Safe C++

Fixing Memory Safety Issues

David Spuler

Aussie AI Labs

Safe C++: Fixing Memory Safety Issues

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

https://www.aussieai.com

First published: October 2024.

Preface

Why a Book on Safe C++?

This book examines the new proposals for a standardized "Safe C++" extensions, along with a variety of pragmatic techniques for coding safe C++ programs, from beginner to advanced, along with a catalog of common C++ bugs to avoid.

Who This Book is For

Anyone programming in C++ or trying to learn the language will benefit from better safety! skills! This book examines safety in coding from beginner to advanced, starting with basic techniques. In the later chapters, the book then covers a variety of advanced techniques.

How This Book is Organized

This book is organized with safety in mind, from its rounded corners to soft crumple zones.

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

Safe C++ Projects

Learn more about our C++ projects at https://www.aussieai.com/safe/projects:

- Aussie Debuglib — debug wrapper library for C++ primitives.
- Aussie Lint — linter capability for C++.

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 inference algorithms (hardware acceleration).
- Inference optimization algorithms (i.e., software methods).

C++ Source Code

Most of the source code examples are excerpts from the Aussie AI C++ library, in many of the C++ source code examples. Details about source code availability can be found in the Aussie AI Safe C++ book area:

https://www.aussieai.com/safe/overview

Some code examples are abridged with various code statements removed for brevity or elucidation. For example, assertions, self-checking code, or function argument validation tests have sometimes been removed.

Most of the code is standard C++, and should run across most platforms.

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 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.

More Safe C++

A whole book on Safe C++ isn't enough for you? You can find more on our website.

Updates and Bonus Materials: Additional book materials, bonus articles and chapters, updates and errata will be made available over time online at the Aussie AI website. Visit this URL: https://www.aussieai.com/safe/overview

Errata: Any bugs or slugs that we learn about in this work will be posted online on the Aussie AI website in the Errata section of Aussie AI research. Visit this URL to view these details: https://www.aussieai.com/safe/errata

AI Research Literature Review: Ongoing updates to the AI research literature review are found in the Aussie AI Research pages, categorized by topic, starting at the entry page: https://www.aussieai.com/research/overview. The main Safe C++ research is available at: https://www.aussieai.com/research/safe. If you have a correction to a citation or a paper to suggest for a category, please email `research@aussieai.com`.

Blog: Add a regular dose of *Safe C++* to your feed. Review the Aussie AI blog at https://www.aussieai.com/blog/index, with a variety of articles on AI and Safe C++ programming.

Future Editions: Please get in touch with any contributions or corrections as future editions of the book are planned. I welcome suggestions for improvement or information on any errors you find in the book.

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

Acknowledgements

This book would not have been possible without the help of others. Thank you to Michael Sharpe who lent his AI and C++ expertise to the project with industry guidance and technical reviews. Data scientist and architecture expert Cameron Gregory also provided much assistance with many contributions to various chapters on coding, architecture, and DevOps.

I would like to acknowledge the many hardware engineers, AI researchers and open source contributors who have made the AI revolution possible. In particular, the advanced coding skills shown in the many C++ projects and examples are acknowledged with both admiration and appreciation.

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`.

About the Author

David Spuler is a serial technology entrepreneur who has combined his love for 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 *Generative AI in C++*, which covers AI internals, two books on CUDA C++ programming for the GPU, and four books on general C++ programming covering introductory and advanced C++ programming, efficiency/optimization, debugging/testing, and software development tools.

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. Prior roles in the corporate world have been as a software industry executive at BMC Software, M&A advisor, strategy consultant, patent expert, and prolific C++ coder with expertise in autonomous agents, compiler construction, internationalization, ontologies and AI/ML. Contact by email to `research@aussieai.com` or connect via LinkedIn.

About the Contributors

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

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

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

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

Table of Contents

1. Memory Safety

Why Safe C++?

If you're a C++ programmer, then why are you asking? You already know all about the issues with C++ and its lack of memory safety. The only real question is why everyone suddenly cares.

There's really been two driving forces towards the need for a "Safe C++" language:

- Large software companies, and
- The U.S. Federal Government.

Large companies in the software development business, notably Microsoft and Google, have been complaining about C++ memory weaknesses for years. Notably, both Microsoft and Google reported the same percentage of failures attributable to memory safety issues: 70% of errors. These were mainly in the area of security, which is a subset of "safety," but the number 70% is so high that people started to pay attention to C++ memory issues.

Following on from this, several initiatives in the U.S. Federal Government cited memory safety as not just an economic issue, but also a security issue. Guidelines are available for the use of memory-safe programming languages, and C++ is definitely not on that list. This culminated in public releases from the White House with reports about memory safety in software.

Sounds like it's time to do something!

What To Do?

An important point to make is: *this isn't the 1980s*. Computers are massively faster than when the C and C++ programming languages emerged. Modern computers have the capability to defer more computational power to safety concerns without unduly reducing performance for users. Hence, whereas slowing down C++ for performance would have been anathema in decades past, it is quite plausible to do in the modern tech stack.

Hence, drawing attention to the memory safety issues in C++ is not unwarranted, and some would say it's long past time to do so. Reactions by affected parties to the White House materials were varied, and included:

- Push to use Rust over C++,
- Defenders of C++ emerged, and
- The "Safe C++" initiative was launched.

It's not like nobody's ever done anything before. Indeed, those very same companies, Microsoft and Google, have done extensive work on their internal C++ software development practices. The whole industry has developed tools and techniques aimed towards higher quality C++ software in terms of memory safety and other quality concerns.

Notably, the recent high-level attention has also spawned a new initiative: the Safe C++ language proposal, which is also connected to the official ISO standardization organization. The idea of Safe C++ is to define an extension to the standard C++ language that incorporates additional safety capabilities.

This new proposal, which will hopefully evolve into an ISO-ratified standard, uses some advanced and brain-bending ideas. Many of these ideas are borrowed from Rust (surprise!), which has a model of memory "borrows" and "lifetimes" that can give a compile-time guarantee of memory safety, while not incurring the runtime cost of garbage collection methods.

Choices for Safe C++ Projects

Organizations are well aware of software quality issues, and their software development organizations already use many tools and techniques. Nevertheless, how should you respond to the Safe C++ initiative? At a high-level, the main choices are:

- Switch — move to a memory-safe programming language (e.g., Rust),
- Stay — do nothing different and keep using C++.
- Wait — for Safe C++.

Note that Rust is not the only alternative language to C++. There are other memory-safe languages, such as Go or Java, but these have the performance cost of garbage collection.

I'm not a fan of the Rust rewrite, mainly because I'm a C++ programmer. Thus, I have disclosed my bias. But most developers love a good rewrite, so they might be happy to learn a new programming language for that. The pay rates for Rust are higher at the moment, so there's that incentive, too.

Don't count on me to switch to Rust, but, on the other hand, as an experienced C++ programmer for multiple decades, I'm well aware of all the memory problems inherent to C++ programs. It's surprising the industry has taken so long to make it more of a priority.

People just like speed, I suppose?

The Safe C++ Proposal

This book examines the proposal for "Safe C++" from the C++ Alliance. This is an early working-draft proposal for extensions to the C++ language, intended to move towards an ISO standard for safer C++.

The first draft of the standard was released in September 2024, by authors Sean Baxter and Christian Mazakas at `safecpp.org`. The Safe C++ proposal includes an extensive document and a Github repository.

It is important to note that this is only a proposal at the time of writing. You should check whether anything more recent has changed in the C++ standardization area, although formal ratification of new ISO standards may take years.

Another area is that the industry C++ compilers, such as GCC, Clang, and Microsoft Visual C++, often lead in terms of adding new language functionality. These compilers already have significant safety features that can be used already, and more will undoubtedly be added over time, especially with the current interest in safer C++ practices.

The Pragmatic Plan

Are these your only options for addressing C++ memory safety: "stray, stay or play"? By which I mean, move to Rust, stay with standard C++, or "play" while awaiting Safe C++.

How about this for a fourth option:

- Fix or mitigate as many C++ problems as possible now, and
- Move to Safe C++ when available.

Let's do something pragmatic, right now!

There are incremental ways to partially address the C++ safety issue, instead of a massive dislocation (Rust), or doing nothing, And it sure beats just sitting around to wait for compiler vendors and the standardization organizations to finish their work on a new Safe C++ language.

There are many ways to proactively intervene now to improve the safety of C++ code. Some suggestions that we'll examine in this book include:

- Use existing safety capabilities of compilers (there are many!)
- Exploit all of the various toolchains for C++.
- Run the memory debugging tools at full scale.
- Use debugging libraries available already.
- Use linters and static analyzers as another defensive method.
- Adopt defensive C++ coding practices that can prevent or mitigate against failures.

It's not like nobody's ever done anything about safe C++. Let's max it out!

Sliding Scale of Safety

There's not a binary distinction between "safe" or "unsafe" programs. Rather, there's a sliding scale where there is a trade-off between levels of safety and the runtime performance cost of achieving that safety. At the two ends of the scale are:

- 100% safe — pointer accesses disallowed or in bubble wrap.
- 100% unsafe — super fast, no checks at all.

Now, 100% memory safety is effectively achievable today, but it runs slow. I don't mean the Safe C++ language proposal, which would run fast, because we don't yet have that.

Instead, I mean the well-known memory checker tools such as `valgrind` and the other various "sanitizer" tools, which effectively wrap all erroneous pointer and memory issues, and make them harmless, with a reported error message. Hence, we certainly can run C++ programs in a very safe way, just by using these tools that already exist.

These runtime memory checker tools are widely used by C++ developers during the testing and debugging phases. But we can't make our customers use these tools, because they'd be just too slow. Hence, the real question is not how to get C++ safety, but how to achieve C++ safety *with acceptable speed*.

The software development industry knows this issue only too well. There are a number of techniques to improve C++ quality as part of the software development processes. We already mentioned one of them, which is the regular use of sanitizers and memory checkers by developers. There are numerous other things that C++ coders do "in the lab" to improve code quality:

- Unit testing and regression testing (i.e., automated testing).
- Compiler warnings and linter tools that examine source code.
- Standardized libraries and third-party code (i.e., reusing pre-tested C++ components).

The list is rather long, and we're going to examine various other ideas in later chapters. But here's the big question:

What's missing?

The answer is simply there's no safety net when running C++ for customers. This book examines two main types of bubble wrap to use for running a C++ program:

- Safe C++ language proposal — guaranteed memory-safety.
- Dynamic safer C++ libraries — detect and make harmless a large subset of problems.

Again, there's a sliding scale. The proposal for a Safe C++ language achieves memory safety with zero cost to run-time efficiency and an enforceable compile-time guarantee. The various other "safe C++ library" ideas can improve safety, but their effectiveness is on the sliding scale: simple error checks can prevent some memory failures, at a low runtime cost, whereas more comprehensive memory checking can prevent more errors, but at a greater slow-down in execution speed.

Organizational Safety Initiatives

Most of this book is down in the details of the C++ language, but that's not the only area to examine. At a high level, here are some company-wide actions that will improve C++ safety:

- Hire experienced C++ programmers.
- Train your existing C++ staff with a focus on safety practices.
- Review existing C++ development practices from a safety perspective.
- Ensure correct processes and project management are in place for software development.

With regard to processes for software development, some additional specific ideas include:

- Professionalize workflow with source code control systems, requirements and design, etc.
- Document policies and practices related to C++ safety.
- Define coding standards and risk areas with regard to safety.
- Implement code review and CI/CD approval practices.
- Consider safety-specific refactoring projects.
- Don't expect AI to save you!

The reality is that C++ safety won't improve much unless you make it a business priority. Doing so is not without cost in terms of time and money, so you should also consider what you're going to de-focus away from in order to have time for a coding safety initiative.

2. Rust versus C++

Why Rust Memory Safety?

We didn't get here overnight. The concerns about C++ memory glitches have been well-known for years. A variety of techniques and tools have arisen to mitigate these problems, but they're not perfection.

More recently came the focus on security vulnerabilities. The problem with memory safety issues is not only that it causes a crash or a glitch for our users, but it also exposes a security vulnerability that can be exploited by malicious actors.

The classic attack vector is to use a buffer overrun to cause the program to execute malicious code. The attack is rather involved, meaning that the buffer overrun has to trigger the machine code to be executed. However, exploiting these memory errors, especially on the stack, has become routine.

Many companies have been running defence against security exploits, and have spent a lot of resources doing so. Both Microsoft and Google report that over 70% of their C++ vulnerabilities are related to memory safety failures.

Maybe we should fix that!

But the initiatives for C++ safety didn't really get a head of steam until the U.S. Government began reporting on security vulnerabilities related to memory safety weaknesses in programming languages. The recent White House initiative to convert usage to memory-safe programming languages was seen as a challenge to the very existence of C++ as a programming language.

Hence, we get a full debate on Rust versus C++ and whether programmers should switch to a memory-safe language. This has subsequently led to the development of the Safe C++ proposal.

C++ Versus Rust

Rust is newer and is now gaining a lot of supporters in the development community. The pros of Rust over C++ include:

- Memory safety
- Thread safety (concurrency control)
- Advanced modern language features

The advantages that C++ retains over Rust include:

- Tested and well-understood
- Developer community
- Longstanding codebases
- Large ecosystem of tools and libraries.
- Standardized (i.e., C++11/C++14/C++17/C++20/C++23)

Rust vs C++ Syntax. Some of the differences in the low-level syntax of the two languages:

- Rust uses "`let`" for assignments.
- Rust memory allocation uses the "borrow" and "lifetime" annotations for compile-time validated memory safety.
- Rust does not need a garbage collection mechanism (unlike various other memory-safe languages such as Go or Java).
- Rust has a "`println`" command for output.
- Ruse uses "`struct`" (structure) and "`impl`" (implementation) for class-like modularity.

Why Not Other Memory-Safe Languages?

The push for an alternative memory-safe programming language has coalesced around Rust as the main alternative. But it's not the only memory-safe language. Why not others like Go or Java?

The primary reasons are:

- Memory safety compile-time enforcement, and
- No garbage collection.

Whereas Java has memory-safety, it also requires garbage collection for memory allocation. This is a significant runtime cost, and hinders the use of Java in latency-critical applications and low-level operating system code.

By way of comparison, Rust's use of borrows means that there's no need for garbage collection. The de-allocation of memory is automatic. Hence, Rust has a reputation as a strong choice for low-latency coding, and notably, is now being used as part of the code for the Linux kernel.

In Defense of C++

C++ has a lot going for it, and a wholesale move to Rust would involve massive upheaval. Advantages include:

- Large number of experienced and new developers.
- Strong ecosystem of tools and components.
- Standardized libraries of code (huge efforts).
- Existing installation codebase around the world.

Such an ecosystem didn't grow without a reason. Let us take a moment to remind ourselves of the inherent positives of the C++ programming language itself:

- Object oriented programming
- Modularity (classes)
- Type safety
- Speed and efficiency
- Portability (high-level language)
- Exception handling mechanisms

Types of Memory Safety

Memory errors are a large class of problems in C++ programs. Both Microsoft and Google reported that approximately 70% of their C++ program issues were related to memory safety. The main impacts are:

- Safety — glitches and crashes in programs.
- Security — buffer overflows and related memory vulnerabilities are attack vectors.

Memory safety errors can be split into two main types:

- Spatial (location-based)
- Temporal (time-based)

Spatial memory errors are those related to a bad address. Examples in C++ would include:

- Array address out-of-bounds
- Array address underflow

Temporal memory errors are time-related errors in the sequence of memory usage. The memory was previously valid, but is now invalid. Uninitialized memory is another example where the memory is not yet valid. Examples in C++ include:

- Double de-allocation.
- Use of stack addresses after stack unwinding.
- Use of de-allocated memory.

Concurrent and multi-threaded programming in C++ gives additional examples of temporal issues in parallel programming:

- Race conditions (write-after-read, read-after-write, write-after-write).
- Synchronization errors (underlying cause).

Detection versus Toleration

There are many areas where there is tension between detecting errors and resilient toleration of problems. These are the age-old debates about whether to leave debug code in production or not. If there is a failure for a customer, do we want it to be detected, bearing in mind that this will be perceived by the customer as a software failure, little different to other less graceful crashes. Or would we rather that the software quietly handles the error, and is thus resilient for the customer. An intermediate method would be to do both:

(a) Detect the internal error and log it, and

(b) Tolerate the error and continue execution.

Speed. The other issue: speed versus safety. How much more performance in terms of compute efficiency are we willing to give up to achieve these different types of error detection and resilient capabilities?

Generally speaking, there is a trade-off between how many errors can be detected, versus the execution time penalty for doing the additional checking. For example, in trying to detect memory errors via filling the block with magic values, we could use:

- None
- Magic value in the first address of a memory block.
- Magic values in the whole memory block.
- Hash table of addresses for tracking of blocks.
- Hash table for addresses and magic values in blocks.

Uninitialized memory errors. The problems with incorrect use of uninitialized memory error present a classic example of using detection versus resilience. Our debugging memory library can fill the uninitialized memory with data, with two strategies:

- Canary strategy — fill with magic non-zero values.
- Toleration strategy — fill with zeros (i.e., initialize it).

The canary strategy will detect the error, whereas the toleration strategy will make it harmless. Which one is better?

Compiler-Supported Memory Safety

Some of the specific features that could be used to improve memory safety include:

- Heap memory initialization (e.g., `malloc`, `new`)
- Stack memory initialization
- Double-deallocation detection
- Uninitialized memory detection
- Use-after-free memory detection
- Use after stack unwind memory detection

Note that these methods that initialize memory could either use a canary strategy with non-zero magic values to detect memory issues, or could zero the memory to make uninitialized-use errors harmless.

Hence, these safety methods should have multiple different options for handling uninitialized memory usage checking:

- Nothing
- Canary (detection with magic value filling)
- Zeroing (toleration harmlessly).

Memory safety is only one aspect in overall safe programming, although it's a major problem in C++. Other issues in C++ (and other languages, too) include:

- Arithmetic overflow and underflow
- Undefined behavior (non-standardized features)
- String and character processing
- File processing

Languages Can't Fix Everything

There are some things that neither Rust nor Safe C++ could possibly fix:

- Platform-specific features
- Low-level features

Areas of portability that are unlikely to be sorted out by your programming language include:

- Data type sizes (e.g., 32-bit vs 64-bit).
- Files and directories
- Database integrations
- Devices and peripherals
- Signals and interrupts
- Assembly language

There are also some more obscure C++ coding issues that are problematic for all languages:

- Endian-ness of numeric representations.
- ASCII versus EBCDIC character set.
- Internationalization with UTF8 and Unicode.

3. The Safe C++ Proposal

Introducing Safe C++

The movement to address safety issues in the C++ programming language has been spearheaded by major software vendors, notably Microsoft and Google, over many years. However, the issue went mainstream when the U.S. federal government, specifically the White House, went public with policies aimed at addressing memory safety issues in software.

This led to work on extending C++ so that it has full memory-safety with a view to creating an ISO standard for a language called Safe C++. The draft proposal for Safe C++ is available online here: `https://safecpp.org/draft.html`

This is an immense body of work that has been completed over the prior 18 months. The results are not only the proposed standard for Safe C++, but also a compiler that implements the proposal. The expected timeframe for full completion is another 18 months after this announced draft in September, 2024, which puts final completion into early 2026 by my calculations.

Note that there have been several other initiatives in regard to a "Safe C" language, and also "Safe C Standard Library" versions. However, this is the first initiative for C++ safety that attempts to address memory safety using a Rust-like model with borrows and lifetimes. The main advantages of this policy are:

- Memory safety guarantees (at compile-time!)
- Low performance cost (no garbage collection needed)

Note that the performance benefit is not only the lack of garbage collection overhead, but also that the compile-time guarantee of memory safety means that there is not a runtime cost to enforcing pointer safety. For example, pointers and arrays would not need a costly validation of their address at runtime.

The way of achieving this is quite involved, and the draft standard also provides some advice for compiler implementers. The main features of the Safe C++ proposal are examined below.

Goals of Safe C++

In a word: safety. But in a way that extends C++ with features similar to Rust's memory safety guarantees, while maintaining backward compatibility with all the existing C++ code. The over-arching goals are therefore:

- Reducing memory errors and their consequent failures.
- Increased security from the absence of memory errors.

The Safe C++ proposal aims to prevent memory errors rather than detecting them via an intrinsic modification to the memory management model. Some memory errors that should be prevented inside the safe sections of code include:

- Buffer overflows
- Null pointer dereferences
- Dangling pointers
- Array out-of-bounds errors
- Use-after-free memory errors
- Double-free errors
- Memory leaks

Note that memory leaks are resolved by guaranteeing automatic de-allocation of memory. The borrow method of managing all allocated memory does not require garbage collection, which is an inherent advantage of Rust added to Safe C++.

Safe C++ Extensions

The main methods used for safety in the Safe C++ proposal include:

- Safe contexts — code is split into safe and unsafe areas.
- Memory safety — prevention of buffer overflows, array bounds errors, and null pointer dereferences.
- Explicit mutation — clarification of when memory is modified.
- Borrow checking — guarantees memory addresses and de-allocation.

Note that Safe C++ is inspired by the borrow-lifetimes model in Rust for memory safety, but does not adopt the Rust language syntax. Rather, Safe C++ maintains the C++ style of syntax, while adding various safety-related extensions to the language. Safe C++ also does not adopt other neat features of Rust unrelated to safety, such as algebraic types or pattern matching. Instead, Safe C++ limits its focus to safety-related additions, which is more than enough to start!

Safe C++ Syntax

The Safe C++ syntax builds on standard C++ syntax, so that existing developers only need to learn the extensions. Some of the more interesting features of Rust are not added to Safe C++.

Safe C++ introduces a number of new keywords to the language, such as:

- `safe` — mark safe contexts for statements or functions.
- `unsafe` — unsafe contexts and also a specifier.
- `mut` — explicit mutation contexts indicating memory modification.
- `borrow` — use of an object without transferring ownership.
- `owned` — marks objects as being "owned" rather than transferred.
- `checked` — marks risky areas that are checked at compile-time.

The features of Safe C++ are initially enabled by this line at the top of the C++ code:

```
#feature on safety
```

Blocks of code can be declared as "safe" using that as a keyword:

```
safe {
    // code block
}
```

In a safe code block, all of the code must follow additional rules in relation to memory safety, such as for pointers and arrays.

Various safe versions of the standard C++ library are available via "`std2::`" rather than "`std::`" prefixes. Thus, the Safe C++ proposal requires significant additional changes to the standard C++ libraries.

The safe keyword is not only for code blocks, but can also be used as a specifier in declarations. You can declare a function as being "safe" via a special specifier keyword that is part of its type (like "`noexcept`"):

```
void myfunc() safe;
```

A special keyword "`mut`" specifies a mutable context, whereby C++ memory allocation is changed to Rust-like semantics with borrows and lifetimes.

Unsafe code blocks can be explicitly created inside Safe C++ functions using the "`unsafe`" keyword. Here's an example: using:

```
unsafe {
    // Block of horrible code
}
```

The `unsafe` keyword can also be used in other contexts, such as types or array-deference operators.

Supporting Safe C++

The draft proposal for Safe C++ needs support from the overall C++ community. The intention is to develop the standard and then attempt to ratify it as an official ISO standard. Please take the time to review the draft proposal and give it the full attention that it deserves.

This proposal needs additional support from industry and feedback to make its way through the standards process. It already represents over 18 months of work on both the standards document and a Circle compiler that implements the proposal. This plan is expected to take another 18 months to complete the proposal and to implement a compiler and standard library for Safe C++.

Unfortunately, the Safe C++ language is not here already for businesses to use, and will take some time to come to fruition, with the current estimate putting project completion in early 2026. I'm not recommending a switch to Rust programming in the interim. That would be a massive dislocation in software development efforts, although I'm sure some organizations will make that choice.

Personally, I am recommending a strategy for memory safety that involves doing some short-term pragmatic actions toward C++ quality improvements. There are numerous ways to address memory safety and other areas of weakness in the C++ language, while retaining acceptance performance cost. However, the main problem with this approach is that, although pragmatic in many ways, fully resolving some of the intractable memory safety issues are too compute-expensive to achieve in the current C++ language model. Hence, the long-term goal has to be the compile-time memory safety guarantees, with zero runtime cost, that are achieved in the Safe C++ proposal.

4. Pragmatic Safe C++

Don't Wait!

There are many actions you can do now to improve C++ safety and resilience. Many techniques can improve the quality of the code and harden it against bugs and security glitches. I am certainly not an advocate of switching to Rust, and these techniques will improve C++ safety while awaiting the full compile-time guarantees of the Safe C++ standard.

Many of these approaches are internal to the software development processes, and do not impact the execution speed of the product at the customer's site at all. Examples of these zero-impact approaches include:

- Automated testing (i.e., unit testing, regression testing)
- Source code analysis tools
- Using runtime memory checkers in the lab

However, there are a few approaches that may impact performance for customers, whether this means paying external customers or the internal "customers" using your code in production.

Approaches that can reduce speed include:

- Leaving self-testing code in the production build (e.g., assertions, self-tests, parameter validation).
- Running dynamic "debug libraries" that can detect and mitigate various memory issues.

The performance impact can range from minimal (e.g., testing return codes) to quite expensive (e.g., memory address validation to the same level as `valgrind`).

I don't think your customers want that last one.

Hardening C++

Here are some ways to "harden" C++ code against both memory bugs and security vulnerabilities:

- Unit testing
- Assertions
- Self-testing code blocks
- Debug tracing
- Function parameter validation
- Module-level self-tests
- Error checking of function return codes

Tools and environments are another area to optimize settings for safety:

- Compiler safety options
- Linters and static analyzers
- Debug wrapper libraries
- Memory error detection tools (sanitizers)

Automation of the "nightly builds" and other build-related automatic testing can be improved:

- Warning-free builds
- Run unit tests via CI/CD approval automation (run-time hardening)
- Run longer regression tests on nightly builds (if too slow for CI/CD).
- Build on multiple compilers and platforms (compile-time and run-time hardening)
- Build with different optimization levels
- Have multiple build paths with more or less warnings from compilers and/or linters.
- Make sure someone's fixing all these warnings!

General policies around the development of C++ for greater reliability include:

- Coding policies
- Code review on pull requests
- CI/CD automation
- Automated testing harnesses

Software development management issues include overall reliability of the whole workflow:

- Source code control systems (i.e., `git`, svn, `cvs`)
- Bug tracking systems
- Support case tracking systems
- Third-party library management and updates policy
- Release management
- Executables and debug versions management
- Backups policy

Safer Production C++

There are some safe C++ techniques that are fast enough to be considered for production release. As already mentioned, a lot of assertions, function return checks, parameter validation checks, and simple self-tests can be optionally left in production code. The assumption here is that these are all only single value comparisons, and are thus not costly.

