Advanced C++ Memory Techniques

Efficiency and Safety

David Spuler

Aussie AI Labs

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

https://www.aussieai.com

First published: June 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.

Preface

Why a Book on C++ Memory?

Memory is everything! It's space, it's speed, and it's many bugs. There are so many aspects to using memory optimization techniques in C++, while avoiding all the pitfalls, that the humble RAM chip full deserves its own book.

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

If you want fast code, here are a number of other books on efficient C++ coding:

- Efficient Modern C++ Data Structures: Container and Algorithm Optimizations
- C++ Low Latency: Multithreading and Hotpath Optimizations
- Safe C++: Fixing Memory Safety Issues

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

- 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

Although I hope the information is useful to you, neither the content nor code in this work is guaranteed for any particular purpose. Nothing herein is intended to be personal, medical, financial or legal advice. You should make your own enquiries to confirm the appropriateness to your situation of any information.

Many code examples are simplistic and have been included for explanatory or educational benefit, and are therefore lacking in terms of correctness, quality, functionality, or reliability. For example, some of the examples are not good at handling the special floating-point values such as negative zero, `NaN`, or `Inf`.

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

Table of Contents

Part I: Memory Optimization Techniques

1. C++ Memory Primitives

Memory Functions

Compiler vendors provide a variety of useful library functions to help with memory safety. Some of these are defined in C++, whereas others are platform-specific. The main classes of functions include:

- Heap memory management
- Stack memory management
- Text string buffer management

If we want to manage memory safely, we first need to examine all the different ways that a C++ program can get some memory.

The main long-standing heap management functions are:

- `malloc`
- `calloc`
- `free`

And in C++ there are the basic operators:

- `new` — object or primitive type allocation.
- `delete` — de-allocation operator.
- `new[]` — array allocation version.
- `delete[]` — array de-allocation.

Some other ones for allocating C-style strings:

- `strdup`
- `_strdup`
- `Strndup`

And there's also the rarely-used standard functions:

- `std::allocate` (and custom allocators).
- `realloc` — resize a heap block, possibly moving it.
- `alloca` and `_alloca` — dynamically allocate a stack block of memory.
- `mmap` — memory-mapped blocks representing disk files.
- `sbrk` — low-level allocation of memory to processes.

Stack Memory Management

It is less commonly used, but possible, to dynamically allocate stack memory. Functions include:

- `alloca` — the main stack memory allocation function (`<alloca.h>`.
- `_malloca` — stack memory allocation (Microsoft CRT)
- `_freea` — free memory on the stack or heap (Microsoft CRT)
- `__builtin_alloca_with_align` (GCC version with alignment)

Note that "de-allocation" of a stack block is technically not required, because the memory is reclaimed when the function returns and the stack is unwound.

Platform-Specific Memory Management

There are also a variety of platform-specific and newer functions. The main header files are:

- `<stdlib.h>` — standard memory allocation functions.
- `<malloc.h>` — Linux or Windows.
- `<crtdbg.h>` — C++ Run-Time debug (Microsoft MSVS).
- `<Strsafe.h>` — Microsoft MSVS.

The platform-specific or newer memory-related functions include:

- `_expand` (MSVS) — lengthen a heap block, in place, without moving it.
- `_malloc_dbg` (MSVS) — debug versions of basic memory primitives in `<crtdbg.h>`.
- `reallocarray` — array version of `realloc`.
- `free_sized` (C23)
- `set_new_handler` and `get_new_handler` (C++11)

Address Alignment

One of the main problems with memory primitives was handling of alignment. The standard methods of achieving alignment in C++ include:

- `alignas` specifier
- `__declspec(align(N))`
- `aligned_alloc` (C11/C++17)

Some other functions and language features include:

- `_alloca` (aligned version).
- `free_aligned_sized` (C23)
- `_aligned_malloc` (Microsoft CRT)
- `_aligned_realloc` (Microsoft CRT)
- `_aligned_free` (Microsoft CRT)
- `_aligned_msize` (Microsoft CRT)
- `_aligned_offset_malloc` (Microsoft CRT)
- `_aligned_offset_realloc` (Microsoft CRT)
- `posix_memalign` (POSIX)
- `_aligned_storage` (deprecated)
- `std::aligned_storage`
- `aligned_union` (deprecated)
- `alignment_of`
- `_Alignas`

Unix and Linux Memory Management

There are a variety of Linux memory management primitives available via GCC, mostly defined in `<malloc.h>`:

- `malloc_usable_size` — size of an allocated memory block.
- `mallinfo`, `mallinfo2` — get allocated memory block information.
- `malloc_info` — exports XML info about the heap state.
- `malloc_stats` — allocation statistics.
- `mallopt` — set memory allocation options (e.g., can control how glibc handles a double-`free` error.)
- `getrlimit` and `setrlimit` — manage resource limits, including the heap.

Some of the other non-standard memory functions in early Unix and Linux include extra versions with bit flag controls:

- `mallocx`
- `rallocx`
- `xallocx`
- `sallocx`
- `dallocx`
- `sdallocx`
- `nallocx`

There are also "memory allocation control" and other memory allocation primitives in older UNIX and Linux:

- `mallctl`
- `mallctlnametomib`
- `mallctlbymib`
- `malloc_stats_print`
- `malloc_usable_size`
- `malloc_message`

Windows Memory Management

Windows has a variety of additional functions, some in `<Strsafe.h>` and others are in the C++ Runtime (CRT) functions and its debug versions in `<crtdbg.h>`:

- `_malloc_dbg` and other "debugging heap" versions (`<crtdbg.h>`).
- `_CrtCheckMemory` — check heap for integrity.
- `_CrtSetDbgFlag` — control debug flags.
- `_CrtMemState` memory block structure in crtdbg.h
- `_heapmin` — reclaim some heap memory (heap minimize).
- `_heapadd` — increase heap size.
- `_heapchk` — self-test heap for consistency.
- `_heapset` — fill all unallocated heap memory with a canary byte!
- `_heapwalk` — traverse through the heap blocks.

Windows has a feature that I especially like: callbacks for memory allocation operations! Here are the details:

- `_CrtSetAllocHook` — set a "hook" (callback) when allocation occurs.
- `_CrtGetAllocHook`

There are also various other calls about memory addresses:

- `_CrtIsMemoryBlock` — check addresses.
- `_CrtIsValidHeapPointer` — check heap addresses.
- `_CrtIsValidPointer`
- `_CrtReportBlockType`

Size of Allocated Heap Block

There's no standardized way to take the address of a heap block and return its size. This is unfortunate, because that would be helpful in several ways for memory safety. Hence, there are platform-specific versions:

- `_msize` — Windows MSVS version.
- `malloc_usable_size` — GCC version.
- `malloc_size` — MacOS version.

Note that the size of the memory block returned from these functions is not necessarily the same as the original size of the request. It shouldn't be less, but it can often be larger, because the system memory allocator has padded out the allocated block for alignment or other optimization reasons. When you run simple tests, it will probably appear to always be the correct size, but after a longer execution with a lot of allocations and deallocations, the algorithm for the memory allocator can get trickier, and it may vary significantly.

If a platform-specific block size function does return a larger value for the block size, it's not easy to know this has occurred. Hence, don't assume that this size value will point exactly to your redzone area, or whatever other tricks you're doing with the end of your allocated memory blocks.

As a further warning, note that `_msize` on Windows is a little fragile, because it throws a runtime exception if the address is either:

(a) a non-heap address, or

(b) not the start of a heap address.

Hence, it's not that useful in testing whether a random address is a heap block or not. Maybe it needs to be combined with `_CrtIsValidHeapPointer`.

C++ Compiler Hardening

Personally, I think that C++ compilers should have extra modes that harden the language. There are some standardization efforts to create a "hardened standard C++ library" and these are laudable, but there are language-specific issues that only the compiler can fix.

As computers have gotten faster, the relative cost of addressing these issues becomes relatively low against the expense of tracking memory issues. Some of the areas where the compiled code could be safer and tolerate issues include:

- `malloc` and `new` should zero memory (like `calloc`).
- `alloca` should also zero memory.
- `realloc` should zero any extra allocated memory areas.
- `new` and `delete` operators should be interchangeable with `malloc` and `free` (e.g., `free` on a `new` block should work, although it won't run any destructor).
- `new/delete` should also work with the `new[]/delete[]` array versions (though this also misses some destructors).
- Stack variables should be zeroed when a function starts (like global variables).

There are a lot of little "undefined" areas that are glitches in the standard C++ library, which probably should be detected and tolerated, or at least warned about, by the library functions instead:

- `std::list` crashes if deleting an object during an iterator scan.
- `fflush(stdin)` should be detected and tolerated.
- Mismatched `fread/fwrite` on a file without intervening `fseek` would be easy to detect.
- `strncpy` should have a warning when it truncates and leaves the string without a null.
- `cos` or `sin` of a number larger than two pi probably means the caller has confused radians and degrees.
- `strlen(NULL)` should not crash.

We can "fix" some of these issues by defining our own intercepted versions of these functions, either via using our own wrapper function names instead, or via automatic preprocessor macro intercepts or link-time changes.

2. 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 total cache locality of 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 binary 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 constant globals (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 for 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 — this is dynamically allocated by the `new` operator or the `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/

3. 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 maintains a warm cache, so that when the hot path is executed for real (e.g., a trade execution in HFT), then memory accesses are very fast.

There are multiple ways to trigger the prefetching of data needed 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 with the "cache lines" of 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. 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 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 only 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/

4. 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 tries hard to predict which of the two paths of a branch is more likely to be taken. Some 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 (e.g., the newer `[[likely]]` and `[[unlikely]]` specifiers).
- Keep unavoidable branches in the same 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()` condition marker (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)

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.
- Tricks can confuse the optimizer (undermining any benefit).

The 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
- Lookup tables
- Conditional move (CMOV) assembly statements
- Ternary operator (`?:`)

Some of the more traditional C++ optimizations techniques can also reduce branching:

- 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;
```

I'm betting that it's actually slower, since multiplication is an expensive operation, but who's to know without 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;
```

Again, the ternary operator's CMOV instruction is probably faster than this de-optimization.

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.

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). We can generate this using the right bitshift operator:

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

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

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 basic idea for a branchless, 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. 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 with 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.

Instruction Reordering Optimizations

Instruction reordering 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 undesirables 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. Some CPUs can 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.

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. Hence, I haven't quite solved the problem of a faster vector dot product using instruction out-of-order execution.

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. GNU, May 2025 (accessed), Adjusting the Instruction Scheduler, https://gcc.gnu.org/onlinedocs/gccint/Scheduling.html

5. 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 many 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 in a simple contiguous array of memory. Less obviously, similar comments apply to the 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 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 of the 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 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 use 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]; // 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 of 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 awful 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 (regardless of 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)
{
    // Bug!
    return memcmp(&a, &b, sizeof(MyComplex)) == 0;
}
```

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 in the computations, 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 might have to manually intercept all of the `new` and `new[]` operators with your own wrapper that does `memset` on the block, rather than use constructor tricks.
- 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-memory 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 would like the determinant function to look:

```
double determinant(double matrix[][], int n); // Wrong
```

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 to cater 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.

6. Pointer Arithmetic

What is Pointer Arithmetic?

Pointer arithmetic is a tricky C++ optimization that can be used to get rid of 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 are 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 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 creating 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++.

7. 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 of 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, one of the user's objects, and another as the free list (with either indices, pointers, or permutations).

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]; // Ptr 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 of `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 obj 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.

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 has 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 obj 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 object
    };
private:
    alignas(alignof(T)) std::array<Node, N> arr_;
    int freelist_[N]; // array of free indexes (stack)
    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 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) {
   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. Everything in this index array version has constant-time efficiency.

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 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 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 all the arguments to the 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 with custom allocators is not just to enhance the 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 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 `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 (or iginally 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)

8. 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 that is dynamically allocated using the `new/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 the storage of 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 values containing the arguments and also 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 standard C++ is to use `std::bitset`. Note that bit-fields are a form of 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 compilers support the special "`#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 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 in overall bytes 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 very large and long functions that do 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 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). If 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 used depends on the size of blocks, the number of blocks and how quickly allocated blocks are deallocated. The size of blocks 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 ways to avoid this pattern. Small amounts of data could possibly be stored in extra fields of the object.

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` member operations in `std::vector` can lead to 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 and manage that process. Overall, this can massively reduce the number of allocated blocks.

Memory fragmentation. Reduce memory fragmentation by reducing both allocations and deallocations. It's also important to manage the different sizes for 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 their requests for more storage, using some reasonable incremental block size, rather than attempting to guess 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.)

9. 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 the newer special standard 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 60 bytes in 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

Part II: Memory-Efficient Data Structures

10. 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 a sorted 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 use with 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; }
```

Sorted Arrays

There is no standard C++ sorted array class, so we've got to implement our own. A sorted array has a good search lookup cost, being logarithmic in the 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 good because sorting an unsorted array is expensive with an *O(n log n)* complexity (e.g., `std::sort` typically uses a quicksort-heapsort hybrid).

If we need sorted data, there are other options in 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 used 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` — alternative for moving or copying right (i.e., create a gap for insertion).

Note that using `std::copy` or `std::copy_backward` functions also work here, but copying 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.

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.

Unsorted Arrays

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.

The use of binary search reduces our searches to logarithmic complexity, which is much faster than linear search.

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 need to benchmark it!

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 even 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 single integral and floating-point 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 arithmetic
}
```

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.

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, the 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 efficient with constant time complexity.

The code 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. 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 error");
    }
    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 array");
    }
    else if (count_ == 1) { // ***
        if (!std::is_trivially_destructible<T>::value) {
            arr_[0].~T(); // Explicit destructor (if needed)
        }
        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) {
            // Explicit destructor (at end)
            arr_[count_ - 1].~T();
        }
        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.

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 loop 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!

Extensions

1. Benchmark the unsorted array implementation above using a raw array type versus an alternative approach of using a `std::vector` member object to store the data.
2. 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.
3. 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.

4. Explore the efficiency of calls to move constructors in a "shuffle" for a sorted array implemented using `std::vector` or `std::array`.
5. 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.)
6. Benchmark inserting into an unsorted array and then sorting using `std::sort`, because 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)?
7. Implement a hybrid binary-linear search where the binary search reverts to linear search once the interval is small enough.
8. Implement an AVX SIMD version of linear search over integers that tests a number of integers in the array at once.
9. 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.
10. 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.

11. 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 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 from the start of the string.

Multi-Character Search. If you want to search for the prefix or suffix with a set of 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 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*` strings which are only the size of a pointer. In fact, a string object typically contains a small buffer for short strings that is packed into the object itself.

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 << "Char buffer start string (off=0)" << endl;
    }
    else if (offset + bytes == sizeof(std::string)) {
        cout << "Char buffer end string (offset = "
             << offset << ")" << endl;
    }
    else {
        cout << "Char 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
Character 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!

12. 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).

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 of 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.

13. 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 of `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 — efficient hashed 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 `std::list` container separately (yes, 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 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

14. 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 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 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:

- 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 to be faster. 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 assnt
    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, maybe we can call the `reserve()` function 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 type
```

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 allocation
```

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 as data 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!

15. Loop Optimizations

Sequential vs Parallel Loop Optimizations

Loop optimizations are the basic of many speedups to the processing of contiguous array data. Loops are often sources of inefficiency and can be optimized in numerous ways, such as:

- Cache locality — process data in fast order for CPU caches (sequential).
- Parallelization — allow vectorization via CPU SIMD instructions or GPU.

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 — repeat the loop body to reduce loop test overhead and parallelize the loop 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 the inner/outer loop iterators of nested loops.
- Loop reordering — change the ranges and arrangements of inner/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 a subset of 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 loop body into two loops.
- 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 we don't know a better way to initialize a vector in C++ like `memset` or `calloc` or load-time `static` variable initialization.

Loop Perforation

The intentional introduction of randomness to code is known as a "stochastic" algorithm. Personally, I'm more familiar with the idea of 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 some types of applications, such as 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) {
            continue; // Skip it... perforated
        }
        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.

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 many applications, we could choose our data structure sizes 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. But it's not great for vectorization as it's likely to confuse the compiler, so 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 note that there are no `case` statements with 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 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 multiplications.

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 declared: `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; // 4x4 tile size
    const int TILEY = 4;
    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 to improve the efficiency of MatMul's and thereby get better throughput of tensor calculations from a GPU. Matrix multiplication is a good candidate for this optimization because it has complexity of *O(n^3)* arithmetic calculations, but uses only *O(n^2)* data. Hence, a naive matrix multiplication algorithm that doesn't address cache 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 in the loop of 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

Loop reordering is the general class of optimizations that involves reordering loops or their iterations. In complex algorithms, there are many loops, and many ways for nesting them, or running them in sequence. Such optimizations can involve changing the ordering of two sequential loops or two nested loops.

The reordering optimization to reverse the inner and outer nested loops is more often called "loop interchange." One loop can be reordered with "loop reversal."

Loop reordering is an optimization that doesn't reduce the total number for 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 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 in 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_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_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" of 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, or 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 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

16. Vector Algorithms

Vector Dot Product

Vector dot product is an algorithm that has received a lot attention lately, because it's the most basic computation algorithm in an AI engine. All of the tensor operations and matrix multiplications break down into instances of a dot product calculation. The dot product is so-named because its mathematical notation is a dot. It is also known as the "scalar product" because its result is a scalar (single number), rather than a vector.

The vector dot product takes two vectors as input, and computes a single `float` number. The algorithm is a product of the elements of each vector, added together. Here's the code:

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

Properties of the dot product include:

- Two vectors as input.
- Scalar output (single number).
- Can be positive or negative.
- Is zero if either vector is all zeros.
- Can also be zero for two non-zero vectors (e.g., if the vectors are "perpendicular" in 2-D or 3-D space).
- Has a physical meaning related to the "angle" between the two vectors.
- Is an integer if both vectors contain integers.
- Dot product of a vector with itself is the square of the vector's magnitude (equivalently, the vector's L2-squared norm).
- Is very slooow. Dot product-based operations inside matrices and tensors are the main culprit for AI needing all those GPUs.

The dot product differs from the "vector product" of two vectors (also called "cross product") that returns a vector, and is a completely different mathematical operation. The vector cross product is interesting mathematically in that it computes a vector perpendicular in 3 dimensions, but it's not very useful in practical applications. The dot product is where the action's at in big tensors.

Vector Norms

Vector norms are measurements of vectors that indicate features of a vector. For example, we can measure if two vectors are "close" to each other. Again, these used to be obscure linear algebra algorithms, but are now widely used in various AI algorithms.

Vector norms map vectors to a single number. Note that vector norms are not the same thing as the "normalization" layer in a Transformer (i.e., LayerNorm or BatchNorm). Note also that a vector "norm" is not at all related to the similarly-named "normal vector" (a vector perpendicular to a surface). The *norm* is a number, whereas the *normal* is a vector, and they're not on speaking terms since that incident last summer.

L2 Norm: The basic norm of a vector is the level-2 (L2) norm, and you probably already know it. This is the length of the vector in physical space, also called the vector's "modulus" or "magnitude" in Mathematics 101. If you treat a vector as a "point" in space, the L2 norm is its straight-line distance from the origin.

The calculation of the L2 norm of a vector is a generalization of Pythagoras's Theorem: sum the squares of all the vector elements, and then take the square root. The code looks like:

```
float aussie_vector_L2_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += (v[i] * v[i]);   // Square
    }
    return sqrtf(sum);
}
```

Because we square every element, they all get turned positive. Zero squared is still zero. Once we've summed all the squares, we usually get a big positive number, which we then square root to get a smaller positive number. Hence, the result of the L2 norm is compressing a whole vector down to a single positive floating-point number.

The properties of the L2 norm are:

- Floating-point number (e.g., 0.567 or 5.6789 or 3.0 or whatever)
- Positive number (not ever negative)
- Zero only if the whole vector is zero.
- Represents the "length" (or "modulus" or "magnitude") of a vector, called the "Euclidean distance".
- Usually a non-integer, even if the vector was all integers.

For a simple 2-D or 3-D vector in Senior Math, the L2 norm is the physical length of the vector in 2-D or 3-D space (or the length of the line from the origin to the equivalent point). For AI, which has vectors in 1024-dimensions, or N-dimensional vectors for whatever N is being used, there's not really a physical explanation of the L2 norm that's easy to visualize, but it's kind of a measure of the length of the vector in N-dimensional space. The value of the L2 norm can be zero, but only if all the vector's elements are zero.

Note that the value of the L2 norm is not unique. Two different vectors can have the same value for the L2 norm. In fact, an infinite number of vectors can have the same value, and those vectors are the set of vectors with the same length (magnitude), which will define a sphere in N-dimensional space.

L2-squared norm: A minor modification of the L2 norm is the "squared L2 norm", which is, as you may have guessed, the square of the L2 norm. To put it another way, it's just the L2 norm without the square-root at the end. The code looks like:

