than `memmove` if you're sure the arguments won't overlap.
- Move local non-`static` objects outside of a critical loop. Reuse the same object rather than re-running constructors and destructors every loop iteration. Add a "`reset`" member function if needed.
- Use scaling factors that are a power-of-two, so that multiplication or division can be a bitshift.
- Specialize a function with a `void` and non-`void` version if you find yourself ignoring the return value sometimes. This avoids all of the calculations to determine the return value inside the `void` function, because the function itself cannot tell whether or not the caller will use its return value.
- Prefer pre-increment (`++i`) to post-increment (`i++`) for non-scalar values. And it's better to use pre-increment even for "`int`" types, even though it's the same, just to get into the habit.
- Use the GCC `__builtin_unreachable()` statement and the "`noreturn`" function attribute to help the GCC optimizer identify dead code paths, allowing unreachable code removal (not that we care that much) and also better optimization of path-specific optimizations on other live paths (e.g., compile-time constant propagation).
- Test the first character of two strings directly with character tests before calling `strcmp`.
- Replace calls to "`round`", "`floor`" or "`ceil`" functions with a type cast to `int` (as an approximation).
- Consider using the simpler `putchar`, `putc`, `fputc`, `puts`, `fputs` functions rather than `printf` or `fprintf`.
- Write your own versions of `abs` and `fabs`/`fabsf` (but benchmark it).
- Avoid the floating-point `pow` function for computing integer powers.
- Instead of `strlen("literal")` declare it as an initialized `char[]` array variable and use `sizeof(arr)-1`.

- Merge a large number of function parameters into an object. Don't pass 10 Boolean flags as differently named function parameters. Create an object or structure and make them fields instead.
- Avoid calling `strlen` in a "`for`" loop conditional.
  Compute `strlen` before the loop, or test for the null byte.
- Merge multiple Boolean function parameters into a bit set. packed into an `int` or `long`. The gain from passing fewer values as function arguments will be offset by the cost of packing and unpacking bits, but still should be better.
- Use `int` type mostly, not `char` or `short`. Maybe prefer `int` to `size_t`, too.
- Specialize functions being called with a constant for an argument using a template function with an integer field. This will increase code size, but the constant will be propagated more at compile-time, and you also don't have the cost of passing it as an argument.
- Add "`noexcept`" specifiers to functions wherever it applies, because this allows the compiler to know not to worry about adding any extra exception handling code.
- If you're "searching" an array or set of constant integers, known at compile-time, consider "proceduralization" by putting the numbers as cases in a `switch`. (Trust the compiler engineers.)
- Consider writing your own faster `atoi`/`itoa` functions, as the standard libraries need to handle lots of rare cases, making them slower. (I'm not sure I agree and you might want to benchmark.)
- Don't overuse "`alignas`" to specify address alignments if you don't need them, as the enforcement of alignment requirements can impose runtime cost.
- `sprintf` is a slow and unsafe function. `snprintf` is safer but still slow. Find another way.
- Post-increment can be faster in pointer arithmetic, so prefer using the normal idiom "`*ptr++`" rather than "`*++ptr`" to scan a vector.