Let us examine a few more speedy self-tests. Some of my targets in this section include:

- Uninitialized memory usage.
- Already-deallocated memory usage.
- Double-deallocation errors.

These are whole categories of C++ memory safety errors that could be prevented. Let us examine these ideas in more detail.

Uninitialized memory usage. The simplest idea is to initialize all memory. The main primitives that create uninitialized memory are `malloc` and `new`, and there is also `realloc` whenever it expands the block. There are also uninitialized stack memory blocks from `alloca`. There are two ways to fix this:

- Auto-intercepts via macros or link-time wrappers that zero the memory.
- Coding policies requiring an immediate `call` to memset after these routines.

Surely using `memset` is very efficient, and this policy will prevent a whole swathe of common memory errors.

Note that this does not address all types of uninitialized memory, as it does not intercept stack-local variables, such as simple variables and uninitialized non-`static` arrays. These can be addressed by a coding policy of never declare any variable without an initializer. This is not necessary for global variables or `static` variables as these are already initialized to zero as part of standard C++.

De-allocated memory detection. A simple trick to prevent most de-allocated memory usage errors is to write a four-byte magic value into the first bytes of the deallocated block. Since this memory is being de-allocated, you can write whatever values you want into it. This method requires the macro or link-time interception of `free` and `delete`.

This method can especially prevent (and detect) any double-deallocation memory errors. It is easier to do this check because both `free` and `delete` should always be given an address at the start of an allocated memory block. Hence, these deallocation primitives should first check for the magic value (and avoid deallocation if found), before setting the magic value themselves before deallocation. This method can trigger a few false positives, resulting in only memory leaks, and also requires intercepted memory allocation primitives to ensure that no blocks are less than four bytes.

Safer coding policies. If your preference is the use of coding policy guidelines for safer C++ (rather than macro-interception of primitives), some of the ways to address memory safety include:

- Prefer `calloc` to `malloc`
- Follow calls to `malloc` or non-object `new` operations with `memset` to zero.
- Precede calls to `free` with `memset`, if the size is known.
- Add `memset` at the end of destructors (assuming the size is known).
- Alternatively, write magic values before `free` or non-object `delete`, and at the end of destructors.

There are various other coding policies available on the internet for safer C++ coding. Many of these are focused on secure C++ coding, which mostly achieves the same thing.

Building QA-Enabled Products

Developers love to get assigned work by the QA department. Hence, it's beneficial to build testing enablement capabilities directly into the product itself, to make the life of the QA staff easier.

Some ideas for building testing enablement into your product:

- Build separate "debug" versions of your executable with more enabled self-testing code (this is not just the debug symbols, but enabling memory checking, stack canaries, or other internal safety features).
- Command-line interface for easier automated testing with scripts and test harnesses.
- Test-containing version: a debug version that is linked with the unit tests and has a "`-test`" command-line option that runs the self-tests itself.
- Add a "`-safe`" command-line option that enables additional internal memory safe-checking.

Many of these ideas can also be combined with "supportability" initiatives. After all, product support is like on-site QA. Some of the opportunities to increase supportability for customers include:

- Simple way to detect full context details (e.g., build dates and numbers, versions, etc.)
- Unique error codes in all error messages that customers might see.
- Printing error context details or full stack backtrace for serious failures or also in logging of less serious problems such as "soft assertions."

Note that these product features are not just for the QA process, since these safety capabilities can also be used during development in the automated test runs or the nightly builds.

Triggering Bugs Earlier

A lot of bugs can be found using the techniques already mentioned. The above approaches are very powerful, but they can also be limited in some less common situations:

- Intermittent bugs — hard to reproduce bugs.
- Silent bugs — how would you even know?

You can't really find a bug with `gdb` or the `valgrind` memory checker if you can't reproduce the failure. It's probably a memory error nevertheless. On the other hand, an intermittent failure might be a race condition or other synchronization error.

Silent bugs are even worse, because you don't know they exist. I mean, they're not really a problem, because nobody's logged a ticket to fix it, but you just know it'll happen in production at the biggest customer site in the middle of Shark Week.

How do you shake out more bugs? Here are some thoughts:

- Add self-testing code with more complex sanity checks.
- Consider debug wrapper functions with extra self-testing.
- Add more function parameter validation
- Auto-wrap function calls ensure error return checking for *all* calls.

Consider non-memory bugs with changes such as:

- Arithmetic overflow or underflow is a very silent bug for both integers and floating-point (e.g., check `unsigned` integers aren't so high they'd be negative if converted to `int`).
- Add some assertions on arithmetic operations (e.g., tests for floating-point `NaN` or negative zero).

With all of these things, any extra runtime testing code requires a shipping policy choice: remove it for production, leave it in for production, only leave it in for beta customers, leave in only the fast checks, and so on.

If you're still struggling with an unsolvable bug, here are a few "hail Mary" passes into the endzone:

- Review the latest code changes; it's often just a basic mistake hidden by "code blindness."
- Add a lot of calls to synchronization primitives or run single-threaded to rule out concurrency issue.
- Try `memset` after `malloc` or `new`, or change to `calloc`.

And some other practical housekeeping tips can help with detecting new bugs as soon as they enter the source code.

Plan ahead for future failure detection.

- Examine compiler warnings and have a "warning-free build" policy.
- Have a separate "`make lint`" build path with lots more warnings enabled.
- Keep track of random number generator seeds for reproducibility.
- Add some portability sanity checks, such as: `static_assert(sizeof(float)==4);`

I guarantee that last one will save your bacon one day!

Compiler Vendor Safety

The various compiler vendors could assist in increasing the level of safety in C++. Let us examine the use of additional safety in existing practices, as an interim step before moving to full memory safety in a future Safe C++ standard. The ideas below assume the compiler vendors could make changes to:

(a) The code generation features of some operators, and

(b) The Standard C++ library routines (e.g., `malloc` and `new`).

This is also just a first commentary. I am sure that the compiler designers who do this kind of stuff all day long could come up with a much more extensive proposal, perhaps with additional levels of trade-offs and individual settings for various sub-types of techniques.

The focus here is to go beyond what is possible via macro or link-time intercepts in your own safety wrapper library. There are additional techniques that can only be applied by the compiler, which are difficult or impossible to do via intercepts. The idea of this section is not necessarily for the safety modes to be used in production, although perhaps the lower levels could be, but to allow multiple levels of safety runs to be used in development practices. For example, a separate build path to run the unit tests at each different safety level.

The basic idea is a simple option:

```
-safe
```

This would turn on safety levels for all of the code by default.

This could be overridden by `unsafe` and `safe` blocks, as in the Safe C++ proposal. The situations where some memory blocks are allocated or initialized in an `unsafe` block needs to be considered carefully.

The extra capabilities that this first-level safety option could enable would include ways to increase the overall safety of the program, focused on tolerance rather than detection. Such ideas include:

- `malloc` and `new` should initialize memory bytes to zero
- `realloc` also, when it extends the memory
- Stack memory for automatic variables should initialize all to zero.
- The `alloca` stack allocation primitive should also zero the bytes.

These methods would not allow toleration of errors, but only reducing their occurrence. Other possibilities involving both detection and toleration include:

- The standard library functions should all tolerate a null pointer argument without crashing for all routines.
- Invalid parameters to the standard library should also be blocked (e.g., zero size to `memset`).
- The `*` and `[]` operators can prevent null pointer uses with a basic test.
- `free` and `delete` should use bit flags in the memory header block to detect and avoid many invalid address de-allocations.
- Similarly, double deallocation errors could be detected and made harmless.

There are many other possibilities on the theme of "only requires an integer or pointer test" for higher safety with detection and/or toleration. Obviously, the compiler could offer each of these capabilities as a separate option, too, rather than grouped into a particular safety level.

Level 2 Safety

The next level of safety would be possible via a compiler option:

```
-safe=2
```

The idea of the second-level safety is to use magic values and/or canaries as part of the safety net. The initial check is an integer test of a simple four-byte magic value (or canary), which indicates the high likelihood of an error, in which case a more expensive analysis of an address can be performed. Overall, this would be very efficient, but occasionally having false positives. This level two safety check goes beyond integer or pointer tests, and adds a superset of additional safety checks.

Examples of capabilities include:

- `malloc` and `new` add magic bytes in their header control block, before their allocated bytes as usual, to detect overwrites from array underruns. Alternatively, the header control block itself can be checked for consistency, without using extra bytes.
- Automatic simple variables are initialized to zero.
- Array memory blocks on the stack have a canary region at both ends, and a magic value in their first bytes if not initialized.
- The `alloca` stack allocation function similarly uses two canary regions and a magic value at the start.
- Global arrays and memory blocks also have a small canary region at both ends (but they are initialized to zero in standard C++, so there is no magic value in the first bytes).
- Similarly, they add extra bytes (e.g., four) at the end with canary magic bytes to detect array overruns.
- malloc and `new` `put` a single magic value, possibly a four-byte integer, at the start of their block indicating "not yet initialized." The rest of the bytes inside the block are zeroed for safety.
- Deallocation by `free` or `delete` would set a marker in the header control block, and also a magic value at the first address of the memory block to indicate "already freed" status. (They would first validate the magic value to detect freeing uninitialized memory, and check canary overruns at both ends of the block.)
- Library routines that use or write to an address can check for this magic byte at the start of the memory block, and only if the value appears to be a faulty magic byte (indicating never-initialized or already-freed blocks), initiate more complex tests to check if it's an invalid block. This only detects cases involving addresses at the start of a block, rather than in the middle.

The goal of level two safety is to check magic values, typically a single integer value, which is relatively efficient. This finds an increasing level of errors, but does not detect all cases:

- Addresses in the middle of a block are not easily validated.
- Array overruns or underruns via `*` and `[]` operators are not detected immediately, but may be detected by overruns.

It would be too expensive to modify every `*` and `[]` operator to detect these memory errors.

However, compilers are already able to auto-detect certain types of loops (e.g., auto-vectorization), in which case the very first access could be checked, and possibly the full extent of the loop could be analyzed to determine array overruns by the end of the loop.

Level 3 Safety

The idea of level 3 safety is similar to what is available from sanitizers such as `valgrind`. As such, its performance may be sluggish and inappropriate for production usage. Since this type of performance is already available via sanitizers, it may be unnecessary for compiler vendors to add this functionality directly. However, it should be noted some of the checkers and sanitizers have limitations (e.g., `valgrind` cannot detect overruns on non-allocated global memory or stack memory blocks), whereas compiler designers could offer greater capabilities.

The basic design of this capability involves:

- All addresses are validated and checked, even those in the middle of a block, or completely outside all blocks.
- All types of memory blocks are validated, including stack memory, read-only memory, and global memory, whereas the previous levels focused on heap blocks.
- All library functions have their address parameters validated.
- All address-related operations, such as `*` and `[]`, will have their addresses validate.

I remain unconvinced as to the necessity of this high-level safety capability in compilers. This would be extremely slow to run, and serves as a reminder that what is really needed is the compile-time memory safety guarantees in the Safe C++ proposal, as these do not have any performance impact at all!

5. AI Safety

What is AI Safety?

This chapter explores a number of the additional safety issues that arise in making an AI application based on Large Language Models (LLMs). Although other chapters focus more on lower-level C++ issues, they are nevertheless applicable to AI applications because most of the low-level AI engine code is in C++.

This is mainly because C++ is fast, and AI engines tend to run slow, because they have billions of parameters to slow them down.

The main additional safety issues in AI application arise because of the weird properties of LLMs. These are not coding glitches, but are inherent properties in neural networks. Some of the issues include

- Hallucinations — LLMs make up plausible but false answers.
- Bias and fairness
- Toxicity
- Refusal issues

This chapter is a broad overview of some of these issues. The issue of AI safety would fully deserve a whole book on the topic.

AI Quality

A quality AI would predict my wishes and wash my dishes. While we wait for that to happen, the desirable qualities of an AI engine include:

- Accuracy
- Sensitivity
- Empathy
- Predictability
- Alignment

Much as I like code, a lot of the "smartness" of the LLM starts with the training data. Garbage in, garbage out! Finding enough quality data that is ratified to use for model fine-tuning or a RAG database is one of the hurdles that delays business deployment of AI applications. Another problem with data quality is that new models are starting to be trained using the outputs of other models, and this "synthetic data" is leading to degradation in these downstream models.

At the other end of the quality spectrum, we've seen the headlines about the various types of malfeasance that a low-quality AI engine could perform, such as:

- Bias
- Toxicity
- Inappropriateness
- Hallucinations (i.e., fake answers)
- Wrong answers (e.g., from inaccurate training data)
- Dangerous answers (e.g., mushroom collecting techniques)
- Going "rogue"

And some of the technical limitations and problems that have been seen in various AI applications include:

- Lack of common sense
- Difficulty with mathematical reasoning
- Explainability/attribution difficulty
- Overconfidence
- Model drift (declining accuracy over time)
- Catastrophic forgetting (esp. in long texts)
- Lack of a "world view"
- Training cut-off dates
- Difficulty with time-related queries (e.g., "What is significant about today?")
- Problems handling tabular input data (e.g., spreadsheets)
- Banal writing that lacks emotion and "heart" (it's a robot!)

If you ask me, almost the exact same list would apply to any human toddler, although at least ChatGPT doesn't pour sand in your ear or explain enthusiastically that "Dad likes wine" during show-and-tell. Personally, I think it's still a long road to Artificial General Intelligence (AGI).

Unfortunately, every single bullet point in the above paragraphs is a whole research area in itself. Everyone's trying to find methods to improve the smartness and reduce the dumbness.

Failure Stories for Generative AI

Cautionary tales abound about Generative AI. It's a new technology and some companies have released their apps without fully vetting them.

Arguably, sometimes it's a simple fact that it's too hard to know all the possible failures ahead of time with such a new tech stack, but risk mitigation is nevertheless desirable.

Here's a list of some public AI failures:

- ChatGPT giving potentially dangerous advice about mushroom picking.
- Google's release of AI that had incorrect image generation about historical figures.
- Air Canada's lost lawsuit over a chatbot's wrong bereavement flight policy advice.
- Google Gemini advising to "eat rocks" for good health, and "use glue" on pizza so the cheese sticks.
- Snapchat's My AI glitch that caused it to "go rogue" and post stories.

There are some conclusions to draw on the causes of generative AI failures. Many possible problems can arise:

- Hallucinations
- Toxicity
- Bias
- Incorrect information
- Outdated information
- Privacy breaches.

And that's only the short list! More details are available later in the chapter.

Consequences of AI Failures

The public failures of AI projects have tended to have severe consequences for the business. The negative results can include:

- PR disasters
- Lawsuits
- Regulatory enforcement

- Stock price decline

However, these very public consequences are probably in the minority, although they've become known in the media. The more mundane consequences for generative AI projects include:

- Not production-ready. Generative AI projects often get stuck in proof-of-concept status.
- Excessive costs.
- Poor ROI.
- Not business goal focused. There's a tendency to use generative AI for a project because it's gotten so much attention, but where the project goal itself is not well aligned with the business.
- Team capabilities exceeded. Some of this AI stuff is hard to do, and may need some upskilling.
- Limitations of generative AI. There are various types of projects for which generative AI is not a good fit, and it would be better to use traditional predictive AI, or even non-AI heuristics (gasp!).
- Legal signoff withheld or delayed (probably for good reason).

A lot of these project-related issues are improving quite quickly. Whereas a lot of AI projects for businesses were stuck in POC status, they are starting to emerge now into production usage. The outlook is optimistic that AI will start to deliver on its promised benefits for businesses and individuals.

Data Causes of AI Failures

Not all failures are due to the model or the AI engine itself. The data is another problematic area, with issues such as:

- Surfacing incorrect or outdated information (e.g., everything on the company's website gets potentially read by the AI engine, and it doesn't know if it's incorrect).
- Sensitive data leakage. Accidentally surfacing confidential or proprietary data via an AI engine can occur, such as if the training data hasn't been properly screened for such content. If you're putting a disk full of PDF documents into your fine-tuning or RAG architecture, better be sure there's no internal-use-only reports in there.
- Private data leakage. Another problem with using internal documents, or even public website data, is that they may accidentally contain private personally-identifying individual information about customers or staff.

- IP leakage. For example, if your programmers upload source code to cloud AI for analysis or code checking, it might be exposing trade secrets or other IP. Worse, the secret IP could end up used for training and available to many other users.
- History storage. Some sensitive data could be retained in the cloud for a much longer time than expected, if your cloud AI is maintaining session or upload histories about its users.

The LLM isn't "self-aware" enough to know when the data is faulty. In typical usage, the LLM will take any data at face value, rather than trying to judge its authenticity. LLMs are not particularly good at identifying sarcasm or the underlying bias of a particular source.

Types of AI Safety Issues

There are a variety of distinct issue in terms of appropriate use of AI. Some of the categories include:

- Bias and fairness
- Inaccurate results
- Imaginary results ("hallucinations")
- Inappropriate responses

There are some quality issues that get quite close to being philosophy rather than technology:

- Alignment (ensuring AI engines are "aligned" with human goals)
- Overrideability/interruptibility
- Obedience vs autonomy

There are some overarching issues for AI matters for the government and in the community:

- Ethics
- Governance
- Regulation
- Auditing and Enforcement
- Risk Mitigation

Code reliability. A lot of the discussion of AI safety overlooks some of the low-level aspects of coding up a Transformer. It's nice that everyone thinks that programmers are perfect at writing bug-free code. Even better is that if an LLM outputs something wrong, we can just blame the data. Hence, since we may rely on AI models in various real-world situations, including dangerous real-time situations like driving a car, there are some practical technological issues ensuring that AI engines operate safely and reliably within their basic operational scope:

- Testing and Debugging (simply avoiding coding "bugs" in complex AI engines)
- Real-time performance profiling ("de-slugging")
- Error Handling (tolerance of internal or external errors)
- Code Resilience (handling unexpected inputs or situations reasonably)

Jailbreaks

Let us not forget the wonderful hackers, who can also now use words to their advantage. The idea of "jailbreaks" is to use prompt engineering to get the LLM to answer questions that it's trained to refuse, or to otherwise act in ways that are different to how it was trained.

It's like a game of trying to get your polite friend to cuss.

Sometimes the objectives of jailbreaking are serious misuses, and sometimes it's just to poke fun at the LLM. Even this is actually a serious concern if something dumb the LLM says goes viral on TikTok.

There are various types of jailbreaks for each different model. Sometimes it's exploiting a bug or idiosyncrasy of a model. There was a recent example with a prompt that contained long sequences of punctuation characters, which for some reason caused some models to get confused.

Another type is to use the user's prompt text to effectively override all other global instructions, such as "forget all previous instructions" or overriding a persona with "pretend you are a disgruntled customer."

These prompt-based instruction-override jailbreaks work because of the way that global instructions and user queries are ultimately concatenated together, and many models don't know which is which.

Models need explicit training against these types of jailbreaks, which is usually part of refusal training. This type of training is tricky and needs broad coverage.

For example, a recent paper found that many refusal modules could be bypassed simply by posing the inappropriate requests in past tense ("how did they make bombs?") rather than present tense ("how to make a bomb?"), which shows the fragility and specificity of refusal training.

Risk Mitigations

When building and launching a generative AI project, consider taking risk mitigation actions, such as:

- Data cleaning
- LLM safety evaluations
- Red teaming
- Expedited update process

Safety in, safety out. Data quality issues can cause a variety of harms. Some of the areas to filter in a training data set or a RAG content datastore, include:

- Profanity (cuss words)
- Sensitive topics
- Insults, anger, or harmful tones
- Personally identifiable information (e.g., names, cell phone numbers, postal addresses, email addresses, etc.)
- Personal financial details (e.g., credit card numbers, bank details, credit reports, lists of transactions)
- Personal identity numbers (e.g., social security numbers, drivers' licenses, passport details)
- Personal histories (e.g., what products they bought from you, or what web pages they visited).
- Out-of-date information
- Company proprietary information
- Internal conversations about issues (e.g., in an internal support database)

Being update-ready. LLMs are too flexible for you to realistically cover all the problems ahead of time. Hence, when you launch a new AI-based application, your team should be ready to quickly address issues as they arise with users.

If an odd response from your chatbot goes viral on social media, you'll want to block that problem quickly. It's not good if you have a 48-hour build process to put a correction live. Rather, you ideally would have a configurable prompt shield method, which can be configured on-the-fly with new query strings to block, so that users get a very polite refusal message instead of fodder for all their TikTok followers.

Refusal Modules and Prompt Shields

LLMs have "refusal" modules designed to stop it from telling you how to build a nuclear weapon in your garage. Mostly, these responses are trained into the weights of the module using specialized data sets, but there are also "prompt shield" modules designed to stop dubious queries ever getting to the model.

There are literally dozens of different types of malfeasance that LLM refusal training data sets have to contend with. Maybe don't look too closely into the text of that data.

Some models do better than others at refusing inappropriate requests, and there are even leaderboards for "security" of LLMs on the internet.

Prompt shields are modules that block inappropriate queries. They differ from refusal modules in LLMs in that they block the query *before* it goes to the LLM.

These modules can be designed in heuristic ways (e.g., block all queries with cuss words), or, for more generality, use a small LLM to do a quick check via "sentiment analysis" of the appropriateness of the topic of the query as a pre-check.

Prompt shields can also act as a minor speedup to inference engines because they reduce the load on the main LLM. They can block not only inappropriate questions, but other miscellaneous incorrect queries, such as all blanks, or all punctuation marks.

On the other hand, maybe you want to send those typo-like queries through to your bot so that it can give a cute answer to the user. On the other, other hand, one of the recent obscure jailbreak queries that was discovered used a query with dozens of repeated commas in the text, so maybe you just want to block anything that looks weird.

AI Engine Reliability

If your C++ application is an AI engine kernel, there are a lot of issues with reliability. We want our AI model to be predictable, not irrational. And it should show bravery in the face of adversity, rather than crumble into instability at the first sign of prompt confusion. At a high-level, there are various facets to AI engine reliability:

- Accuracy of model responses
- Safety issues (e.g., bias, toxicity)
- Engine basic quality (e.g., not crashing or spinning)
- Resilience to dubious inputs
- Scalability to many users

How to make a foundation model that's smart and accurate is a whole discipline in itself. The issues include the various training and other algorithms in the Transformer architecture, along with the general quality of the training dataset. Such issues aren't covered in this chapter.

Aspects of the C++ code inside your Transformer engine are important for its basic quality. Writing C++ that doesn't crash or spin is a code quality issue with many techniques. This involves coding methods such as assertions and self-testing, along with external quality assurance techniques that examine the product from the outside.

Resilience is tolerance of situations that were largely unexpected by programmers. Appropriate handling of questionable inputs is a cross between a coding issue and a model accuracy issue, depending on what type of inputs are causing the problem. Similarly, the engine should be able to cope with resource failures, or at least to gracefully fail with a meaningful response to users in such cases. Checking return statuses and exception handling is a well-known issue here.

A system is only as reliable as its worst component. Hence, it's not just the Transformer and LLM to consider, but also the quality of the other components, such as:

- Backend server software (e.g., web server, request scheduler)
- RAG components (e.g., retriever and document database)
- Vector database
- Application-specific logic (i.e., whatever your "AI thingy" does)
- Output formatting component
- User interface

Most other chapters in this book are about how to make your C++ code reliable, whether it's in an AI engine or other components. This includes various aspects of "code quality" and also ways to tolerate problems such as exception handing and defensive programming.

6. Safe C++ Tools

Tools Overview

There are several distinct types of C++ tools that improve the overall quality. Some of them focus on memory safety issues, whereas other tools have a broader range of features. The general categories with a specific focus on safety issues include:

- Runtime memory checkers (i.e., sanitizers, `valgrind`)
- General sanitizers (non-memory issues)
- Linters and static analysis tools
- Automated test harnesses
- Test coverage tools
- Security vulnerability analysis tools

In addition, some of the general programming tools are important in making a significant impact on programmer productivity and overall quality. These include:

- IDE environments (without and without AI copilots)
- Compilers (with useful warnings and runtime features)
- Interactive debuggers (IDE-based debuggers and `gdb`)
- Performance profilers (IDE-based or command-line)
- Tracing tools

Runtime Memory Checkers

There are a variety of runtime memory checkers available to find memory errors in C++. I remember using Purify back in the 1990s, and it still exists today. However, there are several free high-quality memory sanitizer tools now. Some examples include:

- Valgrind
- AddressSanitizer (Asan)
- MemorySanitizer (Msan)
- LeakSanitizer

There are also runtime checkers with a broader focus than memory safety issues:

- ThreadSanitizer (Tsan)
- UndefinedBehaviorSanitizer (UBSan)

Several compilers and IDEs have builtin support for running sanitizers. GCC has options such as:

```
-fsanitize=address
-fsanitize=kernel-address
-fsanitize=hwaddress
-fsanitize=thread
-fsanitize=leak
-fsanitize=undefined
-fsanitize=signed-integer-overflow
-fsanitize=bounds
```

That's only some of the GCC runtime error checking options for sanitizing. There are many more options, including granular control over specific types of error checks.

Valgrind

The Linux version of Valgrind Memcheck is very capable and well supported. The method to use the Valgrind tool for Linux on your application is simply to run the executable:

```
valgrind a.out
```

If Valgrind is not installed in your Linux environment, you'll need to do something like this:

```
apt install valgrind
```

The start of the Valgrind output is like this:

```
==1143== Memcheck, a memory error detector
==1143== Copyright (C) 2002-2017, and GNU GPL'd, by Julian
Seward et al.
==1143== Using Valgrind-3.18.1 and LibVEX; rerun with -h for
copyright info
==1143== Command: ./a.out
```

As it executes your program, the output from your program will be interleaved with error reports from Valgrind. Hopefully, there won't be any!

The end of the Valgrind execution gives you a nice summary of memory leaks and errors (abridged):

```
==1143== HEAP SUMMARY:
==1143==   in use at exit: 12,710,766 bytes in 10,810 blocks
==1143==   total heap usage: 15,851 allocs, 5,041 frees,
           47,396,077 bytes allocated
==1143==
==1143== LEAK SUMMARY:
==1143==  definitely lost: 0 bytes in 0 blocks
==1143==  indirectly lost: 0 bytes in 0 blocks
==1143==      possibly lost: 30,965 bytes in 199 blocks
==1143==  still reachable: 12,679,801 bytes in 10,611 blocks
==1143==      suppressed: 0 bytes in 0 blocks
==1143== Rerun with --leak-check=full to see leaked memory
==1143==
==1143== ERROR SUMMARY: 0 errors from 0 contexts
```

Running programs in Valgrind is obviously slower because of the instrumentation, but this is also true of similar sanitizer tools.