```
float aussie_vector_L2_squared_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += (v[i] * v[i]);  // Square
    }
    return sum;  // NOT sqrtf(sum);
}
```

The value of the L2-squared norm is a positive number, but a much larger one. The physical meaning is the square of the physical/Euclidean length of the vector. The L2-squared norm also equals the vector's dot product with itself.

Why use the L2-squared norm? Because it's faster to skip the square-root operation, of course. Also, if the vector contains integers, then the L2-squared norm is also an integer, which can make it even faster to compute in integer-only mode. The L2-squared norm is just as good as basic L2 for some uses.

The properties of L2 and L2-squared norms are very similar except that one is a much larger number. Both are positive and related to Euclidean distance, and both increase monotonically the further the vector is away from the origin.

Level 1 Norm: As you can guess from my calling it the L2 norm, there's also an L1 norm, and there's L3 norms, and more. Let's look at the L1 norm, because it's even simpler, although it's *not* usually something that's covered when studying vectors in Math class.

The L1 norm is simply the sum of the absolute values of all the vector elements. We don't square them. We don't take the square root. We just make them positive and add them up. The code looks like:

```
float aussie_vector_L1_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += fabsf(v[i]);   // Absolute value
    }
    return sum;
}
```

Using the absolute values of elements reverses any negative vector elements to positive. The absolute value ensures the whole total can't go negative, and any negative value also adds to the total. A zero element is fine in the vector, but does nothing. The result of the L1 norm is a single positive float number, which can be fractional or whole, ranging from zero to as high as it goes (i.e., if you have big numbers in the vector elements, then the L1 norm will also be large).

The properties of the L1 norm are:

- Floating-point number (fractional or whole).
- Positive number (never negative).
- Zero only if all vector elements are zero.
- Physical meaning is an obscure distance measure (the "Manhattan distance").
- Will be an integer if the vector elements are integers.

What does an L1 norm mean? It's kind of like the distance you'd travel if you walked the longest way by going along each element/dimension of the vector, one at a time, and not going backwards (no negatives). So, the L2 norm was the fastest diagonal direct way to get to a point, but the L1 norm is going the scenic route, and the L1 norm is usually bigger than the L2 norm.

Like the L2 norm, the L1 norm is not unique. Multiple vectors can have the same L1 norm. For example, the vectors `(1,2)` and `(0.5,2.5)` will have L1 vector norm of 3.0. I'm not really sure what the set of all the vectors with the same L1 norm means. Maybe it's this: all the points that you can walk to from the origin when you travel a certain distance (going forwards-only)?

L3 Norms and Above: The mathematical vector norms can be generalized to L3 and higher norms, even to infinity. For an L3 norm, you cube all the vector elements (made positive by absolute value), and take the cube root at the end. It's tricky to find the cube root in C++ until you remember that a cube root is exponentiation to the power of 1/3 (from Year 10 math), so we can use the "`powf`" function. Here's the code:

```
float aussie_vector_L3_norm(float v[], int n)
{
    float sum = 0.0f;
    for (int i = 0; i < n; i++) {
        sum += (v[i] * v[i] * v[i]);  // Cube
    }
    const float frac_third = 1.0f / 3.0f;
    return powf(sum, frac_third);
}
```

Can you guess what an L4 norm is? The higher order versions are really fun and interesting if you wear socks with your sandals, but not very useful in any practical applications of AI coding.

Matrix Norms

There are norms for matrices, but they're not really that often used. Taking a "measurement" of a matrix via a "norm" (or a "metric") to compare it to other matrices isn't a common task.

The silly ones are element-wise matrix norms. You can define an L1 or L2 norm on a matrix using the same algorithm over all its elements. You can also find the maximum element inside a matrix, and call that the "max norm" if you like to sound math-ish. The reason I say these are dumb? Because they ignore the structure in the matrix, so it's a kind of "pseudo-norm" of a matrix. It's really just treating a matrix like it's a big, flat vector, and to me it seems more like misusing a vector norm on a matrix.

More sensible matrix norms consider the rows or columns of the matrices as separate vectors. An NxN matrix has N column vectors or N matrix vectors, so there are N vector norms. Should we add them up?

No, taking the *maximum* of the vector-wise L1 or L2 row/column vector norms has a more useful meaning as a matrix norm than the element-wise matrix L1 or L2 pseudo-norms. You can do this maximum-of-vector-norms either for rows or columns, but not both.

Vector Min and Max

Finding the maximum or minimum element of a vector is useful, and somewhat relevant to the L1/L2 norms. The maximum is a kind of "metric" of the size of a vector. Also, the maximum function over a vector is used in "greedy decoding" to pick the word with the highest predicted probability, which is then output. The minimum function would give us the least likely word, which might also be interesting if useless.

The simple linear code for vector max is:

```
float aussie_vector_max(float v[], int n)  // Maximum
{
    float vmax = v[0];
    for (int i = 1 /*not 0*/; i < n; i++) {
        if (v[i] > vmax) vmax = v[i];
    }
    return vmax;
}
```

The vector minimum function looks similar in sequential C++ code:

```
float aussie_vector_min(float v[], int n)  // Mininum
{
    float vmin = v[0];
    for (int i = 1 /*not 0*/; i < n; i++) {
        if (v[i] < vmin) vmin = v[i];
    }
    return vmin;
}
```

These functions are crying out for optimizations: loop unrolling, pointer arithmetic, etc. However, what they really need is vectorization. There are parallelized *max* and *min* primitives in GPUs and CPU-based AVX intrinsics that you can use.

Top-K Vector Algorithm

The top-k algorithm is more complicated than vector max or min: find the largest *k* elements in a vector. Note that "maximum" is the same as top-k with *k=1*. If you want the short version of the top-k story in C++, there's a `std::partial_sort` standard function that sorts the top k elements, and there's also `std::sort` for a full array sort. However, let's hand-code some top-k algorithms for more clarity.

Note that the top-k algorithm is a somewhat obscure algorithm that used to be rarely used, but now it's a very important piece of code in AI engines. It gives us "top-k decoding" which is how to choose which word to output. The whole encoder-decoder computes a vector giving us the probability that each word should be output next. Using the maximum probability word gives us "greedy decoding" which always outputs the most likely word. But that's kind of boring and predictable, so top-k decoding randomly chooses between the *k* most likely words (e.g., top-50), which is still very accurate and also more interesting because it has some creative variation.

Example: Hand-coded top-2 algorithm: Since top-1 is the maximum of a vector, we can also find a fairly simple linear scan for *k=2*. The basic idea is to scan through and keep track of the two largest values as we go.

```
void aussie_vector_top_k_2(
    float v[], int n, float vout[])
{
    // Order the first 2 elements
    float vmax1 = v[0], vmax2 = v[1];
    if (v[1] > v[0]) {
        vmax1 = v[1];  // Reverse them
        vmax2 = v[0];
    }
    for (int i = 2 /*not 0*/; i < n; i++) {
        if (v[i] > vmax2) {
            // Bigger than the smallest one
            if (v[i] > vmax1) { // Bigger (shuffle)
                vmax2 = vmax1;
                vmax1 = v[i];
            }
            else {
                // In the middle (fix 2nd only)
                vmax2 = v[i];
            }
        }
    }
    vout[0] = vmax1;  // Biggest
    vout[1] = vmax2;  // 2nd biggest
}
```

Note that the above top-2 algorithm is still impractical for our word decoding algorithm. We need to know not only the top probabilities, but also which two indices in the vector had those probabilities, because that's how we know which words map to which probabilities. So, we'd need to modify the above code to track and return the two array indices as well (or instead).

Shuffle Top-K Algorithm

For a larger value of *k* the code becomes more complicated. The above code for *k=2* motivates the general idea for a brute-force algorithm: shuffle sort the first *k* elements, and then scan the rest, shuffling any larger items up into place. We can merge the two shuffling phases into one block of code that handles both the startup and ongoing scan.

```
void aussie_vector_top_k_shuffle(
    float v[], int n, int k, float vout[])
{
    vout[0] = v[0];
    int nout = 1;
    for (int i = 1 /*not 0*/; i < n; i++) {
        float fnew = v[i];
        int maxj;
        if (nout < k) {
            vout[nout++] = fnew;
            maxj = nout - 2;
        }
        else {
            maxj = nout - 1;
        }
        maxj = nout - 1;
        for (int j = maxj; j >= 0; j--) {
            if (fnew > vout[j]) {
                // Shuffle & insert
                if (j + 1 < k)  // Shuffle down
                    vout[j + 1] = vout[j];
                vout[j] = fnew;
                // Keep going
            }
            else {      // Done.. insert it
                if (j != maxj) {
                    if (j + 1 < k)
                        vout[j + 1] = vout[j];
                    vout[j] = fnew;
                }
                break;
            }
        } // end for j
    } // end for i
}
```

The above example is a simplistic and inefficient top-k algorithm, not to mention that it was horribly fiddly and failed my unit tests for hours (i.e., that's a special kind of "fun"). Several loop optimizations suggest themselves: loop sectioning for the outer `i` loop to do the first `k` iterations as a separate loop (avoiding lots of tests against `k`), and loop peeling of the first iteration of the inner `j` loop (i.e., `j==maxj`). This version also should be extended to track the indices from where the top-k values came.

Theoretical Top-K Algorithms

There's a lot of theory about computing the top-k function of an array for large *k* values. These theoretical top-k algorithm papers mainly consider sequential processing, rather than vectorization. Even so, it's not a simple linear scan like *max* or *min* functions, but doesn't need to be as slow as shuffling.

Example: Top-k with qsort sorting: The simplest method for large *k* is to sort the array with a fast method (e.g., the quicksort algorithm) and then pick off the top *k* elements from the sorted array. In C++ there are the `std::sort` methods or the older style `qsort` function. Here's an example using the C++ standard `qsort` function:

```
int aussie_top_k_qsort_cmp(
    void const* addr1, void const* addr2)
{
    float f1 = *(float*)addr1;
    float f2 = *(float*)addr2;
    if (f1 < f2) return +1;  // Reversed (descending)
    else if (f1 > f2) return -1;
    else return 0;
}

void aussie_vector_top_k_qsort(
     float v[], int n, int k, float vout[])
{
    // Top-k with general k (qsort algorithm)
    // Sort the array
    qsort(v, n, sizeof(vout[0]),
                  aussie_top_k_qsort_cmp);
    // Copy top-k elements
    for (int i = 0; i < k; i++) vout[i] = v[i];
}
```

Top-k with qsort and permutation array: We really need a version that returns the indices of the probabilities, rather than just their values. So, I coded up a `qsort` version that sorts via a permutation array, and then returns the top-k of these permutation indices.

```
void aussie_permutation_identity(int permut[], int n)
{
    for (int i = 0; i < n; i++) permut[i] = i;
}

float* g_float_array_for_qsort = nullptr;

int aussie_top_k_qsort_permutation_cmp(
    void const* addr1, void const* addr2)
{
    int index1 = *(int*)addr1;
    int index2 = *(int*)addr2;
    float f1 = g_float_array_for_qsort[index1];
    float f2 = g_float_array_for_qsort[index2];
    if (f1 < f2) return +1; // Reverse (descending)
    else if (f1 > f2) return -1;
    else return 0;
}

void aussie_vector_top_k_qsort_permut(
    float v[], int n, int k,
    float vout[], int permut_out[]
)
{
    // Create a dynamic permutation array
    int* permut_arr = ::new int[n];
    // Identity permutation
    aussie_permutation_identity(permut_arr, n);

    // Sort the array (by permutation)
    g_float_array_for_qsort = v;
    qsort(permut_arr, n, sizeof(permut_arr[0]),
       aussie_top_k_qsort_permutation_cmp);
    // Copy top-k elements
    for (int i = 0; i < k; i++) {
            permut_out[i] = permut_arr[i];
            vout[i] = v[permut_arr[i]];
    }
    delete[] permut_arr;
}
```

Top-k without sorting: Sorting the whole array is somewhat wasteful if we only want the top 50 elements. There are various faster top-k algorithms that don't fully sort the array. These algorithms are called a "partial sort" and can achieve the top-k output with better performance

Standard C++ top-k libraries: As mentioned earlier, the standard C++ libraries have support for sorting algorithms in `std::vector`, such as with:

- `std::sort` — full array sort (simplest idea).
- `std::partial_sort` — partial sort of *k* elements (faster).

There is a top-k specialized version in the modern C++ libraries called `std::partial_sort`, which sorts the top *k* elements of an array, which can then be selected for the top-k algorithm.

Presumably, the `std::partial_sort` function is a faster algorithm than `std::sort`, by not fully sorting the whole array, but I haven't tested it. There is also `std::nth_element`, which is similar to top-k.

17. Tensors

What are Tensors?

Tensors are terrifying at first! I avoided learning about them for ages. All those nested loops are scary. But eventually it dawned on me that they're just three-dimensional arrays, and the computations are nothing harder than multiplication and addition.

An important point is that "tensors" in Computer Science are much different to the mathematical forms used in Physics. AI tensors are used in "linear algebra" for LLMs and are much more basic than the 4-D space-time tensors in Einstein's theory of general relativity. Which may explain why all those brainy physicists are so smug, despite being unable to predict if it'll rain tomorrow.

Tensors are simply multi-dimensional arrays, and are usually 3-dimensional. Each slice of a 3-D tensor is a two-dimensional matrix. And like vectors and matrices, tensors have these basic properties:

(a) Each element stores a single number (i.e., no strings or objects).

(b) All elements have the same data type (e.g., `int` or `float`).

(c) Elements may be positive, negative or zero.

(d) There are no missing elements. The concept of "missing" can only be represented by zero in a normal tensor.

There are exceptions, of course. There are "sparse tensors" that can represent elements as missing. Also, you can technically store strings or objects in a C++ three-dimensional array, but then it's more of a misuse of a tensor. Numbers are where it's at.

Tensors are technically the superset of all of the computational structures, and the number of dimensions is called the "rank" or "dimension" or "axes" of a tensor. Matrices are rank-2 tensors, vectors are rank-1 tensors, and even scalars are rank-0 tensors.

Conceptually, there's a hierarchy of complexity for tensor operations:

- 3-D tensor operations break down into 2-D matrix multiplications.
- 2-D matrix multiplications break down into vector dot products.
- 1-D vector dot products break down to a single `float` number (a scalar).
- 0-D scalars are single numbers.

Another way to think about tensors is in terms of nested loops. Scanning a vector requires one loop, and a matrix needs two nested loops. Tensor operations require three or more nested loops to process all their data.

Neural Network Tensors

I'm not going to take you in detail through the theory of how neural networks function. But in broad strokes, there are "neurons" in layers, where each neuron has a "signal," and there are also connections between neurons that forward the strength of a signal on to the next layer of neurons. So, each neuron in connected to *every* neuron in the previous layer by an "arc" and on that arc is a "weight" that says how strong or weak to consider the incoming neuron's signal.

But how do we get to tensors from that? Not obvious.

Let's step back a little and be one with the neuron. So, we are just one neuron in a layer of 100 neurons. And the previous layer has 100 neurons, and we are "fully connected" with arcs from every one of those 100 prior neurons. With 100 neurons in the previous layer, our little lonely neuron has to consider the signals from all of the 100 neurons in the prior layer, with 100 weights on the arcs to help decide how much attention to pay to each of the 100 prior neurons.

If we consider the previous layer of 100 neurons as a "vector" of each neuron's computed values. What this means is that every one of the 100 prior neurons has a number of its computed signal, so we have a vector of 100 signal numbers from the prior layer (i.e., a vector full of 100 neuron computed values).

Again, our little neuron has to receive a computed signal value from every one of the 100 prior layer neurons, so we have 100 arcs coming into our little neuron, each with a different number, that is the "weight" of that arc. The computed value of a prior neuron is multiplied by the "weight" that's on each arc (i.e., there's 100 weights, one for each arc). So, every one of the arcs from the 100 neurons in the prior layer has a weight, and what does that sound like? A vector of weights.

So, we have a bunch of 100 prior-layer neuron's computed values in a vector, where each one of those 100 signal values is multiplied by a weight that's in a vector with 100 weights. Hence, we've got to pairwise multiplication, where we multiply 100 neuron values times 100 associated weights. Hence, we've got a bunch of element-wise multiplications of two vectors (100 values times 100 weights), which creates a vector of 100 multiplication computations.

But our little neuron cannot have 100 computed values, but can really only have one number, the total computed signal for our current neuron. There are various things we could do to "reduce" our interim vector of 100 multiplications, but the simplest is to add them all up, and this gives us one number. Now we have one number, and it's the computed signal value for our current neuron.

Umm, I remember that from High School. If we multiply two vectors together with the numbers in pairs, and then add it all up: *vector dot product*.

In summary, we have a vector dot product for our single neuron in the current layer, based on two vectors from the prior layer (the vector of 100 calculated neuron values, and the vector of 100 weights).

But this is just for our one lonely neuron. Except, it's not lonely, since it has 99 friends, because it's in a layer of 100 neurons itself. So, our neuron and its 99 friends in the current layer, all have to do a different dot product computation because the weights are different for each set of arcs into each neuron. We have a whole vector of 100 neurons in the current layer, for which we have to compute dot products for 100 values times 100 weights (i.e., using the prior layer). So, we have to do 100 vector dot products to calculate the result for our neuron and its 99 friends. If we do 100 repetitions of vector dot products, this sounds like...*matrix multiplication*.

But that's not all. There's a third dimension based on the "tokens" in the prompt, which is represented by an "embeddings" vector. And with this third dimension thrown in, well, then it's a whole vector worth of matrix multiplications, and we get to a 3-D operation called a "tensor product." Tensors are three-dimensional blocks full of numbers (i.e., cubes or rectangular prisms), which generalize two-dimensional matrices, which generalize one-dimensional vectors, which generalize zero-dimensional scalars. And if you have any common sense, you've stopped reading this section by now, so I'm not going to try explaining this mind-bending tensor stuff any further.

Tensor Arithmetic

Tensors are a convenient and efficient representation of multi-dimensional data. Since complex computations may involve a lot of matrix multiplications, it is useful to represent a sequence of matrix operations as a tensor operation.

Importantly, the arithmetic performed is the same. Using a tensor is computationally efficient for parallelization of algorithms, and also mathematically concise for theoretical analysis, but is not some fantastically amazing matrix algorithm. It's just crunching lots of numbers with the standard matrix multiplication methods. Usually, it's the same as an array of matrices, where you do matrix multiplication on each one.

In practice, tensor kernels will send out different chunks of that computation all over the place for parallel speedup, but it's still computing the exact same numbers as if you did it all brute-force in nested loops. You could even follow along with a pen and paper, except that the computer is better because it won't forget to carry the negative sign.

Tensor shape. Another point is the shape of a tensor. I'm sure you know that matrices may be square or rectangular in shape, but can't be a skewed parallelogram or a circle. Yes, you're right, there are triangular matrices, but now you're messing up my nice clean point.

Anyway, a 3-D tensor can have different sizes on each of its three dimensions. Hence, a 3-D tensor can be a cube if all three sizes are identical, but usually they have the shape of a more general rectangular prism. And it still has a brick-like shape, and can't really represent a triangle, cone, or sphere. Tensors are much less scary if you sing *Everything is Awesome* while you code the nested loops.

Unary Tensor Operations

Like a 2-D matrix, there are various simple operations we can define on a single tensor. The various element-wise operations apply individually to each tensor item.

- Clear or set to a value
- Add or subtract a scalar
- Multiply or divide by a scalar

Similarly, we could apply a particular unary mathematical function to each element separately: square root, exponentiation, natural logarithm, and more.

Binary Elementwise Tensor Operations

Adding two matrices means simply adding each pair of elements in the matrix, which only works if the two matrices have the same size and shape. The same idea generalizes to the addition of tensor elements of two tensors with the same size (i.e., all three dimensions are the same). Hence, we can do element-wise binary arithmetic on each element in two tensors to create a third tensor of the same size:

- Addition or subtraction
- Multiplication or division
- Maximum or minimum

Note that element-wise multiplication of tensor elements is not "tensor multiplication" in the same way that matrix multiplication isn't just paired multiplications of the elements in two matrices. Such an element-wise multiplication is called the "Hadamard product" of matrices, and is so useless that I don't think I was ever taught that in High School. The Hadamard product is not what is used by normal multiplication computations, but I've seen a few research papers where it was proposed as an optimization (probably unsuccessfully). Matrix multiplication is more complex, with its row-by-column vector dot product multiplications, and so is generalizing that to tensors.

That's how we get to "tensor product" of two tensors. It's really just nested loops doing matrix multiplications on slices of each tensor. And then matrix multiplications are just nested loops doing vector dot products. Like I said, tensors are just three-dimensional arrays doing multiplication and addition.

Sparse Tensors

Sparse tensors occur when most of the values are zero. These are a generalization of sparse vectors and sparse matrices, and offer the same advantages: compressed storage and faster arithmetic operations (by skipping operations involving zero).

The level of sparsity required for optimization usually means 80-90% of the weights are zero. With so few non-zero values, tensor arithmetic involves fewer operations and the memory requirements are low (i.e., store only the non-zero weights). Such sparsity is often the result of a "pruning" optimization, but there are also obscure theoretical means to get sparse tensors using tensor algebra (let's not even go there!).

When there is a high degree of sparsity, such as when 80-90% of the values are zero, it becomes more efficient to use alternative algorithms. Sparse tensors can be stored in a permutation index format, where only the index locations of non-zero

items are stored (e.g., storing a four-tuple with the non-zero value and the three indices at which it is located in the tensor). Operations on sparse tensors can use the alternative storage format to create much more efficient kernels that avoid most of the computations involving the missing zero values.

Parallelization of sparse tensor operations is a double optimization, because there are fewer operations (only on non-zero weights), and you can parallelize them as well. Although a permuted index data format is not the usual contiguous memory space amenable to vectorization, there are other methods to vectorize permutation indices, such as with "gather" and "scatter" SIMD operations.

18. 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 algorithms 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 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 index accesses (e.g., crashes on negatives or larger values).

3. `unsigned int` types might overflow and spin for precomputing tables of size "`1<<32`" (need to 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_precomp_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 of 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,
1.79366203433576585078237386661109264803 9e-43f,
3.58732406867153170156474773322218529607 7e-43f,
5.38098610300729755234712159983327794411 6e-43f,
7.17464813734306340312949546644437059215 5e-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, can be null
)
{
     // 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;
                // No extra comments for file output version
                add_commented_number = false;
        }
        unsigned int u = 0;
        fprintf(fp, "// Precomputed table source code: %s, \"%s\"\n",
nickname, outfname);
        fprintf(fp, "float g_gelu_table_precompute_24bits[] = { \n");
        char numbuf[5000] = "";
        for (; u < maxn /*1<<24*/ ; u++) { // For all 2^24=~16.7M
                unsigned int uval = u << 8;  // put zeros in the least
significant 8 mantissa bits
                float f = AUSSIE_UINT_TO_FLOAT(uval);
                float g = fnptr(f);  // Call GELU or whatever
                if (arrout) arrout[u] = g;  // Store precomputed data

                // Format: %g means the smaller of %e or %f
                // ... %e is the exponent format (scientific-like format)
                char* buf = numbuf;
                // Format %g (Number) and suffix "f" (float constant)
                sprintf(buf, "%40.40gf", g);
                if (strchr(buf, 'n')) {
                        // Nan or "-nan" ...
                        strcpy(buf, "0.0 /*nan*/"); // Dummy for 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, but leave 'f'
                                            buf[len - 1] = 'f';
                                            buf[len] = 0;
                                        }
                                        else {
                                            buf[len - 1] = 0;  // remove
                                            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) {
                // Progress to stdout...
                if (writingfile)
                    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

19. 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 vector has *N* elements. The resulting 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 is a dot product of a matrix row with the input. Again, 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 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 of 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.

Spot the Buggy MatMul

Have a look at this code for a matrix-vector multiplication using vector dot product. It took me a long time to realize what was wrong with this. Can you spot the bug?