References

1. Agner Fog, 2023, *Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms*, PDF: https://www.agner.org/optimize/optimizing_cpp.pdf
2. Kurt Guntheroth, 2016, *Optimized C++: Proven Techniques for Heightened Performance*, O'Reilly Media, https://www.amazon.com/dp/1491922060
3. Dov Bulka and David Mayhew, 1999, *Efficient C++: Performance Programming Techniques*, https://www.amazon.com//dp/0201379503
4. Fedor G. Pikus, 2021, *The Art of Writing Efficient Programs: An advanced programmer's guide to efficient hardware utilization and compiler optimizations using C++ examples*, Packt Publishing, https://www.amazon.com/dp/1800208111
5. ISO/IEC, Feb 15, 2006, *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E), https://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (Design of the C++ language from an efficiency perspective, including discussion of virtual functions and other language features.)
6. Nicolai M. Josuttis, 2012, *The C++ Standard Library: A Tutorial and Reference*, Second Edition, Supplementary Chapter, https://www.amazon.com/Standard-Library-Tutorial-Reference-2nd/dp/0321623215, PDF (extra chapter): http://www.cppstdlib.com/cppstdlib_supplementary.pdf (C++ optimizations such as bit sets and user-defined memory allocators.)
7. Bjarne Stroustrup, 2013, *The Essence of C++ with examples in C++84, C++98, C++11, and C++14*, PDF Slides: http://www.staroceans.org/e-book/essenceOfC++.pdf
8. Wikibooks, 2023, *Optimizing C++/Writing efficient code/Performance improving features*, Wikibooks, https://en.wikibooks.org/wiki/Optimizing_C%2B%2B/Writing_efficient_code/Performance_improving_features
9. Dave Abrahams et. al., 2003, *Technical Report on C++ Performance*, http://web.archive.org/web/20040608203404/http://www.research.att.com/~bs/performanceTR.pdf
10. Jakob Engblom, 2001, *Getting the Least Out of Your C Compiler*, https://www.engbloms.se/publications/engblom-esc-sf-2001.pdf
11. Jon Louis Bentley, 1982, *Writing Efficient Programs*, Prentice Hall.
12. Thomas Plum and Jim Brodie, 1985, *Efficient C*, Plum Hall Inc.
13. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, 1986, *Compilers—Principles, Techniques and Tools*, Addison-Wesley.
14. Donald E. Knuth, 1973, *The Art of Computer Programming (Vol. 3): Sorting and Searching*, Addison-Wesley.
15. James O. Coplien, 1992, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley.
16. Jonathan S. Shapiro, 1991, *A C++ Toolkit*, Prentice Hall.
17. Bjarne Stroustrup, 1991, *The C++ Programming Language (2nd edition)*, Addison-Wesley.

Appendix C: Source Code

Tester Object Instrumentation Class

This code is for "object instrumentation" that can be useful for performance analysis, and also for debugging and unit testing.

Here's a test usage to see what constructors and move operations are performed by `push_back` in the `std::vector class`:

```
Tester::reset_counters();
std::vector<Tester> vectest4;
for (int i = 1; i <= 100; i++)
    vectest4.push_back(i);
Tester::print_report();
```