Gnu Debugger: gdb

I'm a big fan of `gdb` for debugging standard C++ on Linux. The basic commands for `gdb` are:

- `r` or `run` — run the code (with optional arguments), or restart if already running.
- `c` or `continue` — continue running (after stopping at a breakpoint).
- `s` or `step` — stepping through statements (also just `Enter`).
- `where` — stack trace (also aliased to "`bt`" for backtrace).
- `list` — source code listing
- `p` or `print` — print a variable or expression.
- `up`
- `n` or `next`

Abnormal program termination

gdb batch mode. You can detect this program crash better in gdb as it will trap the signals, so just run an interactive debugging session. Alternatively, if you have a simple reproducible case, you can automate this with batch mode, where the command to run is like this:

```
gdb -batch -x=gdbtest.txt a.out
```

The batch input file is a set of gdb commands:

```
run
where
exit
```

Here's an example output (abridged):

```
Thread 1 "a.out" received signal SIGSEGV, Segmentation fault.
0x00007ffff7cdfa4e in ?? () from /lib/x86_64-linux-gnu/libc.so.6
#0  0x00007ffff7cdfa4e in ? from /lib/x86_64-linux-gnu/libc.so.6
#1  0x000055555555fdb5 in aussie_malloc(void**, int) ()
#2  0x0000555555562ea9 in aussie_run_clear_vector(int) ()
#3  0x00005555555633aa in main ()
A debugging session is active.
Inferior 1 [process 5143] will be killed.
Quit anyway? (y or n) [answered Y; input not from terminal]
```

There are various other useful things that can be automated using batch gdb and various script commands. For example, you can use it as a trace mechanism that prints out the stack trace at every call to a certain function.

Pre-Breakpointing Trick

One advanced tip for using gdb is to define a function called "breakpoint" in your C++ application. Here's an example:

```
void breakpoint()
{
    volatile int x = 0;
    x = 0;  // Set breakpoint here
}
```

It looks like a silly function, but it serves one useful purpose. The idea is that when you start a new interactive debugging session with `gdb`, or automatically in your "`.gdbinit`" resource file, you can set a breakpoint there:

```
b breakpoint
```

Why do that? The reason is that you also add calls to your "`breakpoint`" function at relevant points in various places where failures can occur:

- Error check macros
- Assertion macros
- Debug wrapper function failure detection
- Unit test failures

Hence, if any of those bad things happen while you're running interactively in the debugger, you're immediately stopped at exactly that point. If you're not running in the debugger, this is a very fast function (though admittedly, it can't be `inline`!), so it doesn't slow things down much. You can even consider leaving this in production code, since the `breakpoint` function is only called in rare situations where a serious failure has already occurred, in which case execution speed is not a priority.

This technique is particularly useful because don't have to go back and figure out how to reproduce the failure, which can be difficult to do for some types of intermittent failures from race conditions or other synchronization problems. Instead, it's already been pre-breakpointed for you, with the cursor blinking at you, politely asking you to debug it right now, or maybe after lunch.

Postmortem Debugging

Postmortem debugging involves trying to debug a program crash, such as a "core dump" on Linux. In this situation, you should have a "core" file that you can load into `gdb`. The command to use is:

```
gdb a.out core
```

Unfortunately, not all errors in an application will trigger a core dump, so you might have nothing to debug if it doesn't.

Programmatic C++ core dumps. One way to ensure that you get a `core` file is to trigger one yourself with the `abort` function. For example, you might do this in your assertion failure routines or other internally self-detected error states.

You can even do this without exiting your application! If you're wanting to have your application to take control of its own core dumps (e.g., exceptions, assertion failures, etc.), there are various points:

- You can always `fork`-and-`abort` on Linux.
- Surely you can write some code to crash!

On the other hand, maybe you're only thinking about core dumps because you want to save debug context information. Doing this might obviate the need for a core dump:

- Use `std::backtrace` or another backtrace library.
- Print error context information (e.g., user's query)
- Print platform details

Customer core dumps. One of the supportability issues with postmortem debugging is that you want your customers to be able to submit a `core` file that they have triggered in your application. These are usually large files, so there are logistical issues to overcome with uploads.

Another issue is that in order to run `gdb` on a `core` file, the developer needs to have exactly the right executable that created the core dump. Hence, your build and release management needs to maintain available copies of all executable files in versions shipped to customers or in beta testing (or to internal customers for in-house applications). This means not only tracking the production releases of stripped executables, but also the correlated debug version of the executable with symbolic information.

Also, there needs to be a command-line option or other method whereby the phone support staff can instruct customers to report the exact version and build number of the executable they are using. It's easy to lose track!

References

1. GNU, Sep 2024 (accessed), *3.8 Options to Request or Suppress Warnings (GCC warning options)*, https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html
2. GNU, Oct 2024 (accessed), *3.12 Program Instrumentation Options* https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
3. GNU, Oct 2024 (accessed), *3.9 Options That Control Static Analysis*, https://gcc.gnu.org/onlinedocs/gcc/Static-Analyzer-Options.html

7. Non-Memory Safety Issues

The Other 30%

If Microsoft and Google both report that 70% of issues are related to memory safety problems, that means there are 30% that are *not*. What type of non-memory errors are problematic? Some of the main examples include:

- Logic errors (i.e., simple programming mistakes and other reasons).
- Arithmetic overflow and underflow
- File I/O errors
- Multithreading concurrency problems (e.g., race conditions).

Some of these are language-specific in relation to C++ syntax, whereas others arise in all programming languages. Multithreaded code or other types of concurrency is simply an order-of-magnitude more complex for programmers than sequential coding. Programmers are also prone to making all sorts of dumb mistakes or simple misunderstandings within the algorithm they are coding.

Code Blindness and Copy-Paste Errors

Serious errors in C++ software don't need to come from intrinsic properties of the programming language. Nor are they all related to memory safety or other undefined arithmetic issues.

There can also be simple logic errors arising from programmer fallibility. They are also very common in any use of "copy-paste" in coding.

There are many programming idioms that are commonly used by programmers and yet carry the risk of occasional serious errors. One of the main ways these errors get introduced is "copy-paste" of a block of code.

For example, one of the most common idioms is the use of an integer loop variable in a `for` loop. A correct `for` loop header looks like:

```
for (i = 1; i <= 10; i++)
```

However, when programmers "copy-and-paste" program statements there are some errors that often arise. When asked to loop down from 10 to 1, a lazy programmer will copy and change the above `for` loop header, a highly error-prone practice.

One such error is that ++ is not changed to -- as below:

```
for (i = 10; i >= 1; i++) // ERROR
```

This will cause a loop that is (almost) infinite. It will terminate only when integer overflow causes `i` to become negative.

A similar use of copy-and-paste without due care has caused a similar error in the code below with nested loops:

```
for (i = 1; i < n; i++)
    for (j = 0; j < n; i++) // ERROR
        arr[i][j] = 0;
```

Can you see the bug? It's hidden by "code blindness" if you can't.

Arithmetic Overflow and Underflow

Surprisingly, arithmetic errors are a reasonably common attack vector for malicious actors. Many C++ programs largely ignore the issue of overflow.

However, when you consider that integers are often used as indices in arrays and strings, it becomes clear that intentionally overflowing an index variable could cause modification to memory in any other locations. This is one level of indirection removed from buffer overflows, but it gets to the same thing.

What are the solutions to arithmetic overflow?

- Compiler-supported "safe arithmetic" modes.
- Manual self-checks for overflow and underflow.
- Safe integer wrapper classes.

Having a compiler safe mode that fixes arithmetic overflow is likely to be prohibitively expensive. Consider having every operator needing to check for integer overflow, or similarly all floating-point arithmetic needing to check for `NaN` or similar problems.

Fortunately, the effects of arithmetic overflow and underflow are well-defined in practice, even if they are officially "undefined behavior" in code. The effects are:

- Signed integer overflow — from `INT_MAX` to a large negative (`INT_MIN`).
- Signed integer underflow — from `INT_MIN` (negative) to `INT_MAX` (positive).
- Unsigned integer overflow — from `UINT_MAX` around to zero.
- Unsigned integer underflow — from zero around to `UINT_MAX`.

Self-testing arithmetic overflow. Self-tests for integer overflow can be things like this:

```
int i = sz * sizeof(float);
assert(i > 0);
```

However, note that the above may detect overflow in the lab, but if you have "soft assertions" that don't abort, then it doesn't actually prevent a malicious actor from abusing it. Instead, you could manually define the code:

```
if (i < 0) {
    assert(i > 0);
    abort();
}
```

However, that block of code reeks of copy-paste errors. Maybe you need a method of defining "hard assertions" like:

```
assert_abort(i > 0);
```

Testing for signed integer overflow becomes:

```
i++:
assert_abort(i > 0);
```

Testing for signed integer underflow becomes:

```
i--;
assert_abort(i < 0);
```

Testing for unsigned integer overflow is:

```
u++;
assert_abort(u != 0);
```

Testing for unsigned integer underflow is:

```
i--;
assert_abort(u != UINT_MAX);
```

Note that all of these are tests *after* the overflow or underflow has already occurred. This idea of "post-testing" for integer overflow also generalizes to other arithmetic operations, such as addition or multiplication. There is also hardware support in some CPUs for detecting an arithmetic operation that caused an overflow.

Pre-testing. Post-testing is probably acceptable, since it's not the actual arithmetic overflow or underflow that causes the vulnerability, but the misuse of the integer variable afterwards.

However, you can also do "pre-testing" of simple forms of integer overflow, such as from increment or decrement.

```
assert_abort(i != INT_MAX); // Int overflow pre-test
i++;

assert_abort(i != INT_MIN); // Int underflow pre-test
i--;

assert_abort(u != UINT_MAX); // Unsigned overflow
u++;

assert_abort(u != 0);  // Unsigned underflow pre-test
u--;
```

Note that unsigned arithmetic testing also applies to various commonly-used builtin types, such as `size_t`.

Insidious C++ Coding Errors

If you're one of the many who often ignore C++ compiler warnings, here's a few examples of things that cause insidious program failures. The only redeeming point: many of them get warnings from the C++ compiler.

Aliasing in the overloaded assignment operator

The definition of an overloaded "`operator=`" function for a class should always check for an assignment to itself (i.e., of the form "`x=x`"). Consider the following simple `MyString` class:

```
class MyString {
private:
    char* m_str;
public:
    MyString() { m_str = new char[1]; m_str[0] = '\0'; }
    MyString(char* s)
    {
            m_str = new char[strlen(s)+1];
            strcpy(m_str, s);
    }
    void operator =(const MyString& s);
    ~MyString() { delete[] m_str; }
    void print() { printf("STRING: %s\n", m_str); }
};

void MyString::operator = (const MyString& s)
{
    delete[] m_str; // delete old string
    m_str = new char[strlen(s.m_str) + 1]; // alloc memory
    strcpy(m_str, s.m_str); // copy new string
}
```

The above code looks fine, but this contains a hidden error that appears only if a string is assigned to itself. Consider the effect of the code:

```
MyString s("abc");
s = s;
s.print();
```

When the assignment operator is called, the argument `s` is the same as the object to which the member function is applied.

Therefore, the addresses `m_str` and `s.m_str` are the same pointer, and the `delete` operator deallocates an address that is immediately used in the subsequent `strlen` and strcpy `function` calls. Thus, these operations apply to an illegal address with undefined behavior, and it fails with a crash or garbage output.

This error is an example of a general problem of aliasing in the use of overloaded operators, especially the = operator. The object to which the operator is applied is an alias for the object passed as the argument. Any modifications to the data members also affect the data in the argument object. This type of error is very difficult to track down because it occurs only for one particular special case, and this case may not occur very often. This error is not restricted to `operator=` ,although this is its most common appearance. Similar aliasing errors may also occur in other operators such as +=, or in non-operator member functions that accept objects of the same type.

The correct idiom to avoid this problem of aliasing is to compare the implicit pointer, `this`, with the address of the argument object (which must be passed as a reference type). If these addresses are the same, the two objects are identical and appropriate action can be taken for this special case. For example, in the `MyString` class the correct action when assigning a string to itself is to make no changes, and the `operator=` function becomes:

```
void MyString::operator = (const MyString& s)
{
    if (this != &s) {  // Correct!
            delete[] m_str;
            m_str = new char[strlen(s.m_str) + 1];
            strcpy(m_str, s.m_str);
    }
}
```

Accidental empty loop

A common novice error with loops is to place a semicolon just after the header of a `for` or `while` loop. Syntactically, this is correct, so the compiler gives no error message. However, it changes the meaning of the loop. For example, consider the code:

```
for (i = 1; i <= 10; i++); // Extra semicolon
{
    ... // body of loop
}
```

This is interpreted as:

```
for (i = 1; i <= 10; i++)
    ;   // empty loop
{
    ... // body of loop executed only once
}
```

Semicolons are statements in C++.

The effect of this is that the body of the loop is assumed to be an empty loop by the compiler. The block after the loop header (the real loop body) is only executed after the loop has finished, and is executed only once.

Worse still, the accidental empty loop may cause an infinite loop if the condition is not being changed in the header.

Dangling else error

The rule that an `else` always matches the closest `if` is usually satisfactory. However, there are occasions where "dangling `else`" errors can arise in nested `if` statements such as:

```
if (y < 0)
    if (x < 0)
        x = 0;
else   // Bug!
    y = 0;
```

Based on the indentation used by the programmer, the `else` clause is presumably intended to match the first `if`. However, the compiler matches the else with the second (closest) `if`, and compiles the code as if it were written as:

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

The method of avoiding this error is to always use braces around the inner `if` statement when using nested `if` statements.

```
if (y < 0) {    // Correct
    if (x < 0)
        x = 0;
}
else
    y = 0;
```

sizeof array parameter

There is another situation when the `sizeof` operator computes surprising results when applied to a function parameter of array type. The error is illustrated by the following function:

```
void test_sizeof(int arr[3])
{
    printf("Size is %d\n", (int) sizeof(arr));
}
```

The computed size is expected to be `3*sizeof(int)`, usually 12. However, the actual result will usually be 4 or 8. This is because the `sizeof` operator is actually being applied to a pointer type. An array parameter is converted to the corresponding pointer type and it is this type that `sizeof` applied to. Therefore, the output result is exactly `sizeof(int*)`, which is the size of a pointer, commonly 4 or 8.

Accidental string literal concatenation

String concatenation is a relatively obscure feature of C++ that allows consecutive string literals to be merged into a single string literal. Concatenation of string literals takes place after the usual preprocessing tasks (i.e., after macro expansion), but before parsing.

An example of its usage is that the following code:

```
char *prompt = "Hello "
               "world";
```

This looks like a typo to beginner C++ programmers, but is totally valid C++ that will be equivalent to:

```
char *prompt = "Hello world";
```

Once you get used to it, this is a very helpful C++ feature that is most useful for writing long string literals on multiple lines. In particular, it avoids the pitfalls that line splicing, with backslashes at the end of a line, has involving whitespace inside string literals.

Unfortunately, the fact that the compiler (or preprocessor) performs this concatenation automatically without any warning can also lead to strange errors. Consider the following definition of an array of strings:

```
char *arr[] = { "a", "b" "c" }; // Missing comma!
```

The absence of the second comma causes "b" and "c" to be concatenated to produce "bc" and `arr` is defined to hold 2 strings instead of 3. Even if the array size were explicitly declared as 3 (i.e., `char*arr[3]`) many compilers would still not produce a warning, since having too few initializers is not an error.

Octal integer constants

Any integer constant beginning with 0 is treated as an octal constant. This creates no problem with 0 itself since its value is the same in both octal and decimal, but there are dangers in using prefix zeros on integer constants. Nevertheless, the temptation to use initial zeros in numbers can arise occasionally. For example, consider representing 4-digit phone extension numbers as integers:

```
struct { char *name; int ext_number; } arr[] = {
    { "Mary", 7234 },
    { "John", 3467 },
    { "Elaine", 0135 }   // Bug!
};
```

The phone number `0135` will be interpreted as an octal constant, and won't equal decimal `135`. It's value in octal is `1*64+3*8+5=93`.

Nested Comments Hide Statements

Nested /* comments are not allowed in C++, although they might trigger a warning. This creates an insidious problem if you accidentally leave off the closing */ for a comment. Note that there's no similar issue with the // style of commenting. Consider this CUDA C++ code:

```
 __global__ void matrix_add_safe_puzzle8(
    float *m3, const float *m1, const float *m2,
    int nx, int ny)
{
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   if (x < nx /*X* / && y < nx /*Y*/ ) {
       int id = x + y * nx; // Linearize
       m3[id] = m1[id] + m2[id];
   }
}
```

There's a nested comment problem that will comment-out the "`y < nx`" test, because there's an accidental space between "*" and "/" in the first comment. You'd probably get a compiler warning, and hopefully you pay attention to them!

8. Undefined C++ Features

What are Undefined Behaviors?

The C++ programming language is very portable despite its low-level focus on efficiency. However, there are numerous areas of the C++ language that are "undefined behaviors" and some of them are relatively common.

Technically, there are two types:

- "undefined behaviors" — compilers can do whatever they like, even differently each time, or
- "implementation-specific behaviors" — compilers can do whatever they like, but they have to be consistent in doing the same thing every time.

If you've never heard of this, well, actually you have. You're certainly familiar with some of the undefined behaviors, like accessing an uninitialized value, a null pointer dereference, or accessing a memory address that has already been de-allocated.

In other words, all the memory bugs are from undefined behaviors.

Some of the indicators that you've accidentally used an undefined behavior include:

- Portability problems where the code crashes on different platforms.
- Use higher optimization levels causes the code to crash.
- Code runs fine in the debugger, but fails without it.

These are also indicators of a memory safety failure, which underscores my point that the most common type of undefined behavior is related to memory errors.

Safety Issues for Compiler Vendors

We've examined various types of problems that can be addressed by memory debug libraries and other debug wrapper libraries. But we can't do everything that way.

Here are some of the issues that are hard to address without changing the code-generation of the compiler. In other words, they can't be detected or resolved by changes to the standard C++ library alone.

Some of the problematic areas include:

- Initialization of automatic stack variables without an explicit initializer (i.e., auto-initialization to zero).
- Checking pointer de-references have a valid address.
- Checking array accesses have a valid address.
- Race conditions and other concurrency problems.
- Integer overflow and underflow (signed or unsigned types).
- Floating-point overflow and underflow.
- Order-of-evaluation issues with binary operator operands.
- Order-of-evaluation issues with function argument expressions.
- Undefined integer operations on signed negative values (e.g., remainder, integer division, right bitshift).
- Throwing an exception inside a destructor.
- Calling `exit` or `abort` inside a destructor.

That's not the full list. There are literally hundreds of these obscure "undefined behaviors" in C++, although most of them are very rare.

Note that some of the most common memory safety issues are not on the above list. For example, it would be easy for a vendor to change their standard library so as to guarantee that `malloc` and `new` would zero the allocated memory.

These could be fixed in the vendor's standard library, rather than needing changes to the compiler engine.

C++ Operator Pitfalls

Most of the low-level arithmetic code in C++ algorithms looks quite standardized. Well, not so much. The general areas where C++ code that looks standard is actually non-portable includes trappy issues such as:

- Arithmetic overflow of integer or `float` operators.
- Integer `%` remainder and / division operators on negatives.
- Right bitshift operator >> on a negative signed integer is not division.
- Divide-by-zero doesn't always crash on all CPUs and GPUs.
- Order of evaluation of expression operands (e.g., with side-effects).
- Order of evaluation of function arguments.
- Functions that should be Boolean are not always (e.g., `isdigit`, `isalpha`)
- Functions that don't return well-defined results (e.g., `strcmp`, `memcmp`, etc.)
- Initialization order for `static` or global objects is undefined.
- `memcmp` is not an array equality test for non-basic types (e.g., structures).

Note that these errors are not only portability problems, but can arise in any C++ program. In particular, different levels of optimization in C++ compilers may cause different computations, leading to insidious bugs.

Signed right bitshift is not division

The shift operators << and >> are often used to replace multiplication by a power of 2 for a low-level optimization. However, it is dangerous to use >> on negative numbers. Right shift is not equivalent to division for negative values. Note that the problem does not arise for unsigned data types that are never negative, and for which shifting is always a division.

There are two separate issues involved in shifting signed types with negative values: firstly, that the compiler may choose two distinct methods of implementing >>, and secondly, that neither of these approaches is equivalent to division (although one approach is often equivalent). It is unspecified by the standard whether >> on negative values will:

(a) sign extend, or

(b) shift in zero bits.

Different compilers must choose one of these methods, document it, and use it for all applications of the >> operator. The use of shifting in zero bits is never equal to division for a negative number, since it shifts a zero bit into the sign bit, causing the result to be a nonnegative integer (dividing a negative number by two and getting a positive result is not division!). Shifting in zero bits is always used for unsigned types, which explains why right shifting on unsigned types is a division.

Divide and remainder on negative integers

Extreme care is needed when the integer division and remainder operators / and % are applied to negative values. Actually, no, forget that, because you should never use division or remainder and if you must, then you choose a power-of-two and use bitwise operations instead. Division is unsigned right bitshift, and remainder is bitwise-and.

Anyway, another reason to avoid these operators occurs with negatives. Problems arise if a program assumes, for example, that -7/2 equals -3 (rather than -4) . The direction of truncation of the / operator is undefined if either operand is negative.

Order of evaluation errors

Humans would assume that expressions are evaluated left-to-right. However, in C++ the order of the evaluation of operands for most binary operators is not specified and is undefined behavior. This makes it possible for compilers to apply very good optimizing algorithms to the code. Unfortunately, it also leads to some problems that the programmer must be aware of.

To see the effect of side effects, consider the increment operator in the expression below. It is a dangerous side effect.

```
y = (x++) + (x * 2);
```

Because the order of evaluation of the addition operator is not specified, there are two orders in which the expression could actually be executed. The programmer's intended order is left-to-right:

```
temp = x++;
y = (temp) + (x * 2);
```

The other incorrect order is right-to-left:

```
temp = x * 2;
y = (x++) + (temp);
```

In the first case, the increment occurs before `x*2` is evaluated. In the second, the increment occurs after `x*2` has been evaluated.

Obviously, the two interpretations give different results. This is a bug because it is undefined which order the compiler will choose.

Function-call side effects

If there are two function calls in the one expression, the order of the function calls can be important. For example, consider the code below:

```
x = f() + g();
```

Our first instinct is to assume a left-to-right evaluation of the "+" operator. If both functions produce output or both modify the same global variable, the result of the expression may depend on the order of evaluation of the "+" operator, which is undefined in C++.

Order of evaluation of assignment operator

Order of evaluation errors are a complicated problem. Most binary operators have unspecified order of evaluation — even the assignment operators. A simple assignment statement can be the cause of an error. This error can occur in assignment statements such as:

```
a[i] = i++;    // Bug
```

The problem here is that "`i`" has a side effect applied to it (i.e., ++), and is also used without a side effect.

Because the order of evaluation of the = operator is unspecified in C++, it is undefined whether the increment side effect occurs before or after the evaluation of `i` in the array index.

Function-call arguments

Another form of the order of evaluation problem occurs because the order of the evaluation of arguments to a function call is not specified in C++. It is not necessarily left-to-right, as the programmer expects it to be.

For example, consider the function call:

```
fn(a++, a);  // Bug
```

Which argument is evaluated first? Is the second argument the new or old value of *a*? It's actually undefined in C++.

Order of initialization of static objects

A special order of evaluation error exists because the order of initialization of static or global objects is not defined across files.

Within a single file the ordering is the same as the textual appearance of the definitions. For example, the Chicken object is always initialized before the Egg object in the following code:

```
Chicken chicken; // Chicken comes first
Egg egg;
```

However, as for any declarations there is no specified left-to-right ordering for initialization of objects within a single declaration.

Therefore, it is undefined which of `c1` or `c2` is initialized first in the code below:

```
Chicken c1, c2;
```

If the declarations of the global objects "chicken" and "egg" appear in different files that are linked together using independent compilation, it is undefined which will be constructed first.

Standard Library Problems

Not everything is well-defined in the standard C++ library. There are numerous pitfalls that can jump up and bite you in apparently innocuous uses of the library functions. Here's a selection of them below.

memcmp cannot test array equality

For equality tests on many types of arrays, the `memcmp` function might seem an efficient way to test if two arrays are exactly equal. However, it only works in a few simple situations (e.g., arrays of `int`).

This idea is buggy in several cases:

- Floating-point has two zeros (positive and negative zero), so it fails.
- Floating-point also has multiple numbers representing `NaN` (not-a-number).
- If there's any padding in the array, such as arrays of objects or structures.
- Bit-field data members may have undefined padding.

You can't skip a proper comparison by looking at the bytes.

EOF is not a char

The `EOF` constant is a special value for C++ file operations. One problem related to signed versus unsigned chars is comparing a `char` type variable with `EOF`. Because `EOF` is represented as integer value `-1`, it should never be directly compared with a `char` type. Although it will usually work if characters happen to be signed for a particular implementation, if any characters are unsigned, the comparison of a default `char` type with `EOF` is not correct since `-1` is promoted to `unsigned int`, yielding a huge value not representable by a `char`.

An example of this type of bug:

```
char ch = getchar();
if (ch == EOF) { ... }   // Bug
```

The correct definition is to use "`int ch`" as the declaration.

fflush on an input file

The `fflush` function is used to flush the buffer associated with a file pointer. Unfortunately, it can only be used to flush an output buffer, causing output to appear on screen (or be flushed to a file). Applying `fflush` on an input file leads to undefined results; it will succeed on some systems, but cause failure on others. The problem is typified by the following statement that often appears in code:

```
fflush(stdin);
```

The intention is to flush all input characters currently awaiting processing (i.e., stored in the buffer), so that the next call to `getchar` (or another input function) will only read characters entered by the user after the `fflush` call. This functionality would be very useful, but is unfortunately not possible in general, as the effect of `fflush` is undefined on input streams. There is no portable way to flush any "type ahead" input keystrokes; `fflush(stdin)` may work on some systems, but on others it is necessary to call some non-standard library functions.

fread and fwrite without intervening fseek

When a binary file is opened for update using a mode such as "`rb+`", the programmer must be careful when using `fread` and `fwrite`. It is an error to mix `fread` and `fwrite` operations without an intervening call to a repositioning function such as `fseek` or `rewind`.