```
void aussie_matmul_vector_basic1_buggy(
          ymatrix m, float v[], int n)
{
    // Basic matrix-by-vector using vector dot products..
    for (int i = 0; i < n; i++) {
        float* rowvector = &m[i][0];
        // Dot product
        float sum = aussie_vecdot_basic(rowvector, v, n);
        v[i] = sum;
    }
}
```

The bug is a kind of aliasing problem here:

```
v[i] = sum;  // Bug!
```

It looks correct, but it's wrong. The computation of `v[i]` is setting its value in the middle of the loop, and then going around for the next matrix row, which will then use that newly calculated `v[i]` value as if it was part of the input vector. Because I'm misusing "v" as both the input and output vector, parts of the output vector will get used as the input vector. It's a very insidious type of aliasing bug, and many of my simple unit tests with zero matrices and identity matrices were still succeeding. It's my fault for trying to do matrix-vector multiplication as an element-wise vector method. The solution is simple: matrix-vector multiplication needs a third operand for the output vector.

Optimizing Matrix-Vector Multiplication

The fixed-up version of matrix-vector multiplication with row-wise vector dot products simply outputs to another separate destination vector operand.

```
void aussie_matmul_vector_basic_out1(
      const ymatrix m, const float v[], int n, float vout[])
{
    // Basic matrix-by-vector using vector dot products..
    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 mult (MatMulVec) benchmarks (N=2048, ITER=300):
Matrix-vector nested loops: 3480 ticks (3.48 seconds)
Matrix-vector nested loops hoisted: 3489 ticks (3.49 seconds)
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 of 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 of 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 with 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 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 multiplication basic naive n^3 algo
    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 of 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 of 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 of 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 mult (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 multiplication 8 iter 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 multipl (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 re-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 of 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 multiplications, 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 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 the transposed matrix
    // into the output matrix (square 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 III: Memory Safety Techniques

20. Memory Safety Techniques

Memory Safety Thoughts

As a response to Bjarne Stroustrup's call to action regarding "C++ attacks" related to memory safety [1], I decided to compile a list of the possible methods, and count them. It's disheartening to see ongoing knee-jerk reactions to C++ memory issues, being effectively just to ban it. The plan to replace C++ with another programming language, usually Rust, is not a good idea because:

- It's expensive,
- It's not necessary, and
- It's too slow.

It's much more expensive to hire new programmers and do a ground-up rewrite with your application, than it is to refactor the code with extra memory safety mitigations. Admittedly, your C++ programmers are already expecting to get fired because of AI, so go crazy if you really like firing people. I guess you could be kinder and ask your C++ programmers to retrain in Rust, but then what you really have is a bunch of newbie programmers writing your business applications.

And it's not necessary. The cost of upgrading C++ to better memory safety is much less. Even upcoming standards improvements, such as "Profiles" as espoused by Bjarne Stroustrup [2], will only require code changes similar to getting C++ to work with a very picky compiler (because Profiles are statically enforced by the compiler). Similarly, adding pragmatic memory safety approaches is similar to a code refactoring effort, not a rewrite. Furthermore, some approaches that you can do today involve the use of new builds, new compiler tools, and upgraded standard libraries, rather than code changes.

Which one is faster? The ground-up rewrite really doesn't sound that fast to me. Many of the pragmatic code changes are a refactoring effort, where many existing techniques can be integrated into existing code bases in a day or so. Similarly, changes to the tools in build, compilers, static analyzers, and runtime checkers, are all very fast, with no code changes (except when they find bugs, haha!).

Over 100 Memory Safety Techniques

Herewith, I provide a list of all the C++ memory safety risk mitigation methods of which I'm aware, in the categories of:

- Upcoming C++ language safety features (e.g., Profiles [2], Safe C++).
- Already-completed and ongoing C++ mitigation work (e.g., C++ Core Guidelines, hardening standard C++ libraries).
- Pragmatic memory safety coding approaches (e.g., safety wrapper functions).

Without further ado...

C++ Safety Future Standardization Efforts:

1. Profiles [2] — supported by Bjarne Stroustrup, with C++ memory-safety enforceable by the compiler statically.
2. Safe C++ — C++ language extensions with "`safe`" and "`unsafe`" keywords.
3. TrapC
4. FilC
5. Mini-C

Existing C++ Safety Guidelines:

6. C++ Core Guidelines — from C++ standardization gurus Bjarne Stroustrup and Herb Sutter.
7. SEI CERT C++ Secure Coding Guidelines
8. SEI CERT C Secure Coding Guidelines

Big Quality Improvements (General Approaches):

9. Automate full test runs regularly — use CI/CD, or nightly builds when it gets too slow for CI/CD.
10. Nightly build tests with sanitizers/runtime checkers — detect bugs as early as possible.
11. Fuzzing — thrash your code with lots of weird stuff, long inputs, etc.
12. Fuzzing with runtime sanitizers — it's slow, but worth it.
13. AI code checking and debugging — it's already good, and will be great.
14. Review "technical debt" — but fixing it is not usually as impactful as programmers think.

Build Improvements for Safety:

15. Run memory safety checkers and sanitizers regularly — e.g., nightly builds.
16. Use multiple runtime memory safety checkers — Valgrind, ASan, MSVS.
17. Build a custom memory-safe coding style enforcer — e.g., even `grep` for a file of patterns containing the names of memory-unsafe functions works quite well.
18. Extra warnings — separate build path for running compilers enabled with extra picky warnings.
19. Optimizer levels — separate build path for running code with different optimizer levels to shake out rare but insidious memory errors that only occur when optimized.
20. Run builds on multiple platforms of your core platform-independent code (Windows, Linux, Mac) — more compiler warnings, more ways to thrash the code at runtime, more sanitizers (you can always spin up a cloud virtual machine for whatever platforms you need).

General Safe Coding Style Improvements:

21. Assertions
22. Check return codes
23. Validate incoming parameter values in functions
24. Debug tracing macros (logging)
25. Unit tests (can never have too many)
26. Module-level tests
27. Automated integration tests
28. Exception handling — consider whether to use C-style return codes versus C++ try/catch exceptions.
29. Painstaking work — adding reliability to code is endless small improvements, not just throw an exception and you're done.

Specific Coding Improvements for C++ Memory Safety:

30. Use macro wrappers to ensure checking of return codes for common functions (both library and custom code) — this is more general than `[nodiscard]` which guarantees only that the return code is assigned somewhere, but not that it's well handled, whereas a wrapper guarantees that failures are at least logged somewhere (and then hopefully properly handled by the caller).
31. Unreachable code marked with assertions or other handling.
32. Not-yet-implemented code marked with assertions or other handling.
33. Safety wrapper functions for common library or non-library functions — validate inputs, check for common usage errors, and check the return value for failure so it's never undetected.

34. Detect uncommon "undefined behavior" in wrapper functions — e.g., overlapping memory blocks in `memcpy`, file read and write without intervening seek, etc.
35. Intercept fatal signals (e.g., `SIGSEGV`) with a handler that at least prints a nice message and ideally even a stacktrace — no, you can't really recover at this point, but you can provide extra debugging context for supportability; also watch out it doesn't get re-raised.

DIY Memory Safety Classes:

36. Safe smart buffer class (two-variable method) — add a second "buffer safety checker" object to watch an existing buffer.
37. Safe smart buffer class (one-variable method) — replace simple buffer variables with a templated smart buffer object of the required size

Heap Memory Safety Methods for C++:

38. Macro interception of C-style memory primitives (malloc, calloc, free, strdup, etc.)
39. Link-time interception of C++ new/delete memory primitives — it's been a standard feature of C++ for many years.
40. Implement a randomized delayed deallocator — this blocks most Use-After-Free attack vectors.

DIY Heap Memory Safety Wrapper Libraries for C++:

41. Validate all memory primitives (e.g., detect free non-heap, double-deallocation, mixed C/C++ memory primitives, null pointers, overlapping memory blocks, and more.)
42. Use canary values for a fast way to detect buffer overflows.
43. Use "last-byte-null" canary values in string buffers.
44. Use redzones to detect buffer overflows.
45. Detect buffer underflows with canaries and redzones.
46. Poison uninitialized or freed memory blocks
47. Poison after the null byte in long buffers containing short strings.
48. Implement a "never-free" approach to detect all Use-After-Free errors — probably only in testing, not production.
49. Check memory block size parameters are not equal to `sizeof(char*)` — indicating `sizeof` used on a pointer or array parameter, then passed to `memset` or other functions.
50. Find memory block sizes in safety wrapper functions — each platform has its own way to find the size of an allocated block using the address of the start of the block (but not from the middle of the block).

Stack Memory Safety Methods for C++:

51. Macro interception of C-style stack memory primitives (alloca, other variants, etc.)
52. Use a runtime sanitizer that detects stack buffer overflows (i.e., not Valgrind).
53. Use a stack overflow canary object as a local variable — the constructor sets it to a fixed value, and the destructor checks the value hasn't changed, or calls `abort` if it has. Use `volatile` to prevent a smart optimizer removing it.
54. Use a semi-randomized stack overflow canary object — to further thwart stack buffer overflow attacks, don't just use a fixed value, but a semi-random value that changes with time.
55. Use an obscured pair of numbers in a semi-randomized stack object — both values are stored in the stack object and can be overwritten, so even further you can either: (a) allocate heap memory for the expected value (effective but inefficient), or (b) use some obscure formula to generate and validate the two values.

Compile-Time Memory Safety Methods for C++:

56. Warning-free compilation policy — fix all compiler warnings, even the "unused variable" ones, lest they hide serious bugs.
57. Use `static_assert` — e.g., check the `sizeof` for your types at compile-time (avoids portability glitches later).

Specific Coding Improvements for Non-Memory Safety:

58. Add loop counters to detect and prevent infinite loops
59. Add `[nodiscard]` to your functions to detect thrown-away return codes.
60. Add a standardized stack trace reporting library — you can use `std::backtrace`, Boost or Gnu; useful for assertions, return code failures, memory failures, etc.
61. Add math function wrappers — e.g., `cos(90)` is probably mistaking radians and degrees.
62. Add fopen/fclose file function wrappers — prevent double `fclose` errors, and other crashes.
63. Safe integer classes to detect overflow — a little slow for my taste.

Weird arithmetic problems:

64. Check for arithmetic overflow and underflow
65. Check for floating-point overflow and underflow
66. Beware floating-point denormalized values — tiny and weird.

67. Two floating-point zeros — there is now a level negative-zero.
68. Floating point has many different values for infinity and negative infinity.
69. Order of evaluation errors on various binary operators.
70. Order of evaluation errors on function call arguments.
71. Floating-point runtime error checker tools — seems like these are only in research papers, not used commercially yet.

Basic Standard C++ Coding Changes:

72. Use references not pointers — references are a longstanding compiler-enforced way to replace pointers, with zero extra runtime cost.
73. Use "const correctness" — I'm not a fan of this, because it's a lot of busy work if the code doesn't already adhere to proper `const` usage, and fixing it rarely finds any real bugs.

Unsafe C-style Functions:

74. Change `sprintf` to `snprintf` — but beware using its return value.
75. Use safe versions of unsafe string primitives (e.g., `strcpy`, `strcat`)
76. Avoid or wrap `strncpy` — it's actually unsafe despite having a buffer size parameter because it can leave a string without a null byte.
77. Avoid `fflush(stdin)` — officially undefined behavior, although I've seen it used and it's simply a nop.
78. Avoid the old `strncat` function — Write your own safe version of `strcat` instead.
79. Prefer `memmove` to `memcpy` — it handles overlapping ranges without failure.
80. Avoid `longjmp` and `setjmp` for exception handling — they're old, unsafe, and superceded by many other options.
81. Avoid `tmpfile` and `mktemp` — they have a race condition that's a security risk.
82. Avoid `scanf` and `sscanf` — they're just a hot mess!

Standard C++ Library Memory Safety Techniques:

83. Use `std::string` not `char*` or `char[]`.
84. Use `std::span` — safe view onto other array data.
85. Use `std::mdspan` — safe view onto multi-dimensional data.
86. Use standard data structure classes — don't write your own hash table ever again.
87. Use standard smart pointer types (e.g., `shared_ptr`, `weak_ptr`, etc.)

Sanitizer Tools for Runtime C++ Memory Safety Checking:

88. Valgrind (free on Linux)
89. ASan (address sanitizer, free)
90. Various commercial runtime memory checkers

Compilers for C++ Memory Safety:

91. GCC warning options (`-Wall` is my favorite, and there's `-Wextra`, `-Wpointer-arith`, and various others)
92. CLANG warnings — enable more to check more.
93. MSVS warnings — click on some checkboxes.

Linters for C++ Memory Safety:

94. Use compilers and extra warnings as linters
95. Set up a separate "lint" build path (e.g., "`make lint`")
96. Use freeware linters
97. Use commercial linters

Library Improvements for C++ Memory Safety:

98. C++ standard library hardening efforts
99. libc++ hardening
100. Debug versions of libc++ — with extra error checking, self-validation, and instrumentation.

Sigh. Okay, so I admit they're not all about memory safety, but some are more about software quality and non-memory undefined behaviors.

On the other hand, I didn't list out every different warning to use for each compiler, every free and commercial lint checker, every different type of memory error, and so on.

References

1. Thomas Claburn, Sun 2 Mar 2025, C++ creator calls for help to defend programming language from 'serious attacks': Bjarne Stroustrup wants standards body to respond to memory-safety push as Rust monsters lurk at the door, The Register, https://www.theregister.com/2025/03/02/c_creator_calls_for_action
2. Bjarne Stroustrup, March 2025 (accessed), ``Profiles" -- What we need, https://github.com/BjarneStroustrup/profiles

21. DIY Memory Safety

Why DIY Memory Safety?

Well, because you fix some bugs yourself! Instead of waiting for compiler vendors to add a "`-safe`" option, or the standards organizations to define "Safe C++" language, you do it yourself!

These are the main memory safety issues in C++:

- Array bounds writes (buffer overflow writes)
- Array bounds reads (buffer overflow reads)
- Uninitialized memory usage (e.g., `malloc`, `new`, stack buffers).
- Use-after-deallocation (i.e., reads or writes after `free` or `delete`).
- Double-deallocation (i.e., double-`free`, double-`delete`).

There are also other special cases of memory issues:

- File pointer misuses (e.g., double-`fclose`).
- Text buffer overruns (e.g., string copy overwrites).

Strategies for DIY Memory Safety

There are two overarching strategies, which are the opposite of each other:

- Make some failures harmless (e.g., get rid of uninitialized memory usage errors by always initializing memory to zero).
- Detect more failures by automatically causing memory problems intentionally.

You can pick one of these and do it for both developer testing and production runs by customers.

Or you can vary the idea:

- Detect more bugs in developer mode.
- Make the bugs harmless in production mode.

Why would we do this? Why not just run `AddressSanitizer` or `valgrind`? There's a few reasons:

- The sanitizers run too slow, so we cannot use them all the time, or in production.
- If we implement fast DIY methods, we can use them continually during testing.
- If they're really fast, we might even leave the self-checks in for production runs.

The DIY techniques to detect more bugs inside your own code include:

- Canary regions ("redzones") around memory blocks.
- Poisoning memory inside the blocks with error-triggering values.
- Magic values for statuses stored in buffers.
- Full address tracking (i.e., your own hash table of memory block addresses).

Hence, there are multiple levels of error detection, ranging from super-fast to almost-as-slow-as-`valgrind`.

Making Uninitialized Accesses Harmless

There's another option: just fix it! Instead of trying to find the bugs, just make them disappear by becoming harmless. This is particularly true of the whole class of memory bugs base on uninitialized memory reads.

Why are these even bugs? They seem more like language design failures, with too great a focus on speed. The basic problem with standard C++ and memory initialization is this patchwork of choices:

- Global variables are initialized to zero (hooray!).
- Local `static` local variables are initialized to zero (hooray!).
- Stack variables are not initialized to zero (boo!).
- Heap-allocated memory blocks are sometimes initialized to zero (boo!).

For heap memory allocation, we have again a patchwork:

- `malloc` memory is never initialized.
- `calloc` initializes to zero always.
- `new` of object types relies on constructors to initialize.
- `new` of arrays of objects relies on (many) constructors to initialize.
- `new` of primitive data types does not initialize at all (single variables or arrays).
- `realloc` does not initialize extra memory.

Really we want: change all `malloc` and `new` calls to `calloc`. Then a whole class of memory safety issues just disappears! Honestly, rather than detecting uninitialized memory uses, shouldn't we just make them a non-issue? Why would we even bother trying the other strategy of filling uninitialized memory with poisoned values, when we could just fix it everywhere?

Intercepting C++ Primitives

Here are the basic strategies for how to integrate safety into your code with DIY fixes to your codebase:

- Coding style to require calling safe functions
- Wrapper functions to automatically fix or detect issues.

The way that debug wrapper functions work includes these ideas:

- Macro intercepts of `malloc`, `calloc`, and `free`.
- Link-time intercepts of `new` and `delete` operators.
- Macro intercepts for `strlen` and `strcpy`, etc.
- Macro intercepts for `fopen` and `fclose`.

We have to be aware of a few issues:

- Macro intercepts won't get any allocations from any less-used primitives we don't intercept.
- Macro intercepts won't see anything in third-party libraries (including Standard C++/STL).
- Link-time `new` and `delete` intercepts will see Standard C++ calls (which can be good or bad).
- Link-time `new` and `delete` intercepts must define four versions, two for objects, and two array versions.

- There's no simple way to intercept stack-based memory operations for local variables (i.e., from function calls or returns).
- We can macro-intercept stack-based `alloca` calls, but it's hard to know when the function returns.
- We can macro-intercept `fopen` type file operations, but it's hard for C++ `fstream` types.

Overall, the DIY memory safety approach is a patchwork of techniques in itself. It would be so much easier if the compiler vendors would just add a "`-safe`" flag that does all this!

22. Intercepting Memory Primitives

Interception Methods

Intercepts can be useful for both performance instrumentation and memory error detection or prevention. There are two main ways to intercept memory primitives in your C++ code.