Here's the full code:

```
class Tester {
private:  // Static data members
    static bool traceall_;
    static int count_default_constructor;
    static int count_copy_constructor;
    static int count_move_constructor;
    static int count_copy_assignment;
    static int count_move_assignment;
    static int count_destructor;
    static int count_int_constructor;

private:  // Object data members
    int ival_;
    bool trace_;
public:
    Tester() {
        ival_ = 0;
        count_default_constructor++;
        trace_ = false;
        if (traceall_) {
            cout << "Tester: default constructor: "
                 << ival_ << endl;
        }
    }
    Tester(int val) {
        count_int_constructor++;
        ival_ = val;
        trace_ = false;
        if (traceall_) {
            cout << "Tester: int constructor: "
                 << ival_ << endl;
        }
    }

    Tester(const Tester &other)  // Copy constructor
    {
        ival_ = other.ival_;
        trace_ = other.trace_;
        count_copy_constructor++;
        if (trace_ || traceall_) {
            cout << "Tester: copy constructor: "
                 << ival_ << endl;
        }
    }

    Tester(Tester&& other) noexcept  // Move constructor
    {
        ival_ = other.ival_;
        trace_ = other.trace_;
        other.ival_ = -1;  // Invalidate moved data
        count_move_constructor++;
        if (trace_ || traceall_) {
            cout << "Tester: move constructor: "
                 << ival_ << endl;
        }
    }

    Tester& operator=(const Tester& other)  // Copy assign
    {
        count_copy_assignment++;
        if (this != &other) {  // Avoid aliasing
            ival_ = other.ival_;
            if (trace_ || traceall_) {
                cout << "Tester: copy assignment: "
                     << ival_ << endl;
            }
        }
        else {
            if (trace_ || traceall_) {
                cout << "Tester: copy assignment aliasing: "
                     << ival_ << endl;
            }
        }
        return *this;
    }
```

```
Tester& operator=(Tester&& other) noexcept  // Move
{
    count_move_assignment++;
    if (this != &other) {  // Avoid aliasing
        ival_ = other.ival_;
        if (trace_ || traceall_) {
            cout << "Tester: move assignment: "
                 << ival_ << endl;
        }
    }
    else {
        if (trace_ || traceall_) {
            cout << "Tester: move assignment aliasing: "
                 << ival_ << endl;
        }
    }
    other.ival_ = -1;  // Invalidate moved data
    return *this;
}

~Tester()
{
    count_destructor++;
    if (trace_ || traceall_) {
        cout << "Tester: destructor: " << ival_ << endl;
    }
    ival_ = -1;  // Safety
}

// Equality operators
bool operator==(const Tester& other) {
      return ival_ == other.ival_;
}


// Setters for object members
void trace(bool bval) { trace_ = bval; }

// Setters for static data members
static void traceall(bool bval) { traceall_ = bval; }
static void reset_counters() {
    count_default_constructor = 0;
    count_copy_constructor = 0;
    count_move_constructor = 0;
    count_copy_assignment = 0;
    count_move_assignment = 0;
    count_destructor = 0;
    count_int_constructor = 0;
}
```

```
    static void print_report() {
        cout << "Tester Count Report" << endl;
        cout << "- Default constructor: "
             << count_default_constructor << endl;
        cout << "- Int constructor: "
             << count_int_constructor << endl;
        cout << "- Copy constructor: "
             << count_copy_constructor << endl;
        cout << "- Move constructor: "
             << count_move_constructor << endl;
        cout << "- Copy assignment: "
             << count_copy_assignment << endl;
        cout << "- Move assignment: "
             << count_move_assignment << endl;
        cout << "- Destructor: "
             << count_destructor << endl;
    }

    static void selftest() {
        // Constructors should equal destructors
        // ... but move constructors don't increase count
        int errors = 0;
        int total_constructors = count_default_constructor
               + count_int_constructor
               + count_copy_constructor;
        if (total_constructors != count_destructor) {
            if (total_constructors > count_destructor) {
                cout << "Tester selftest: constructors ("
                     << total_constructors
                     << ") more than destructors ("
                     << count_destructor << ")" << endl;
                errors++;
            }
            else {
                cout << "Tester selftest: destructors ("
                     << count_destructor
                     << ") more than constructors ("
                     << total_constructors << ")" << endl;
                errors++;
            }
        }

        if (errors == 0) {
            cout << "Tester selftest: no errors found"
                 << endl;
        }
    }
};
```

```
// Define Tester static data members
bool Tester::traceall_ = false;
int Tester::count_default_constructor = 0;
int Tester::count_copy_constructor = 0;
int Tester::count_move_constructor = 0;
int Tester::count_copy_assignment = 0;
int Tester::count_move_assignment = 0;
int Tester::count_destructor = 0;
int Tester::count_int_constructor = 0;
```

Intercepted new and delete

This source code is the global scope intercept functions for the `new` and `delete` operators. The library tracks basic statistics about calls and bytes allocated.

```
// Global counters
unsigned long int s_new_count = 0;
unsigned long int s_newarr_count = 0;
unsigned long int s_delete_count = 0;
unsigned long int s_deletearr_count = 0;
unsigned long int s_new_bytes = 0;
unsigned long int s_newarr_bytes = 0;

void memory_reset_counters()
{
    s_new_count = 0;
    s_newarr_count = 0;
    s_delete_count = 0;
    s_deletearr_count = 0;
    s_new_bytes = 0;
    s_newarr_bytes = 0;
}

void memory_report()
{
    cout << "MEMORY CALLS REPORT" << endl;
    cout << "- new calls: " << s_new_count << endl;
    cout << "- new[] calls: " << s_newarr_count << endl;
    cout << "- delete calls: " << s_delete_count << endl;
    cout << "- delete[] calls: " << s_deletearr_count
         << endl;
    cout << "MEMORY SIZE REPORT" << endl;
    cout << "- new bytes: " << s_new_bytes << endl;
    cout << "- new[] bytes: " << s_newarr_bytes << endl;
}

void* operator new(size_t n)
{
    s_new_count++;
    s_new_bytes += n;
```

```
        return malloc(n);
    }

    void* operator new[](size_t n)
    {
        s_newarr_count++;
        s_newarr_bytes += n;
        return malloc(n);
    }

    void operator delete(void* v)
    {
        s_delete_count++;
        free(v);
    }

    void operator delete[](void* v)
    {
        s_deletearr_count++;
        free(v);
    }
```