For example, do not assume, after sequentially reading all records in a file using `fread`, that a call to fwrite will write data to the end of the file. Instead, use an `fseek` call to explicitly reach the end of the file before writing. The best method of avoiding this file access error is to call the `fseek` function immediately before every `fread` or `fwrite` call.

Modification of string literals

String literals should not be modified in C++ because they could potentially be stored in read-only memory. They should be thought of as having type `const char*`. Therefore, using `char*` string types without caution can lead to errors, such as applying `strcpy` to a pointer that currently points to a string literal, as below:

```
char *result = "yes";
if (...)
    strcpy(result, "no"); // WRONG
```

The effect of this code is to try to modify the memory containing the string literal "yes". If this is stored in read-only memory the `strcpy` function has no effect or possibly a run-time failure.

Even if string literals happen to be modifiable for the particular implementation this form of modification can lead to strange errors. Overwriting "yes" with "no" means that the initialization of result will never again set result to "yes".

The code can be thought of as equivalent to the following code:

```
char yes_addr[4] = { 'y', 'e', 's', '\0' };
char *result = yes_addr;
if (...)
    strcpy(result, "no"); // WRONG
```

Hence, the `strcpy` call changes `yes_addr` and the initialization will always set "`result`" to whatever `yes_addr` currently contains.

Worse still is the problem that many compilers merge identical string literals so as to save space. Hence, the above `strcpy` call will change *all* uses of the constant "yes" to be "no" throughout the program! Therefore, one change to a string constant will affect all other instances of the same string constant — a very severe form of aliasing.

Avoiding the modification of string literals is not all that difficult, requiring only a better understanding of strings. One solution to the above problem is to use an array of characters instead of a pointer:

```
char result[] = "yes";
if (...)
    strcpy(result, "no"); // RIGHT
```

In this case the compiler allocates 4 bytes for result, rather than making it point at the 4 bytes for the string literal (which was the same address that all uses were given).

Backslash in DOS filenames

Windows and DOS use backslashes for directory paths in filenames, whereas Linux uses a forward slash character. A common error with file operations occurs when a DOS filename is encoded with its full path name. The backslash starts an escape inside the string constant. Hence, the filename below is wrong:

```
fp = fopen("c:\file.cpp", "r");      // Bug
```

The backslash character starts the escape `\f`, which is a formfeed escape. The correct statement uses two backslash characters:

```
fp = fopen("c:\\file.cpp", "r");     // Correct
```

Summary

The above is just a selection of some of the undefined and non-portable aspects of C++. The solution to a fully safe C++ programming language rests not only in addressing memory safety, but in fixing a lot of these areas. The goal would be to change all the "undefined behaviors" into "well-defined" parts of the safe C++ standard. Unfortunately, there are many issues to choose from!

9. Error Checking

Error Checking

Everyone's always known that it's good programming style to check all error return codes. It's extra work, but everyone does it anyway, because it's so important. I've been coding for years, and I've never seen a `printf` or `fopen` without an `if` statement immediately after it.

Yeah, right! Many people don't even know that `printf` returns a value, and as for `fopen`, this is common:

```
fp = fopen(fname, "r");
assert(fp != NULL);
```

Oh, well, technically that is an error return check! A better way to code it is:

```
fp = fopen(fname, "r");
if (fp == NULL) {
    // Complain...
}
```

This is a very common approach in C++ programming, but when you think about it, there's a few things wrong with it:

- Relies on the programmer to add this code.
- Copy-paste errors in having error handling sequences everywhere.
- No way to enforce consistency in error handling.
- Enforcing the requirement for return checking is difficult.

This chapter examines some solutions to these problems.

Types of Error Checking

Everyone knows the basic idea of checking error returns from system functions. Let's have a look at the basic ideas:

- Checking standard library error returns
- Checking your own functions

But there are some other related issues:

- Validating input parameter values (in your own code).
- Validating standard C++ system parameter values (in a debug wrapper).
- Checking `errno` values (these are not "returned" but set).

And there's the issue of how to integrate your basic error return checking in a way that's consistent with:

- Exception handing (i.e., `try...catch`).
- Assertion failure handling
- Self-testing code
- Unit tests and regression tests
- Signal handling

For example, would you want an error detected in a function's error return to `throw` an exception?

Function Return Attribute: nodiscard

The `[[nodiscard]]` attribute can be used in the return type of function declarations. It encourages the compiler to issue a warning (not an error) if the function is then used in a way that discards its return check. There are two ways to use it: the basic way, or a way with an optional message parameter.

```
[[nodiscard]] int my_important_function();
[[nodiscard("Returns important status")]]
int my_important_function();
```

Note that the optional parameter can only be a string literal. It cannot be a computed expression — not even a compile-time constant expression.

Obviously, this is helpful for enforcing a policy that certain functions should always get error-checked. Also, some of the standard library functions will have this setting, but it's implementation-specific which ones will.

Suppressing with void casts. You can suppress a warning about discarding a return value, simply by adding a type cast to `void`. Examples include:

```
(void)my_important_function();  // Not that important
void(my_important_function());  // Also not
```

Compatibility before C++17. Note that `nodiscard` was defined in C++17, but there have been some earlier attributes used in some C++ compilers, such as the `_Noreturn` attribute. You probably shouldn't use these old attributes in newly-written code, but you might see them during code maintenance.

There is also a generalized version, added in C++20, that allows an optional string literal containing a message or an explanation. This is obviously useful, but I feel like it could also get a little abused. Here's an example:

```
[[nodiscard("You idiot!")]]
int my_important_function();
```

Detecting void casts. I feel like there should be a way to detect where the code uses a type cast to `void` so as to override these settings. Otherwise, programmers can simply work around the `nodiscard` settings without getting publicly shamed on the internal Slack channel.

Unfortunately, I'm not aware of a compiler setting for this. Maybe some of the static analyzer tools have this capability, but you could hand-code a simple solution with `grep`:

```
grep '[()[ ]*void[ ]*[)]' *.cpp
grep 'void[ ]*[()[ ]*[a-zA-Z_].*[ ]*[)]' *.cpp
```

The first one is very specific, but the second one might need some refining to avoid false positives.

Recursive Macro Error Checks

C++ allows macros to be recursive in the sense that they can use their own name. It's not actually "recursive" and is actually limited to a once-only expansion, rather than an infinitely recursive expansion. This feature is a longstanding feature of C and C++ languages since they were created, so you can rely upon it.

For example, these would be harmless:

```
#define memset(a,b,c)   memset(a,b,c)
#define memcpy(a,b,c,d) memcpy(a,b,c,d)
```

The idea is to automatically add the error check macros:

```
#define memset(a,b,c) \
    AUSSIE_ERRORCHECK(memset(a,b,c))
#define memcpy(a,b,c) \
    AUSSIE_ERRORCHECK(memcpy(a,b,c))
```

But that doesn't quite work, when used with this type of call:

```
errval = memcpy(....);
```

The `do...while(0)` trick expands out to give a compilation syntax error:

```
errval = do { ... // etc.
```

Similarly, the version with a combined macro and `inline` function also gets a different type of compilation error:

```
errval = aussie_check_function(....)
```

The problem is that the return type of the `inline` function is `void`. Hence, we'd need to go back and fix any code that uses the return value of `memcpy` or `memset`, which would be a good job for a coding copilot, if only I didn't have so many trust issues.

Instead, we can just fix the return type to be `void*` and use a pass-through of the return value:

```
#define AUSSIE_ERRORCHECK3(codeexpression) \
    aussie_check_function2((codeexpression), \
       __func__, __FILE__, __LINE__)

inline void * aussie_check_function2(
        void *expression,
        const char *func, const char *fname, int lnum)
{
   if (expression == NULL) {
      fprintf(stderr,
        "ERROR: Function returned NULL: in %s at %s:%d\n",
        func /*__func__*/,
        fname /*__FILE__*/,
        lnum /*__LINE__*/ );
   }
   return expression;  // pass through!
}
```

And we really should add a ridiculous number of round brackets around the macro parameters, and also use `#undef` for total macro safety:

```
#undef memset  // safety
#undef memcpy
#define memset(a,b,c) \
    (AUSSIE_ERRORCHECK3(memset((a),(b),(c))))
#define memcpy(a,b,c,d) \
    (AUSSIE_ERRORCHECK3(memcpy((a),(b),(c),(d))))
```

Voila! Now we have a set of macros that automatically adds return code error checking around all calls to `memcpy` and `memset`. And it should work irrespective of whether their returned values are used or not in the calls.

To use them properly, we just need to `#include` a header file near the top of every C++ source file. But it has to be after any system header files because those system header files have prototype declarations of functions like `memcpy` that our tricky macros will break.

Now we only have to add similar recursive macros for all 1,657 of the Standard C++ API functions. No, relax, I'm just kidding. There's only a few that matter.

Macro Intercepted Debug Wrapper Functions

Is there any way you can level up? We've already auto-added the error checking macros around all the standard library function calls. Can we do better? Of course, we can!

One extension is to build debug wrapper function versions for the main API calls. These functions can then perform more extensive error self-checking than is performed within the standard library.

```
#undef memcpy
void* aussie_memcpy_wrapper(
        void *destp, const void *srcp, size_t sz)
{
    void *ret =
        AUSSIE_ERRORCHECK3(memcpy(destp,srcp,sz));
    return ret;
}
#define memcpy(a,b,c) \
    aussie_memcpy_wrapper(a,b,c)  // Intercept!
```

Note that the `#undef` is really important here, and must be before the wrapper function body. If we're not careful, our wrapper function can wrap itself, and become infinitely recursive.

The above example doesn't do any extra error checking, other than what we've already put into the error checking macro (i.e., `AUSSIE_ERRORCHECK3`). However, we could add extra self-checking code for common errors that arise from `memcpy` copy-pasting:

- Destination or source pointers are null
- Destination or source pointers are the wrong address scopes
- Destination pointer equals source pointer

The standard library may already find some of those errors, and `valgrind` or other sanitizers would find even more. However, we could go further with our analysis. For example, some more extensive error checks possible could be:

- `memcpy` size argument is zero or negative (after conversion to `size_t`).
- `memcpy` arguments appear to be in reverse order.

The possible error checks from this type of system function interception are discussed further in the full chapter on debug wrapper functions.

Reporting and Handling Errors

What should an error checking macro do on failures? Some options include:

- Print an error message
- Print the error code number, such as errno and its name with `strerror`
- Give source code context information
- Exit the program (or not?)

That's not the full list, and advanced ideas for production-error handling include:

- Throw an exception and hope someone's listening.
- Full stack trace (e.g., `std::backtrace` in C++23).
- Report a full error context for supportability in the wild.
- Log information to a file, not just to `stderr`.
- Abort the program to generate a "`core`" file.

Reporting Error Context

A key aspect of reporting the error context is the C++ statements that triggered the issue. The basics of error context are these macros:

- `__func__`
- `__FILE__`
- `__LINE__`

I don't know why one is lower case and two are upper case, but it's called international standardization. That's what makes C++ programming so fun.

However, I have to say that I think these source code context macros are on their way out. Once reporting the full stack trace in C++23 with `std::backtrace` is widespread, why would we need those macros? Also gone would be lots more preprocessor macro tricks that only exist in order to report the source code context. Instead, use an `inline` function and `std::backtrace`. More advanced error context that can help with supportability includes things like:

- Date and time of error.

- User query that triggered the failure.
- Random number seed (for reproducibility of AI errors).
- Full stack trace (if available)

Limitations of Macro Error Checking

Some problem areas include:

- Cannot intercept everything (e.g., can't intercept arithmetic operators to check for overflow or divide-by-zero).
- Macro interception is not perfect, with some valid syntax causing compile errors.

Two of these methods rely on preprocessor macro interception to auto-wrap the calls with debug checks. Unfortunately, macro interception isn't a perfect solution, and some of the problems that macros may have include:

- Interception of `new` and `delete` operators is only possible at link-time.
- Namespace-scoped calls fail: e.g., `std::memcpy(...)` or `std::memset(...)`
- Use of these standard function names as function pointers won't work.
- Non-standard calling syntax: e.g., parentheses around the function name.

Much better than macro interception would be a way to link to a debug version of the standard C++ library. Many more complex error checks are possible than are performed, and this would significantly improve the timeframe to detect many types of coding errors.

But I have to finish by saying that the really major limitation is this:

Remembering to add it every time!

I've given a few suggestions for auto-fixing that issue above, but they're far from perfect. Maybe the Standard C++ library needs a callback mechanism, or some other method whereby programmers can ensure that they never miss an error return.

10. Safe Builds

Build Management

Proper build management can be an important part of C++ safety initiatives. The aspects of builds related to C++ safety in the development environment are many.

Some examples include:

- Warnings analysis from compiler output.
- Automated unit tests and regression testing.
- Integrated testing with the nightly builds.
- CI/CD approval processes (e.g., run unit tests).

In regards to external management of builds and releases, there are also opportunities to improve overall quality:

- Tracking builds and releases (the basics)
- Keeping executables for all builds
- Matching debug versions of executables (for postmortem debugging purposes).
- Maintaining hash signatures for executable security.

Some of the pitfalls in build management include:

- Inadvertent disclosure of security credentials used in testing.
- Security tracking to ensure hackers cannot add viruses to your builds.

Build engineering is not just about building!

Leveraging More Builds

Instead of thinking about how to get the product built, let's think about ways to leverage builds for extra quality. The basic method is simply "nightly builds" whereby:

- Unit tests automatically run.
- Full regression test suite automatically run.
- Failures are detected.
- Notification via email to the developer team about any failures.

This is a very efficient system. Once it's setup, there's very little to maintain. But we can level it up:

- Run unit tests with `valgrind` and/or other sanitizers.
- Run the full regression test suite with `valgrind` and/or other memory checkers and sanitizers (if it takes more than a day, don't run it every night).
- Automate analysis of compiler warnings (e.g., remove unimportant ones).
- Add multiple runs of unit tests under different build conditions (e.g., with and without debug code enabled, with different optimizer levels, with different compilers).
- Add linters and static analysis tool pathways.

The incremental cost of setting up more builds is relatively low. Hence, if you really want to finesse things:

- Build and run tests on different hardware platforms (e.g., with local hardware or via remote virtual machines).
- Run multiple sanitizers, and/or use your own home-grown memory debug library (e.g., as in this book).
- Use multiple pathways for compiler warnings (e.g., the basic build and one with many optional compiler warnings enabled).
- Use multiple linter pathways (i.e., one for bug-focused warnings, and one with more pedantic settings for style issues).

One final point about all these builds: don't just email the output. A huge ream of informational messages and compiler warnings causes immediate overload.

Instead, someone needs to take the time to `grep` out the unimportant messages. Otherwise, anything major detected by unit tests or compiler/linter warnings gets lost in the snow.

Maybe you shouldn't take your build engineer for granted. They're probably less likely to be replaced by AI than you!

Warning-Free Build

Don't ignore compiler warnings! A very good goal for C++ software quality is to get to a warning-free compile. You should think of compiler warnings as doing "static analysis" of your code. To maximize this idea, turn on more warning options, since the warnings are rarely wrong in modern compilers, although some are about harmless things.

Harmless doesn't mean unimportant. And anyway, the so-called "harmless" warnings aren't actually harmless, because if there's too many of them in the compilation output, then the bad bugs won't get seen. Hence, make the effort to fix the minor issues in C++ code that's causing warnings. For example, fix the "unused variable" warnings or "mixing `float` and `double`" type warnings, even though they're rarely a real bug. And yet, sometimes they are! This is why it's powerful to have a warning-free compile.

Tracking compilation warnings. One way to take warning-free compilation to the next level is to actually store and analyze the compiler output. It's like log file analysis in DevOps, only it's not for systems management, but for debugging. On Linux, I typically use this idea:

```
make build |& tee makebuild.txt
```

Here's an actual example from a `Makefile` in an Aussie AI project on Linux:

```
build:
    -@make build2 |& tee makebuild.txt
    -@echo 'See output in makebuild.txt'
```

The `Makefile` uses prefix "-" and "`@`" flags, which means that it doesn't echo the command to output, and doesn't stop if one of the steps triggers an error.

When the build has finished, then we have a text file "`makebuild.txt`" which can be viewed for warning messages. To go further, I usually use `grep` to remove some of the common informational messages, to leave only warning messages.

Typically, my Linux command looks like:

```
make warnings
```

Here's an example of the "warnings" target in a Makefile for one of my Aussie AI projects:

```
warnings:
    -@cat makebuild.txt | grep -v '^r -' \
    | grep -v '^g++ ' | grep -v '^Compiling' \
    | grep -v '^Making' | grep -v '^ar ' \
    | grep -v '^make\[' | grep -v '^ranlib' \
    | grep -v '^INFO:' | grep -v 'Regressions failed: 0' \
    | grep -v 'Assertions failed: 0' | grep -v SUCCESS \
    |more
```

Note that this uses the grep command to also remove the various informational messages from g++, ar, ranlib, and make. And it also removes the unit testing success messages if all tests pass (but not if they fail!). The idea is to show only the bad stuff because log outputs with too many lines get boring far too quickly and then nobody's watching.

One annoying thing about using grep with make is that you get these kind of error messages:

```
make: [annoying] Error 1 (ignored)
```

Here's a way to fix them in a Makefile on Linux:

```
-@grep tmpnam *.cu *.cpp || true
```

The "true" command is a shell command that never fails. Note that this line uses the double-pipe "||" shell logical-or operator, so it only runs "true" if grep fails. But don't accidentally use a single "|" pipe operator, which would actually be a silent bug! This idea makes the command line calling grep return a non-zero status, and then make is silent.

Finally, your warning-free tracking method should ideally be part of your "nightly builds" that do more extensive analysis than the basic CI/CD acceptance testing. You should email those warnings to the whole team, at about 2am ideally, because C++ programmers don't deserve any sleep.

Advanced Build Issues

There are various other aspects of build management that can improve overall quality. These include:

- Security issues
- CI/CD/CT integration issues
- Documentation generation issues
- Release management

Build security. Security issues with builds are both internal and external. There are two main issues:

- Accidental release of internal security credentials.
- Protection against security issues in third-party licenses.
- Avoiding malicious contamination of your releases (don't be part of a "supply chain attack").

It's common for internal security credentials to get added into the source code control system. This is mostly problematic if your build is releasing an open source package, whereas if it's building software executables, these credentials probably won't be in the release.

However, if credentials are hard-coded into the source code for testing purposes, these will still be disclosed publicly as part of an executable. Don't underestimate the power of hackers to disassemble binaries, or the simple capabilities from the `strings` Linux tool.

External security issues arise in terms of the third-party libraries that you are using in your application (i.e., dependency management). Alternatively, you can be a direct victim of a hacking attempt, which may damage your business.

Even more insidious are the cases where hackers have embedded payloads into software that is distributed to other customers, which are known as "supply chain attacks." You don't want to be the source of a virus distributed to all your customers!

Release management. Supportability can be greatly improved by good build management. The release process needs to carefully manage which executables go out in which build release.

Some of the issues include:

- Mapping customer releases to internal build numbers and versions.
- Tracking which versions of third-party licenses were used in which builds.
- Storing a permanent copy of any executable that went out.
- Keeping a correlated "debug" copy of the executable (for use in post-mortem debugging any customer core dumps).
- Tagging the source code to mark the release numbers and builds.

The build management aspects of software are less heralded than the exciting algorithms in the latest AI engines. But a good, solid foundation in your build management is critical for high-quality software.

11. Linters and Static Analysis

Linters for C++

Linters, or "static analyzers," are tools that examine your source code for errors or stylistic concerns. The main advantage of these tools is that they improve safety "for free" without any runtime impact for your customers. Judicious use of static checkers on your C++ source code can detect a variety of errors before they impact customers. The main advantages of linters include:

- Detect coding errors before the code is even run.
- Both security vulnerabilities and bugs can be flagged.
- General improvement in coding quality.
- Reduced debugging time because the number of live bugs is reduced.
- Can be used for stylistic issues or coding policy guidelines.

There are also some linters that focus on "reformatting" and "beautification" for source code. Similarly, there are source code analysis tools that aim to auto-generate internal code documentation. I'm not really talking about those ones in this section. I'm hunting bugs!

Linters are not for everyone, and are less popular with developers than runtime memory checkers. Disadvantages of linters include:

- Additional cost and time to implement and address issues in an ongoing way.
- Fixing harmless warnings (e.g., need to fix warnings that aren't real bugs).
- Stylistic warnings are never popular with developers.
- Configuration of some linters is onerous (e.g., they need all the include paths setup).

You should consider linters as an additional safety technique, which is orthogonal to runtime techniques such as runtime memory checking tools. Linting is an add-on technique for additional improvements to overall quality.

General advice in regard to using linters for C++ programming is:

- Use compiler warnings as free linting.
- Turn off less serious stylistic warnings when introducing linting.
- Use a separate linter build sequence.
- Have two linter paths (i.e., one for bugs, one for style).
- Use multiple compilers and linters for extra coverage.
- Automate linting into the nightly build.

In the past, linters have gained a somewhat poor reputation because of two factors:

- Not finding many bugs, and
- Emitting a huge swathe of warnings for minor, stylistic nitpicks.

These concerns are largely no longer true of both open source and commercial linting tools. Linting tools can now detect a huge range of real bugs in your code, and many can be focused to only emit serious bugs or security vulnerabilities. You can, of course, turn on all of the stylistic warnings if you want to use a linter for enforcing a company-wide C++ coding policy, in which case, they won't be popular with the team.

Using GCC as a Linter

If you want more warnings, and who doesn't, you can enable more warnings in `gcc` on Linux. You can either do this in your main build by enabling more compiler warnings, or use a separate build path (e.g., choose an inspiring name like: "`make lint`") so that the main build is not inundated with new warnings.

There are some `gcc` flags that are specific to static analysis of source code:

- `-fanalyzer` — enables the static analyzer.
- `-Wanalyzer-SUBAREA` — controls the static analyzer's warnings.

Some useful `gcc` warning flags include:

- `-Wall` — "all" warnings (well, actually, some).
- `-Wextra` — the "extra" warnings not enabled by "`-Wall`".
- `-Wpedantic` — yet more of the fun ones.

You know, I really cannot say that I am a fan of endlessly scrolling warnings from the "pedantic" mode. Maybe, turn that one off, and pick-and-choose from the list of flags in the "pedantic" list. For example, I have used "`-Wpointer-arith`" in projects.

Fixing Linter Warnings

Here's some advice about fixing the code to address linter concerns:

- Aim for a warning-free compilation of bug-level messages.
- Don't overdo code changes to fix any stylistic complaints.

Fix the bugs found by warnings (obviously), but as far as the stylistic type warnings are concerned, be picky. I say, aim for code quality and resilience, not code aesthetic perfection.

Warning-free linting. As with the main build, if you're not fixing the less severe linter warnings, turn them off, or have two separate build sequences for the main anti-bug linting versus stylistic linting. You want any newly found serious problems to be visible, not lost in a stream of a hundred other spurious warnings. Hence, high quality code requires achieving a warning-free linting status for the main warnings.

On the other hand, you don't want programmers doing too much "busy work" fixing minor coding style warnings with little practical impact on code reliability. Hence, you might find that your policy of "warning-free linting" needs to suppress some of the pickier warnings. And that'll be a fun meeting to have.

Linter Products

There are many existing linter tools that are available in open source or commercially. Some examples include:

- Sonar Lint
- `cppcheck`
- `cpplint`
- `oclint`
- `clang-tidy`

Existing compilers and IDEs also include linters and static analysis tools:

- Microsoft Visual Studio's "static analyzer"
- Clang static analyzer
- GCC static analyzer

There are many more. Wikipedia has an extensive list of them on its "List of tools for static code analysis" page.

Note that we have an active project for a C++ linter. Find more information about Aussie Lint at https://www.aussieai.com/safe/projects.

Linter Capabilities

There are many linters available, and a whole range of features. There are various different types of linting capabilities that you might consider in a project:

- Bug detection
- Security vulnerability detection
- Coding policy adherence

Note that the state-of-the-art has progressed rapidly in the area of static analysis. These tools can identify a variety of pitfalls in C++ programming, including:

- Lexical oddities (e.g., nested comments).
- Preprocessor errors (e.g., macro operator precedence errors).
- Expression errors (e.g., wrong logical operators).
- Control flow errors (e.g., unreachable code, never-failing conditions, etc.)
- C++ class errors (e.g., consistent types of any constructor and destructor declarations).
- Function call graph errors (e.g., indirect recursion)

Some examples of specific and simple warnings in coding style may include:

- Deprecated functions (e.g., `gets` versus `fgets`).
- Inefficient older functions (e.g., `rand` and `srand`).
- Security-vulnerable functions (e.g., `tmpnam`).
- Buffer-overflow prone functions (e.g., `sprintf`).
- Unsafe functions to change to "safe" versions (e.g., `strcpy` vs `strncpy`).

Linter Research

Linters have an advanced base of theory these days. It's similar to compiler design theory, but with a different focus, since linters do not need the "code generation" phase of compilation.

Some of the main techniques that linters use include:

- Expression trees
- Control flow graphs
- Function call graphs

Expression trees expression operator precedence and parenthesized sub-expressions into a hierarchical tree. This is not a graph, since there are no cross-edges between subtrees.

Control flow graphs express the flow of control through a function or a code block. These primarily focus on `if` statements, loops, and `switch` statements. Aspects of short-circuited operators and the ternary operator may sometimes be involved, or these may be handled in the expression trees. Note that control flow graphs may contain cycles due to loops.

Function call graphs express the hierarchy of function calls. Never-returning functions such as `exit` or `abort` need to be handled specially. This analysis is primarily based on the static calls to function names, and will have difficulty if `virtual` functions or function pointers are used for dynamic function calls. Nevertheless, the call graph can be useful to detect various errors.

Note that the call graph may contain cycles in the event that recursion, directly or indirectly, is used in any functions.

Variable analysis. Particularly interesting is that static analyzers now use "flow propagation" to track errors throughout execution pathways. The idea is similar to a compiler's "constant propagation" but can relate to categories of values for a variable, rather than just a single constant value.

Aspects of the value of a variable can be propagated through expression trees and also along the edges of the control-flow graph. In advanced cases, it may also be propagated through the function call graph.

For example, pointers can be tracked as null versus non-null, as a simple binary condition. Integral variables can have sets of possible values propagated through control flow statements, so that always-succeed or always-fail tests on these variables can be detected.