- Preprocessor macro interception — used for old-style allocation functions like `malloc` and `free`.
- Link-time interception — used for `new` and `delete` operators.

Once you have a successful intercept, it's amazing the things you can do! Here are some ideas:

- Detect various types of memory errors
- Prevent some types of memory crashes
- Track statistics on memory allocations
- Detect and prevent errors in various non-memory functions

Preprocessor Macro Intercepts

There are different approaches to consider when wrapping system calls, which we examine using `memset` as an example:

- Leave "`memset`" calls in your code (auto-intercepts)
- Use "`memset_wrapper`" in your code instead (manual intercepts)

Macro auto-intercepts: You might want to leave your code unchanged using `memset`. To leave "`memset`" in your code, but have it automatically call "`memset_wrapper`" you can use a macro intercept in a header file.

```
#undef memset // ensure no prior definition
#define memset memset_wrapper  // Intercept
```

Note that you can also use preprocessor macros to add context information to the debug wrapper functions. For example, you could add extra parameters to "`memset_wrapper`" such as:

```
#define memset(x,y,z) \
memset_wrapper((x),(y),(z),__FILE__,__LINE__,__func__)
```

Note that in the above version, the macro parameters must be parenthesized even between commas, because there's a C++ comma operator that could occur in a passed-in expression. Also note that these context macros (e.g., `__FILE__`) aren't necessary if you have a C++ stack trace library, such as `std::stacktrace`, on your platform.

Variadic preprocessor macros: Note also that there is varargs support in C++ `#define` macros. If you want to track variable-argument functions like `sprintf`, `printf`, or `fprintf`, or other C++ overloaded functions, you can use "`...`" and "`__VA_ARGS__`" in preprocessor macros as follows.

```
#define sprintf(fmt,...) \
    sprintf_wrapper((fmt),__FILE__,__LINE__, \
                          __func__, __VA_ARGS__ )
```

Manual Wrapping: Alternatively, you might want to individually change the calls to `memset` to call `memset_wrapper` without hiding it behind a macro. If you'd rather have to control whether or not the wrapper is called, then you can use both in the program, wrapped or non-wrapped. Or if you want them all changed, but want the intercept to be less hidden (e.g., later during code maintenance), then you might consider adding a helpful reminder instead:

```
#undef memset
#define memset dont_use_memset_please
```

This trick will give you a compilation error at every call to `memset` that hasn't been changed to `memset_wrapper`.

Link-Time Interception: new and delete

The idea of a tool to test memory allocations is to shine light on the hidden calls that create and destroy allocated memory. This helps examine how containers are using allocated memory, and it's not usually pretty!

Macro interception does not work for the `new` and `delete` operators, because they don't use function-like syntax. Fortunately, you can use link-time interception of these operators instead, simply by defining your own versions. This is a standard feature of C++ that has been long supported.

Note that defining class-level versions of the `new` and `delete` operators is a well-known optimization for a class to manage its own memory allocation pool, but this isn't what we're doing here. Instead, this link-time interception requires defining four operators at global scope:

- `new` and `new[]`
- `delete` and `delete[]`

There's a pitfall in implementing our intercepted versions. You cannot use the real `new` and `delete` inside these link-time wrappers. They would get intercepted again, and you'd have infinite stack recursion.

However, you can call `malloc` and `free` instead, assuming they aren't also macro-intercepted in this code. Here's the simplest versions:

```
void * operator new(size_t n)
{
    return malloc(n);
}

void* operator new[](size_t n)
{
    return malloc(n);
}

void operator delete(void* v)
{
    free(v);
}

void operator delete[](void* v)
{
    free(v);
}
```

This method of link-time interception is an officially sanctioned standard C++ language feature since the 1990s. Be careful, though, that the return types and parameter types are precise, using `size_t` and `void*`, as you cannot use `int` or `char*`.

Also, declaring these intercept functions as `inline` gets a compilation warning, and is presumably ignored by the compiler, as this requires link-time interception.

Memory Debug Wrappers

I've always used this intercept method for some self-testing debug wrappers. Here's an example of some ideas of some basic possible checks you can do in these intercepted operators:

```
void * operator new(size_t n)
{
    if (n == 0) {
        AUSSIE_ERROR("new operator size is zero");
    }
    void *v = malloc(n);
    if (v == nullptr) {
        AUSSIE_ERROR("new operator: alloc failure");
    }
    return v;
}
```

Note that you can't use `__FILE__` or `__LINE__` as these are link-time intercepts, not macros. However, you could use `std::backtrace` in C++23 instead.

Memory Performance Analysis

We can also use the idea of link-time interception to do performance improvement on memory allocation. This helps us find the slugs in both standard containers and our own code.

The modified version of these link intercepts is shown in the Appendix, with full source code. The idea is that you can examine the behavior of code by wrapping memory debug calls around it:

```
memory_reset_counters();
std::vector<int> v;
memory_report();
```

This allows investigation of the memory characteristics of any sequence of code. It's quite enlightening to investigate what sort of actions in the standard C++ libraries will trigger memory allocations.

Unit Testing of Memory Allocation. Another useful idea is to add unit tests to your build, so as to ensure that nobody's accidentally added some memory allocations to the code.

```
memory_reset_counters();
std::vector<MyClass> v;
TEST(s_new_count == 0);  // No memory allocations!
```

You know what I mean: trust but verify!

23. Smart Pointers

Overview of Smart Pointers

Smart pointers are a major addition to the C++ language in C++11. The three main types of smart pointers are:

- `std::unique_ptr` — exclusive ownership of an object.
- `std::shared_ptr` — reference-counted multiple owners.
- `std::weak_ptr` — not an owner, but keeping an eye on things.

These classes are defined in the `<memory>` header file, and are templated by the type of the object. There was also `std::auto_ptr` in C++98, but that was deprecated in C++11 and finally removed in C++17.

The main features of the smart pointer library include:

- Use smart pointers like raw pointers via the `*`, `[]` and `->` operators.
- Automatic deallocation of the object when the smart pointer disappears.
- Destructors called for the underlying object.
- Automatically chooses `delete` for objects and `delete[]` for arrays.

And some of the advanced features include:

- Thread-safety of the smart pointers library.
- Custom deleters can be used for actions on smart pointer destruction.

But we're getting ahead of ourselves there.

Basic Smart Pointer Usage

Here's a couple of empty smart pointer declarations that are managing nothing yet:

```
std::unique_ptr<Object> up1;
std::shared_ptr<Object> sp1;
std::weak_ptr<Object> wp1;
```

However, those declarations are smart pointers without a pointer! The normal scheme of things is to use the `new` operator in the initialization of the smart pointer:

```
std::unique_ptr<Object> up1(new Object);
```

Here's the key point: auto-deallocation! The destructor for the `unique_ptr` type will automatically call the `delete` operator on the object that was created with the `new` operator. In fact, it's even smarter and will know whether to call `delete` or `delete[]`, depending on whether you called `new` or `new[]` in the initializer.

Copying smart pointers. Note that you can "copy" a smart pointer in a constructor or an assignment. This not only copies the address being managed to the new shared pointer, but increases the reference count by one in the control block. But copying a unique pointer makes no sense, because it's the exclusive owner. Hence, copy construction or copy assignment has these effects:

- Shared pointer — increases the reference count with the copy now also managing the object.
- Unique pointer — copying is explicitly disallowed via the "`=delete`" syntax!

Hence, the unique pointer class has deleted copy constructor and copy assignment declarations. You can only move a unique pointer, not copy it.

Moving smart pointers. There are move constructors and move assignment operators for all types of smart pointer objects. The smart pointer being moved to will first unmanage its object, if any, which could call its destructor (i.e., always for a unique pointer, or based on the reference count for shared pointers, but never for a weak pointer). The details of the underlying object are then moved to the left operand smart pointer, and the smart pointer on the right becomes an empty smart pointer (managing nothing).

Take care with choosing between copying and moving, which have very different semantics for shared pointers, and copying is disallowed for unique pointers. Use of the `std::move()` type cast is helpful to ensure move semantics.

Details of auto-deallocation. The idea of auto-deallocation is to avoid memory leaks. It also allows smart pointers to embody the RAII idiom, provided your initializer does allocate the object.

The call to `delete` is made in the destructor of the smart pointer object, if it hasn't already been destroyed.

So, there are a few ways for the destructor to run:

- The destructor at the end of scope.
- The `reset()` member function to deallocate earlier.

Note that the deallocation differs for unique pointers with exclusive object control versus shared pointers with non-exclusive control:

- Unique pointers — always deallocates its fully-controlled object.
- Shared pointers — only deallocated when the reference count says it's the last shared pointer managing this object.

Note the exceptions to deallocation in the destructor:

- Weak pointers don't deallocate.
- You can define "custom deleters" in the declaration of a smart pointer.

Hence, technically, you can override the calls to `delete` in the destructor. The idea is mainly when using custom allocators such as memory pools (as an optimization), but it means you can workaround some limitations if you really need to. For example, you could define a "do-nothing" deleter to manage addresses of stack or global objects. Or you could define a custom deleter that calls `free()` if you want to manage `malloc()` blocks.

Pitfalls. There is no magic whereby the `unique_ptr` object knows that `new` was called in its initializer. It just assumes that you've given it an allocated pointer to manage. Hence, you can do this:

```
Object *objptr = new Object;
std::unique_ptr<Object> up1(objptr);  // Dangerous
```

This will also be auto-deallocated correctly. But you have to be careful that the scope of `objptr` does not outlast the `up1` smart pointer object. Because `objptr` will point to an already-deallocated pointer after the destructor for `up1` runs. Here's an example:

```
Object *objptr = new Object;
{
   std::unique_ptr<Object> up1(objptr);  // Dangerous
} // Destructor calls delete
// ....
obj->my_method();  // Kaboom!
```

There are also various other pitfalls whereby you can get a double-`delete` error on a pointer managed by a smart pointer:

- Declaring two smart pointers on the same raw pointer.
- Calling `delete` on your raw pointer (before or after the smart pointer destructor).

There also pitfalls whereby `delete` is called on the wrong type of address:

- Using smart pointers with a stack or global/static address.
- Using `malloc` addresses with smart pointers.

Note that the `nullptr` is not a crash with the smart pointer classes. It simply means an "empty" smart pointer that isn't managing anything yet.

These memory address problems don't arise with `std::weak_ptr`, which doesn't ever do any deallocation of the managed pointer. Also, if you were desperate, you could use a custom deleter that doesn't call `delete`. But the main defence is to stick to the idiom whereby the `new` operator is expressly called in the initializer of your smart pointer.

Pointer Templating. Note that templating with a pointer type "`Object*`" rather than "`Object`" is a misunderstanding, and will get a compilation error for this declaration:

```
std::unique_ptr<Object*>up1(new Object); // Error
```

You can use that type of templating if you're really wanting your smart pointers to manage other raw pointers, but why would you want to? Anyway, this would compile:

```
std::unique_ptr<Object*> up1(new Object*);   // Strange!
```

Weak Pointers

Weak pointers are used much less than unique or shared pointers. They are "observers" that do not change the lifetime of the managed object. In fact, the object can be destroyed or deallocated before the weak pointer has finished watching.

The main features of weak pointers:

- Observer idiom without control.
- Does not destroy the object ever.
- Can be "upgraded" to a shared pointer.

Weak pointers cannot be initialized with a raw pointer. They can only be initialized with nothing (or `nullptr`), or via another shared pointer. Weak pointers can stop watching in two ways: they can go out-of-scope, in which case their destructor reduces the reference count (of weak pointers) on any object they've been watching. Or you can expressly call the `reset()` member function for an early release, which releases the object, and causes the weak pointer object to be empty thereafter.

One misunderstanding about weak pointers is that you might have read something that implies the objects managed by shared pointers are not deallocated if the number of weak pointers is not zero. And yes, there's a separate reference count for weak pointers that's used in a shared pointer. However, this is only that the *control block* is not deallocated until the weak pointer reference count is zero (and also the shared pointer reference count). This is an internal allocated block that contains the reference counters and other stuff. As mentioned above, the object being managed by a shared pointer is destroyed when zero shared pointers are managing it (ignoring the weak pointer reference count), so a weak pointer can be pointing to a deallocated object if all the shared pointers have disappeared. There is the `expired()` member function to test whether the weak pointer still has a valid non-destroyed object.

Limitations of Smart Pointers

The smart pointer library has many advanced features, but there are still some things you cannot do. Some of the limitations of smart pointers include:

- Only work with heap pointers via `new` — not addresses of stack objects or global objects.
- Cannot be used with old-style `malloc` or `calloc` objects.
- Weak pointers cannot deallocate the pointer — whereas unique pointers and shared pointers do deallocate (and must!).
- No way to avoid deallocation in shared pointers — you can deallocate early with `reset()`, but cannot specify that a shared pointer destructor shouldn't do so (except by adding a custom "do-nothing" deleter at the declaration of the smart pointer, and unique pointers have the `release()` function).
- Smart pointers don't know about anything you do with its raw pointer — e.g., if you extract the raw pointer using the `get()` method.

Okay, so I'm wrong. I've written that you cannot do these things, but really you can work around most of these by defining a custom deleter. Your custom deleter might simply do nothing, rather than deallocating memory. If you really had to use `malloc` addresses, your custom deleter could call `free`.

Note that there's a valid and complicated reason that `shared_ptr` does not have a `release()` function, whereas `unique_ptr` does. The idea with shared pointers is that they are reference counted and a sharing ownership of an item. Hence, it makes less sense for a shared pointer to "release" an object to the wild, whereas unique pointer is the only manager of an object.

Furthermore, the last point about the shared pointer not knowing what you're doing if you call `get()`, this means that if you use a shared or unique pointer, it will 100% be deallocated. You cannot return your pointer to the wild, except that, again, you could use a do-nothing custom deleter.

Smart Pointer Safety

The proper use of smart pointers can significantly improve the safety of pointer-related code. Some of the errors avoided include:

- Wild pointer addresses — the smart pointer object is safe within its scope.
- Memory leaks — instead, `delete` is automatically called.

Best practices for using smart pointers for safety include:

- Choose carefully between unique pointers and shared pointers (occasionally also weak pointers).
- Initialize smart pointers using `new` directly as the initializer (rather than an already-allocated raw pointer).
- Use smart pointers with function-local scope (i.e., stack variables, not global or `static` smart pointer objects).
- Use scope of the smart pointer to control when `delete` is called.

Some problematic styles to avoid with smart pointers include:

- Avoid using the raw pointer via `get()` by using `*` and `->` operators on the smart pointer object itself.
- Avoid using smart pointers on raw pointers that already exit (i.e., prefer to allocate the objects in the smart pointer's initializer).
- Avoid allocating smart pointer objects via `new` (it gets very confusing!).

Smart Pointer Inefficiencies

Generally, smart pointers focus more on safety than speed, so that add some inefficiency to your code. However, they are relatively efficient with a limited amount of extra overhead for each smart pointer:

- The smart pointer object itself, and
- A "control block" with details about the managed object.

The control block is an internal data structure, which is not explicitly part of the smart pointer object (by default). The contents of the control block include:

- Address of managed object (i.e., the raw pointer).
- Reference count of shared pointers to the object.
- Reference count of weak pointers to the object.
- Deleter to be used (e.g., by default, it's automatically chosen as the `delete` or `delete[]` operator).

Smart Pointer Optimizations

What can you do to reduce the inefficiencies of smart pointers? I mean, other than going back to the use of raw pointers, which is not ideal. Using "dumb pointers" would lose all the safety advantages of smart pointers.

Some of the optimizations include:

- Avoid two separate objects per smart pointer (with the extra control block).
- Minimize the scope of the smart pointer.
- Call `reset()` to deallocate the memory for the object earlier.

Making Smart Pointers. By default, smart pointers have two separate objects: the smart pointer object itself, and an internal allocated object called the "control block." One of the main ways to optimize smart pointer objects is to merge the smart pointer object with its control block. The way to do this is by calling either of the "make" methods for smart pointers:

- `std::make_unique()` (C++14)
- `std::make_shared()` (C++11)

Both of these standard functions create a single object with both the smart pointer and its control block. Note that there's no "`make_weak()`" version of these functions.

Smart Pointer Bugs

Although the idea of smart pointers is to prevent common problems with raw pointers, such as wild pointers or memory leaks, there are also some new types of bugs that can occur due to misuses of smart pointer objects. Some of the possible bugs include:

- Templating the smart pointer classes with a pointer type — usually a compilation error.
- Smart pointer leaks — if you lose track of your smart pointers, their destructors never run, and the underlying objects are never cleaned up either.
- Using the `delete` operator on a raw pointer used with a smart pointer — it's also double-deallocated by the smart pointer's destructor.
- Creating two smart pointers from the same raw pointer — also causes a double `delete` memory error, since both destructors run.
- Accessing the raw pointer from a smart pointer via `get()` — very risky, allowing various raw pointer problems.
- Using smart pointers with `malloc()` blocks — this causes `delete` on a `malloc()` block (a bad error).
- Using a non-heap pointer to initialize a smart pointer — will cause `delete` on a stack or static address in the destructor (crashing).
- Weak pointer refers to an object that has "expired" (i.e., been destroyed) — the last `shared_ptr` or the single `unique_ptr` has already deallocated the object via `reset()` or its destructor, although this can be avoided by always testing the `weak_ptr::expired()` member function.

Note that a weak pointer does not ever deallocate the object, but only "observes" the object (or "refers" to it). Only unique pointers and shared pointers can actually `delete` the object from memory. Note also that you can also define "custom deleters" for your smart pointers, if you really need to avoid some of these problems.

Fortunately, there are quite a few bugs that smart pointers avoid. For example, the smart pointer destructor should automatically know whether to use `delete` or `delete[]`, depending on whether it was initialized by a simple object pointer or an array type. This is important, because a call of `delete` on a `new[]` block will not properly run the destructors of all objects in the array.

24. Canaries and Redzones

What are Canaries and Redzones?

The two terms are related to memory safety for prevention and detection of memory areas. Redzones are regions of bytes around a memory block that are marked as invalid or "poisoned" for use. Canary values are a special type of redzone, with a single value, which is examined to see if it has changed.

There are various other terms used for these two approaches. Redzones are also called memory poisoning, memory tainting, memory tagging, memory coloring, and I've probably missed a few. Canaries are sometimes called sentinel values or guard values. The general techniques are referred to as memory safety or buffer overflow protection.

The main usage of redzones and canaries is to detect buffer overflows that result in array bounds violations, which are a common C++ bug and also a security vulnerability. These types of array buffer overflow attacks are more likely to be a security vulnerability if they occur in stack memory (rather than the heap), because the program stack can be corrupted intentionally.

However, never underestimate human creativity, and many other memory errors can also be used as an attack vector. Surprisingly, one of the other major vulnerabilities is by abusing "dangling pointers" that arise from use-after-free errors.

What are Array Bounds Violations?

There are a lot of imprecise names used for basically the same thing:

- Array bounds violation
- Array overflow or array underflow
- Buffer overrun or underrun
- Buffer overwrite

Enough with the terminology; let's look at code!

An example with string buffers on the stack looks like:

```
char buf[3];
strcpy(buf, "abcd");  // Boom! (buffer overflow)
```

An example of a buffer overwrite or overflow with an array looks like this:

```
int arr[10];
arr[10] = 0;    // Write error
val = arr[55];  // Read error
```

And this is an array "underflow" error:

```
arr[-1] = 0;  // Boom!
```

Note that watching for changes in canary values can only detect "write" array bounds errors, rather than reads, but more advanced methods with redzones can also detect some read accesses to redzone memory.

Text Buffer Last Byte Canaries

One of the simplest methods of using a canary is useful for text buffers. The very last byte of a memory block containing a text string can be used as a canary. This last byte must either be the null byte, if the string buffer is full, or an unused byte if the string is shorter. Hence, there is a trick where we *set* the last byte of a text buffer to the null byte, even if it's not going to be used. Then this last byte is a canary, where a non-zero value being found afterwards means it has overflowed at some previous point (i.e., a bounds overflow write error).

Here's a raw example of how it works:

```
char buf[100] = "";
buf[99] = 0;  // Set canary null value in last byte
// ... Do stuff with buf
if (buf[99] != 0) {  // Check at the end
    // Canary squawks!
    // Text buffer has bounds-violation
}
```

The advantage of this method is that it has no extra memory overhead, and only two fast single-byte operations (null byte assignment and testing for the null byte). However, it only works for text string with a null byte at the end, rather than for other types of arrays, and can only detect write errors (not reads).

Array Extra Element Canaries

The idea of using the last element in text buffers as a canary can be generalized to non-text arrays. An array overflow error would look like this:

```
int arr[10];
arr[10] = 0;  // Error
```

To add a canary, we need to do this:

- Allocate one more element for the array.
- Set it to a magic value at the start.
- Check it still has the magic value at the end.

Here's the basic hand-coded idea:

```
const int sz = 10;
int arr[sz + 1];  // +1 for the canary

// Set up the canary (last element)
const unsigned magic = 0x12345678;
arr[sz] = magic;

// Do stuff....
arr[10] = 0;  // Error

// Check canary afterwards
if (arr[sz] != magic) {
    // Overflow write error detected!
}
```

Note the features of this canary technique:

- Works for any basic data type.
- Works for any array memory type (e.g., heap, stack, global, etc.)
- Canary value can be checked multiple times, not only at the very end.

The disadvantages include:

- After-the-fact detection of the overwrite, rather than immediately.
- Memory overhead is one extra array element per array.
- Time cost is setting one array element, and then checking it later.
- Need to disable this trick, or use a poisoning API, when running a sanitizer, because it interferes with their checks.

Redzones and Canaries for Memory Allocation Overflows

Array buffer overflows are the main reason to use redzones or canary values. These occur where an array access goes beyond the end of a valid array block, whether for a read access or a write access. Canary values can only detect writes, because they rely on the code changing the canary value, but redzones can also be used to detect reads.

The general idea is to add some extra memory to the end of an allocated block. We can intercept `malloc` or `new` memory primitives and replace them with wrapper versions that set aside some extra memory for use in error checking. Then we can check for modifications to these redzone or canary bytes, in which case an array write has occurred that is a bounds violation.

Hence, the basic steps are:

- Macro-intercepts of the memory allocation functions `malloc`, `calloc`, and `free` (also `strdup` and `realloc`, amongst others).
- Linker intercepts of `new` and `delete` (four versions with two basic and two array versions).
- Add extra bytes to be allocated in the memory block we return from our wrapper versions.
- Fill these extra redzone bytes with a special value.
- Detect uses of these special bytes later.

In advanced implementations, we can mark these redzone bytes with binary instrumentation or hardware-assisted pointer tagging.

Detection of Heap Underflows

Checking for underflows of heap addresses, such as `addr[-1]`, is trickier because we cannot just add more memory to the start of the block. The region prior to an allocated heap block contains a system header block, which is used by the system allocator (i.e., the system's `malloc` or `new` primitives). Hence, we cannot just write a canary value to `addr[-1]`, because doing so would be an underflow write error in itself, which will trigger a crash. Our technique is supposed to prevent memory glitches, not cause them!

The tricky way to detect underflow is to allocate extra memory for this underflow redzone, but not in the system header block.

The idea is to take a simple allocation:

```
char *str = (char*)malloc(100);
// Do stuff with 'str'
free(str);
```

Instead, we allocated more memory, say 16 bytes, like this:

```
char *mem = (char*)malloc(100 + 16);
// Set up the redzone mem[0]..mem[15]
char *str = mem + 16;
// Do stuff with 'str' ...
// Check the redzone mem[0]..mem[15]
free(str - 16);
```

This is messy, because we need to keep adding and subtracting the size of the redzone block (i.e., 16 bytes) but that's the overall idea. The first 16 bytes of the larger block are the redzone for underflow checking. The original code is passed a pointer to the middle of the block for use with the original code.

More generally, we can do this in a debug memory allocation library. As usual with this approach, the code needs to macro-intercept `malloc` and `free`, and link-intercept `new` and `delete` operators.

There are several problems that make this plan difficult:

- Memory alignment of addresses
- Calls to `free` need to be offset (or it crashes!)
- `new` and `delete` cannot be used in manual code sequences like the above.
- Non-intercepted calls to `malloc` in third-party linked libraries will not have redzones and are thus problematic to deallocate.

Memory Read Errors

Read errors are those that access memory, but don't change the value. Some examples of memory safety concerns with read accesses include:

- Uninitialized memory usage
- Array bound overflow reads
- Array bound underflow reads
- Use-after-free errors
- Use-after-delete errors

Superficially, it might seem that these reads are less likely to be dangerous than writes. However, it's not really the case, because read errors can still be important to detect and prevent because they can be:

(a) crashes — e.g., segmentation faults.

(b) invalid results — e.g., reading the wrong values.

(c) attack vectors — use-after-free exploits are a major category of vulnerabilities.

Redzones can be used to detect read errors if it is possible to intercept read operations on an invalid block. There are several techniques in detecting memory read errors including uninitialized memory and use-after-free:

- Instrumentation of assembly or binary code
- Memory tagging (pointer tagging)
- Hardware-assisted exceptions
- Shadow memory

The technologies for hardware-assisted memory management include:

- ARM Memory Tagging Extension (ARM MTE)
- Intel Memory Protection Extensions (MPX)
- Sparc Application Data Integrity (ADI)

All of these techniques are somewhat beyond a basic DIY memory safety technique. These are the types of methods used in runtime memory checker tools such as Valgrind and AddressSanitizer.

The basic idea is to set aside a redzone area of unused memory around every memory block, such as heap and stack memory, and then various methods are used to check every memory access for an invalid redzone address.

Prevention Versus Detection

Most of the above DIY techniques are about detecting memory safety issues, rather than preventing crashes or blocking security attackers using exploits.

Full prevention requires instrumentation or hardware-assisted shadow memory, as done by sanitizers, but then the code tends to run too slow for use in production.

However, these techniques are great to use continually during development and testing. Some of the simpler methods are also fast enough to leave in production code, or at least when shipping to beta customers.

The idea is to find as many of these issues as possible. Hence, canary and redzone techniques should be combined with fuzzing and other types of stress testing, such as passing invalid or very long inputs to the code.

And these methods are complementary to sanitizers, which should still be run in nightly builds of the regression test suites, and also sometimes combined with fuzzing and other longer tests.

Limitations of Canaries and Redzones

The canary and redzone techniques are not perfect, and won't do as well as a real runtime sanitizer tool.

Some of the problems include:

- Canary value checks only detect prior failures (not immediately).
- Redzone techniques to detect overflows immediately are difficult for DIY.
- Extra memory overhead to store the canary values and redzone bytes.
- Extra time cost of setting up canaries/redzones, and then later testing them.
- Read errors are much harder to detect than writes (almost impossible in DIY techniques).
- Crashes and memory corruption are not actually prevented (in most cases).
- Bounds violations further away than the length of the redzone will be missed.
- Security attacks are not actually prevented (and redzones can be worked around anyway).

Nevertheless, the goal of DIY canaries and redzones is to add some checking that detects a subset of failures, but is much faster than sanitizers, so it can be run 100% of the time, maybe even in production for customers.

25. Use-After-Free

What is Use-After-Free?

Use-after-free errors arise when heap memory is de-allocated, but there is still a pointer to that address. This becomes a "dangling pointer" (or "dangling reference") and any use of that memory via the pointer is a "use-after-free" error.

Note that the word "free" means any memory deallocation primitive, such as the `free` function or the `delete` operator.

Although the error usually refers to heap memory addresses, it can also occur with stack addresses.

A stack-based use-after-free type of error can occur if the address of a stack variable is returned to a caller, and then it can be misused later when the call stack expands deeper again. This is a rarer type, but it's still an error and security risk.

There are several problems with use-after-free errors:

- Crashes
- Insidious program errors
- Portability issues
- Security exploits

Programs with use-after-free errors often exhibit unpredictable behavior with intermittent failures. They may also work fine on one platform, but crash when ported to a different platform, or when the optimizer level is turned up.

Use-After-Free Security Vulnerabilities

Surprisingly, use-after-free errors in heap memory are a very common security vulnerability, second only to buffer overflow attacks on stack memory. The attack involves these steps:

(a) Intentionally triggering a problematic free to gain a dangling pointer,

(b) Waiting for something important to get allocated into the previously-freed memory, and

(c) Accessing or modifying the important data (e.g., Unix suid bits) via the dangling pointer.

This sounds very complicated and unwieldy, but it's been a very successful method of targeting vulnerabilities in C++ software.

Detecting Use-After-Free

The methods to detect use-after-free errors include:

- Memory sanitizer runtime tools
- Memory tagging
- Memory poisoning (magic bytes)
- Hardware-assisted memory block exceptions

The main way to detect these sorts of errors is to use memory sanitizers, such as Valgrind or AddressSanitizer. These tools are very good at this stuff, and you should be running them in your nightly builds with a full regression test suite. It's also useful to run these tools when using "fuzzing" (testing with many large random inputs), as a way to detect these memory errors on unexpected inputs.

Some of the ways to reduce these errors, or to mitigate them as a security attack vector, include:

- Never-free policies (where possible).
- Delayed-free policies (with various configurations).
- Random delayed free (less predictable delayed-free sequences).

Note that if you change the memory deallocation policy, you need to do it at a low level, such as in your own custom memory allocators, or in debug wrappers for allocation functions. You can't just comment out all the `delete` statements in destructors, because it's sometimes important that the destructors for these sub-objects can still run.

The idea of never deallocating any memory is horror-inspiring for most programmers. However, it's a plausible idea for short batch programs that aren't hanging around long enough for the leaks to matter.

Also, one particular case is that you can disable memory deallocation whenever the program is shutting down, whether it's a batch program or a long-running service. Program termination commonly triggers a huge volume of deallocation requests in destructors for stack, heap, and global objects, making it a fertile field for memory deallocation errors, not to mention that it also causes annoying slow program exits! Plenty of inadequately tested programs will crash on exit due to earlier heap corruptions. And yet, these deallocations don't actually matter because the operating system will reclaim all the memory once the program shuts down.

Double Deallocation Errors

One special case of the use-after-free error is a double-`free` or a double-`delete`. A program crash is likely from this:

```
char *s = (char*)malloc(100);
free(s);
free(s);  // Boom!
```

One minor mitigation is to clear the pointer to null whenever using any deallocation:

```
free(s);
s = NULL;  // safety
```

Hence, the second call will do `free(NULL)`, which is not a crash, and supposedly harmless according to the standards.

You can do self-referential macro tricks with the comma operator:

```
#define free(s)  ( free((s)), (s) = NULL )
```

Another way is that you can define wrapper functions with reference parameters:

```
#define free free_wrapper

inline void free_wrapper(void *&v)
{
    free(v);
    v = NULL;  // change reference parameter
}
```

However, it's harder to do these types of tricks for the `delete` operator because its syntax is not function-like. If only C++ had a more powerful preprocessor mode!

26. Array Bounds Violations

What are Array Bounds Violations?

Array bounds violations are memory errors where an array or buffer has its memory block bounds exceeded. For an array block of memory `arr` of size `N`, the valid range for the array index is `0..N-1`. Array bounds violations come in two types:

- Overflow — accessing `arr[N]` or larger `N`.
- Underflow — accessing `arr[-1]` or earlier.

Each of these two types of bounds violations also has two subtypes:

- Write — modify the out-of-bounds memory.
- Read — get a value from out-of-bounds memory.

All types of memory blocks can be affected by overflows or underflows:

- Global variables — these are stored in global memory.
- C-style allocated memory — `malloc` and `calloc` allocations.
- C++-style allocated memory — `new` and `new[]` memory.
- Local variables in functions on the stack — such as string buffer variables.
- Local `static` variables in functions — in global memory, not the stack.
- Class data members — in whatever type of memory that contains the object (i.e., any).
- Class `static` data members — these are in global memory.
- Read-only memory regions — string literals and numeric constants, and simple `const` variables.

There are a variety of lesser-known memory allocation functions, and also platform-specific functions that allocate memory:

- `realloc` — when it increases memory block size or moves the block.
- `aligned_alloc` — allocation with address alignment restrictions.
- `cudaMalloc` — CUDA C++ GPU memory allocation.
- `alloca` memory — dynamically allocated stack memory.
- `sbrk` — lower-level memory allocation controls.

Bounds Violation Detection Methods

The methods to detect memory errors in general, including array bounds violations, include:

- Sanitizer runtime tools — e.g., `valgrind` and `AddressSanitizer`.
- DIY methods — as described in this chapter.

The main advantage of the sanitizer tools is that they catch the errors immediately, as they happen. Unfortunately, they're too slow to run all the time, or in production, but still should be running every night with all the automated regression tests.

The DIY methods aim to be much faster, but tend to only catch buffer overruns after they have occurred, so it is not always clear when the buffer was previously overrun or what code caused it. However, some DIY methods can catch and prevent buffer overruns beforehand. The various DIY methods range in efficiency from adding only a single byte test (very fast) to a fully instrumented "memory wrapper library" that is as slow as the sanitizers.

Sanitizers typically detect multiple types of errors in different memory. However, `valgrind` notably does not check stack buffers. The DIY methods for array overruns can also be combined with other techniques:

- Uninitialized memory read detection.
- Poisoned memory blocks usage.
- Basic parameter validation (e.g., deallocation of a null address).

The main techniques for DIY buffer overflow techniques include:

- Canary regions ("redzones") of extra bytes around the memory block.
- Explicit checking of sizes and addresses at intercepted points.
- Checking the last byte of a text buffer is the null byte.
- Checking the last element of a non-text buffer (e.g., `float` array).

The remainder of this chapter is about text buffers and detecting overruns without any canary redzone areas. Canaries and redzone memory regions for text and non-text buffers are shown in Chapter 24.

Text Buffer Overruns

The classic case of a text buffer overrun occurs on the stack:

```
char buf[3];
strcpy(buf, "abcd");
```

The typical method to avoid such overflows is the "safe" string functions:

- `strncpy` (with a big proviso!)
- `snprintf`
- `strcpy_s`

There are a few disadvantages of these functions. Firstly, `strncpy` has issues (discussed below). These functions also have the problem that they silently truncate the string, without giving the programmer a way to detect that an overflow has occurred. No error messages!

strncpy problems

The funny thing is that `strncpy` in standard C or C++ is intended to help with array bounds, and yet it is literally the worst function. Sure, if the string is too long, it will avoid a buffer overrun right there. But it fills the whole buffer, which then leaves the string without a null byte at the end. Any subsequent use of the string (e.g., `strlen`) will be a buffer overrun.

The solution is to manually add your own null byte:

```
strncpy(buf, sz, s);
buf[sz - 1] = 0;   // ensure null
```

The better way is to declare your own `strncpy` safety wrapper:

```
inline char *safe_strncpy(char *dest, char *src, int n)
{
    #undef strncpy  // remove wrapper
    char *s = strncpy(dest, src, n);
    dest[n - 1] = 0;  // ensure nulled
    return s;
}
```

Then you should macro intercept all calls to `strncpy`, or otherwise ban them.

```
#define strncpy safe_strncpy
```

A more advanced version of the safety wrapper would check for null parameter values. We'd also like to check the last byte was already null at the start, and that any canary redzones have not been changed by a prior buffer overrun. However, in the general case of intercepts, we cannot necessarily be sure that `strncpy` is occurring at the start of the buffer, or that the size is that of the whole buffer.

Checking the Last Byte of Text Buffers

This method is a buffer overrun detection method that uses the very last byte of a text string buffer. It only works for text strings, not for other types of arrays. The advantages include:

- No extra memory overhead
- Fast single-byte tests

The main disadvantages of this quick approach:

- After-the-fact detection (does not prevent the overrun).
- No information on when and where it was overrun.

Here's how it works. Let's assume that we have a simple buffer variable on the stack:

```
char buf[100];
```

Slightly better is to initialize it:

```
char buf[100] = "";
```

This avoids uninitialized memory usage, with a null at the first byte (and 99 uninitialized characters), but this variable still has no overflow checking.

The trick is to think about the *last* byte, not the first. Now, if we have such a text buffer that contains strings, then the last byte in the buffer is either:

(a) the null byte (for a full buffer), or

(b) unused (for a shorter string).

We'd like to use this byte for overflow checking, but in the latter case, it could have a random value. Hence, the insightful trick is to always *set* the last byte to zero right at the start, even if we aren't necessarily going to use it. Then we can be sure it must be zero at all times when using the buffer, or else there's been a buffer overflow (at some time previously). We can be sure the last byte is zero for global text buffers, but not for stack variables or allocated buffers, so we have to add our own "set" method near the buffer initialization.

With this idea, we can add some checks:

```
char buf[100];
DEBUG_SET_BUFFER_OVERRUN(buf, 100);  // Set zero
// ... rest of function
DEBUG_CHECK_BUFFER_OVERRUN(buf, 100);  // Check zero
```

The macros are quite small and efficient, only setting and checking a single byte of the array:

```
#define DEBUG_SET_BUFFER_OVERRUN(buf, len) ( \
    ((buf)[(len)-1] = 0))

#define DEBUG_CHECK_BUFFER_OVERRUN(buf,len) \
    (( (buf)[(len)-1] == 0) ? \
       true /*ok*/ :\
       debug_buffer_overrun_failed((buf),(len)))
```

This idea will work with any kind of memory block, where we know the size of the buffer, whether local, global, or heap memory. If you have a class object with a text buffer data member, then add the "set" macro in the constructor, and the "check" macro in the destructor (and optionally also other places along the way).

We can clean this up a little with the `sizeof` operator. But be aware that there's an insidious `sizeof` error if the buffer is ever a function parameter, in which case it returns the size of a pointer (too small).

Here's the version:

```
char buf[100];
DEBUG_SET_BUFFER_OVERRUN(buf, sizeof buf);  // Set zero
// ... rest of function
DEBUG_CHECK_BUFFER_OVERRUN(buf, sizeof buf);  // Check zero
```

Note that we can actually use the check as often as we like, at any point where we think that a buffer overflow might have occurred.

```
char buf[100];
DEBUG_SET_BUFFER_OVERRUN(buf, sizeof buf);  // Set zero
// ... some of the function
DEBUG_CHECK_BUFFER_OVERRUN(buf, sizeof buf); // Check middle
// ... rest of the function
DEBUG_CHECK_BUFFER_OVERRUN(buf, sizeof buf);  // Check final
```

We can hide the `sizeof` operator behind a macro. Here are some macros based on this idea:

```
#define DEBUG_SET_BUFFER_OVERRUN(buf) ( \
    ((buf)[(sizeof(buf))-1] = 0))

#define DEBUG_CHECK_BUFFER_OVERRUN(buf) \
    (( (buf)[(sizeof(buf))-1] == 0) ? \
       true /*ok*/ :\

     debug_buffer_overrun_failed((buf),(sizeof(buf))))
```

If you don't like typing, you can do this:

```
#define SET DEBUG_SET_BUFFER_OVERRUN
#define CHK DEBUG_CHECK_BUFFER_OVERRUN
```

Note that `sizeof` only works on local variables and global variables, but not for heap buffers or array function parameters. Hence, you can choose between both versions, and prefer the additional macro version with a separate length parameter in some cases, where the caller can provide the memory block size.

Finally, note that we need to change the last byte, so this doesn't work for read-only constants (e.g., string literals), but they can't really have buffer overruns anyway.

Smart Buffer Variable with Bounds Checking

One way to add bounds checking of text buffers is to replace a simple `char` buffer with a smart buffer object. This is a "one-variable" solution because we change the original buffer to our smart object. The simple code is this:

```
char buf[100];
```

We write this instead:

```
SafeStackBuf<100> buf;
```

The full `class` code is a `template` with an integer parameter, like this:

```
template<int bufsize>
class SafeStackBuf {
    const char magicbyte = '@';
    static_assert(bufsize > 0);
private:
    char m_buffer[bufsize];  // The stack buffer
private:
    SafeStackBuf(const SafeStackBuf&) = delete; // disallow
    void operator=(const SafeStackBuf&) = delete;
public:
    SafeStackBuf() {   // Constructor
        m_buffer[0] = 0;  // Ensure initialized
        // Mark end for later overrun detection..
        m_buffer[bufsize - 1] = 0;  // Sentinel byte
    }
    void check_overflow() {
        // Check for buffer overrun... (at some prior time)
        if (m_buffer[bufsize - 1] != 0) {
            // Sentinel byte changed
            // Overrun detected (at some previous time)
            AUSSIE_ERROR("ERROR: SafeStackBuf overflow");
        }
    }
    ~SafeStackBuf() { // Destructor
        check_overflow();
    }
    // Type conversion to "char*" type...
    operator char* () { return m_buffer; }
};
```

Note that we defined a type conversion operator so that this smart buffer variable can hopefully be used without changing much of the other code in the function.

In theory, we should be able to compile-out the checking for production mode with this style (and no other code changes):

```
#if DEBUG
    SafeStackBuf<100> buf;
#else
    char buf[100];
#endif
```

An important advantage is that there's literally no extra memory overhead. We've simply put the original text buffer inside an object framework, but it's the same size. As for runtime overhead, there's the extra "set" of the last byte in the constructor, and the "check" in the destructor, but these are `inline` functions, and should be the same as using the macro versions earlier.

Two-Variable Smart Buffer Wrapper Class

The two-variable version of using a smart buffer object puts the bounds overflow checking on the "outside" in a different object. This extra object does the "set" in its constructor (clearing the last byte to zero), and the "check" in its destructor.

The way to set up the bounds overflow detection works looks like this with two separate variables:

```
char buf[100];
SafeBufferWrap bufwrap(buf, sizeof buf);
```

This is more elegant that the original macro versions, in that you don't need to add an explicit "check" call at the end of the function, because the wrapper object's destructor is automatically called when it goes out of scope. The wrapper object does the bounds overflow detection in the destructor, just before it disappears.