The level of error detection from these approaches is quite amazing. If you've tried static analysis tools for C++ in the past, and been underwhelmed, you really should give them another try!

12. Self-Testing Code

What is Self-Testing Code?

Instead of doing work yourself, get a machine to do it. Who would have ever thought of that?

Getting your code to test itself means you can go get a cup of coffee and still be billing hours. The basic techniques include:

- Unit tests
- Regression tests
- Error checking
- Assertions
- Self-testing code blocks
- Debug wrapper functions

The simplest of these is unit tests, which aim to build quality brick-by-brick from the bottom of the code hierarchy. The largest techniques are to run full regression tests suites, or to add huge self-testing code blocks.

Self-Testing Code Block

Sometimes an assertion, unit test, or debug tracing printout is too small to check everything. Then you have to write a bigger chunk of self-testing code. The traditional way to do this in code is to wrap it in a preprocessor macro:

```
#if DEBUG
   ... // block of test code
#endif
```

Another reason to use a different type of self-testing code than assertions is that you've probably decided to leave the simpler assertions in production code. A simple test like this is probably fine for production:

```
assert(ptr != NULL);  // Fast
```

But a bigger amount of arithmetic may be something that's not for production:

```
assert(aussie_vector_sum(v, n) == 0.0);  // Slow
```

So, you probably need macros and preprocessor settings for both production and debug-only assertions and self-testing code blocks. The simple way looks like this:

```
#if DEBUG
assert(aussie_vector_sum(v, n) == 0.0);
#endif
```

Or you could have your own debug-only version of assertions that are skipped for production mode:

```
assert_debug(aussie_vector_sum(v, n) == 0.0);
```

The definition of "`assert_debug`" then looks like this in the header file:

```
#if DEBUG
#define assert_debug(cond) assert(cond) // Debug mode
#else
#define assert_debug(cond) // nothing in production
#endif
```

This makes the "`assert_debug`" macro a normal assertion in debug mode, but the whole coded expression disappears to nothing in production build mode. The above example assumes a separate set of build flags for a production build.

Self-test Code Block Macro

An alternative formulation of a macro for installing self-testing code using a block-style, rather than a function-like macro, is as follows:

```
SELFTEST {
    // block of debug or self-test statements
}
```

The definition of the `SELFTEST` macro looks like:

```
#if DEBUG
#define SELFTEST // nothing (enables!)
#else
#define SELFTEST if(1) {} else // disabled
```

```
#endif
```

This method relies on the C++ optimizer to fix the non-debug version, by noticing that "`if(1)`" invalidates the `else` clause, so as to remove the block of unreachable self-testing code that's not ever executed.

Note also that `SELFTEST` is not function-like, so we don't have the "forgotten semicolon" risk when removing `SELFTEST` as "nothing". In fact, the nothing version is actually when `SELFTEST` code is *enabled*, which is the opposite situation of that earlier problem. Furthermore, we cannot use the "`do-while(0)`" trick in this different syntax formulation.

Self-Test Block Macro with Debug Flags

The compile-time on/off decision about self-testing code is not the most flexible method. The block version of `SELFTEST` can also have levels or debug flag areas. One natural extension is to implement a "flags" idiom for the debug areas, to allow configuration of what areas of self-testing code are executed for a particular run (e.g., a decoding algorithm flag, a normalization flag, a MatMul flag, etc.). One Boolean flag is set for each debugging area, which controls whether or not the self-testing code in that module is enabled or not.

A macro definition of `SELFTEST(flagarea)` can be hooked into the run-time configuration library for debugging output. In this way, it has both a compile-out setting (`DEBUG==0`) and dynamic runtime "areas" for self-testing code. Here's the definition of the self-testing code areas:

```
enum self_test_areas {
    SELFTEST_NORMALIZATION,
    SELFTEST_MATMUL,
    SELFTEST_SOFTMAX,
    // ... more
};
```

A use of the `SELFTEST` method with areas looks like:

```
SELFTEST(SELFTEST_NORMALIZATION) {
    // ... self-test code
}
```

The `SELFTEST` macro definition with area flags looks like:

```
extern bool g_aussie_debug_enabled; // Global override
```

```
extern bool DEBUG_FLAGS[100];     // Area flags


#if DEBUG
    #define SELFTEST(flagarea) \
        if(g_aussie_debug_enabled == 0 || \
            DEBUG_FLAGS[flagarea] == 0) \
        { /* do nothing */ } else
#else
#define SELFTEST if(1) {} else // disabled completely
#endif
```

This uses a "debug flags" array idea as for the debugging output commands, rather than a single "level" of debugging. Naturally, a better implementation would allow separation of the areas for debug trace output and self-testing code, with two different sets of levels/flags, but this is left as an extension for the reader.

Debug Stacktrace

There are various situations where it can be useful to have a programmatic method for reporting the "stack trace" or "backtrace" of the function call stack in C++. Some examples include:

- Your assertion macro can report the full stack trace on failure.
- Self-testing code similarly can report the location.
- Debug wrapper functions too.
- Writing your own memory allocation tracker library.

C++ is about to have standard stack trace capabilities with its standardization in C++23. This is available via the "`std::stacktrace`" facility, such as printing the current stack via:

```
std::cout << "Stacktrace: "
          << std::stacktrace::current()
          << std::endl;
```

The C++23 `stacktrace` library is already supported by GCC and early support in MSVS is available via a compiler flag "`/std:c++latest`". There are also two different longstanding implementations of stack trace capabilities: glibc `backtrace` and Boost `StackTrace`. The C++23 standardized version is based on Boost's version.

13. Assertions

Why Use Assertions?

Of all the self-testing code techniques, my favorite one is definitely assertions. They're just so easy to add! The use of assertions in C++ programs can be a very valuable part of improving the quality of your work over the long term. They ensure that you find bugs early in the life cycle of code, and they don't have much impact on performance (if used correctly). I find them especially useful in getting rid of obvious glitches when I'm writing new code, but then I usually leave them in there.

The standard C++ library has had an "`assert`" macro since back when it was called C. The simplest idea is therefore to use the builtin `assert` macro. The `assert` macro is a convenient method of performing simple tests. The basic usage is illustrated to validate the inputs of a simple vector function:

```
#include <assert.h>
...
float vector_sum(float v[], int n)
{
    assert(v != NULL);  // Easy!
    // ... etc
}
```

Compile-Time Assertions: static_assert

Runtime assertions have been a staple of C++ code reliability since the beginning of time. However, there's often been a disagreement over whether or not to leave the assertions in production code, because they inherently slow things down.

The modern answer to this conundrum is the C++ "`static_assert`" directive. This is like a runtime assertion, but it is fully evaluated at compile-time, so it's super-fast. Failure of the assertion triggers a compile-time error, preventing execution, and the code completely disappears at run-time.

Unfortunately, there really aren't that many things you can assert at compile-time. Most computations are dynamic and stored in variables at runtime.

However, the `static_assert` statement can be useful for things like blocking inappropriate use of template instantiation code, or for portability checking such as:

```
static_assert(sizeof(float)==4, "float not 32 bits");
```

This statement is an elegant and language-standardized method to prevent compilation on a platform where a "`float`" data type is 64-bits, alerting you to a portability problem.

Custom Assertion Macros

An important point about the default "`assert`" macro is that its failure handling may not be what you want. The default C++ `assert` macro will literally crash your program by calling the standard "`abort`" function, which triggers a fatal exception on Windows or a core dump on Linux.

That is fine for debugging, but it isn't usually what you want for production code. Hence, most professional C++ programmers declare their own custom assertion macros instead.

For example, here's my own "`aussie_assert`" macro in my own header file:

```
#define aussie_assert(cond) \
      ( (cond) || \
     aussie_assert_fail(#cond, __FILE__, __LINE__))
```

This tricky macro uses the short-circuiting of the "`||`" operator, which has a meaning like "`or-else`". So, think of it this way: the condition is true, *or else* we call the failure function. The effect is similar to an `if-else` statement, but an expression is cleaner in a macro.

The `__FILE__` and `__LINE__` preprocessor macros expand to the current filename and line number. The filename is a string constant, whereas the line number is an integer constant. The expression "`#cond`" is the "stringize" operator, which works in preprocessor macros, creating a string of its argument.

Note that you can add "`__func__`" to also report the current function name if you wish. There's also an older non-standard `__FUNCTION__` version of the macro. Note that the need for all these macros goes away once there is widespread C++ support for `std::stacktrace`, as standardized in C++23, in which case a failing assertion could report its own call stack in an error message.

When Assertions Fail. This `aussie_assert` macro relies on a function that is called only when an assertion has failed. And the function has to have a dummy return type of "`bool`" so that it can be used as an operand of the `||` operator, whereas a "`void`" return type would give a compilation error. Hence, the declaration is:

```
// Assertion failed
bool aussie_assert_fail(char* str, char* fname, int ln);
```

And here's the definition of the function:

```
bool aussie_assert_fail(char* str, char* fname, int ln)
{
    // Assertion failure has occurred...
    g_aussie_assert_failure_count++;
    printf("AUSSIE ASSERTION FAILURE: %s, %s:%d\n",
            str, fname, ln);
    return false;  // Always fails
}
```

This assertion failure function must always return "`false`" so that the assertion macro can be used in an `if`-statement condition.

Assertion Failure Extra Message

The typical assertion macro will report a stringized version of the condition argument (i.e., `#cond` is the special stringize operator), plus the source code filename, line number, and function name. This can be a little cryptic, so a more human-friendly extra message is often added. The longstanding hack to do this has been:

```
aussie_assert(fp!=NULL && "File open failed"); // Works
```

The trick is that a string constant has a non-null address, so `&&` on a string constant is like doing "*and true*" (and is hopefully optimized out). This gives the extra message in the assertion failure because the string constant is stringized into the condition (although you'll also see the "`&&`" and the double quotes, too). Note that an attempt to be tricky with comma operator fails:

```
aussie_assert(fp!=NULL, "File open failed"); // Bug
```

There are two problems. Firstly, it doesn't compile because it's not the comma operator, but two arguments to the `aussie_assert` macro.

Even if this worked, or if we wrapped it in double-parentheses, there's a runtime problem: this assertion condition will never fail. The result of the comma operator is the string literal address, which is never false.

Optional Assertion Failure Extra Message: The above hacks motivate us to see if we could allow an optional second parameter to assertions. We need two usages, similar to how "`static_assert`" currently works in C++:

```
aussie_assert(fp != NULL);
aussie_assert(fp != NULL, "File open failed");
```

Obviously, we can do this if "`aussie_assert`" was a function, using basic C++ function default arguments or function overloading. If you have faith in your C++ compiler, just declare the functions "`inline`" and go get lunch. But if we don't want to call a function just to check a condition, we can also use C++ variadic macros.

Variadic Macro Assertions

C++ allows #define preprocessor macros to have variable arguments using the "`...`" and "`__VA_ARG__`" special tokens. Our `aussie_assert` macro changes to:

```
#define aussie_assert(cond, ...) \
   ( (cond) || \
    aussie_assert_fail(#cond, \
        __FILE__, __LINE__, __VA__ARG__))
```

And we change our "`aussie_assert_fail`" to have an extra optional "message" parameter.

```
bool aussie_assert_fail(
  char* str, char* fname, int ln, char *mesg=0
);
```

This all works fine if the `aussie_assert` macro has 2 arguments (condition and extra message) but we get a bizarre compilation error if we omit the extra message (i.e., just a basic assertion with a condition). The problem is that `__VA_ARG__` expands to nothing (because there's no optional extra message argument), and the replacement text then has an extra "`,`" just hanging there at the end of the argument list, causing a syntax error.

Fortunately, the deities who define C++ standards noticed this problem and added a solution in C++17. There's a dare-I-say "hackish" way to fix it with the `__VA__OPT__` special token. This is a special token whose only purpose is to disappear along if there's zero arguments to `__VA_ARG__` (i.e., it takes the ball and goes home if there's no-one else to play with). Hence, we can hide the comma from the syntax parser by putting it inside `__VA_OPT__` parentheses.

```
#define aussie_assert(cond, ...) \
   ( (cond) || \
      aussie_assert_fail(#cond, __FILE__, __LINE__ \
           __VA_OPT__(,) __VA__ARG__) )
```

Note that the comma after `__LINE__` is now inside of a `__VA_OPT__` special macro. Actually, that's not the final, final version.

We really should add "`__func__`" in there, too, to report the function name. Heck, why not add `__DATE__` and `__TIME__` while we're at it? Why isn't there a standard `__DEVELOPER__` macro that adds my name?

Assertless Production Code

Not everyone likes assertions, and coincidentally some people wear sweaters with reindeer on them. If you want to compile out all of the assertions from the production code, you can use this:

```
#define aussie_assert(cond)  // nothing
```

But this is not perfect, and has an insidious bug that occurs rarely (if you forget the semicolon). A more professional version is to use "`0`" and this works by itself, but even better is a "`0`" that has been typecast to type "`void`" so it cannot be accidentally used in any expression:

```
#define aussie_assert(cond) ( (void)0 )
```

The method to remove the variadic macro version uses the "`...`" token:

```
#define aussie_assert(cond, ...) ( (void)0 )
```

Personally, I don't recommend doing this at all, as I think that assertions should be left in the production code for improved supportability. I mean, come on, recycle and reuse, remember? Far too many perfectly good assertions get sent to landfill every year.

Assertion Return Value Usage

Some programmers like to use an assertion style that tests the return code of their `assert` macro:

```
if (assert(ptr != NULL)) {  // Risky
    // Normal code
    ptr->count++;
}
else {
    // Assertion failed
}
```

This assertion style can be used if you like it, but I don't particularly recommend it, because it has a few risks:

1. The failure function returns `false` so the `if` test fails when the assertion fails.

2. Embedding assertions deeply into the main code expressions increases the temptation to use side effects like "++" in the condition, which can quietly disappear if you ever remove the assertions from a production build:

```
if (assert(++i >= 0)) { // Risky
    // ...
}
```

3. The usual assertion removal method of "`((void)0)`" will fail with compilation errors in an `if` statement. Also using a dummy replacement value of "0" is incorrect, and even "1" is not a great option, since the "`if(assert(ptr!=NULL))`" test becomes the unsafe "`if(1)`". A safer removal method is a macro:

```
#define assert(cond) (cond)
```

Or you can use an `inline` function:

```
inline void assert(bool cond) { } // Empty
```

This avoids crashes, but may still leave debug code running (i.e., a slug, not a bug). It relies on the optimizer to remove any assertions that are not inside an "`if`" condition, which just leave a null-effect condition sitting there. Note also that this removal method with "`(cond)`" is also safer because keeping the condition also retains any side-effects in that condition (i.e., the optimizer won't remove those!).

Generalized Assertions

Once you've used assertions for a while, they begin to annoy you a little bit. They can fail a lot, especially during initial module development and unit testing of new code. And that's the first time they get irritating, because the assertion failure reports don't actually give you enough information to help debug the problem. However, you can set a breakpoint on the assertion failure code when running in `gdb`, so that's usually good enough.

The second time that assertions are annoying is when you ship the product. That's when you see assertion failures in the logs as an annoying reminder of your own imperfections. Again, there's often not enough information to reproduce the bug.

So, for your own sanity, and for improved supportability, consider extending your own assertion library into a kind of simplified unit-testing library. The extensions you should consider:

- Add `std::stacktrace` capabilities if you can, or use Boost Stacktrace or GCC `backtrace` as a backup. Printing the whole stack trace on an assertion failure is a win.
- Add extra assertion messages as a second argument.
- Add `__func__` to show the function name, if you haven't already.

And you can also generalize assertions to cover some other common code failings.

- Unreachable code assertion
- "Null pointer" assertion
- Integer value assertions
- Floating-point value assertions
- Range value assertions

Creating specialized assertion macros for these special cases also means the error messages become more specific.

Unreachable code assertion

This is an assertion failure that triggers when code that should be unreachable actually got executed somehow. The simple way that programmers have done this in the past is:

```
aussie_assert(0);  // unreachable
```

And you can finesse that a little with just a better name:

```
#define aussie_assert_not_reached() \
    ( aussie_assert(false) )
...
aussie_assert_not_reached(); // unreachable
```

Here's a nicer version with a better error message:

```
#define aussie_assert_not_reached() \
    ( aussie_assert_fail("Unreachable code", \
     __FILE__, __LINE__) )
```

Once-only execution assertion

Want to ensure that code is never executed twice? A function that should only ever be called once? Here's an idea for an assertion that triggers on the second execution of a code pathway, by using its own hidden "`static`" call counter local variable:

```
#define aussie_assert_once()  do { \
    static int s_count = 0; \
        ++s_count; \
        if (s_count > 1) { \
        aussie_assert_fail("Code executed twice", \
                __FILE__, __LINE__); \
        } \
   } while(0)
```

Restricting any block of code to once-only execution is as simple as adding a statement like this:

```
aussie_assert_once();   // Not twice!
```

This can be added at the start of a function, or inside any `if` statement or `else` clause, or at the top of a loop body (although why is it coded as a loop if you only want it executed once?). Note that this macro won't detect the case where the code is never executed. Also note that you could customize this macro to return an error code, or throw a different type of exception, or other exception handling method when it detects double-executed code.

Function Call Counting

The idea of once-only code assertions can be generalized to a count. For example, if you want to ensure a function isn't called too many times, use this code:

```
aussie_assert_N_times(1000);
```

Here's the macro, similar to `aussie_assert_once`, but with a parameter:

```
#define aussie_assert_N_times(ntimes)  do { \
        static int s_count = 0; \
            ++s_count; \
            if (s_count > (ntimes)) { \
            aussie_assert_fail( \
           "Code executed more than " \
                #ntimes " times", \
                __FILE__, __LINE__); \
            } \
   } while(0)
```

This checks for too many invocations of the code block. Checking for "too few" is a little trickier, and would need a `static` smart counter object with a destructor.

Detecting Spinning Loops

Note that the above call-counting macro doesn't work for checking that a loop isn't spinning. It might seem that we can use the above macro at the top of the loop body to avoid a loop iterating more than 1,000 times. But it doesn't work, because it will count multiple times that the loop is entered, not just a single time. If we want to track a loop call count, the counter should not be a "`static`" variable, and it's more difficult to do in a macro. The simplest method is to hand-code the test:

```
int loopcount = 0;
while (...) {
    if (++loopcount > 1000) {  // Spinning?
        // Warn...
    }
}
```

Generalized Variable-Value Assertions

Various generalized assertion macros can not only check values of variables, but also print out the value when the assertion fails. The basic method doesn't print out the variable's value:

```
aussie_assert(n == 10);
```

A better way is:

```
aussie_assertieq(n, 10);  // n == 10
```

The assertion macro looks like:

```
#define aussie_assertieq(x,y) \
     (( (x) == (y)) || \
      aussie_assert_fail_int(#x "==" #y, \
          (x), "==", (y), \
         __FILE__, __LINE__))
```

The assertion failure function has extra parameters for the variables and operator string:

```
bool aussie_assert_fail_int(char* str, int x,
        char *opstr, int y, char* fname, int ln)
{
    // Assert failure has occurred...
    g_aussie_assert_failure_count++;
    fprintf(stderr, "INT ASSERT: %s, %d %s %d, %s:%d\n",
            str, x, opstr, y, fname, ln);
    return false;  // Always fails
}
```

If you don't mind lots of assertion macros with similar names, then you can define named versions for each operator, such as:

- `aussie_assertneq` — `!=`
- `aussie_assertgtr` — `>`
- `aussie_assertgeq` — `>=`
- `aussie_assertlss` — `<`
- `aussie_assertleq` — `<=`

If you don't mind ugly syntax, you can generalize this to specify an operator as a parameter:

```
aussie_assertiop(n, ==, 10);
```

The macro with an "op" parameter is:

```
#define aussie_assertiop(x, op, y) \
     (( (x) op (y)) || \
      aussie_assert_fail_int(#x #op #y, \
          (x), #op, (y), \
         __FILE__, __LINE__))
```

And finally, you have to duplicate all of this to change from `int` to `float` type variables. For example, there's macros named "`aussie_assertfeq`", "`aussie_assertfop`", and a failure function named "`aussie_assert_fail_float`". There's probably a fancy way to avoid this using function overloading or C++ templates and compile-time type traits, but only if you're smarter than me.

Assertions for Function Parameter Validation

Assertions and toleration of exceptions have some tricky overlaps. Consider the modified version of vector summation with my own "`aussie_assert`" macro instead:

```
float vector_sum(float v[], int n)
{
    aussie_assert(v != NULL);
    // etc..
}
```

What happens when this assertion fails in a custom assertion macro? The execution will progress after the assertion, in which case any use of `v` will be a null pointer dereference. The code is not very resilient.

Hence, the above code works fine only if your custom "`aussie_assert`" assertion macro throws an exception. This requires that you have a robust exception handling mechanism in place above it, for the caught exception, which is a significant amount of work. There are also problems with using assertions in destructors if you throw exceptions on assertion failure.

The alternative is to both assert and handle the error in the same place, which makes for a complex block of code:

```
aussie_assert(v != NULL);
if (v == NULL) {
     return 0.0;  // Tolerate
 }
```

Slightly more micro-efficient is to only test once:

```
if (v == NULL) {
    aussie_assert(v != NULL); // Always triggers
    return 0.0;  // Tolerate
}
```

This is a lot of code that can get repeated all over the place. Various copy-paste coding errors are inevitable.

Assert Parameter and Return

An improved solution is an assertion macro that captures the logic "check parameter and return zero" in one place. Such a macro first tests a function parameter and if it fails, the macro will not only emit an assertion failure message, but will also tolerate the error by returning a specified default value from the function.

Here's a generic version for any condition:

```
#define aussie_assert_and_return(cond,retval) \
    if (cond) {} else { \
        aussie_assert_fail(#cond " == NULL", \
              __FILE__, __LINE__); \
        return (retval); \
    }
```

The usage of this function is:

```
float aussie_vector_something(float v[], int n)
{
        aussie_assert_and_return(v != NULL, 0.0f);
    ...
}
```

The above version works for any condition. Here's another version specifically for testing an incoming function parameter for a `NULL` value:

```
#define aussie_assert_param_tolerate_null(var,retval) \
    if ((var) != NULL) {} else { \
        aussie_assert_fail(#var " == NULL", \
              __FILE__, __LINE__); \
        return (retval); \
    }
```

The usage of this function is:

```
  aussie_assert_param_tolerate_null(v, 0.0f);
```

If you want to be picky, a slightly better version wraps the "`if-else`" logic inside a "`do-while(0)`" trick. This is a well-known trick to make a macro act more function-like in all statement structures.

```
#define aussie_assert_param_tolerate_null2(var,retval) \
    do { if ((var) != NULL) {} else { \
        aussie_assert_fail(#var " == NULL", \
              __FILE__, __LINE__); \
        return (retval); \
    }} while(0)
```

The idea of this macro is to avoid lots of parameter-checking boilerplate that will be laborious and error-prone. But it's also an odd style to hide a `return` statement inside a function-like preprocessor macro, so this is not a method that will suit everyone.

Next-Level Assertion Extensions

Here are some final thoughts on how to further improve your assertions:

- Change any often-triggered assertions into proper error messages with fault tolerance. Users don't like seeing assertion messages. They're kind of like gibberish to ordinary mortals.
- Add extra context information in the assertion message (i.e., add an extra information string). This is much easier to read than a stringized expression, filename with line number, or multi-line stack trace.

- Add unique codes to assertion messages for increased supportability. Although, maybe not, because any assertion that's triggering often enough to need a code, probably shouldn't remain an assertion!
- `inline` assertion function? Why use macros? Maybe these assertions should instead be an `inline` function in modern C++? And it could report context using `std::backtrace`. All I can say is that old habits die hard, and I still don't trust the optimizer to actually optimize much.

The downside of assertions is mainly that they make you lazy as a programmer because they're so easy to add. But sometimes no matter how good they seem, you have to throw an assertion into the fires of Mordor. The pitfalls include:

- Don't use assertions instead of user input validation.
- Don't use assertions to check program configurations.
- Don't use assertions as unit tests (it works, but bypasses the test harness statistics).
- Don't use assertions to check if a file opened.

You need to step up and code the checks of input and configurations as part of proper exception handling. For example, it has to check the values, and then emit a useful error code if they've failed, and ideally it's got a unique error code as part of the message, so that users can give a code to support if they need. You really don't want users to see the dirty laundry of an assertion message with its source file, function name, and line number.

14. Safe Standard C++ Library

Debug Standard Library Versions

Various vendor capabilities exist in terms of debugging features and debug versions of the standard library. GCC appears to be leading the way, and Clang has a lot of features too. The Microsoft Visual Studio capabilities are less fully formed in this area, although it does have static analysis capabilities, runtime checks and some other debugging features.

GCC Debug Library. GCC does have a debug version of its `glibcxx` library, which you can link in by using the `/usr/lib/debug` path. You can add linker options like these:

- "`-L/usr/lib/debug`" for static libraries, or
- "`LD_LIBRARY_PATH=/usr/lib/debug`" for dynamic libraries.

But that's not really what I'm talking about. This is a version which has the symbolic information retained for better debugging, rather than a version which offers additional safety and debugging features.

But GCC also has one of those!

GCC has a debug-enabled version of `libstdc++` with additional error checking enabled. Interestingly, this is implemented via a "wrapper model" on top of the production versions of the standard library. The GCC debug library with error checking is `_GLIBCXX_DEBUG` and there is also a flag `_GLIBCXX_ASSERTIONS`.

Clang debug capabilities. Clang has been following GCC with a lot of common features. For example, Clang has options to launch a variety of sanitizer tools at runtime (e.g., ASan, UBSan, TSan, MSan, etc.), and has the Clang Static Analyzer and `clang-tidy` for source code analysis.

Safe Standard Libraries

This section is a list of bugs and undefined behaviors that could probably be detected by a linkable debug version of the standard libraries. This could mean:

- Macro-intercepted debug wrapper library, or
- Vendor offered debug versions of the standard library.

The standard library could perform a lot more internal self-tests to detect and prevent serious internal errors. This could be done better by compiler vendors, but a lot could also be done by users with a debug wrapper library based on macro or link-time intercepts.

For error detection and prevention, I'm thinking in terms of what could be done quickly, just by testing a few flags or magic values, rather than a full-scale memory debugging library (i.e., not to the extent of `valgrind` or other sanitizers). A lot of internal errors could be changed from crashes to harmless logged warnings.

Error preventions. The standard library is in a position to prevent several categories of memory errors. For example, why don't `malloc` and `new` clear their memory to zero? Similarly, the `alloca` function could clear stack memory. This would prevent a whole range of "uninitialized memory use" errors. Surely, this wouldn't be very slow, or it could be offered as an option to users of the library.