One of the advantages of this two-variable approach over the one-variable smart buffer is that we can easily compile-out the bounds checking object for production.

The checking is not inherent to the buffer itself, and we can do this style and the overhead of the bounds checking completely disappears:

```
    char buf[100];
#if DEBUG
    SafeBufferWrap bufwrap(buf, sizeof buf);
#endif
```

Another advantage of the two-variable approach is that the original variable is unchanged, so there is no need to fuss about whether type conversions are working. The original variable uses the original code. No problems at all!

Here's what the full `class` looks like to implement this wrapper. Note that it's not a `template`.

```
class SafeBufferWrap { // Safe wrapper for char[] buffer
    const char magicbyte = '@';
private:
    char* m_string;  // Address this wrapper is tracking
    int m_bufsize;   // Number of bytes allocated
public:
    SafeBufferWrap() = delete; // disallow without string...
    SafeBufferWrap(char* addr, int bufsize) {  // Initialize
        ASSERT_RETURN(addr != NULL);
        m_string = addr;
        m_bufsize = bufsize;
        // Set the overrun detection sentinel byte to zero
        m_string[m_bufsize - 1] = 0;
    }
    void check_overflow() { // Check for overrun (prior)
        if (m_string[m_bufsize - 1] != 0) {
            // Detected overflow (but don't know when)
            AUSSIE_ERROR("ERROR: SafeBufferWrap overrun");
        }
    }
    ~SafeBufferWrap() { // Destructor
        check_overflow();
    }
    char* string() { return m_string; }
    int size() { return m_bufsize; }
};
```

The downside to this approach, when compared to the simple "set" and "check" macro versions, is two-fold:

- Memory overhead from the extra object (a pointer and an integer).
- Runtime overhead from storing data in the extra object (a couple extra assignments).

Note that there's nothing requiring this to be used on a stack buffer. Hence, you can use a wrapper object for allocated memory blocks, global arrays, or any other memory object, provided you supply the correct buffer size. The last byte has to be writeable, so this doesn't work on read-only memory.

Furthermore, this approach can be used in other ways, because the wrapper object does not need the same lifetime as the original buffer object. You can use a wrapper object multiple times for the same buffer, and you can also combine this approach with other calls to the earlier macros that check that the last byte is null. Too many options!

27. Poisoning Memory Blocks

What is Poisoned Memory?

Poisoning memory is a technique where memory blocks are intentionally set to non-zero bytes, hoping to provide a failure if this memory block is used. The general breakdown of DIY memory safety C++ techniques includes:

- Canary regions ("redzones") around memory blocks.
- Poisoned memory blocks inside the memory block.
- Magic values stored at the start of a block.

Hence, poisoned memory aims to detect some of these memory failures:

- Uninitialized allocated memory use (e.g., `malloc`, `new`).
- Uninitialized stack memory buffer usage.
- Use-after-`delete` heap memory.
- Use-after-`free` heap memory.
- Use-after-`return` for stack memory blocks

Hence, here are some of the places where we want to poison memory blocks:

- `new` or `new[]` heap block — uninitialized heap memory.
- `malloc` block — old-style uninitialized heap memory.
- `delete` or `delete[]` — de-allocated heap block.
- `free` — old-style de-allocated heap block.

Those above examples are for the heap, but we also care about stack memory, and ideally we also want to poison:

- Local buffer variables on entry to a function (uninitialized stack memory).
- Returning from a function with a local buffer variable (invalid memory after stack unwind).

Note that we don't need poisoning for these cases:

- Global variable or object (already initialized to zero).
- Class-level `static` data members (are initialized).
- Function-local `static` variable (also zeroed in C++).

And this makes a good point: if the C++ compiler auto-zeroed all the allocated and stack memory, we wouldn't have to worry about this. Hence, I want a "`-safe`" flag for my compiler.

Marking Poisoned Memory Blocks

The simplest way to "poison" a block with bytes is simply to put a special value into every byte:

```
char buf[100];
memset(buf, '@', sizeof buf);
```

Here is a general utility routine to poison a buffer more elegantly. Note that this code does not poison the final byte in the buffer, so that any inadvertent use of the string in the buffer won't actually go beyond the buffer. Whether you do or don't want this to crash depends on context.

```
inline void aussie_poison_buffer(char* s, int bufsize,
                                 char magicchar /* = '@'*/)
{
    // PURPOSE: buffer is unused, mark with poison bytes.
    // Put some very visible magic letters e.g., @@@@@
    // They can be tested in other use of the buffer,
    // .. and also make any errors visible in output...
    memset(s, bufsize - 1, magicchar); // Clear all but last
    // Note: null byte after many @@@'s means
    // it won't crash on strlen/etc.
    s[bufsize - 1] = 0;   // Put null byte at end for safety
}
```

I like the use of multiple @ characters as a poisoned value, because it's highly visible in a printout or HTML page. It's also possible to quickly test for a likely poisoned address:

```
bool is_poisoned = s[0] == '@' && s[1]=='@' && s[2] == '@';
```

We can make this into a macro:

```
#define is_poisoned(s) \
    ((s)[0] == '@' && (s)[1] == '@' && (s)[2] == '@')
```

The preprocessor macro version really needs all those parentheses to avoid operator precedence errors, but also isn't fully safe against any side-effects in the argument expression. Safer is to use a modern `inline` function version:

```
inline bool is_poisoned(const char *s)
{
    return s[0] == '@' && s[1] == '@' && s[2] == '@';
}
```

This example is looking for three @'s in a row. It's up to you whether you want to check for 1, 2, 3, or 4 bytes in a row. Fewer means more false positives, and one @ is probably too few, as it will get a false positive for every email address or social media handle in your input text.

However, you can also use other poison byte values, such as special numbers `(char)1` or `(char)127` or some other escape. I prefer to use the range 1..127 because you needn't worry about `signed` versus `unsigned char`. Using an explicit type cast of the byte is annoying but omitting the cast is non-portable across different compilers, too. Note also that most 128..255 values are used in valid UTF8 for European or DBCS languages (or emojis!), but there are a few bytes that are not valid UTF8 (in which case, you have to be careful to cast to `unsigned char` when testing).

Obviously, you cannot use the null byte or any commonly used character as the poison marker. Also, you would usually repeat the same byte in sequence, which is fast to set using `memset`. However, if you really prefer slower code with fewer false positives, you can use alternating byte patterns or other variations.

Macro Intercepts of malloc and free

The simplest method of poisoning newly allocated blocks with `malloc` is with preprocessor macro intercepts. Note that we don't want to poison `calloc`, because it's already initialized. Here's the basic idea for the macro intercept in a header file:

```
#define malloc aussie_malloc
```

And here's the basic idea for the wrapper function that initializes:

```
void* aussie_malloc(int sz)
{
    #undef malloc  // avoid wrapper
    void* v = malloc(sz, 1);   // Call real malloc
    if (v) memset(v, '@', sz);  // Poison
    return v;
}
```

Link-Time Intercepts of new and delete

The C++ memory allocation operators cannot be macro-intercepted because they are not a function-like syntax. However, link-time interception is a standard feature of C++ that has been supported for decades.

Here's the basic code to create a global link-time intercept for `new`, simply by defining your own version:

```
void* operator new(size_t n)
{
    #undef malloc // avoid macro intercept
    void* v = malloc(n); // Call malloc (Not ::new)
    if (v) memset(v, '@', n);  // Poison
    return v;
}
```

Note that you need to exactly match the types, with a `size_t` parameter and a `void*` return type. And we also need to intercept `delete`, so that we can change it to `free`; otherwise there is a mismatch error.

```
void operator delete(void* v)
{
    #undef free // avoid macro intercept
    free(v);  // call the real free (Not delete!)
}
```

And we also need the pair of intercepted array allocate and deallocate versions:

```
void* operator new[](size_t n)
{
    #undef malloc // avoid macro intercept
    void* v = malloc(n1);  // Call malloc (Not ::new)
    if (v) memset(v, '@', n);  // Poison
    return v;
}

void operator delete[](void* v)
{
    #undef free // avoid macro intercept
    free(v);  // call the real free (Not delete here!)
}
```

Poisoning Deallocated Memory Blocks

Note that the above macro intercept of `free` and link-time intercept for the `delete` operator are not really doing anything. There's no poisoning, and it just calls another deallocation routine.

The main problem is that we don't know the size of the block being deallocated, so how can we poison it? There's no standard function for the size of a memory block.

However, non-portable code to the rescue! The methods to get the size of a block from its address include:

- `_msize` — Windows MSVS version.
- `malloc_usable_size` — GCC version.
- `malloc_size` — MacOS version.

So, here's what a semi-portable block size function would look like:

```
int size_of_block(void *addr)
{
    #if DOS || MSVS || _MSC_VER
        return _msize(addr);
    #elif LINUX || UNIX || GCC
        return malloc_usable_size(addr);
    #elif MACOS
        return malloc_size(addr);
    #else
    #error What is this platform?
    #endif
}
```

Note that the `_msize` function actually fails with a runtime exception if the address is not the start of an allocated block (e.g., the middle of an allocated block, or a non-heap address). However, we can certainly use this in a deallocation sequence, which would crash anyway if we passed it a non-block address.

Hence, we can use this idea to poison de-allocated memory in `free` using a macro interception:

```
#define free aussie_free
```

And here's the basic definition for the wrapper function that poisons freed memory:

```
void aussie_free(void *v)
{
    int sz = size_of_block(v);
    memset(v, '@', sz);
    #undef free // avoid macro intercept
    free(v);  // call the real free
}
```

And here is the C++ `delete` operator version:

```
void operator delete(void* v)
{
    int sz = size_of_block(v);
    memset(v, '@', sz);
    #undef free // avoid macro intercept
    free(v);  // call the real free (Not delete here!)
}
```

Poisoning Stack Buffer Memory

Stack variables are still a problem, even if we're intercepting all heap allocation primitives. The simple example of an uninitialized stack variable looks like this:

```
void my_stack_crash_function()
{
    char buf[100];
    std::cerr << buf << std::endl;
}
```

Fixing stack buffer usage is more difficult than heap memory. We cannot easily intercept when the stack frame is increased on function entry, nor when it is released on function returns.

Compiler vendors could do this, but it's hidden from the programmer. There's no way to use macros, and I'm not aware of any callback mechanisms or compiler settings to control the memory on the stack.

Some of the possible approaches to poisoning uninitialized stack variables include:

- Explicit calls to `memset`
- Use smart buffer objects instead of local array buffers (i.e., a one-variable wrapper).
- Use two-variable methods with smart buffer wrapping objects.
- Macro-intercept the `alloca` dynamic stack block allocation method (but it's rarely used, so this isn't that valuable).

This is the usual way of requiring an initialization, which obviates the need to do poisoning completely (except see below about partial buffers):

```
char buf[100] = "";
```

This is a worthwhile policy, and it fixes the bug in my above code example. The downside is that the whole buffer is not zero.

Here's the manual way to poison a stack variable:

```
char buf[100] = "";
memset(buf + 1, '@', 100 - 1);
```

And here's the slightly improved way of poisoning with `sizeof` operator:

```
char buf[100] = "";
memset(buf + 1, '@', sizeof buf - 1);
```

And we can use a macro to reduce the chances of copy-paste errors:

```
#define POISON_STACK_BUFFER(buf) \
      memset((buf)+1, '@', sizeof(buf)-1)
// ....
char buf[100] = "";
POISON_STACK_BUFFER(buf)
```

But beware the trap of using `sizeof` on a *parameter* of a function, rather than a local variable. An array function parameter is a pointer, rather than a real array type, the result of `sizeof` is the 4 or 8 byte size of a pointer rather than the size of array buffer (i.e., too small).

Don't do this:

```
void poison_my_buffer(char buf[100])
{
    memset(buf, '@', sizeof buf);  // Bug with sizeof!
}
```

The above methods are fine for poisoning the uninitialized part of a stack buffer, to detect a future use of uninitialized stack memory from the poisoned characters. But this doesn't poison the stack memory once the function returns. Instead, to achieve this, we need to use a smart buffer class.

Smart Stack Buffer Classes

Another way to handle stack buffers, with poisoning both before usage and after function return, is to use smart buffer classes. There are two approaches:

(a) One-variable method replacing the buffer with a class object, or

(b) Two-variable method with a second variable that is a wrapper or "watcher" object of the buffer.

The way to replace the buffer with a class looks like this:

```
char buf[100];  // Original
SmartStackBuffer<100> buf; // Template-based size
```

Or you can do this, but it's inefficient because it has to allocate on the heap instead of using stack memory, because it doesn't rely on compile-time sizing of the object:

```
SmartStackBuffer buf(100);  // Really it's on the heap
```

The two-variable method looks like this:

```
char buf[100];
SmartStackWrapper bufwrap(buf, sizeof buf);
```

In this two-variable method, we use the character array buffer as usual. But the extra smart stack wrapper object does some extra work at the start, and at the end in its destructor.

The performance downside of the one-variable or two-variable smart buffer approach is that we've added class overhead to a very primitive type. On the other hand, we can make them all short functions that are declared as `inline`, so the performance hit is minimal.

The overhead of smart buffer classes is more worthwhile when used to do a variety of checks. Using them on stack buffers can do all of these things (some of which are shown in other chapters):

- Poison the stack buffer on entry to catch uninitialized memory usage.
- Poison the unused portion of a partially-filled buffer.
- Detect buffer overrun writes (after they occur, in the destructor).
- Detect some buffer overrun reads/writes as they occur (with extra member functions).
- Poison the stack memory on function return (in the destructor), to detect use-after-return.
- Track stack memory block addresses in more detail.

Stack Buffer Destructors

The neatest thing about smart stack buffer objects it that the destructor runs whenever it goes out of scope, at the end of a code block or the end of the function. Hence, we don't need to do anything extra to detect when the stack has unwound and the buffer is no longer valid memory.

Here's an example of the two-variable class wrapper method, which works like this:

```
char buf[1000];
SafeBufferWrap bufwrap(buf, sizeof buf);
```

Here's the code and note that the stack object wrapper has both types of poisoning and also buffer overrun post-detection:

```
class SafeBufferWrap { // Safe wrapper for char[] buffers
    const char magicbyte = '@';
private:
    char* m_string;  // Address this wrapper is tracking
    int m_bufsize;   // Number of bytes allocated
public:
    SafeBufferWrap() = delete; // disallow without string
    SafeBufferWrap(char* addr, int bufsize) {  // Initialize
        m_string = addr;
        m_bufsize = bufsize;
        memset(m_string, magicbyte, m_bufsize); // Poison!
        // Set the overrun detection sentinel byte to zero
        m_string[m_bufsize - 1] = 0;
    }
```

```
    void check_overflow() {
        // Check for buffer overrun... (at some prior time)
        if (m_string[m_bufsize - 1] != 0) {
            // Detected overflow (but don't know when)
            AUSSIE_ERROR("SafeBufferWrap overrun");
        }
    }
    ~SafeBufferWrap() { // Destructor
        check_overflow();
        // Poison on stack unwind
        memset(m_string, '@', m_bufsize);
    }
    char* string() { return m_string; }
    int size() { return m_bufsize; }
};
```

Handling False Positives

The idea with the above poison method is three @'s in a row indicates poisoned memory, as defined by the "`is_poisoned`" function above. If you prefer, it could be two or four characters. Regardless of the length, you'll get a false positive if any input text contains that sequence. This is a "false positive" where an error is detected that is not real.

How to handle false positives?

The simplest idea is to ignore them, since the poisoning technique is mainly for use in development and testing phases, rather than in production. It's better to suppress false positives, as they may otherwise hide real errors. For example, if your regression tests are somehow triggering a false positive error on every nightly build, add some code to suppress it. You can build a suppression method into your error reporting mechanism, such as simply searching for other string patterns related to the error, or by suppressing it based on context values found via `__func__`, `__FILE__` or `__LINE__`.

Poisoning Partial Memory Buffers

It is useful to detect errors where there are "semantically unusable" memory bytes, even where the memory is still officially safe in C++ terms. A good example is copying a string into a larger buffer.

```
char buf[100] = "";
snprintf(buf, 100, "abc");
```

There is nothing wrong with the start of the buffer, and it has been safely copied using `snprintf`. However, any use of the end of the buffer beyond the string stored there has no valid meaning.

In this case, it's also uninitialized stack memory, but even if it was a fully-initialized global buffer, any use of that memory is still suspect.

Hence, we want to mark indices 4..99 as invalid memory. There's no standard way to do this in C++, but we can "poison" this area with special byte values. Here is the hand-coded version to do that with the above buffer:

```
int len = (int)strlen(buf);
memset(buf + len + 1, 100 - len - 1, '@');
```

Obviously, you can generalize that into a useful utility function.

```
void aussie_poison_unused_part_buffer(char* s,
             int maxbufsize, char magicchar /* = '@'*/)
{
    // PURPOSE: buffer contains string, poison unused bytes
    if (!s) {
        AUSSIE_ASSERT(s != NULL);
        return;
    }
    int len = (int)strlen(s);
    int validbytes = len + 1;  // add 1 for the null byte
    if (validbytes > maxbufsize) {
        // Too many bytes (overrun the buffer already?)
        AUSSIE_ASSERT(validbytes <= maxbufsize);
        return;  // avoid this overrun!
    }
    int remaining_bytes = maxbufsize - validbytes;
    if (remaining_bytes == 0) return;  // Buffer full
    if (remaining_bytes > 1) {
            // Poison bytes ... except last byte
            memset(s + validbytes, magicchar,
                 remaining_bytes - 1);
    }
    s[maxbufsize - 1] = 0;  // Null at very end for safety
}
```

Advanced Poisoning

But wait, there's more! If you really want the poisoning approach to be complete, there are various ways to uplevel:

- Add automated checks for poisoned addresses via the "`is_poisoned`" function in intercepts of functions such as: `strlen`, `strcmp`, `strcpy`, etc.
- Ensure the macro intercept header file is at the top of each C++ source file (after the system headers, but before any application headers).
- Either include the macro intercept at the top of your header files, or ensure there's no `malloc` or `free` used in `inline` functions in header files.
- Macro-intercept other functions (e.g., `realloc`, `alloca`).
- Linked third-party libraries will not get macro-intercepted, but will still work for link-time interception.
- Header-only third-party libraries might need review of their memory allocation usage (e.g., maybe add your macro intercept header file before including them, or maybe not).
- Any other custom allocators, such as class-specific ones, may need changes for this approach.
- Add a compile-out preprocessor macro, because you'll need to remove some of your poisonings when using a sanitizer or `valgrind`.
- Detect whether a runtime memory sanitizer is already running (e.g., the `RUNNING_ON_VALGRIND` variable) and modify the approach (e.g., don't use your own redzones, because these become valid memory in the silicon mind of the sanitizer).
- Call the sanitizer APIs to poison memory blocks when running in a sanitizer mode (e.g., not usually necessary for heap or stack memory block issues, but useful for partially empty buffers).

Poisoning API Usage

One advanced usage is modifying your approach if a sanitizer such as ASan or Valgrind is running. You can detect these with features such as:

- `RUNNING_ON_VALGRIND` — true if Valgrind is currently running.
- `__SANITIZE_ADDRESS__` preprocessor macro.
- `__has_feature(address_sanitizer)` preprocessor test.

The ASan examples above are preprocessor constructs that detect whether GCC is compiling the C++ in ASan mode (e.g., the `-fsanitize=address` option).

This isn't exactly the same thing as whether ASan is currently active at runtime, but it's a good proxy.

The AddressSanitizer tool has macros whereby you can poison custom memory blocks, so that ASan will treat their use as an error. Valgrind also has a much larger range of functions, from custom allocator controls to explicit "poisoning" calls.

ASAN poisoning API. The usage of the AddressSanitizer macros to control poisoning of memory blocks looks like:

```
ASAN_POISON_MEMORY_REGION(addr, size)
ASAN_UNPOISON_MEMORY_REGION(addr, size)
```

There is also a runtime flag "`allow_user_poisoning`" that controls these, and can remove them for production code.

Valgrind API. There's also a lot of API macros for fine-grained control of memory blocks in Valgrind. These macros can be useful:

- `VALGRIND_MALLOCLIKE_BLOCK` — mark a block as if it's newly allocated.
- `VALGRIND_FREELIKE_BLOCK` — mark as if this block is now freed.
- `VALGRIND_MAKE_MEM_UNDEFINED` — data in this memory is undefined.
- `VALGRIND_MAKE_MEM_UNDEFINED` — reset memory to be defined.
- `VALGRIND_MAKE_MEM_NOACCESS` — any use of this memory is an error.

The `VALGRIND_MAKE_MEM_NOACCESS` macro can be used to mark redzones or other poisoned regions, and `VALGRIND_MAKE_MEM_UNDEFINED` can mark memory as uninitialized.

What are these used for? Manually marking of memory blocks as poisoned or freed can be useful to manage the status of memory for ASan, including situations such as:

- Partial string buffer poisoning (as shown above).
- Memory pools or class-specific memory allocators that pre-allocate a large memory block using the system allocator, but it is then later "allocated" in small chunks.
- Data structures with initialized memory, whether cleared by constructors or allocated physically via `calloc`, but where the memory is not all used at a logical level.

- Custom memory allocators with fine-grained control over the memory blocks.
- Implementing a "never-free" or "delayed-free" memory management method for better detection of use-after-free errors, thereby getting more warnings from ASan about uses of the pseudo-deallocated memory blocks, even if they haven't really been freed yet.
- High-level logic whereby a memory block is known to be no-longer-used by the program (e.g., after move semantics), or is otherwise invalid, but is still valid from a low-level system allocator perspective.

In conclusion, the above has presented a variety of methods of poisoning both the uninitialized memory on the heap or stack, de-allocated heap memory, and unwound stack memory. The goal is to detect reads of uninitialized or invalid already-freed memory blocks.

This chapter shows a variety of techniques, and these are a lot of extra work for the programmer. It would be better if the compiler vendors did this for us! Hence, I vote for a "`-poison`" option in the next compiler release.

28. Uninitialized Memory Safety

What are Uninitialized Memory Errors?

There are fundamental problems with memory initialized in C++. This is the standard C++ situation:

- Global variables are initialized to zero.
- Basic stack local variables are not initialized (buggy!).
- Local static variables are initialized to zero.
- Heap-allocated variables are sometimes initialized (buggy!).

There are two main strategies for dealing with uninitialized memory:

- Detect the problems (e.g., run sanitizers, or use poisoned memory DIY methods), or
- Just fix them!

This chapter is about ways to fix uninitialized memory usage in C++ by DIY initialization-to-zero tricks. Really, there should be a compiler "`-safe`" option that does this for you, but I'm not aware of a vendor that offers it yet.

Initializing C++ Heap Memory

The situation with memory initialization on the heap in C++ includes:

- `malloc` memory is never initialized.
- `calloc` initializes to zero always (hooray!).
- `new` of object types relies on constructors to initialize.
- `new` of arrays of objects relies on (many) constructors to initialize.
- `new` of primitive data types does not initialize at all (single variables or arrays).
- `realloc` does not initialize extra memory.

A first approach would be to fix these via a coding standard:

- Never use `malloc`; only use `calloc`.
- Never use `new` for basic data types (e.g., `int`).

Here's one simple try to automate this:

```
#define malloc(n)   calloc(1,(n))
```

Note that we cannot macro-intercept the `new` operator because it's not function-like. Further, we can't really institute a coding policy of replacing the `new` operator with `malloc`, or `delete` with `free`, for any object types, because we need the constructors and destructors to run. We could do that for non-object types, such as basic data type arrays, but it becomes a problematic patchwork in itself.

These are all worthwhile ideas, and will fix some issues. But it doesn't address these uninitialized memory usage errors:

- Forgetting to initialize a data member in a constructor.
- Stack variables are not addressed.
- Less common methods like `realloc` still have the problem.
- Easy to get confused and mix-up the matching `free` and `delete`.

Here's another idea for fixing the uninitialized data member problems:

```
memset(this, 0, sizeof(*this)); // in constructor
```

But this is an annoying manual coding intervention, and also doesn't fully handle the issue, because it may get confused about the object size in base versus derived objects.

Intercepting Memory Allocation

A more comprehensive approach is to intercept all of the memory allocation primitives. This is possible in this way:

- Macro intercepts of `malloc`, `calloc`, and `free`.
- Link-time intercepts of `new` and `delete`.

There are also some platform-specific tricks that are neat. Microsoft CRT has a callback mechanism called "hooks" that gets called whenever an allocation occurs. You simply register your own callback functions.

What do we do in these intercepts? The basic idea is:

- Change `malloc` to `calloc`
- Change `new` and `new[]` to use `calloc`.
- Change `delete` and `delete[]` to use `free` (avoids mismatches).

Note that there's no problems with constructors and destructors with these intercepts, because they are low-level memory primitives. The `new` intercepts run before the constructors, and the `delete` intercepts run after the destructors.

The bugs that we can fix with memory allocation interception include:

- Uninitialized heap memory.
- Mismatched `new`/`delete` with `malloc`/`free`.

Macro Intercepts

Here's the basic idea for the macro intercept in a header file:

```
#define malloc aussie_malloc
```

And here's the basic idea for the wrapper function that initializes:

```
void* aussie_malloc(int sz)
{
    #undef calloc  // avoid wrapper
    void* v = calloc(sz, 1);   // Call real calloc
    return v;
}
```

Link-Time Intercepts

Here's the basic code to create a global link-time intercepts:

```
void* operator new(size_t n)
{
    #undef calloc // avoid macro intercept
    void* v = calloc(n,1);  // Call calloc (Not ::new)
    return v;
}

void operator delete(void* v)
{
    #undef free // avoid macro intercept
    free(v);  // call the real free (Not delete here!)
}
```

And we also need the array versions:

```
void* operator new[](size_t n)
{
    #undef calloc // avoid macro intercept
    void* v = calloc(n,1); // Call calloc (Not ::new)
    return v;
}

void operator delete[](void* v)
{
    #undef free // avoid macro intercept
    free(v); // call the real free (Not delete here!)
}
```

Advanced Intercepts

To make these ideas as robust as possible, it's necessary to do this work:

- Ensure macro intercept header file is included at the top of every C++ file
- Macro intercept headers may be needed at the top of some header files, too (e.g., for `inline` functions).
- Add four C++ link intercept functions: basic and array overrides for `new`/`new[]` and `delete`/`delete[]` operators.
- Intercept less common functions: `realloc`, `aligned_alloc`, etc.
- Examine third-party allocation functions in non-header linked libraries (C++ allocation will be handled automatically by the link-time intercepts, but C-style allocations won't be seen by the macro intercepts.)
- Class-specific allocators may bypass this method, or not, depending on how they are implemented.
- The Standard C++ library/STL uses a lot of C++ memory allocation, which isn't necessarily a problem, but be aware of it.
- Global or `static` C++ objects of your own or STL global variables will run your link-time intercept functions before the `main` function starts (again, not usually a problem).
- Add an option to compile-out these initializations, such as for use when running sanitizers to detect uninitialized memory errors.

On the other hand, ignore that last point. Why bother ever detecting them now? They're fixed! Just initialize the memory to zero for ever after.

One of the main downsides of the above methods is that these interception methods only work for the heap, and don't help with the stack. We can't use these two approaches of function-like macro interception or link-time interception with local stack variables.

Stack Buffer Initialization

Stack variables are still a problem, even if we're intercepting all heap allocation primitives. The simple example of an uninitialized stack variable looks like this:

```
void my_stack_crash_function()
{
    char buf[100];
    printf("%s\n", buf);
}
```

Fixing stack buffer usage is more difficult than heap memory. We cannot easily intercept when the stack frame is increased on function entry, nor when it is released on function returns. Compiler vendors could do this, but it's hidden from the programmer. There's no way to use macros, and I'm not aware of any callback mechanisms or compiler settings to always zero the stack.

Some of the possible approaches include:

- Require all local variables to be initialized.
- Coding standard requirement to use `memset` or other methods after every stack array variable.
- Use smart buffer objects instead of local array buffers (i.e., a one-variable wrapper object).
- Use two-variable methods with smart buffer wrapping objects.
- Macro-intercept the `alloca` dynamic stack block allocation method (but it's rarely used, so this isn't that valuable).

There's no easy method to do this comprehensively for stack memory, and I'm not aware of any compiler flags that guarantee zeroing of the stack frame on function entry.

This is the usual way of requiring an initialization:

```
char buf[100] = "";
```

This is a worthwhile policy, and it fixes the bug in my above code example. The downside is that the whole buffer is not zero.

Here's the manual way:

```
char buf[100] = "";
memset(buf, 0, 100);
```

And here's the better way with `sizeof` operator:

```
char buf[100] = "";
memset(buf, 0, sizeof buf);
```

And we can use a macro to reduce the chances of copy-paste errors:

```
#define INIT_MY_BUFFER(buf) \
        memset((buf), 0, sizeof(buf))
// ....
char buf[100] = "";
INIT_MY_BUFFER(buf)
```

But beware the trap of using `sizeof` for a *parameter* rather than a local variable. An array function parameter is a pointer, rather than a real array type, so it'll be the size of a pointer rather than the size of a buffer (i.e., too small). Don't do this:

```
void init_my_buffer(char buf[100])
{
    memset(buf, 0, sizeof buf);   // Bug!!
}
```

Smart Buffer Classes

Another way to handle stack buffers is to use smart buffer classes. There are two approaches: either replace the buffer with a class object, or use a second variable that is a wrapper or "watcher" of the buffer.

The way to replace the buffer with a class looks like this:

```
char buf[100];  // Original
SmartStackBuffer<100> buf;  // Smart buffer version
```

Or you can do this, but it's inefficient because it has to allocate on the heap instead of using stack memory, because it doesn't rely on compile-time sizing of the object:

```
SmartStackBuffer buf(100);  // Really it's on the heap
```

The performance downside is that we've added class overhead to a very primitive type. On the other hand, we can make them all short functions that are declared as `inline`, so the performance hit is minimal.

I'm not especially fond of the idea of using smart buffer classes just for fixing uninitialized stack memory. After all, the `memset` ideas above are almost as good, and faster than adding class apparatus around a buffer. However, smart buffer classes are worthwhile because they can also do these things:

- Detect buffer overrun writes (after they occur, in the destructor).
- Detect some buffer overrun reads/writes as they occur (with extra member functions).
- Poison the stack memory on function return (in the destructor), to detect use-after-return.
- Track stack memory block addresses in more detail.

In conclusion, the above has presented a variety of methods of making the uninitialized memory read error into a harmless non-issue. But it's a variety of techniques, and a lot of extra work, so it would be better if the compiler vendors did this for us!

29. Smart Stack Buffers

What are Smart Stack Buffers?

The idea behind smart stack buffers is to use a light wrapper around any text buffers or arrays on the stack (i.e., a local variable inside a C++ function). The idea is reasonably efficient and convenient, because:

- `inline` functions make it fast (even as a class wrapper).
- Fast tests detect buffer overflow with a single arithmetic test.
- Destructors are automatically executed whenever it goes out-of-scope (e.g., function returns), so we don't need to track that.

Here's one way to use a class wrapper to track an automatic array buffer:

```
char stackbuf[1000] = "";
SafeBufferWrap stackwrap(stackbuf, sizeof stackbuf);
```

Note that this is a two-variable method: the original buffer is unchanged, but a second class object is used to check it. There are other methods whereby a class object is used instead of a text buffer variable, which we'll explore further below.

Why Use Smart Buffers?

Why do we need this type of smartness? After all, the various sanitizers such as `valgrind` or `AddressSanitizer` (ASan), can find stack buffer overflows. Well, actually, the `valgrind` memory checker cannot find stack overflows, but only heap-allocated memory overflows, though at least ASan does detect stack variable glitches.The reason to use our own smart buffers is simple: it's faster!

Since the wrapper checking is much faster than sanitizers, we can just leave it running *all the time*. This means that we can detect these overflows:

- Continuous detection during development and testing (by dev or QA).
- Early detection in CI/CD workflows and nightly builds.
- Optionally, could even be shipped to customers enabled (either when beta testing, or maybe even in production).
- Helps tracking down intermittent and hard-to-reproduce cases.

We can't realistically be running the sanitizers in any of those situations. I'm not saying to replace them, because it's critically important to run full sanitizers on the overall regression test suite as part of your nightly builds. We can do both.

Two-Variable Method

Here's the example wrapper class called `SafeBufferWrap`, which is initialized with the raw text buffer variable, and tracks it from the outside:

```
class SafeBufferWrap {   // Safe wrapper for char[] buffers
private:
    char* m_string;  // Address buffer wrapper is tracking
    int m_bufsize;   // Number of bytes allocated
public:
    SafeBufferWrap() = delete; // disallow without a string
    SafeBufferWrap(char* addr, int bufsize) {  // Initialize
        ASSERT_RETURN(addr != NULL);
        m_string = addr;
        m_bufsize = bufsize;
        // Set the overrun detection sentinel byte to zero
        m_string[m_bufsize - 1] = 0;
    }
    void check_overflow() {
        // Check for buffer overrun... (at some prior time)
        if (m_string[m_bufsize - 1] != 0) {
            // Detected overflow (but don't know when)
            AUSSIE_ERROR("SafeBufferWrap overrun");
         }
    }
    ~SafeBufferWrap() { // Destructor
        check_overflow();
    }

    char* string() { return m_string; }
    int size() { return m_bufsize; }
};
```

It's quite a lot of code, which gives it a "heavy" appearance, but note it's actually quite "light" with relative efficiency. Firstly, all the functions can be `inline`. Secondly, the tripwire is to clear a single byte to null, and the test for overflow is a single byte test for non-null. That is very efficient.

Zeros and Canaries

There are two ways to further extend this safe buffer wrapper idea:

- Auto-initialize the buffer to all-zeros (safety).
- Set the buffer to "canary" bytes (triggers failures).

Note that the two ideas are mutually exclusive. We can either go for suppressing all the initialization errors, or we can intentionally set the buffer to non-zero values, so as to shake out more bugs. Less bugs, or more bugs, take your choice. Here's the code with these two additional options:

```
#define SAFE_BUFFER_WRAP_CLEAR  1 // 1 inits bytes to 0
#define SAFE_BUFFER_WRAP_CANARY 1 // 1 inits to canary byte

class SafeBufferWrap {   // Safe wrapper for char[] buffers
    const char magicbyte = '@';
private:
    char* m_string;  // Address this wrapper is tracking
    int m_bufsize;   // Number of bytes allocated
public:
    SafeBufferWrap() = delete; // disallow without a string
    SafeBufferWrap(char* addr, int bufsize) {  // Initialize
        ASSERT_RETURN(addr != NULL);
        m_string = addr;
        m_bufsize = bufsize;
        // Optionally: clear all bytes to zero
#if SAFE_BUFFER_WRAP_CLEAR
        memset(m_string, 0, m_bufsize);  // Clear to zero
#endif
#if SAFE_BUFFER_WRAP_CANARY
        // Mark all buffer with canary bytes
        memset(m_string, magicbyte/*'@'*/, m_bufsize);
#endif
        // Set the overrun detection byte to zero
        m_string[m_bufsize - 1] = 0;
    }
    void check_overflow() {
        // Check for buffer overrun... (at some prior time)
        if (m_string[m_bufsize - 1] != 0) {
            // Detected overflow (but don't know when)
            AUSSIE_ERROR("SafeBufferWrap overrun");
        }
    }
    ~SafeBufferWrap() { // Destructor
        check_overflow();
    }
    char* string() { return m_string; }
    int size() { return m_bufsize; }
};
```

Limitations of Smart Buffers

The limitations of this approach include:

- After-the-fact detection of buffer overruns (we don't know when it occurred, or what code caused the overrun).
- Does not prevent the overrun so it won't stop a crash and isn't a protection against attackers.
- Only detects writes beyond array bounds, not reads.

Hence, we still need to do all that work to make sure that the buffers don't overrun! And we still need to run the sanitizers in auto mode while we sleep.

30. Safe Text Buffers

C-style sprintf is Unsafe

The first assumption here is that you want to use `sprintf` rather than C++ `std::string` for performance. It's hard to get a buffer overflow with `std::string`, because it manages its own allocated buffer, but feel free to try. The only problem is speed, because `std::string` will make calls to the memory allocator once the string exceeds about 16 characters in length.

The C++ `sprintf` function for formatting strings is a long-standing part of C and C++, but it's also unsafe. It can easily overflow a buffer, and there's no way to know without inspecting the parameters in greater detail. Consider this code:

```
char buf[100];
sprintf(buf, "%s", str);   // Buffer overflow?
```

One marginally safer way is to use the precision markers, such as in:

```
char buf[100];
sprintf(buf, "%.100s", str);   // Still overflows
```

In this way, the output is limited to 100 bytes, but this is still an overflow because of the +1 for the null byte. We really need this:

```
char buf[100];
sprintf(buf, "%.99s", str);   // No buffer overflow
```

Somewhat Safer is snprintf

The `snprintf` function is safer than `sprintf`. On some platforms, there is also the `sprintf_s` safe function. Here's how `snprintf` works:

```
char buf[100];
snprintf(buf, 100, "%s", str);   // Safer
```

We can write this more portably:

```
char buf[100];
snprintf(buf, sizeof buf, "%s", str);   // Safer
```

Problems with snprintf

Although using `snprintf` will avoid a buffer overrun and a crash (whereas `sprintf` didn't), there are still some limitations:

- Not easy to detect if any overflow occurs (i.e., was prevented).
- Difficult to use `snprintf` in the middle of a string.
- Appending with `snprintf` is similarly tricky.

Detecting Truncated Overflows with snprintf

In many applications, you might want to know that a buffer overflow was avoided, such as by emitting an error message or throwing an exception. By default, `snprintf` will quietly truncate the output and do nothing else.

It is possible to examine the return value of `snprintf` to know whether an overflow has been prevented and the output truncated. The returned value is an integer and it's rather weird:

> The bytes that *would have been* output if there was enough room in buffer.

If there's no overflow, then `snprintf` returns the bytes output (excluding the terminating null byte), just like unsafe `sprintf`. If there's an overflow, then the return value will be more than (or equal to) the size of the buffer. This seems odd, but it's actually quite useful, because the way to detect an overflow is simply to compare the return code to the buffer size:

```
int bufsize = sizeof buf;
int ret = snprintf(buf, bufsize, "%s", s);
if (ret < 0) {
    // snprintf failure... (can this really occur?)
}
else if (ret >= bufsize) {
    // Overflow has occurred! (Truncated text)
}
else {
    // Normal case.
    // The string and its null byte fit in the buffer.
}
```

Note that if the return code exactly equals the buffer size (i.e., `ret==bufsize`), this is still an overflow because the extra null byte didn't fit, and `snprintf` has truncated one character from the output string so as to leave room for the null byte.

Macro Wrapping snprintf Return Codes

The above code sequence is rather a lot of typing if you're going to do that for every call to `snprintf`. Here's a way to automate it, using a preprocessor macro intercept and an `inline` function to check the return code:

```
#undef snprintf
#define snprintf(dest, bufsize, ...) \
    aussie_snprintf_return_check( \
       snprintf(dest, bufsize,__VA_ARGS__), \
              bufsize, __func__, __FILE__, __LINE__)
```

This looks dangerous since the macro `snprintf` is also in the macro value. However, C++ preprocessor macros that are self-referential are only expanded once. This is standard functionality since inception for both C and C++.

Note that this is using variable-arguments C++ macros, which are also standard C++ for many years now. These include the "`...`" and the "`__VA_ARGS__`" tokens. There's also a useful `__VA_OPT__` macro, but we don't need it here.

The above macro simply wraps the call to `snprintf` with another function whose only task is to check the return value. Here's an example of that definition:

```
inline int aussie_snprintf_return_check(
    int snprintf_retval, int bufsize,
    const char* func, const char* file, int line
    ) {
    // PURPOSE: Wrapper for snprintf return value
    if (snprintf_retval < 0) {
        AUSSIE_ERROR_CONTEXT("snprintf returned negative",
                func, file, line);
        return snprintf_retval;  // pass through
    }
    else if (snprintf_retval >= bufsize) {
        int bytes_truncated = snprintf_retval - bufsize + 1;
        // Optionally: report the bytes truncated, bufsize,
        // etc., as extra error context...
        AUSSIE_ERROR_CONTEXT("snprintf overflow truncated",
               func, file, line);
        return snprintf_retval;  // pass through
    }
    return snprintf_retval;  // pass through
}
```

Unsafe Buffer Appending with sprintf

It's tricky to append to a string using `sprintf` or `snprintf`. Here's the basic idiom for unsafe `sprintf` appending using `strlen`:

```
char xbuf[1000] = "";
sprintf(xbuf + strlen(xbuf), "abc");
sprintf(xbuf + strlen(xbuf), "def");
sprintf(xbuf + strlen(xbuf), "xyz");
```

Note that this works even for the special case of an empty string, where `strlen` will return 0, and add nothing to the location.

If you do this a lot, or the buffer is a massive text string (e.g., a long HTML document in memory), then the call to `strlen` is a slug. Marginally better is to maintain an incremental buffer pointer, so that the `strlen` calls are only from the current location, which is faster.

```
char* where = xbuf;
sprintf(where, "abc");
where += strlen(where); // append
sprintf(where, "def");
where += strlen(where); // append
sprintf(where, "xyz");
```

And you can micro-optimize this using the return code, which works for `sprintf`, which returns the number of bytes output.

```
char* where = xbuf;
where += sprintf(where, "abc");
where += sprintf(where, "def");
where += sprintf(where, "xyz");
```

But beware a pitfall: don't do this trick for `snprintf`, because it doesn't always return the actual bytes output, but returns the bytes it *would have output*, had it been in the right frame of mind.

There's only one problem with all those appending tricks: none of them are safe!