Memory errors. The heap management libraries need to have a hidden block of data for each allocated memory block, anyway. So, add some more flags and magic values in there, which are then checked by all of the primitives trying to access a memory block. Many of the usual suspects could be caught relatively simply:

- Double de-allocation
- Non-heap address for de-allocation
- Mismatched allocation/de-allocations

Detecting usage of uninitialized memory from `malloc` or `new` is less simple, because although you can set a flag to say "newly allocated block," it's harder to know if a simple array access has written to the memory block, thereby initializing it. However, this is also doable with reasonable efficiency via a single magic value in the first few bytes of a block.

Memory overruns are harder to detect only in a library. However, there are two ways to detect this:

- Add canary memory ranges at the end of the allocated blocks (i.e., a magic value just a few bytes afterwards).
- Intercept the standard C++ library byte manipulation library calls (e.g., `memset`, `strcpy`, etc.)

Many of the techniques that can be used by vendor standard libraries are discussed in the chapter on debug wrapper libraries. It's the same problem.

File pointers and file descriptors. All of the file descriptors (integers), file pointer structures, or `fstream` classes manage a block of memory for an open file, which could contain bit flags of its status (e.g., open, just written, just read, closed, etc.). Furthermore, these file data structures are only used or modified via standard library primitives, so a debug version of a standard library could surely detect most errors. Also, the pointers for these file pointers are usually within a fixed-size array, so it's two pointer comparisons to validate the file pointer is inside this block. And finally, the most recent type of file operation could be tracked in a bit flag (e.g., recently read, recently written, recently seeked).

It seems like these file blocks could be self-tested on any file access. Hence, my list of file-related errors and undefined behaviors that could be (a) detected, and (b) made harmless, includes:

- Null file pointer value.
- Invalid file descriptor (or file pointer or `fstream`) is outside the range for valid pointer addresses, or an invalid integer file handle.
- File descriptor is a "never-opened" file (i.e., does not have an "opened" or "closed" bit flag set).
- Double-close (i.e., the "closed" bit flag is already set).
- Read/write/seek operation on already-closed file.
- Read/write sequence without intervening seek on file pointer (a weird "undefined behavior").
- `fflush` on an input file (e.g., `fflush(stdin)` is a common mistake).

All of these file-related errors would become suddenly harmless. I've certainly had crashes myself from "double `fclose`" errors. These are fixable at the performance cost of a few bit flag and pointer comparisons.

Worth doing!

Going further, this debug file management library could have trace capabilities, so you can view not only the above serious internal errors, but also application-level happenings such as what files are opened and closed. This could already record "warnings" for things like "file not found" or read or write failed to read/write enough bytes.

As such, odd file occurrences could get telegraphed to the developer earlier.

Character functions. Various functions could be more carefully implemented to check errors and tolerate unusual usage. Examples include:

- Character category function arguments (e.g., `islower`) should warn and tolerate out-of-range values (e.g., more than 255) and consistently handle negative values (i.e., from signed character types).
- Character category function return values limited to Boolean status (e.g., `islower` should return only `true` or `false`, rather than zero versus non-zero).

String functions. There are various failures that could be detected in libraries.

- Memory addresses — the standard string functions, such as `strcpy`, should be incorporated into the memory safety checking.
- Standard return values — e.g., `strcmp` should return explicitly -1, 0, or 1, rather than only requiring less-than, equal-to, or greater-than zero (this makes harmless a common misuse).

Math functions. At first thought, it seems that there's not that many errors that can be auto-detected by changes to the standard mathematical libraries, without needing changes to the C++ operators. However, I came up with a few that can be done just inside the functions themselves:

- Degrees versus radians — `cos(60.0)` is confusing radians and degrees. The library could warn if any trigonometric function is using large values that are unusual for radians, or degree-like exact integer values (30, 45, 60, 90, etc.)
- Any `errno`-setting happening in the math library could issue or log a warning, rather than relying on the caller to check `errno`.
- Any arithmetic function that returns a result `NaN` could log a warning.
- Invalid ranges of arguments could also log a warning and be made harmless (e.g., division by zero, or infinite results like `tan(0.0)` is one).

Container classes. The various standard C++ container classes could improve their quality, such as:

- Out-of-bounds array accesses on `std::vector` should be expressly handled with warning and make-harmless processing.
- Tracing and logging of uncommonly occurring problem issues, such as the `std::vector` automatically resizing itself (a common slug).

Extra Builtin Functions for Debugging

It's somewhat difficult to build your own debug library, whether macro-intercepted or linked, because some things are hidden in the compiler implementation layer.

Some API primitives that would be useful if provided by compiler vendors include:

- Address categorization — is this address on the stack? In the heap? Global? Read-only data? String literal?
- Address is an allocated block? — valid heap block start address? Inside a valid block? Which block?
- Address of the start of an allocated block, if given an address inside it.
- Size of the allocated memory block, if given the start address of a heap block.
- Address of start of a stack block given an address inside.
- Size of a stack block if given the starting stack block address.
- Heap statistics — size of the heap, remaining free memory, block counts, etc.
- Callbacks that are callable on various internal events (e.g., memory allocation, de-allocation, various primitives called, etc.).

Note that some of these could be offered in two or more versions, with a slower and faster option. Fast memory address checking could be based on a magic value or other trick.

Slower memory address checking would, for example, scan the entire heap to confirm it's a valid address.

These would be valuable for users to create their own debug library infrastructure. It would also be valuable for coding policies aimed at memory safety, whereby the user could decide whether to incur the runtime cost of checking a memory address before using it.

Some of these issues are flat out impossible to do in any C++ platforms. A few of them are possible in a very non-portable way. It seems to me that a compiler vendor could offer a lot of these functions with a relatively low amount of work, because they have a great deal more information available behind the scenes.

15. Safety Wrapper Functions

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.

Wrapping Memory Functions

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!

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.

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.

Here's an example:

```
#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.

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:

```
// Wrap memset
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" above
        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 not char");
       return dest; // fail
   }
   void* sret = ::memset(dest, val, sz);  // Real call!
   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_wrapper(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).

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 you can't 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:

```
#define AUSSIE_ERROR(mesg, ...) \
    ( printf((mesg) __VA_OPT__(,) __VA_ARGS__ ) )

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.

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.

16. Debugging Strategies

General Debugging Techniques

A lot of the work in debugging programs is nothing special: it's just basic C++ coding mistakes. Most of the errors in coding are ordinary, boring coding errors that every C++ programmer is prone to. There are a variety of ways to go wrong in handling pointers and addresses, from basic beginner mistakes to traps that can catch the experienced practitioner.

The best way to catch a bug is to try to make it happen *early*. We want the program to crash in the lab, not out in production. In this regard, some of the best practices are about auto-detecting the failures in your code, rather than waiting for them to actually cause a crash:

- Check every return code (even the harmless functions that can "never" fail).
- Use macro wrappers to help handle errors.
- Add debug wrapper functions and enable them while testing.
- Run `valgrind` or other sanitizers on your code regularly.
- Thrash the code in many ways in the nightly builds.

If you mess up, and a bug happens in the production backend of your AI training run, I suggest this: blame the data scientists. Surely, the problem was in the training data, not in my perfect C++ code. And if that doesn't work, well, obviously the GPU was overheating.

Very Difficult Bugs. Some bugs are like roaches and keep coming out of the woodwork. General strategies for solving a tricky bug include:

- Can you reproduce it? That's the key.
- Write a unit test that triggers it (if you can).
- Try to cut down the input to the smallest case that triggers the fault.
- Gather as much information about the context as possible (e.g., if it's a user-reported error).

Your debugging approach should include:

- Run `valgrind` or sanitizers to check for memory glitches.
- Think about what code you just changed recently (or was just committed to the repo by someone else!).
- Memory-related failures often cause weird errors nowhere near the cause.
- Review the debug trace output carefully (i.e., may be that some other part of the code failed much earlier).
- Step through the code in `gdb` about ten more times.
- Run a static analysis ("linter") tool on the code.
- Run an AI copilot debugger tool. I hear they're terrific.
- Refactor a large module into smaller functions that are more easily unit-tested (often you accidentally fix the bug!).

If you really get stuck, you could try talking to another human (gasp!). Show your code to someone else and they'll find the bug in three seconds.

Bug Symptom Diagnosis

It is very beneficial to the debugging process to be able to identify the cause of an error from its symptoms. Unfortunately, this is a very difficult process — otherwise debugging would be easy! Nevertheless, there are some common run-time errors with well-known causes, and this section attempts to provide a brief catalog of common error causes, mapping observable failure symptoms into the common errors.

Linux core dumps

There are a number of run-time error messages that occur mainly on Linux machines. Some of the common run-time error messages are:

- Segmentation fault
- Bus error
- Illegal instruction
- Trace/BPT trap

The message "`core dumped`" will often accompany the error message if it causes program termination, and this indicates that a file named "core" has been saved in the current directory. The "`core`" file can be used for postmortem debugging to locate the failure with a symbolic debugger.

Note that the dump of the `core` file can be prevented by providing an empty file named "`core`" that is set to protection mode `000` using `chmod`. This may be useful if disk space is limited and the core dumps are huge.

A segmentation fault occurs when the hardware detects a memory access by the program that attempts to reference memory it is not allowed to use. For example, the address `NULL` cannot be referenced, and in fact, the single most common cause of a segmentation fault (at least for the experienced programmer) is a null dereference, but there are many other causes.

A bus error occurs when an attempt is made to load an incorrect address into an address bus. Although this leads us to suspect bad pointers, this error can also arise via stack corruption (because this can cause bad pointer addresses), and so there are a variety of potential causes.

Segmentation faults and bus errors may be reported as the program receiving signal `SIGSEGV` or `SIGBUS` in some situations. The most common causes of a segmentation fault or bus error are listed below. Different architectures will have different results for these errors, but will usually produce either a segmentation fault or bus error.

Here are some of the causes:

- Null pointer dereference.
- Wayward pointer dereference (memory allocation problem).
- Non-initialized pointer dereference.
- Array index out of bounds.
- Wrong number or type of arguments to non-prototyped function.
- Bad arguments to `scanf` or `printf`.
- Forgetting the `&` on arguments to `scanf`.
- Deallocating non-allocated location using `free` or `delete`.
- Deallocating same address twice using `free` or `delete`.
- Executable file removed/modified as being executed (dynamic loading).
- Stack overflow due to function calls or automatic variables.

Another common abnormal termination condition for Linux machines is the message "illegal instruction," which usually causes a core dump.

The most common causes of this method of termination are:

- `assert` macro has failed (causes `abort` call).
- `abort` library function called.
- Data has been executed somehow (uninitialized pointer-to-function?).
- Stack corruption (e.g., write past end of local array).
- Stack overflow (not the website).
- C++ exception problem causing `abort` call.
- Unhandled exception was thrown.
- Unexpected exception from function with interface specification.
- Exception thrown in destructor during exception-related stack unwinding.

Another run-time error message for Linux machines is the message "fixed up non-aligned data access," although this does not necessarily lead to program termination. This indicates that hardware has detected an attempt to access a value through an address with incorrect alignment requirements. Typically, it refers to attempting to read or write an integer or pointer at an odd-valued address (i.e., an address that is not word-aligned). Note that on machines without this automatic "fix-up" the same code will probably cause a bus error.

Program hangs infinitely

When one is faced with debugging a program that seems to get stuck, it is important to determine what type of "hang" has occurred. If the program is simply stuck in an infinite loop, you will still have control of the program and can interrupt it. One method of finding out where the program is stuck is to run the program from a debugger, or to use the keyboard interrupt <ctrl-\> to cause a core dump, which can then be examined by a debugger. Some causes of this form of infinite looping are:

- NP-complete algorithm (i.e., basically anything in AI).
- Infinite loop is occurring due to logic bug or coding error.
- Looping down to zero with a `size_t` index variable (i.e., `unsigned`).
- Accidental semicolon on end of `while`/`for` loop header.
- `exit called` within a destructor of global object (C++ only).
- Handled/ignored signal is recurring (e.g., `SIGSEGV`, `SIGBUS`).
- Waiting for input: `getc`/`getch` assigned to `char`.
- Linked data structure corrupted (contains pointer cycles).

Delayed crash

If the program hangs for a period of time and then crashes, a likely candidate is a runaway recursive function. This will loop (almost) infinitely, consuming stack space all the time, until it runs out of stack space and:

(a) Terminates abnormally, or

(b) The stack overwrites some important memory and the second, more severe form of "hang" occurs.

Non-responsive program

The most severe form of a "hung" program is one that will not respond. You know it's a bad bug when the reset button is the only thing that works. When this occurs, I recommend the use of any compiler run-time checks, especially stack overflow checking and array bounds checking (if available). An additional method is to recompile using a sanitizer or a memory allocation debugging library.

Some possible causes of a non-responsive program are:

- Infinite recursion error.
- Stack overflow for other reasons.
- Array index out of bounds.
- Modification via wayward pointer.
- Modification via non-initialized pointer.
- Modification via null pointer.
- Freeing a non-allocated block.
- Freeing a string constant.
- Non-terminated string was copied.
- Inconsistent compiler/linker options.

Failure after long execution

A very annoying error is that of a program that runs perfectly for a long period of time and then suddenly fails for no apparent reason. This usually indicates a "memory leak" causing the system to use up all available memory and `malloc` to return `NULL`.

However, there are other causes and a more complete list is:

- Untested rare sequence of events is causing the error (try to repeat it).
- Heap memory leak causing allocation failure (allocated memory not deallocated).
- Running out of `FILE*` handles (files opened but not closed).
- Some form of memory corruption (symptom of bug doesn't appear immediately).
- Integer overflow (e.g., of some 16-bit or 32-bit counter).
- Disk filling up (e.g., excessive logging).
- Unhandled peripheral error (e.g., printer out of paper).

Optimizer-only bugs

A program that runs correctly with normal compilation but fails when the optimizer is invoked is a well-known problem. The immediate reaction is to blame a bug in the optimizer. Memory safety errors are a likely cause! However, if your runtime memory checker is showing no errors, it's something else. although such bugs are not so rare as one would wish, there are a number of other potential causes. It is usually an indication that some erroneous or non-portable code has been working correctly more by luck than good programming, and the more aggressive optimizations have shown up the error. Some possible causes are:

- Order of evaluation errors (optimizer rearranges expressions).
- Special location not declared `volatile`.
- Use of an uninitialized variable.
- Wrong number/type of arguments to non-prototyped function.
- Wrong arguments to prototyped function not declared before use.
- Memory access problems (optimizer has rearranged some memory).

In this situation it may be useful to examine what compiler options are available to choose which optimizations are chosen. For example, there may be an option to choose between traditional stack-based argument passing and pass by register. If so, recompilation with and without that option can help to test for argument passing errors.

Failure disappears when debugger used

A really annoying situation is a program that crashes when run normally, but does not fail when run via a symbolic debugger or interpreter. One fairly well-known cause is the use of an uninitialized automatic variable. The error may disappear when run via the debugger, because some debuggers set these local variables to zero or `NULL` initially. Thus, some possible causes are mainly from memory access problems, where the debugger has rearranged memory somehow:

- Using uninitialized variable (especially a pointer)
- Array index out of bounds
- Modification via wayward pointer
- Modification via non-initialized pointer
- NULL pointer dereference
- Modification via null pointer
- Freeing a non-allocated block
- Freeing a string constant

The list of errors possibly causing a memory-related problem is comparable with the list of errors causing a non-recoverable hung program.

Program crashes at startup

When a C++ program crashes on program startup, without even executing the first statement in main, we must suspect constructors of global objects. Use a run-time debugger to determine if main has been entered; but note that some debuggers allow debugging of constructors before main and others do not. Alternatively, place an output statement as the very first statement in main (even before the first declaration!) to ensure that the problem really is arising before main, rather than from instructions in main. Once a constructor problem has been identified, finding the root cause of the problem is a debugging matter. There are no forms of error particular to constructors, so the problem is something being done by a constructor that is probably some type of other error (e.g., a memory stomp error).

Program crashes on exit

The program can fail in a few obscure ways at the end of execution. Careful consideration of what actions are taking place at the end of execution is important (e.g., destructors are invoked in C++; any functions registered with `atexit` will be called). In my experience this failure is most common during the learning phase in C++ programming, when destructor errors are common.

Possible causes include:

- `delete` operation in object destructor is trashing memory.
- Destructor in global object calls `exit`.
- `main` accidentally declared returning non-`int`. e.g., missing semicolon on `class` or `struct` declaration above `main`.
- `setbuf` buffer is a non-`static` local variable of `main`.
- No call to `exit`, and no `return` statement in `main` (a few platforms only).
- File closed twice (e.g., double `fclose` error).

Function apparently not invoked

Consider the situation where you are debugging a program, and discover that a particular function seems to be having no effect. You put an output statement at its first statement and no output appears. Why isn't the function being invoked?

Some possible causes are:

- No call to the function (!), e.g., you didn't rebuild properly (your fault), or a source code repo issue (someone else's fault) or you're looking in the wrong C++ source file (I've done it many times).
- Control flow or conditional test controlling the call is wrong.
- Missing brackets on function call (null-effect).
- Function is a macro at call location.
- Function is a reserved library function name (wrong function is getting called).
- Missig semicolon or statement in `if` statement above the function call.
- Nested comments unclosed and deleting call to function.

Garbage output

When a program runs and produces strange output there are a number of possibilities (mostly related to misusing string variables). Note that it is important to distinguish whether the output of a statement is entirely garbage or whether it has a correct prefix (which may indicate a non-terminated string).

Some causes are:

- Uninitialized local or allocated variable.
- Constructor not initializing all data members.
- Missing argument to `printf %s` format.
- Wrong type argument to `printf %s` format.
- Returning address of automatic local string array.
- Stack corruption (local array buffer overrun).
- `strncpy` leaves string non-terminated.
- Pointer variable not initialized.
- Address has already been deallocated

Failure on new platform

When a program appears to be running successfully on one machine, it is by no means guaranteed that porting the source code and recompiling on a new machine will not lead to new errors.

When a new error is discovered, the first thing that must be tested is whether the same error exists for the same test data on the original machine. The bug might not be a portability problem — it might be an untested case.

However, if the bug appears on one machine but not on another there are a few common causes. The most frequent portability problem is a memory corruption error since these will often lurk undetected on one machine, and appear in the new memory layout of a different environment.

Other possible causes are different compilation results that may arise when a new compiler uses more aggressive optimization. Hence, code that relies on an undocumented compiler feature (e.g., left to right function argument evaluation) may suddenly fail.

Note that this implies that portability errors can arise after a compiler upgrade on the same machine, as well as when moving code to a new machine.

Some common causes of portability errors are:

- Memory corruption errors.
- Array index out of bounds.
- Modification via wayward pointer.
- Modification via non-initialized pointer.
- Null pointer dereference.

- Modification via null pointer.
- Freeing a non-allocated block.
- Freeing a string constant.
- Function has no `return` statement.
- Order of evaluation error.
- Operator order-of-evaluation: `a[i]=i++;`
- Function argument order-of-evaluation: `fn(i,i++);`
- Global object construction in separate files.
- Special location not declared `volatile`.
- Use of an uninitialized variable.
- Constructor not initializing all data members.
- `new` doesn't initialize non-class types.
- `malloc` doesn't initialize any types.
- Bit-field is plain `int`.
- Plain `char` is `signed/unsigned`.
- `getc/getchar` return value assigned to `char`.

Most of these causes are fairly self-explanatory. However, the appearance of "function has no `return` statement" in the list may appear surprising — surely this will cause a bug on all implementations?

In fact, most C++ implementation will offer a compilation error, but this is not always true, and not applicable to C compilers. It has been observed surprisingly frequently that a function that terminates without a `return` statement might accidentally return the correct value. Typically, this surprising outcome occurs if, by coincidence, a local scalar variable that is intended to be returned happens to be in the hardware register that is used to hold the function return value. Since that register is not loaded when no return statement is found, the correct result is accidentally returned, and there is no failure until a different compiler or environment is used.

Some compilers have compilation options to change various compiler-dependent features. For example, there may be options to change the default type of plain `char` and/or plain `int` bit-fields to signed or unsigned. If it is suspected that this may be the cause of the error, the code can be recompiled with different option settings to confirm this. Any run-time error checking options such as memory allocation debugging and stack overflow checking should also be enabled.

Making the Correction

An important part of the debugging phase that is often neglected is actually making the correction. You've found the cause of the failure, but how do you fix it? It is imperative that you actually understand what caused the error before fixing it; don't be satisfied when a correction works and you don't know why.

Here are some thoughts on the best practices for the "fixing" part of debugging:

- Test it one last time.
- Add a unit test or regression test.
- Re-run the entire unit test or regression test suite.
- Update status logs, bug databases, change logs, etc.
- Update documentation (if applicable)

Another common pitfall is to make the correction and then not test whether it actually fixed the problem. Furthermore, making a correction will often uncover (or introduce!) another new bug. Hence, not only should you test for this bug, but it's a very good idea to use extensive regression tests after making an apparently successful correction.

Level Up Your Post-Debugging Routine. Assuming you can fix it, think about the next level of professionalism to avoid having a repetition of similar problems. Consider doing followups such as:

- Add a unit test or regression test to re-check that problematic input every build.
- Write it up and close the incident in the bug tracking database like a Goody Two-Shoes.
- Add safety input validation tests so that a similar failure is tolerated (and logged).
- Add a self-check in a C++ debug wrapper function to check for it next time at runtime.
- Is there a tool that would have found it? Or even a `grep` script? Can you run it automatically? Every build?

Production-Level Code

As with all applications, there's another level needed to get the code out the door into production. Some of the issues for fully production-ready C++ code include:

- Validate function parameters (don't trust the caller or the user).
- Check return codes of all primitives.
- Handle memory allocation failure (e.g., graceful shutdown).
- Add unique error message codes for supportability

Let's not forget that maybe a little testing is required. High-quality coding requires all manner of joyous programmer tasks: write unit tests, warning-free compilation, static analysis checks, add assertions and debug tracing, run `valgrind` or sanitizers, write useful commit summaries (rather than "I forget"), don't cuss in the bug tracking record, update the doc, comment your code, and be good to your mother.

17. Debug Tracing

Debug Tracing Messages

Ah, yes, worship the mighty `printf`!

A common debugging method is adding debug trace output statements to a program to print out important information at various points in the program. Judicious use of these `printf` statements can be highly effective in localizing the cause of an error, but this method can also lead to huge volumes of not particularly useful information. One desirable feature of this method is that the output statements can be selectively enabled at either compile-time or run-time.

Debug tracing messages are informational messages that you only enable during debugging. These are useful to software developers to track where the program is executing, and what data it is processing. The simplest version of this idea looks like:

```
#if DEBUG
printf("DEBUG: I am here!\n");
#endif
```

A better solution is to code some BYO debug tracing macros. Here's a macro version:

```
#define aussie_debug(str)  ( \
    fprintf(stderr, "DEBUG: %s\n", (str)) )
...
aussie_debug("I am here!");
```

Here's the C++ stream version:

```
#define aussie_debug(str) \
    ( std::cerr << str << std::endl )
...
aussie_debug("DEBUG: I am here!");
```

In order to only show these when debug mode is enabled in the code, our header file looks like this:

```
#if DEBUG
    #define aussie_debug(str) \
        ( std::cerr << str << std::endl )
#else
    #define aussie_debug(str) // nothing
#endif
```

Missing Semicolon Bug: Professional programmers prefer to use "`0`" rather than emptiness to remove the debug code when removing it from the production version. It is also good to typecast it to "`void`" type so it cannot accidentally be used as the number "`0`" in expressions. Hence, we get this improved version for removing a debug macro:

```
#define aussie_debug(str) ((void)0)  // better!
```

It's not just a stylistic preference. The reason is that the "nothing" version can introduce an insidious bug if you forget a semicolon after the debug trace call in an `if` statement:

```
if (something) aussie_debug("Hello world") // No semi!
x++;
```

If the "nothing" macro expansion is used, then the missing semicolon leads to this code:

```
if (something) // nothing
x++;
```

Can you see why it's a bug? Instead, if the expansion is "`((void)0)`" then this missing semicolon typo will get a compilation error.

Variable-Argument Debug Macros

A neater solution is to use varargs preprocessor macros with the special tokens "`...`" and "`__VA_ARGS__`", which are standard in C and C++ (since 1999):

```
#define aussie_debug(fmt,...) \
      printf((fmt), __VA_ARGS__ )
...
aussie_debug("DEBUG: I am here!\n");
```

That's not especially helpful, so we can add more context:

```
// Version with file/line/function context
#define aussie_debug(fmt,...)  \
    ( printf("DEBUG [%s:%d:%s]: ", \
        __FILE__, __LINE__, __func__ ), \
      printf((fmt), __VA_ARGS__ ))
...
aussie_debug("I am here!\n");
```

This will report the source code filename, line number, and function name. Note the use of the comma operator between the two `printf` statements (whereas a semicolon would be a macro bug). Also required are parentheses around the whole thing, and around each use of the "`fmt`" parameter.

Here's a final example that also detects if you forgot a newline in your format string (how kind!):

```
// Version with newline optional
#define aussie_debug(fmt,...)  \
    (printf("DEBUG [%s:%d:%s]: ", \
      __FILE__, __LINE__, __func__ ), \
     printf((fmt), __VA_ARGS__ ), \
     (strchr((fmt), '\n') != NULL \
           || printf("\n")))
...
aussie_debug("I am here!");  // Newline optional
```

Dynamic Debug Tracing Flag

Instead of using "`#if DEBUG`", it can be desirable to have the debug tracing dynamically controlled at runtime. This allows you to turn it on and off without a rebuild, such as via a command-line argument or inside a `gdb` session. And you can decide whether or not you want to ship it to production with the tracing available to be used. Your phone support staff would like to have an action to offer customers rather than "turn it off and on."

This idea of dynamic control of tracing can be controlled by a single Boolean flag:

```
extern bool g_aussie_debug_enabled;
```

We can add some macros to control it:

```
#define aussie_debug_off() \
    (g_aussie_debug_enabled = false )
#define aussie_debug_on() \
    (g_aussie_debug_enabled = true )
```

And then the basic debug tracing macros simply need to check it:

```
#define aussie_dbg(fmt,...) \
   ( g_aussie_debug_enabled && \
    printf((fmt), __VA_ARGS__ ))
```

So, this adds some runtime cost of testing a global flag every time this line of code is executed.

Here's the version with file, line, and function context:

```
#define aussie_dbg(fmt,...)  \
    ( g_aussie_debug_enabled && \
    ( printf("DEBUG [%s:%d:%s]: ", \
             __FILE__, __LINE__, __func__ ), \
    printf((fmt), __VA_ARGS__ )))
```

And here's the courtesy newline-optional version:

```
#define aussie_dbg(fmt,...)  \
    ( g_aussie_debug_enabled && \
    (printf("DEBUG [%s:%d:%s]: ", \
      __FILE__, __LINE__, __func__ ), \
    printf((fmt), __VA_ARGS__ ), \
    (strchr((fmt), '\n') != NULL \
        || printf("\n"))))
```

Multi-Statement Debug Trace Macro

An alternative method of using debugging statements is to use a special macro that allows any arbitrary statements. For example, debugging output statements can be written as:

```
DBG( printf("DEBUG: Entered function print_list\n"); )
```

Or using C++ `iostream` output style:

```
    DBG(    std::cerr    <<    "DEBUG:    Entered    function
print_list\n"; )
```

This allows use of multiple statements of debugging, with self-testing code as:

```
    DBG( count++; )
    DBG( if (count != count_elements(table)) { )
    DBG(   aussie_internal_error("ERROR: Count wrong");)
    DBG( } )
```

But it's actually easier to add multiple lines of code or a whole block in many cases. An alternative use of `DBG` with multiple statements is valid, provided that the enclosed statements do not include any comma tokens (unless they are nested inside matching brackets). The presence of a comma would separate the tokens into two or more macro arguments for the preprocessor, and the `DBG` macro above requires only one parameter:

```
  DBG(
      count++;
      if (count != count_elements(table)) { // self-test
          aussie_internal_error("ERROR: Count wrong");
      }
    )
```

The multi-statement `DBG` macro is declared in a header file as:

```
    #if DEBUG
    #define DBG(token_list) token_list  // Risky
    #else
    #define DBG(token_list) // nothing
    #endif
```

The above version of `DBG` is actually non-optimal for the macro error reasons already examined. A safer idea is to add surrounding braces and the "`do-while(0)`" trick to the `DBG` macro:

```
    #if DEBUG
    #define DBG(token_list) \
        do { token_list } while(0)   // Safer
    #else
    #define DBG(token_list)   ((void)0)
    #endif
```

Note that this now requires a semicolon after every expansion of the DBG macro, whereas the earlier definition did not:

```
DBG( std::cerr << "Value of i is " << i << "\n"; );
```

Whenever debugging is enabled, the statements inside the DBG argument are activated, but when debugging is disabled they disappear completely. Thus, this method offers a very simple method of removing debugging code from the production version of a program, if you like that kind of thing.

This DBG macro may be considered poor style since it does not mimic any usual syntax. However, it is a neat and general method of introducing debugging statements, and is not limited to output statements.

Yet another alternative style is to declare the DBG macro so that it follows this statement block structure:

```
DBG {
    // debug statements
}
```

Refer to the implementation of a block "SELFTEST" macro in the prior chapter for details on how to do this.

Multiple Levels of Debug Tracing

Once you've used these debug methods for a while, you start to see that you get too much output. For a while, you're just commenting and uncommenting calls to the debug routines. A more sustainable solution in a large project is to add numeric levels of tracing, where a higher number gets more verbose.

To make this work well, we declare both a Boolean overall flag and a numeric level:

```
extern bool g_aussie_debug_enabled;
extern int g_aussie_debug_level;
```

Here's the macros to enable and disable the basic level:

```
#define aussie_debug_off()   ( \
    g_aussie_debug_enabled = false, \
    g_aussie_debug_level = 0)
```

```
#define aussie_debug_on()   ( \
    g_aussie_debug_enabled = true, \
    g_aussie_debug_level = 1 )
```

And here's the new macro that sets a numeric level of debug tracing (higher number means more verbose):

```
#define aussie_debug_set_level(lvl)   ( \
    g_aussie_debug_enabled = (((lvl) != 0)), \
    g_aussie_debug_level = (lvl) )
```

Here's what a basic debug macro looks like:

```
#define aussie_dbglevel(lvl,fmt,...)   ( \
    g_aussie_debug_enabled && \
    (lvl) <= g_aussie_debug_level && \
    printf((fmt), __VA_ARGS__ ))
...
aussie_dbglevel(1, "Hello world");
aussie_dbglevel(2, "More details");
```

Now we see the reason for having two global variables. In non-debug mode, the only cost is a single Boolean flag test, rather than a more costly integer "<" operation.

And for convenience we might add multiple macro name versions for different levels:

```
#define aussie_dbglevel1(fmt) \
   (aussie_debuglevel(1, (fmt)))
#define aussie_dbglevel2(fmt) \
   (aussie_debuglevel(2, (fmt)))
...
aussie_dbglevel1("Hello world");
aussie_dbglevel2("More details");
```

Very volatile. Note that if you are altering debug tracing levels inside a symbolic debugger (e.g., `gdb`) or IDE debugger, you might want to consider declaring the global level variables with the "`volatile`" qualifier. This applies in this situation because their values can be changed (by you!) in a dynamic way that the optimizer cannot predict. On the other hand, you can skip this, as this issue won't affect production usage, and only rarely impacts your own interactive debugging usage.

BYO debug printf: All of the above examples are quite fast in execution, but heavy in space usage. They will be adding a fair amount of executable code for each "`aussie_debug`" statement. I'm not sure that I really should care that much about the code size, but anyway, we could fix it easily by declaring our own variable-argument debug `printf`-like function.

Advanced Debug Tracing

The above ideas are far from being the end of the options for debug tracing. The finesses to using debug tracing messages include:

- Environment variable to enable debug messages.
- Command-line argument to enable them (and set the level).
- Configuration settings (e.g., changeable inside the GUI, or in a config file).
- Add unit tests running in trace mode (because sometimes debug tracing crashes!).
- Extend to multiple sets or named classes of debug messages, not just numeric levels, so you can trace different aspects of execution dynamically.

Supportability Tip: Think about customers and debug tracing messages: are there times when you want users to enable them? Usually, the answer is yes. Whenever a user has submitted an error report, you'd like the user to submit a run of the program with tracing enabled to help with reproducibility. Hence, consider what you want to tell customers about enabling tracing (if anything). Similarly, debug tracing messages could be useful to phone support staff in various ways to diagnose or resolve customer problems. Consider how a phone support person might help a customer to enable these messages.

18. Portability

Portability Strategy

You do need portability if your users have different platforms. There are also various generic benefits from having most of the C++ code being standardized and portable. Using portable C++ in the general code areas means being able to run unit test on lots of utility code on developer's boxes, no matter what the deployment platform.

Good code design generally dictates that the non-portable parts should at least be wrapped and isolated.

Portability in C++ programming of AI applications involves correctly running on the underlying tech stack, including the operating system, CPU, and GPU capabilities.

Conceptually, in both cases, there are two levels:

1. Toleration. The first level of portability is "toleration" where the program must at least work correctly on whatever platform it finds itself.

2. Exploitation. The second level is "exploiting" the specific features of a particular tech stack, such as making the most of whatever CPU or GPU hardware is available.

This is generally true for any application, but especially true for AI engines. To get it running fast, you'll need a whole boatload of exploitation deep in your C++ backends. Hence, the basic approach to writing portable code is:

1. Write generic code portably, and

2. Write platform-specific code where needed.

Writing portable standard C++ code. Wherever your application doesn't need a GPU, your C++ code should be written in portable C++. The majority of the C++ programming language is well-standardized, and a lot of code can be written that simply compiles on all platforms in a way that it has consistent results. You just have to avoid the portability pitfalls.

Platform-specific coding. Most C++ programmers are familiar with using `#if` or `#ifdef` preprocessor directives to handle different platforms, and the various flavors of this are discussed further below. The newer C++ equivalent is "`if constexpr`" statements for compile-time processing. Small or sometimes large sections of C++ code will need to be written differently on each platform.

Likely major areas that will be non-portable include:

- Hardware acceleration (GPU interfaces)
- Intrinsic functions (CPU acceleration)
- FP16/BF16 floating-point types
- User interfaces (Windows vs Mac vs X Windows)
- Android vs iOS (not just the GUI)
- Multi-threading (Linux vs Windows threads)
- Text file differences (You've heard of `\r`, right?)
- File system issues (Directory hierarchies, permissions, etc.)
- "Endian" issues in integer representations.

Consider your code choices carefully. Some other areas where you can create portability pain for yourself include:

- Third-party libraries (i.e., if not widely used like STL or Boost).
- Newer C++ standard language features (e.g., C++23 features won't be widely supported yet).

Compilation Problems

If you want your C++ code to run on both Linux and Windows, you might need to get past the compiler errors first! C++ has been standardized for decades, or it seems like that. So, I feel like it should be easier to get C++ code to compile. And yet, I find myself sometimes spending an hour or two getting past a few darn compiler errors.

Most compilers have a treat-warnings-as-errors mode. Come on, I want the reverse.

Some of the main issues that will have a C++ program compile on one C++ compiler (e.g., MSVS) but not on another (e.g., GCC) include:

- `const` correctness
- Permissive versus conformant modes
- Pointer type casting

const correctness refer to the careful use of "`const`" to mark not just named constants, but also all unchanging read-only data types. If it's "`const`" then it cannot be changed; if it's non-`const`, then it's writable.

People have different levels of feelings about whether this is a good idea. There are the fastidious Vogon-relative rule-followers who want it, and the normal reasonable pragmatic people who don't. Can you see which side I'm on?

Anyway, to get non-`const`-correct code (i.e., mine) to compile on GCC or MSVS, you need to turn off the fussy modes. On MSVS, there's a "permissive" flag in "Conformance Mode" in Project Settings that you have to turn off.

Pointer type casting is another issue. C++ for AI has a lot of problems with pointer types, mainly because C++ standardizers back in the 1990s neglected to create a "`short float`" 16-bit floating-point type. Theoretically, you're not supposed to cast between different pointer types, like "`int*`" and "`char*`". And theoretically, you're supposed to use "`void*`" for generic addresses, rather than "`char*`" or "`unsigned char*`".

But, you know, this is AI, so them rules is made to be broken, and the C++ standardizer committees finally admitted as much when they created the various special types of casts about 20 years later (i.e., `reinterpret_cast`).

Anyway, the strategies for getting a non-compiling pointer cast to work include:

- Just casting it to whatever you want.
- Turning on permissive mode
- Casting it to `void*` and back again (i.e., "`x=*(int*)(void*)(char*)&c`")
- Using "`reinterpret_cast`" like a Goody Two-Shoes.

Runtime Portability Glitches

A bug that occurs on every platform is just that: a bug. A portability glitch is one with different behavior on different platforms. Some examples of the types that can occur:

- The code doesn't compile on a platform.
- The code has different results on different platforms.
- Sluggish processing on one platform.
- Crashes, hangs, or spins on one platform.

Some other types of weird problems that might indicate a portability glitch:

- Code runs fine in normal mode, but fails when the optimizer is enabled, or if the optimization level is increased.
- Code crashes in production, but runs just fine in the debugger (i.e., cannot reproduce it).
- Code intermittently fails (e.g., it could be a race condition or other timing issue.)

A lot of these types of symptoms are screaming "memory error!" And indeed, that's got to be top of the list. You might want to run your memory debugging tools again (e.g., Valgrind), even on a different platform to the one that's crashing.

However, it's not always memory or pointers. There are various insidious bugs that can cause weird behavior in the 0.001% cases where it's not a memory glitch:

- Uninitialized variables or object members.
- Numeric overflow or underflow (of integers or `float` type).
- Data size problems (e.g., 16-bit, 32-bit, or 64-bit).
- Undefined language features. Your code might be relying on something that isn't actually guaranteed in C++.

Data Type Sizes

There are a variety of portability issues with the sizes of basic data types in C++. Some of the problems include:

- Fundamental data type byte sizes (e.g., how many bytes is an "`int`").
- Pointer versus integer sizes (e.g., do `void` pointers fit inside an `int`?).

- `size_t` is usually `unsigned long`, not `unsigned int`.

Typical AI engines work with 32-bit floating-point (`float` type). Note that for 32-bit integers you cannot assume that `int` is 32 bits, but must define a specific type. Furthermore, if you assume that `short` is 16-bit, `int` is 32-bit, and `long` is 64-bit, well, you'd be incorrect. Most platforms have 64-bit `int` types, and the C++ standard only requires relative sizes, such as that `long` is at least as big as `int`.

Your startup portability check should check that sizes are what you want:

```
// Test basic numeric sizes
aussie_assert(sizeof(int) == 4);
aussie_assert(sizeof(float) == 4);
aussie_assert(sizeof(short) == 2);
```

Another more efficient way is the compile-time `static_assert` method:

```
static_assert(sizeof(int) == 4);
static_assert(sizeof(float) == 4);
static_assert(sizeof(short) == 2);
```

And you should also print them out in a report, or to a log file, for supportability reasons. Here's a useful way with a macro that uses the "#" stringize preprocessor operator and also the standard adjacent string concatenation feature of C++.

```
#define PRINT_TYPE_SIZE(type) \
    printf("Config: sizeof " #type \
           " = %d bytes (%d bits)\n", \
    (int)sizeof(type), 8*(int)sizeof(type));
```

You can print out whatever types you need:

```
PRINT_TYPE_SIZE(int);
PRINT_TYPE_SIZE(float);
PRINT_TYPE_SIZE(short);
```

Here's the output on my Windows laptop with MSVS:

```
Config: sizeof int = 4 bytes (32 bits)
Config: sizeof float = 4 bytes (32 bits)
Config: sizeof short = 2 bytes (16 bits)
```

Standard Library Types: Other data types to consider are the builtin ones in the standards. I'm looking at you, `size_t` and `time_t`, and a few others that belong on Santa's naughty list. People often assume that `size_t` is the same as "`unsigned int`" but it's actually usually "`unsigned long`". Here's a partial solution:

```
PRINT_TYPE_SIZE(size_t);
PRINT_TYPE_SIZE(clock_t);
PRINT_TYPE_SIZE(ptrdiff_t);
```

Data Representation Pitfalls

Portability of C++ to platforms also has data representation issues such as:

- Floating-point oddities (e.g., negative zero, `Inf`, and `NaN`).
- Whether "`char`" means "`signed char`" or "`unsigned char`"
- Endian-ness of integer byte storage (i.e., do you prefer "big endian" or "little endian"?).
- Whether zero bytes represent zero integers, zero floating-point, and null pointers.

Zero is not always zero? You probably assume that a 4-byte integer containing "0" has all four individual bytes equal to zero. It seems completely reasonable, and is correct on many platforms, but not all. There's a theoretical portability problem on a few obscure platforms. There are computers where integer zero or floating-point 0.0 is not four zero bytes. If you want to check, here's a few lines of code for your platform portability self-check code at startup:

```
int i2 = 0;
unsigned char* cptr2 = (unsigned char*)&i2;
for (int i = 0; i < sizeof(int); i++) {
    assert(cptr2[i] == 0);
}
```

Are null pointers all-bytes-zero, too? Here's the code to check `NULL` in a "`char*`" type:

```
// Test pointer NULL portability
char *ptr1 = NULL;
unsigned char* cptr3 = (unsigned char*)&ptr1;
for (int i = 0; i < sizeof(char*); i++) {
    assert(cptr3[i] == 0);
}
```

What about 0.0 in floating-point? You can test it explicitly with portability self-testing code:

```
// Test float zero portability
float f1 = 0.0f;
unsigned char* cptr4 = (unsigned char*)&f1;
for (int i = 0; i < sizeof(float); i++) {
    assert(cptr4[i] == 0);
}
```

It is important to include these tests in a portability self-test, because you're relying on this whenever you use `memset` or `calloc`.

Pointers versus Integer Sizes

You didn't hear this from me, but apparently you can store pointers in integers, and vice-versa, in C++ code. Weirdly, you can even get paid for doing this. But it only works if the byte sizes are big enough, and it's best to self-test this portability risk during program startup. What exactly you want to test depends on what you're (not) doing, but here's one example:

```
// Test LONGs can be stored in pointers
aussie_assert(sizeof(char*) >= sizeof(long));
aussie_assert(sizeof(void*) >= sizeof(long));
aussie_assert(sizeof(int*) >= sizeof(long));
// ... and more
```

Note that a better version in modern C++ would use "`static_assert`" to test these sizes at compile-time, with zero runtime cost.

```
static_assert(sizeof(char*) >= sizeof(long));
static_assert(sizeof(void*) >= sizeof(long));
static_assert(sizeof(int*) >= sizeof(long));
```

In this way, you can perfectly safely mix pointers and integers in a single variable. Just don't tell the SOC compliance officer.

References

1. Horton, Mark, *Portable C Software*, Prentice Hall, 1990, https://www.amazon.com/Portable-Software-Mark-R-Horton/dp/0138680507.
2. Jaeschke, Rex, *Portability and the C Language*, Hayden Books, 1989, https://www.amazon.com/Portability-Language-Hayden-Books-library/dp/0672484285.
3. Lapin, J. E., *Portable C and UNIX System Programming*, Prentice Hall, 1987, https://www.amazon.com/Portable-Systems-Programming-Prentice-hall-Processing/dp/0136864945.
4. Rabinowitz, Henry, and SCHAAP, Chaim, *Portable C*, Prentice Hall, 1990, https://www.amazon.com/Portable-C-Prentice-Hall-Software/dp/0136859674.
5. David Spuler, March 2024, *Generative AI in C++*, https://www.amazon.com/Generative-AI-Coding-Transformers-LLMs-ebook/dp/B0CXJKCWX9/.

19. Supportability

What is Supportability?

Supportability refers to making it easier to support your customers in the field. This means making it easier for your customers to solve their problems, and also making it easier for your phone support staff whenever customers call in.

Hey! I have an idea: how about you build an AI chatbot that knows how to debug your software? Umm, sorry, rush of blood to the head.

Some of the areas where the software's design can help both customers and support staff include:

- Easy method to print the program's basic configuration, version, and platform details (e.g., either an interactive method or logged to a file).
- Printing important platform stats (e.g., what CPU/GPU acceleration was found by the program, what is `sizeof int`, and so on).
- Self-check common issues. Don't just check for input file not found. You can also check if it was empty, blanks only, zero words, punctuation only, wrong character encoding, and so on.
- Verbose and meaningful error messages. Assume every error message will be seen by customers.
- Specific error messages. Lazy coders group two failures: "ERROR: File not found or empty." Which is it?
- Unique codes in error messages.
- Documenting your error messages in public help pages or by making your online support database world-public (gasp!).
- Retain copies of all shipped executables, with and without debug information, as part of your build and release process, so you can postmortem debug later.
- Have a procedure whereby customers can upload `core` files to support.
- Documentation for customers about how to run the support/debug features of the software, how to run self-diagnostics, or how to locate the tracing logs or other supportability features.
- Documenting post-mortem procedures, such as `gdb` on a `core` file, as customers aren't necessarily engineers.

- Not crashing in the first place. Fix this by writing perfect code, please.

Why use unique message codes? Adding unique numeric or symbolic codes in your error messages and even in assertions can improve supportability in two ways: self-help and phone support call-ins.

A unique code allows customers to find these error codes easily on the internet (i.e., via Google or Bing), either in your website's online help web pages, or on the third-party websites (e.g., Stack Overflow and the like), where other customers have had the same problem.

Note that the codes don't really *need* to be completely unique, so don't worry if two messages have the same code, unless you're doing internationalization! And certainly, don't agonize over enforcing a huge corporate policy for all teams to use different numbers or prefixes. However, it does help for your unique code to have a prefix indicating which software application it's coming from, because the AI tech stack has quite a lot of components in production, so maybe you need a policy after all (*sigh*).

Note that supportability is at the tail end of the user experience. It's less important than first impressions: the user interface, installation and the on-boarding experience.

Graceful Core Dumps

Okay, so it's a bit of an oxymoron to say that. But here's a way to at least print a useful support message if your code crashes. Register a signal handler for `SIGSEGV` (seg fault), `SIGILL` (illegal instructions), `SIGFPE` (floating point error), and any other fatal signals.

Here's an example of a shutdown signal handler:

```
void crash_gracefully(int sig)
{
    static bool s_already = false;
    if (s_already) {
        // Already shutting down
        // Probably a re-raised signal
        // Too dangerous here to do anything
        return;  // Just finish
    }
    s_already = true;  // Avoid recursive calls
    fprintf(stderr, "Hi Customer, alas we crashed!\n");
    fprintf(stderr, "Please call 1-800-DEVNULL to report.\n");
```

```
    fprintf(stderr, "Email core to devnull@<MYSITE>.com.\n");
    abort();  // Trigger the core dump
}
```

This is how you install a signal handler, usually at the start of program execution, such as the top of `main`.

```
#include <signal.h>
// ...
signal(SIGSEGV, crash_gracefully);
signal(SIGILL, crash_gracefully);
```

I've successfully used this approach on Unix and Linux platforms, but I'm not sure about on MacOS (which is like BSD Unix) or Windows (which isn't). Note that you must be very careful about re-raised signals here. Otherwise, this function is going to spin for the customer, rather than core dump.

There's not much you can do other than print a message and core dump. You can't try to recover from this for fatal signals. If you try to block a `SIGSEGV` signal and just return, hoping to keep going, it won't work. Instead, the `SIGSEGV` will get raised again by the CPU, and also spin.

You can test this code quite easily by writing bad code that crashes (intentionally, for a change). Feel free to try getting it to print out a stack trace with `std::backtrace` (C++23) or the other stack trace libraries from GNU or Boost. I'm betting against it, because it'll probably be too corrupted to see the stack in a signal handler, but it might work.

Random Number Seeds

Neural network code often uses random numbers to improve accuracy via a stochastic algorithm. For example, the top-k decoding uses randomness for creativity and to prevent the repetitive looping that can occur with greedy decoding. And you might use randomness to generate input tests when you're trying to thrash the model with random prompt strings.

But that's not good for debugging! We don't want randomness when we're trying to reproduce a bug!

Hence, we want it to be random for users, but not when we're debugging. Random numbers need a "seed" to get started, so we can just save and re-use the seed for a debugging session.

This idea can be applied to old-style `rand/srand` functions or to the newer `<random>` libraries like `std::mt19937` (stands for "Mersenne twister").

Seeding the random number generator in old-style C++ is done via the "`srand`" function. The longstanding way to initialize the random number generator, so it's truly random, is to use the current time:

```
srand(time(NULL));
```

Note that seeding with a guessable value is a security risk. Hence, it's safer to use some additional arithmetic on the `time` return value.

After seeding, the "`rand`" function can be used to get a truly unpredictable set of random numbers. The random number generator works well and is efficient. A generalized plan is to have a debugging or regression testing mode where the seed is fixed.

```
if (g_aussie_debug_srand_seed != 0) {
    // Debugging mode
    srand(g_aussie_debug_srand_seed); // Non-random!
}
else {  // Normal run
    srand(time(NULL));
}
```

The test harness (not shown here) would have to set the global debug variable "`g_aussie_debug_srand_seed`" whenever it's needed for a regression test. For example, either it's manually hard-coded into a testing function, or it could be set via a command-line argument to your test harness executable, so the program can be scripted to run with a known seed.

This is better, but if we have a bug in production, we won't know the seed number. So, the better code also prints out the seed number (or logs it) in case you need to use it later to reproduce a bug that occurred live.