Safe Buffer Appending with snprintf

How do we append safely to a buffer? We want to do this:

```
char xbuf[1000] = "abc";
snprintf_append(xbuf, sizeof xbuf, "def");
```

But this function doesn't exist. We have to try to define our own via a macro:

```
#define snprintf_append(dest, bufsize, ...) \
    do { \
     int snplentmp = (int)strlen((char*)dest); \
     snprintf((char*)(dest) + snplentmp,  \
          (bufsize) - snplentmp, __VA_ARGS__); \
    } while(0)
```

As you can see, this figures out how far along the buffer to append using `strlen`. Then it adds that byte count to the location, but also reduces the buffer size by that amount.

It's difficult to return the value of `snprintf` in this statement-like macro. However, if we're using the macro intercept with `#define snprintf` (as in prior sections), then the wrapped return value checking will also be occurring in this usage of `snprintf`, so maybe we don't need to return the value to the caller.

Again, the call to `strlen` can become a slug for large buffers, because it's always scanning from the very start of the buffer. The alternative is to maintain a pointer to the end of the string, which is the location from which to append. Pointer arithmetic can compute the byte count more efficiently.

```
#define snprintf_append_end(dest, bufsize, endstr, ...) \
    do { \
     long int snplentmp = (long) ( \
          (char*)endstr - (char*)dest); \
     snprintf((char*)(dest) + snplentmp, \
          (bufsize) - snplentmp, __VA_ARGS__); \
    } while(0)
```

If we really do need to return the code through, then it's hard to do this in a macro, which looks like a code block rather than a function-like macro. Instead of using a macro, you can define a C++ function with variable arguments, and then have it call the `vsnprintf` function.

```
#include <stdarg.h>

int snprintf_append_function(char *dest, int bufsize,
                             char* format, ...)
{
    va_list ap;

    int len = (int)strlen(dest);
    va_start(ap, format);
    int ret = vsnprintf(dest+len, bufsize-len, format, ap);
    va_end(ap);
    return ret;
}
```

Again, we can avoid the slowdown from the `strlen` call if we maintain another pointer to the end (or middle) of the text buffer:

```
#include <stdarg.h>

int snprintf_append_end_function(char* dest, int bufsize,
                         char *endstr, char* format, ...)
{
    va_list ap;

    if (*endstr != 0) endstr += strlen(endstr);  // Safety
    long int len = (long)((char*)endstr - (char*)dest);
    va_start(ap, format);
    int ret = vsnprintf(dest+len, bufsize-len, format, ap);
    va_end(ap);
    return ret;
}
```

Actually, for a further optimization, the parameter `endstr` probably should be a reference parameter, so that its value is automatically updated in the calling code whenever it gets moved to the end.

And one final safety point: we need to check the return value of `vsnprintf`, so that we know when an overflow caused a truncation. This is possible either through another macro intercept, like we did above for `snprintf`, or by adding extra code directly into the above varargs functions.

31. Preventive Memory Safety

Prevention Versus Detection

This chapter examines the question as to what DIY memory safety techniques can be used to prevent an error from occurring, or to prevent a security exploit being used. There are many other techniques to "detect" a memory error, which are valuable, but do not directly prevent a memory glitch in production. These improve quality indirectly by finding bugs, which can then be fixed.

The list of memory errors to consider for prevention includes:

- Uninitialized memory usage (heap and stack)
- Null pointer dereference
- Buffer overflows (reads and writes)
- Buffer underflows (reads and writes)
- Use-after-free
- Double-deallocation
- Mismatched allocation and deallocation
- Standard library container memory issues
- Standard library function problems

Some of the standard library issues include:

- Unsafe string functions — e.g., `strcpy`, `strcat`, `sprintf`.
- Detecting when the "safe" string functions truncate the text (e.g., `snprintf`, `strcpy_s`).
- `strncpy` is a special problematic case that is easily fixed by a wrapper.
- File pointer problems and file operation sequence errors (e.g., null file pointers, double-`fclose`).
- Removing an object from a container in the middle of an iterator.

The DIY memory techniques that we can consider include:

- Memory sanitizer tools
- Macro intercepts (e.g., `malloc` and `free`)
- Linker intercepts (e.g., `new` and `delete`)

Some more tricks:

- Initialization methods
- Canary values
- Redzone memory regions
- Memory poisoning
- Delayed-deallocation
- Safe wrapper functions
- Smart wrapper classes

Memory Sanitizer Tools

The most obvious method of prevention of memory problems is to use runtime memory checkers and sanitizers. Examples include:

- Valgrind (Linux)
- AddressSanitizer (GCC)
- compute-sanitizer (CUDA C++)

These tools will detect and prevent a vast range of memory errors in the stack and heap. Examples include uninitialized memory usage, array bounds overflows, and use-after-free errors.

But these tools are simply too slow to use in production. They are valuable in terms of indirectly improving memory safety because glitches are detected early and fixed by programmers. But they really don't solve the prevention problem.

Preventing Memory Initialization Errors

One of the simplest DIY fixes is to avoid uninitialized memory errors in C++ by initializing memory ourselves. To do this, we need to use these techniques:

- Intercept `malloc` with macros (or linking) and replace with a wrapper that uses `calloc` (or uses `memset` to zero).
- Intercept other heap allocation primitives (e.g., `strdup`, `realloc`).
- Link-time intercept `new` and change to `calloc` (also requires matching linker intercepts of `delete` to change to `free`).
- Intercept `alloca` dynamic stack memory function (and use `memset` to zero memory).
- Use smart buffer wrapper classes to initialize local buffer variables on the stack (i.e., function local variables).

A whole class of memory errors disappears!

Most of the above techniques require minimal code changes to existing code, such as to add a header file for macro intercepts. Note that C++ already zeroes all memory for global variables and local `static` variables, without needing any special changes.

The most invasive of the above methods is adding safety class wrappers for stack buffers, but there's not really any intercepts possible in C++ for stack memory. Other possible solutions for stack buffers would involve changes to the code itself, such as to use heap memory instead, or changing to dynamic `alloca` stack memory (which can be macro-intercepted).

Overall, there's only a few exceptions to what memory we can initialize with DIY techniques, in that compiler changes are probably needed for:

- Full stack frame initialization to zero on function entry.
- Initialization of small local variables on the stack (without extra class wrapper variables).
- Register variable initialization (also related to local variables).

Mismatched Allocation and Deallocation

Mismatches between the various types of allocation and deallocation cause undefined behavior, and can even crash. In some cases, they won't crash, but will fail to run the correct constructors or destructors. The correct matches are:

- `malloc`, `calloc`, `strdup` — `free`
- `new` — `delete`
- `new[]` — `delete[]`

Any crossover between any of the three categories is technically a failure. However, these are easily resolved by DIY memory primitive wrappers. By using link-time intercepting of the four `new` and `delete` primitives, everything can be converted to `malloc/calloc` and `free`. In this way, there won't be any crashes anymore, even if this error occurs. However, note that many of these failures are still higher-level errors even if they don't crash, because they won't correctly run all the destructors if non-scalar objects are being deallocated.

Why Use Wrapper Functions?

The idea of debug wrapper functions is to fill a small gap in the self-checking available in the C++ ecosystem. There are two types of self-testing that happen when you run C++ programs:

- Self-tests such as error return checks, assertions, and wrappers in the main C++ code.
- `valgrind` or sanitizer detection of numerous run-time errors.

Both of these methods are highly capable and will catch a lot of bugs. To optimize your use of these capabilities in debugging, you should:

- Test all error return codes (e.g., a fancy macro method), and
- Run `valgrind` and/or other sanitizers on lots of unit tests and regression tests in your CI/CD approval process, or, when that gets too slow, at least in the nightly builds.

But this is not perfection! But there's two main reasons that some bugs will be missed:

- Self-testing doesn't detect all the bugs.
- You have to remember to run sanitizers on your code.

Okay, so I'm joking about "remembering" to run the debug tests, because you've probably got them running automatically in your build. But there's some real cases where the application won't ever be run in debug mode:

- Many internal failures trigger no visible symptoms for users (silent failures).
- Customers cannot run `valgrind` on their premises (unless you ask nicely).
- Your website "customers" also cannot run it on the website backends.
- Some applications are too costly to re-run just to debug an obscure error (I'm looking at you, AI training).

Hence, in the first case, there's bugs missed in total silence, never to be fixed. And in the latter cases, there's a complex level of indirection between the failure occurring and the C++ programmer trying to reproduce it in the test lab. It's much easier if your application self-diagnoses the error!

Fast Debug Wrapper Code

But it's too slow, I hear you say. Running the code with `valgrind` or other runtime memory checkers is much slower than without. We can't ship an executable where the application has so much debug instrumentation that they're running that much slower.

You're not wrong, and it's the age-old quandary about whether to ship testing code. Fortunately, there are a few solutions:

- Use fast self-testing tricks like magic numbers in memory.
- Have a command-line flag or config option that turns debug tests on and off at runtime.
- Have "fast" and "debug" versions of your executable (e.g., ship both to beta customers).

At the very least, you could have a lot of your internal C++ code development and QA testing done on the debug wrapper version that self-detects and reports internal errors.

As the first point states, there are "layers" of debugging wrappers (also ogres, like Shrek). You can define very fast or very slow types of self-checking code into debug wrapper code. These self-tests can be as simple as parameter null tests or as complex as detecting memory stomp overwrites with your own custom code. In approximate order of time cost, here are some ideas:

- Parameter basic validation (e.g., null pointer tests).
- Magic values added to the initial bytes of uninitialized and freed memory blocks.
- Magic values stored in every byte of these blocks.
- Tracking 1 or 2 (or 3) of the most recently allocated/freed addresses.
- Hash tables to track addresses of every allocated or freed memory block.

I've actually done all of the above for a debug library in standard C++. Make sure you check the Aussie AI website to see when it gets released.

Standard C++ Debug Wrapper Functions

It can be helpful during debugging to wrap several standard C++ library function calls with your own versions, so as to add additional parameter validation and self-checking code. Some of the functions which you might consider wrapping include:

- `malloc`
- `calloc`
- `memset`
- `memcpy`
- `memcmp`

If you're doing string operations in your code, you might consider wrapping these:

- `strdup`
- `strcmp`
- `strcpy`
- `sprintf`

Note that you can wrap the C++ "`new`" and "`delete`" operators at the linker level by defining your own versions, but not as macro intercepts. You can also intercept the "`new[]`" and "`delete[]`" array allocation versions at link-time.

Example: Wrapping malloc

You can use macros to intercept various standard C++ functions. For example, here's a simple interception of `malloc`:

```
// intercept malloc
#undef malloc
#define malloc aussie_malloc
void*aussie_malloc(int sz);
```

Once intercepted, the wrapper code can perform simple validation tests of the various parameters.

Here's a simple wrapper for the `malloc` function in a debug library for C++ that I'm working on:

```
void *aussie_malloc(int sz)
{
    // Debug wrapper version: malloc()
    AUSSIE_DEBUGLIB_TRACE("malloc called");
    AUSSIE_DEBUG_PRINTF("%s: == ENTRY malloc === sz=%d\n",
         __func__, sz);

    g_aussie_malloc_count++;
    AUSSIE_CHECK(sz != 0, "AUS007", "malloc size is zero");
    AUSSIE_CHECK(sz >= 0, "AUS008", "malloc size negative");

    // Call the real malloc
    void *new_v = NULL;
    new_v = malloc(sz);
    if (new_v == NULL) {
            AUSSIE_ERROR("AUS200", "ERROR: malloc failure");
            // Try to keep going?
    }
    return new_v;
}
```

This actually has multiple levels of tests:

- Validation of called parameter values.
- Detection of memory allocation failure.
- Builtin debug tracing macros that can be enabled.

A more advanced version could also attempt to check pointer addresses are valid and have not been previously freed, and a variety of other memory errors. Coming soon!

Example: memset Wrapper Self-Checks

Here's an example of what you can do in a wrapper function called "`memset_wrapper`" from one of the Aussie AI projects:

```
void *memset_wrapper(void *dest, int val, int sz)
{
    if (dest == NULL) {
        aussie_assert2(dest != NULL, "memset null dest");
        return NULL;
    }
    if (sz < 0) { // Why we have "int sz" not "size_t sz"
        aussie_assert2(sz >= 0, "memset size negative");
        return dest;  // fail
```

```
    }
    if (sz == 0) {
        aussie_assert2(sz != 0, "memset zero size");
        return dest;
    }
    if (sz <= sizeof(void*)) {
        // Suspiciously small size
        aussie_assert2(sz > sizeof(void*),
               "memset with sizeof array parameter?");
        // Allow it, keep going
    }
    if (val >= 256) {
        aussie_assert2(val < 256, "memset value not char");
        return dest; // fail
    }
    void* sret = ::memset(dest, val, sz);  // Call real one!
    return sret;
}
```

It's a judgement call whether or not to leave the debug wrappers in place, in the vein of *speed versus safety*. Do you prefer sprinting to make your flight, or arriving two hours early? Here's one way to remove the wrapper functions completely with the preprocessor if you've been manually changing them to the wrapper names:

```
#if DEBUG
    // Debug mode, leave wrappers..
#else // Production (remove them all)
    #define memset_wrapper memset
    //... others
#endif
```

Compile-time self-testing macro wrappers

Here's an idea for combining the runtime debug wrapper function idea with some additional compile-time tests using `static_assert`.

```
#define memset(addr,ch,n) ( \
    static_assert(n != 0), \
    static_assert(ch == 0), \
 memset_wrapper((addr),(ch),(n),__FILE__,__LINE__,__func__))
```

The idea is interesting, but it doesn't really work, because not all calls to the `memset` wrapper will have constant arguments for the character or the number of bytes, so the `static_assert` commands will fail in that case. You could use standard assertions, but this adds runtime cost. Note that it's a self-referential macro, but that C++ guarantees it only gets expanded once (i.e., there's no infinite recursion of preprocessor macros).

Preventing Null Pointer Dereferences

A huge number of null pointer dereferences can be prevented and detected by wrapping the many standard library functions. Here's a simple example of the intercept:

```
#define strcmp strcmp_safe
```

And here's the wrapper function with parameter validation checks that prevent null pointer crashes:

```
int strcmp_safe(const char* s1, const char* s2)
{
    if (!s1 && s2) {
        AUSSIE_ASSERT(s1);
        return -1;
    }
    else if (s1 && !s2) {
        AUSSIE_ASSERT(s2);
        return 1;
    }
    else if (!s1 && !s2) {
        AUSSIE_ASSERT(s1);
        AUSSIE_ASSERT(s2);
        return 0;  // Equal-ish
    }
    else {
        // Both non-null
        return strcmp(s1, s2);
    }
    // NOTREACHED
}
```

Unfortunately, detecting null pointer usage requires compiler changes for direct pointer or array operations, such as:

```
*ptr = 0;
ptr->value = 0;
arr[0] = 0;
```

Generalized Self-Testing Debug Wrappers

The technique of debug wrappers can be extended to offer a variety of self-testing and debug capabilities. The types of messages that can be emitted by debug wrappers include:

- Input parameter validation failures (e.g., non-null)
- Failure returns (e.g., allocation failures)
- Common error usages
- Informational tracing messages
- Statistical tracking (e.g., call counts)

Personally, I've built some quite extensive debug wrapping layers over the years. It always surprises me that this can be beneficial, because it would be easier if it were done fully by the standard libraries of compiler vendors. The level of debugging checks has been increasing significantly (e.g., in GCC), but I still find value in adding my own wrappers.

There are several major areas where you can really self-check for a lot of problems with runtime debug wrappers:

- File operations
- Memory allocation
- String operations

Wrapping Math Functions

It might seem that it's not worth wrapping the mathematical functions, as their failures are rare. However, these are some things you can check:

- `errno` is already set on entry.
- `errno` is set afterwards (if not already set).
- Function returns `NaN`.
- Function returns negative zero.

Most of these can be implemented as a single integer test (e.g., `errno`) or as a bitwise trick on the underlying floating-point representation (e.g., convert `float` to an `unsigned`). There are also builtin library functions to detect floating-point categories such as `NaN`.

In this way, a set of math wrapper functions has automated a lot of your detection of common issues. These aren't as common as memory issue, but it's yet another way to move towards a safe C++ implementation.

Wrapping File Operations

Many of the file operations are done via function calls, and are a good candidate for debug wrapper functions. Examples of standard C++ functions that you could intercept include:

- `fopen`, `fread`, `fwrite`, `fseek`, `fclose`
- `open`, `read`, `write`, `creat`, `close`

Note that intercepting `fstream` operations in this way is not workable. They don't use a function-like syntax for file operations.

Using the approach of wrapping file operations can add error detection, error prevention, and tracing capabilities to these operations. Undefined situations and errors that can be auto-detected include:

- File did not open (i.e., trace this).
- Read or write failed or was truncated.
- Read and write without intervening seek operation.

Link-Time Interception: new and delete

Macro interception works for C++ functions like the standard C++ functions like `malloc` and `free`, but unfortunately you really can't possibly macro-intercept the `new` and `delete` operators, because they don't use function-like syntax. Fortunately, you can use link-time interception of these operators instead, simply by defining your own versions. This is a standard feature of C++ that has been long supported.

Note that defining class-level versions of the `new` and `delete` operators is a well-known optimization for a class to manage its own memory allocation pool, but this isn't what we're doing here. Instead, this link-time interception requires defining four operators at global scope:

- `new`
- `new[]`
- `delete`
- `delete[]`

You cannot use the real `new` and `delete` inside these link-time wrappers. They would get intercepted again, and you'd have infinite stack recursion.

However, you can call `malloc` and `free` instead, assuming they aren't also macro-intercepted in this code. Here's the simplest versions:

```
void * operator new(size_t n)
{
    return malloc(n);
}

void* operator new[](size_t n)
{
    return malloc(n);
}

void operator delete(void* v)
{
    free(v);
}

void operator delete[](void* v)
{
    free(v);
}
```

This method of link-time interception is an officially sanctioned standard C++ language feature since the 1990s. Be careful, though, that the return types and parameter types are precise, using `size_t` and `void*`, as you cannot use `int` or `char*`. Also, declaring these functions as `inline` gets a compilation warning, and is presumably ignored by the compiler, as this requires link-time interception.

Here's an example of some ideas of some basic possible checks:

```
void * operator new(size_t n)
{
    if (n == 0) {
        AUSSIE_ERROR("new operator size is zero\n");
    }
    void *v = malloc(n);
    if (v == NULL) {
        AUSSIE_ERROR("new operator: alloc failure\n");
    }
    return v;
}
```

Note that you can't use `__FILE__` or `__LINE__` as these are link-time intercepts, not macros. Maybe you could use `std::backtrace` instead, but I have my doubts.

Destructor Problems with Debug Wrappers

The use of a debug wrapper library can be very valuable. However, there are a few problematic areas:

- Destructors should not `throw` an exception.
- Destructors should not call `exit` or `abort`.
- Destructor issues with `assert`.

Any of these happenstances can trigger an infinite loop situation. Exception handlers can trigger destructors, which in turn trigger exceptions again. Exiting or aborting in a destructor may trigger global variable destruction, which calls the same destructor, which tries to exit or abort again (and loops). Be careful of the system `assert` macro inside destructors, because it's a hidden call to `abort` if it fails.

Although these infinite-looping problems are serious, it would seem that these are minor issues to add to your coding standards: don't do these things inside a destructor. However, we're talking about debug wrapper libraries, rather than explicit calls, and destructors often have need to:

- De-allocate memory
- Close files

Both of these tasks are often intercepted by debug wrapper libraries, whether macro-intercepted or at link-time. Hence, the issue we have is that any failure detected by the debug wrapper code may trigger one of the above disallowed calls, depending on our policy for handling a detected failure.

Unfortunately, I'm not aware of an API that checks if "I'm running a destructor" in C++. Hence, it's hard for the debug library to address this issue itself. There are a few mitigations you can use in coding destructors:

- Recursive re-entry detection inside destructors using a `static` local variable.
- Modify the debug library's error handling flags on entry and exit of a destructor
- Have global flags called "I'm exiting" or "I'm failing" that are checked by all your destructors, in which case it should probably do nothing.

Alternatively, you could manage your own global flag "I'm in a destructor" in every destructor function. More accurately, this is not a flag, but a counter of destructor depth. This flag or counter is then checked by the debug library to check if it's in a destructor before it throws an exception, exits, or aborts.

But I'm not sure what the debug library should do instead? Maybe it can itself set a global flag saying "I want to exit soon" and then it will later detect this flag is set on the next intercepted call to the debug library, provided that it's not still inside a destructor. Perhaps your application's main processing loop could regularly check with the debug library whether it wants to quit, by just checking that global variable often.

Ugh! None of that sounds workable.

A better plan is probably that your debugging library wrapper functions should never throw an exception, exit, abort, or use the builtin system `assert` function, because it can't ever be sure it's not inside a destructor. Instead, report errors and log errors in another way, but try to keep going, which is a good idea anyway.

Appendix: 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
{
    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" << 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);
}
```