```
if (g_aussie_debug_srand_seed != 0) {
    srand(g_aussie_debug_srand_seed); // Debug mode
}
else {  // Normal run
    long int iseed = (long)time(NULL);
    fprintf(stderr, "INFO: Random number seed: %ld 0x%lx\n",
            iseed, iseed);
```

```
    srand(iseed);
}
```

An extension would be to also print out the seed in error context information on assertion failures, self-test errors, or other internal errors.

There's one practical problem with this for reproducibility: what if the bug occurs after a thousand queries? If there's been innumerable calls to our random number generator, there's not really a way to reproduce the current situation.

One simple fix is to instantiate a new random number generator for every query, which really isn't very expensive.

Adding Portability to Supportability

The basic best practices are to write portable code until you can't. Here are some suggestions to further finesse your portability coding practices for improved supportability:

1. Self-test portability issues at startup.

2. Print out platform settings into logs.

A good idea is to self-test that certain portability settings meet the minimum requirements of your application. It's necessary to check for the exact feature you want. And you probably should do these feature self-tests even in the production versions that users run, not just in the debugging versions.

It's only a handful of lines of code that can save you a lot of headaches later.

Also, you should detect and print out the current portability settings as part of the program's output (or report), or at least to the logs.

Ideally, you would actually summarize these settings in the user's output display, which helps the poor phone jockeys trying to answers callers offering very useful problem summaries: "My AI doesn't work."

If it's not a PEBKAC, then having the ability to get these platform settings to put into the incident log is very helpful in resolving production-level support issues. This is especially true if you have users running your software on different user interfaces, and, honestly, if you don't support multiple user interfaces, then what are you doing here?

You should also output backend portability settings for API or other backend software products. The idea works the same even if your "users" are programmers who are running your code on different hardware platforms or virtual machines, except that their issue summaries will be like: "My kernel fission optimizations of batch normalization core dump from a `SIGILL` whenever I pass it a Mersenne prime."

20. Quality

What is Software Quality?

Quality is an overarching goal in software design. The terms "software quality" and "code quality" are not the same thing. Software quality is more about product quality from the user or company perspective, which has a more outward looking feel with issues such as functionality and usability. Code quality is what software developers work on every day. Having quality coding practices is a pre-requisite for software quality, so there's much overlap.How do we improve both types of "quality"?

First, let's acknowledge the subjectivity. Some groups of people are more focused on "software quality" than "code quality" as a goal. Salespeople want the product to have the hot features. Marketing wants a nice UI and a "positioning" in the market (what is so great about the letter P?). Support wants nobody to call.

For those working on software, everyone has a different view of code quality. Quality engineers want everything to be perfect before it ships. Project managers want to hit the date by time-boxing features. Developers want, well, who knows, because every developer has a different but deeply-held belief about this topic.

Second, let's examine the metrics for quality code. It's runtime things like: has cool features, doesn't crash or spin, and is performant. And it's static things like: readability, modularity, and so on. And there are future-looking metrics such as: maintainability, extensibility, etc. There are various techniques to enhance these types of metrics, which we examine in the following chapters.

Third, let's take a top-down look. What does "software quality" or "code quality" mean on the executive floor? Probably it means any software that has "AI" features, so the CEO can say that buzzword in the earnings call about a hundred times. I heard on TikTok that McKinsey research proved that stocks appreciate by *sqrt(pi/8)* percentage points for every mention.

Finally, let's take a bottom-up look, which is really most of this chapter and the following chapters. We are talking about C++ coding, after all. There are several practical techniques that can be used to improve the delivery of quality software through improvements to C++ code quality and other areas.

Advanced Software Quality

If you want to write the best C++ software for enterprise purposes in terms of "quality," you need to consider a lot of "abilities":

- Testability
- Debuggability
- Scalability
- Usability
- Installability
- Supportability
- Availability
- Reliability
- Maintainability
- Portability
- Extensibility
- Interoperability
- Reusability

Take a breath. Keep going. Some more:

- Deployability
- Manageability
- Readability
- Upgradability
- Marketability
- Monetizability
- Quality-ability (whatever that means!)
- Security protection (hackability)
- Internationalization (translatability)
- Fault tolerance and resilience (keep-going-ability)
- Modularity (separatability)
- Stability

Oh, and I almost forgot one coding quality issue:

- Adding new features that customers want.

Before we get too wrapped in all those inward-looking "abilities," let us remind ourselves that the customer only cares about a few of them: installability, usability, stability.

For a B2C product, think about the "grandma test"; could your grandma use this software? (After she's called you and made you set up her WiFi, I mean.) For B2B customers, the main thing the users actually care about is "ability-ability" which is whether your software has the *capability* to help users do whatever bizarre things businesses want to do with your code.

Sellability

Oops, I've forgotten about sales yet again, which isn't surprising because all of us in R&D aren't allowed to talk to the reps. I guess they have cooties or they'll stop selling the currently shipped version or they'll blame us for not winning a deal with the currently shipped version. We have drills to practice hiding under our chairs if we see a rep.

Anyway, to get back on topic, marketability and sellability is actually the highest level of quality. If nobody buys it, who cares how beautiful an architecture? Consider broadening the definition of "quality" beyond the C++ code to the "software quality" of the entire product from the perspective of the company.

Sellability is quality!

Most of the "code quality" practices in software engineering are internal inward-focused work, rather than looking "outwards" at the customer. If your company goal is actually financial success of your C++ software product in the B2B market, here's my suggestion of an alternative set of C++ "sellability" processes to consider:

1. Ask your sales reps what new features will close their current deal.

2. Code that in C++.

3. Run your 24-hour or 48-hour automated test suite.

4. Give the executables to your sales reps on a zippy.

Note that I only said to "consider" this method. Nobody in R&D is actually going to do it, I'm sure. I only wrote that so all the sales reps would buy a programming book.

Software Engineering Methodologies

Below is a list of various software engineering paradigms and architectural practices. Let me hereby emphatically state that one of these methods is clearly and by far the absolute best one, far superior to all the rest, and I will defend it to the hilt over a brew any day of the week.

Oh, but I'm not going to tell you which one. Feel free to argue amongst yourselves.

Here's the list:

- Agile development
- Pair programming
- AI copilot programming
- Waterfall method
- DevOps for everyone
- Test-driven development
- Feature-driven development
- Agile scrum
- Lean coding
- GMB
- Don't Repeat Yourself (DRY)
- Structured Design Methodology
- Designated Object Architecture (DOA)
- UML
- Rapid Application Development (RAD)
- eXtreme Programming
- Object Oriented Design (OOD)
- SQA
- Rogue coder model
- Pick Your Favorite Acronym (PYFA)
- Intentional coding
- Joint Application Development Process
- Move fast & break stuff
- Behavior-Driven Development
- SOLID
- Domain-Driven Design
- Product Market Fit (PMF)
- ISO something
- Fingers and toes crossed
- Spiral Model

- TQM or six-sigma or Jack Welch stuff
- Code myself a new minivan
- YAGNI
- Rational Unified Coding
- Product-Led Growth (PLG)

What a fun list! I'm going to make a poster to put on the wall above my "jump to conclusions" mat.

Software Engineering Process Group

The idea of a Software Engineering Process Group (SEPG) is a team of people in your company who aim to help software engineers write better code. It's people helping people, so what could be better than that?

What this SEPG team does is buy everyone in the company a copy of this book, including the valet parking attendants and catering staff, who are integral to your software development strategy, if you ask me (you didn't). After that, it's feet up on the desk and read the newspaper for the rest of the day on the SEPG floor, because it's all sorted in this book.

I really like the idea of the SEPG, but I've also seen it ineffective when product groups simply ignored their advice. I don't know what to say about that. I guess if I were running an SEPG, I'd say try to focus on pragmatic and incremental ways to improve software processes. Some of the ways that an SEPG can add tremendous value across an entire software development organization include:

- Educating engineers on best practices.
- Reviewing coding tools that might be useful.
- Vetting common libraries of low-level functionality (reusability!).
- Documenting and sharing successful methods and ideas.
- Coding up horizontal libraries like debug wrappers.

Oh, yeah, and a coding standards document, because who doesn't love a great one of those.

Coding Standards

I cannot pretend that I am a big fan of having coding style standards. But most large companies tend to have them, and there is certainly a benefit to doing so. You can find Google's on the Internet, and I read it to my toddler to put him to sleep (easier than putting him into a child seat and doing a hundred blockies at 3am while wearing pyjamas; who doesn't love parenting?).

The advantage of a coding policy is a standardization of various activities and processes company-wide, which is something they really like in head office. The disadvantages include things like: (a) a focus on "busy work" coding rather than adding new user features, and (b) practical difficulties merging two different development procedures if you acquire another big company. Newly acquired startups will expend a fair amount of effort to conform to your standards, but they probably need to do similar activities to fix technical debt, anyway.

My preference would rather be that a company has a specific organizational group focused on software engineering excellence, with a focus on practicality, rather than dictate the "one true way" of programming. Coding standards are only one of the many issues for such a cross-company team to address. This is the idea of having an SEPG in your organization, which is kind of like a SEP field, if you know what I mean. So, it is a matter of tone and focus in terms of how high or how low to go in devising the coding standard for your project or organization.

Some high-level issues that could be addressed:

- Which programming language. (C++, of course!)
- Code libraries allowed
- Tech stack: database, app layer, UI, etc.
- Tools: source code control, bug database, etc.
- Naming: e.g., good APIs follow a naming convention that the developer can guess.

A coding style for C++ could specify a variety of factors about which of the advanced language features to use (or avoid):

- Templates
- Operator overloading
- Class inheritance hierarchies
- Namespace management

I'm really not going to suggest your coding standard document should address indentation, variable names, comments, and so on, but some of these wonderful types of documents actually do.

There is also value in specifying standard suggested coding libraries and interfaces:

- Basic data types
- Basic coding libraries
- Basic data structures (e.g., hash tables, lookup tables, etc.)
- Unit testing library/APIs
- Regression testing tools and harnesses
- Assertions and self-testing
- Debug tracing code
- Exception handling
- Testing and debugging tools

I could go on, but I won't.

Project Estimation

Estimating project time and space requirements is an important part of software project management. Although estimating the efficiency of a proposed project is important in ascertaining its feasibility, it is difficult to find anything concrete to say about arriving at these estimates. Producing advance estimates is more of an art than a science, and a typical process goes like this:

1. Pick a random date.
2. Deny programmers sleep until this date.
3. Slip the date.
4. Time-box out all useful features.
5. Ship it!

Experience is probably the best source of methods for producing an accurate estimate. Hence, it is wise to seek out others who have implemented a similar project, or to perform a literature search for relevant papers and books. Unfortunately, neither of these methods is guaranteed to succeed and the implementor may be forced to go it alone. The only other realistic means of estimation relies on a good understanding of the various data structures and algorithms that will be used by the program. Making realistic assumptions about the input can provide some means of examining the performance of a data structure. How a data structure performs under worst case assumptions may also be of great importance.

An alternative to these methods of plucking estimates out of the air is to code up a prototype version of the program, which implements only the most important parts of the project (especially those which will have the biggest impact). The efficiency of the prototype can then be measured using the various techniques. Even if the prototype is inefficient, at least the problem has been identified early in the development cycle, when the investment in the project is relatively low.

Code Quality

Everyone has their own opinions on the best way to write software, so I'll choose to simply offer some possible options for you to discuss. Here is my list of pragmatic and useful ways to ensure code reliability as a professional developer:

- Lots of unit tests.
- Lots of assertions.
- Lots of bigger regression tests.
- Automated acceptance testing in CI/CD.
- Nightly builds that automatically re-run all the bigger tests that are too slow for CI/CD.
- Warning-free compilation (as a coding policy goal).
- Running Valgrind or other memory checkers in the nightly builds (Linux).
- Run big multi-platform tests in the nightly builds.
- Check return codes (as a coding policy).
- Validate incoming function parameters (as a coding policy).
- Use an error logger.
- Use a debug tracing library.
- Add some debug wrapper functions.

And here's an extra bonus one: have an occasional "testing day." Programmers are good at random testing of OPC, but they tend not to do it much.

Extensibility

Extensibility is allowing your customers to extend or customize your AI software. Although your first thought is going to be to run off and build an API or an SDK, there are a few things to consider first. The simpler ways to "extend" are:

- Just add more features.
- Add configuration settings.
- Add command-line options.
- Add minor personalization features.

Adding customer features. The basic problem that customers have is that they want to find a way to do something. If they're looking to extend your software, well, that means that some feature is lacking. If one customer finds this issue, other customers are probably silently suffering. So, rather than building an API, just listen to your customer, and add some more features to your code that will solve the issue, and other reasonably similar issues.

Configuration settings. Think about your AI's configuration settings from the point-of-view of extensibility. If you prefer, call them "declarative extensions." It's much easier for a customer to change a config option than to write a program using your SDK. Consider elevating and documenting some of the different ways that your application can be configured, to give your customers more capabilities. Yes, this does significantly increase the error handling code and QA testing cycle, so this is a careful consideration: which of your internal config options do you hide or publicize?

Personalization options. When you're deep in the guts of an AI application, you're thinking about really brain-intensive stuff like vectorizing your tokenizer. Your customer, however, just wants to put their company's name at the top of their AI-generated report. Hence, focus on adding some of the "smaller" functionality that seems trivial to engineers, but is what customers want. Maybe, like the wheel, the report could even have different colors?

And one final point about extensibility: your customers aren't programmers. They don't even know what the acronyms API and SDK stand for. Your customers need an API like a fish needs a bicycle.

Scalability

Almost this entire treatise is about scalability of your application. Getting a huge behemoth to run fast is the biggest challenge.

But the actual software code is not the only scalability concern. There's also the server on which your application resides, receiving and process requests, sending them on to the backend application, and collating returned results. This server is itself a piece of software, and it could be an off-the-shelf server, or you could write your own in C++ if you like.

User interfaces are another overlooked point in regard to scalability. Not only must the backend be fast, but the user interface layer must handle all of the requirements in a way that people can cope with.

The key point is this:

Humans don't scale.

What that means is that making your human user do anything is a hard problem. People cannot read reams of text fast, they cannot click on a thousand warning messages, and they do dumb things in the interface, like re-clicking the "Load" button a hundred times if it's taking too long. The fact that a human is part of the process flow means that you have to make sure that all of your steps are human-friendly. This is an often-underestimated aspect of scalability.

Reusability

In our commercial world it is frequently the cost of our own time that is the greatest. Using our own time efficiently can be more important than writing fast programs. Although improving programming productivity is not our main topic of this book, let us briefly consider a few methods here.

The basic method of reducing time spent programming is to build on the work of others. The use of libraries, including the wide variety of commercially available source code libraries, and the C++ standard library, is a good way to build on the work of others. There are a few concerns with using third-party libraries:

- Quality concerns — Is it bug-free? How well tested and supported?
- Security issues — Consider the source and its security protections.
- Legal licenses — Don't use "copyleft" or "share-alike" or "non-commercial" code.

Building on your own work is the other main method of productivity improvement. How often have you coded up a hash table? Have you ever written a sorting routine off the top of your head and then spent hours debugging it? You should perform tasks only once. This doesn't necessarily mean writing reusable code in its most general sense, but just having the source code available for the most common problems. Modifying code that has already been debugged is far more time-efficient than writing it from scratch. Organizations should seek to create building blocks of code that programmers can use, but you can also do so in your own personal career.

21. Reliability

Code Reliability

Code reliability means that the execution is predictable and produces the desired results. Sequential coding is hard enough, but parallelized code of any kind (e.g., CPU or GPU vectorization, multi-threaded, multi-GPU, etc.) multiplies this complexity by another order of magnitude. Hence, starting with the basics of high quality coding practices are ever more important for code reliability, such as:

- Unit tests
- Assertions
- Self-testing code
- Debug tracing methods
- Automated system tests
- Function argument validation
- Error detection (e.g., starting with checking error return codes)
- Exception handling (wrapping code in a full exception handling stack)
- Resilience and failure tolerance
- Regression testing
- Test automation
- Test coverage measurement

One useful method of catching program failures is making the program apply checks to itself. Assertions and other self-testing code have the major advantage that they will catch such errors early, rather than letting the program continue, and cause a failure much later.

All of these techniques involve a significant chunk of extra coding work. Theory says that full exception handling can be 80% of a finalized software product, so it's a four-fold amount of extra work!

Maybe that estimate is a little outdated, given improvements in modern tech stacks, but it still contains many grains of truth.

There are many programming tools to help improve code reliability throughout the development, testing and debugging cycle:

- Memory debugging tools (e.g., `valgrind`).
- Performance profiling tools (for "de-slugging").
- Synchronization debugging tools (e.g., race condition checkers).
- Memory usage tracking (i.e., memory leaks and allocated memory measurement).
- Interactive debugging tools (IDE and command-line).
- Static analysis tools ("linters") or turn on more compiler warnings.
- Bug tracking databases (for cussing at each other).

Refactoring versus Rewriting

Refactoring was something I was doing for years, but I called it "code cleanup." The seminal work on refactoring is Martin Fowler's book "*Refactoring*" from 1999. This was the first work to gain traction in popularizing and formalizing the ideas of cleaning up code into a disciplined approach.

Refactoring is a code maintenance task that you do mainly for code quality reasons, and it needs to be considered an overhead cost. True refactoring does not add any new functionality for customers, and marketing won't be happy if you do refactoring all day long. But refactoring is a powerful way to achieve consistency in code quality and adhere to principles such as DRY. In highly technical special cases such as writing an API, you'll need to refactor multiple times until the API is "good."

Rewriting is where you pick up the dusty server containing the old source code repo, walk over to the office window and toss it out. You watch it smash ten floors below, drive over to CompUSA to buy a new server, and then start tapping away with a big smile on your face.

The goals of refactoring and rewriting are quite different. Refactoring aims to:

- Make the existing code "better" (e.g., modularized, layered).
- Add unit testing and other formality.
- Retain all the old features and functionality.
- Not add any new functionality.

Rewriting projects tend to:

- Throw away all the existing code.
- Choose a new tech stack, new UI, new tools, etc.
- Not support backward compatibility.
- Add some new functionality.

Refactoring and rewriting are very close together, and there's a lot of middle ground between them. If you're fixing some old code by rewriting one of the main modules, is it refactoring or rewriting?

The reality is that rewriting versus refactoring is always an engineering choice, and it's a difficult one without a clear right or wrong answer. You can't try both to see which one works better, so there's never any proof either way.

Defensive Programming

Defensive programming is a mindset where you assume that everything will go wrong. The user input will be garbage. Anyone else's code will be broken. The operating system intrinsics will fail. And your poor helpless C++ application needs to keep chugging along.

Many of the high-level types of defensive coding are discussed elsewhere in this book. Good practices that attempt to prevent bugs include: assertions, self-testing code, unit tests, regression tests, check return codes, validate incoming parameter values, exception handling, error logging, debug tracing code, warning-free compilation, memory debugging tools, static analysis tools, document your code, and call your mother on Sunday.

Using Compiler Errors for Good, not Evil: One of the advanced tricks for defensive programming is to intentionally trigger compiler errors that prevent compilation. For example, you can enforce security coding policies:

```
#define tmpnam dont_use_tmpnam_please
```

Or if you are using debug wrappers for some low-level system functions, you can enforce that:

```
#define memset please_use_memset_wrapper
```

Politeness is always required. You don't want to be rude to your colleagues.

Defensive Coding Style Policies: You might want to consider some specific bug-prevention coding styles, for defensive programming, maintainability, and general reliability. Some examples might be:

- All variables must be initialized when declared. Don't want to see this anymore: `int x;`
- All `switch` statements need a `default` clause.
- Null the pointer after every `delete`. You can define a macro to help.
- Null the pointer after every `free`. If you use a debug wrapper for free, make it pass-by-reference and NULL the pointer's value insider the wrapper function.
- Null the file pointer after `fclose`. Also can be nulled by a wrapper function.
- Unreachable code should be marked as such with an assertion (a special type).
- Prefer `inline` functions to preprocessor macros.
- Define numeric constants using `const` rather than `#define`.
- Validate `enum` variables are in range. Add a dummy EOL item at the end of an enum list, which can be used as an upper-bound to range-check any `enum` has a valid value. Define a self-test macro to range-check the value.
- Use `[[nodiscard]]` attributes for functions. All of them.
- Start different enums at different numbers (e.g., token numbers start at 10,000 and some other IDs start at 200,000), so that they can't get mixed up, even if they end up in `int` variables. And you'll need a bottom and top value to range-check their validity. You have to remove the commas from these numbers, though!
- All allocated memory must be zeroed. This might be a policy for each coder, or it could be auto-handled by intercepting the `new` operator and `malloc/calloc` into debug wrappers, and only returning cleared memory.
- Constructors should use `memset` to zero their own memory. This seems like bad coding style in a way, but how many times have you forgotten to initialize a data member in a constructor?
- Zero means "not set" for every flag, `enum`, status code, etc. This is a policy supporting the "zero all memory" defensive idea.

Assume failures will happen: Plan ahead to make failures easier to detect and debug (supportability!), even when they happen in production code:

- Use extra messages in assertions, and make them brief but useful.
- If an assertion keeps failing in testing, or fails in production for users, change it to more detailed self-checking code that emits a more detailed error.
- Add unique code numbers to error messages to make identifying causes easier (supportability).
- Separately check different error occurrences. Don't use only one combined assertion: `assert(s && *s);`
- Review assertions for cases where lazy code jockeys have used them to check return codes (e.g., file not found).

Maintainability

My first Software Engineer job was maintenance of low-level systems management on a lumbering Ultrix box in C code, with hardly any comments. You'd think I hate code maintenance, right? No, I had the opposite reaction: it was the best job ever!

If you think you don't like code maintenance, consider this: Code maintenance is what you do every day. I mean, except for those rare days where you're starting a new project from scratch, you're either maintaining your own code or someone else's, or both. There are two main modes: you're either debugging issues or extending the product with new features, but in both cases it is at some level a maintenance activity.

So, how do you improve future maintainability of code? And how do you fix up old code that's landed on your desk, flapping around like a seagull, because your company acquired a small startup.

Let's consider your existing code. How would you make your code better so that a future new hire can be quickly productive? The answer is probably not that different to the general approach to improving reliability of your code. Things like unit tests, regression testing, exception handling, and so on will make it easier for a new hire. You can't stop that college intern from re-naming all the source code files or re-indenting the whole codebase, but at least you can help them to not break stuff.

One way to think about future maintainability is to take a step back and think of it as a "new hire induction" problem. After you've shown your new colleague the ping pong table in the lunch room and the restrooms, they need to know:

- Where is the code, and how do I check it out of the repo?
- How do I build it? Run it? Test it?
- Where's the bug database, requirements documents, or enhancements list?
- What are the big code libraries? Which directories?

After that, then you can get into the nitty-gritty of how the C++ is laid out. Where are the utility libraries that handle low-level things like files, memory allocation, strings, hash tables, and whatnot? Where do I add a new unit test? A new command-line argument or configuration property?

Maintenance safety nets: How do you make your actual C++ code resilient to the onslaught of a new hire programmer? Assume that future changes to the code will often introduce bugs, and try to plan ahead to catch them using various coding tricks. Actually, the big things in preventing future bugs are the large code reliability techniques (e.g., unit tests, assertions, comment your code, blah blah blah). There are a lot of little things you can do, which are really quite marginal compared to the big things, but are much more fun, so here's my list:

- All variables should be initialized, even if it'll be immediately overwritten (i.e., "`int x=3;`" never just "`int x;`"). The temptation to *not* initialize is mainly from variables that are only declared so as to be passed into some other function to be set as a reference parameter. And yes, in this case, it's an intentional micro-inefficiency to protect against a future macro-crashability.
- Unreachable code should be marked with at least a comment or preferably an attribute or assertion (e.g., use the "`assert_not_reached`" assertion idea).
- Prefer statement blocks with curly braces to single-statements in any `if`, `else`, or loop body. Also for `case` and `default`. Use braces even if all fits on one line. Otherwise, some newbie will add a second statement, guaranteed.
- Once-only initialization code that isn't in a constructor should also be protected (e.g., the "`assert_once`" idea).
- All `switch` statements need a `default` (even if it just triggers an assertion).
- Don't use `case` fallthrough, except it's allowed for Duff's Device and any other really cool code abuses. Tag it with `[[fallthrough]]` if you must use it.

- Avoid preprocessor macros. Prefer `inline` functions rather than using function-like preprocessor macro tricks, and do named constants using `const` or `enum` names rather than `#define`. I've only used macros in this book for educational purposes, and you shouldn't even be looking at my dubious coding style.
- Declare a dummy `enum` value at the end of every long `enum` list (e.g., "`MyEnum_EOL_Dummy`"), and use this EOL name in any range-checking of values of `enum` variables. Otherwise, it breaks when someone adds a new `enum` at the end. EOL means "end-of-list" if you were wondering.
- Add some range-checking of your `enum` variables, because you forgot about that. Otherwise array indices and `enum` variables tend to get mixed up when you have a lot of `int` variables.
- Assert the exact numeric values of a few random `enum` symbols, and put cuss words in the optional message, telling newbie programmers that they shouldn't add a new `enum` at the top of the list.
- `sizeof(varname)` is better than `sizeof(int)` when someone changes it to `long` type. Similarly, it can be significantly safer to use `sizeof(arr[0])` and `sizeof(*ptr)`. No, the `*` operator isn't live in `sizeof`.
- All classes should have the "big four" (constructor, destructor, copy constructor, and assignment operator), even if they're silly, like when the destructor is just `{}`.
- If your class should not ever be bitwise-copied, then declare a dummy copy constructor and assignment operator (i.e., as "`private`" and without a function body), so the compiler prevents a newbie from accidentally doing something that would be an object bitwise copy.
- If your code needs a mathematical constant, like the reciprocal of the square root of pi, just work it out on your calculator and type the number in directly. Job security.
- A `switch` over an `enum` should usually have the default clause as an error or assertion. This detects the code maintenance situation where a newly added `enum` code isn't being handled.
- Avoid long `if-else-if` sequences. They get confusing. They also break completely if someone adds a new "`if`" section in the middle, but forgets it should be "`else if`" instead.
- Instigate a rule that whoever breaks the build has to bring kolaches tomorrow.

But don't sweat it. New hires will break your code, and then just comment out the unit test that fails.

Maintaining OPC. What about brand-new code? It's from that startup that got acquired, and it's really just a hacked-up prototype that should never have shipped. Now it's landed on your desk with a big red bow wrapped around it and a nice note from your boss telling you how much it'll be appreciated if you could have a little look at this. At least it's a challenge, and maybe you could even learn a little Italian, because that's the language the comments are written in.

So, refactoring has to be top of the list. You need to move code around so that it is modular, easier to unit test, and so on. Split out smaller functions and group all the low-level factory type routines. Writing some internal documentation about new code doesn't hurt either! And "canale" means "channel" in Italian so now you're bilingual.

Technical Debt

When programmers talk in disparaging tones about "technical debt" in code, what they often mean is that the code wasn't written "properly." A prototype got shipped long ago, and was never designed well, or in fact, was never designed *at all*. Some other giveaways of high technical debt are basically:

- Unit tests? That's someone else's job.
- Documentation? Never heard of it. Oh, you meant code comments? We don't use those.
- File Explorer is a source code control system.
- And a backup tool.
- Bug tracking tool? Do you mean the whiteboard?
- Requirements documentation. Also the whiteboard.
- Test plan? Eating free bananas while I test it.

Or to summarize all these points into one:

- You work at an AI startup.

Debt-Free Code: The good news is that there is a popular software development paradigm that has zero technical debt. It's called Properly-Written Code (PWC) and programmers are always talking about it in hushed or strident tones. Personally, I've been watching for years, but haven't yet been fortunate enough to actually see any, but apparently it exists somewhere out in the wild, kind of like the Loch Ness Monster, but with semicolons.

Exactly what properly-written code means is rather vague, but the suggested solution is usually a refactor or a rewrite. Personally, I favor refactoring, because I think that technical debt gets *increased* by rewrites, because the brand-new code:

a) Lacks unit tests.

b) Lacks internal documentation.

c) Hasn't been "tested by fire" in real customer usage.

d) Hasn't been tested by anyone, for that matter.

e) Is a "version 1.0" no matter how you try to spin it.

So, here's my probably-unpopular list of suggestions for reducing technical debt without rewriting anything:

- Comment your code!
- Fix compiler warnings to get warning-free compilation.
- Add more assertions and self-checking code.
- Check return codes from system functions (e.g., file operations).
- Add parameter validation checks to your functions.
- Add debug wrapper functions for selected system calls.
- Add automated tests (unit tests or regression tests).
- Port the platform-independent code modules to another platform. Even only to get compiler warnings and run tests.
- Add performance instrumentation (i.e., time).
- Add memory usage instrumentation (i.e., space).
- Add file usage instrumentation.
- Document the architecture, APIs, classes, data formats, or interfaces. With words.
- Add unique codes to error messages (for supportability).
- Document your DevOps procedures.
- Run nightly builds, and with tests running, too.
- Do a backup once in a while.

And if you're at a startup or a new project, get your tools sorted out for professional software development workflows:

- Compilers and IDEs
- Memory error detection
- Source code control (e.g., `svn` or `git` or `cvs`)
- CI/CD/CT build system
- Bug tracking system
- Internal documentation tools
- User support database

What really makes better code? Well, that's a rather big question about the entirety of software development practices, so I'll offer only one final suggestion: *humans*. My overarching view is that the quality of code is most impacted by the ability and motivation of the programmers, rather than by new tools or a trendy programming language (or even an AI coding copilot). A small team that is "on fire" can outpace a hundred coders sitting in meetings talking about the right way to do agile development processes. Hence, morale of the team is important, too.