The C++ In-Depth Series

Bjarne Stroustrup, Editor

"I have made this letter longer than usual, because I lack the time to make it short."

BLAISE PASCAL

The advent of the ISO/ANSI C++ standard marked the beginning of a new era for C++ programmers. The standard offers many new facilities and opportunities, but how can a real-world programmer find the time to discover the key nuggets of wisdom within this mass of information? **The C++ In-Depth Series** minimizes learning time and confusion by giving programmers concise, focused guides to specific topics.

Each book in this series presents a single topic, at a technical level appropriate to that topic. The Series' practical approach is designed to lift professionals to their next level of programming skills. Written by experts in the field, these short, in-depth monographs can be read and referenced without the distraction of unrelated material. The books are cross-referenced within the Series, and also reference *The C++ Programming Language* by Bjarne Stroustrup.

As you develop your skills in C++, it becomes increasingly important to separate essential information from hype and glitz, and to find the in-depth content you need in order to grow. The C++ In-Depth Series provides the tools, concepts, techniques, and new approaches to C++ that will give you a critical edge.

Titles in the Series

Preface

Get into a rut early: Do the same process the same way. Accumulate idioms. ***Standardize.*** *The only difference(!) between Shakespeare and you was the size of his idiom listnot the size of his vocabulary.*

Alan Perlis [emphasis ours]

The best thing aboutstandards is that there are so many to choose from.

Variously attributed

We want to provide this book as a basis for your team's coding standards for two principal reasons:

- *A coding standard should reflect the community's best tried-and-true experience:* It should contain proven idioms based on experience and solid understanding of the language. In particular, a coding standard should be based firmly on the extensive and rich software development literature, bringing together rules, guidelines, and best pr actices that would otherwise be left scattered throughout many sources.
- *Nature abhors a vacuum:* If you don't consciously set out reasonable rules, usually someone else will try to push their own set of pet rules instead. A coding standard made that way usually has all of the least desirable properties of a coding standard; for example, many such standards try to enforce a minimalistic C-style use of C++.

Many bad coding standards have been set by people who don't understand the language well, don't understand software development well, or try to legislate too much. A bad coding standard quickly loses credibility and at best even its valid guidelines are liable to be ignored by disenchanted programmers who dislike or disagree with its poorer guidelines. That's "at best"at worst, a bad standard might actually be enforced.

How to Use This Book

Think. Do follow good guidelines conscientiously; but don't follow them blindly. In this book's Items, note the Exceptions clarifying the less common situations where the guidance may not apply. No set of guidelines, however good (and we think these ones are), should try to be a substitute for thinking.

Each development team is responsible for setting its own standards, and for setting them responsibly. That includes your team. If you are a team lead, involve your team members in setting the team's standards; people are more likely to follow standards they view as their own than they are to follow a bunch of rules they feel are being thrust upon them.

This book is designed to be used as a basis for, and to be included by reference in, your team's coding standards. It is not intended to be the Last Word in coding standards, because your team will have additional guidelines appropriate to your particular group or task, and you should feel free to add those to these Items. But we hope that this book will save you some of the work of (re)developing your own, by documenting and referencing widely-accepted and authoritative practices that apply nearly universally (with Exceptions as noted), and so help increase the quality and consistency of the coding standards you use.

Have your team read these guidelines with their rationales (i.e., the whole book, and selected Items' References to other books and papers as needed), and decide if there are any that your team simply can't live with (e.g., because of some situation unique to your project). Then commit to the rest. Once adopted, the team's coding standards should not be violated except after consulting with the whole team.

Finally, periodically review your guidelines as a team to include practical experience and feedback from real use.

Coding Standards and You

Good coding standards can offer many interrelated advantages:

- *Improved code quality:* Encouraging developers to do the right things in a consistent way directly works to improve software quality and maintainability.
- *Improved development speed:* Developers don't need to always make decisions starting from first principles.
- *Better teamwork:* They help reduce needless debates on inconsequential issues and make it easier for teammates to read and maintain each other's code.
- *Uniformity in the right dimension:* This frees developers to be creative in directions that matter.

Under stress and time pressure, people do what they've been trained to do. They fall back on habit. That's why ER units in hospitals employ experienced, trained personnel; even knowledgeable beginners would panic.

As software developers, we routinely face enormous pressure to deliver tomorrow's software yesterday. Under schedule pressure, we do what we are trained to do and are used to doing. Sloppy programmers who in normal times don't know good practices of software engineering (or aren't used to applying them) will write even sloppier and buggier code when pressure is on. Conversely, programmers who form good habits and practice them regularly will keep themselves organized and deliver quality code, fast.

The coding standards introduced by this book are a collection of guidelines for writing high-quality C++ code. They are the distilled conclusions of a rich collective experience of the C++ community. Much of this body of knowledge has only been available in bits and pieces spread throughout books, or as word-of-mouth wisdom. This book's intent is to collect that knowledge into a collection of rules that is terse, justified, and easy to understand and follow.

Of course, one can write bad code even with the best coding standards. The same is true of any language, process, or methodology. A good set of coding standards fosters good habits and discipline that transcend mere rules. That foundation, once acquired, opens the door to higher levels. There's no shortcut; you have to develop vocabulary and grammar before writing poetry. We just hope to make that easier.

We address this book to C++ programmers of all levels:

If you are an apprentice programmer, we hope you will find the rules and their rationale helpful in understanding what styles and idioms C++ supports most naturally. We provide a concise rationale and discussion for each rule and guideline to encourage you to rely on understanding, not just rote memorization.

For the intermediate or advanced programmer, we have worked hard to provide a detailed list of precise references for each rule. This way, you can do further research into the rule's roots in C++'s type system, grammar, and object model.

At any rate, it is very likely that you work in a team on a complex project. Here is where coding standards really pay offyou can use them to bring the team to a common level and provide a basis for code reviews.

About This Book

We have set out the following design goals for this book:

- *Short is better than long:* Huge coding standards tend to be ignored; short ones get read and used. Long Items tend to be skimmed; short ones get read and used.
- *Each Item must be noncontroversial:* This book exists to document widely agreed-upon standards, not to invent them. If a guideline is not appropriate in all

Acknowledgments

Many thanks to series editor Bjarne Stroustrup, to editors Peter Gordon and Debbie Lafferty, and to Tyrrell Albaugh, Kim Boedigheimer, John Fuller, Bernard Gaffney, Curt Johnson, Chanda Leary-Coutu, Charles Leddy, Heather Mullane, Chuti Prasertsith, Lara Wysong, and the rest of the Addison-Wesley team for their assistance and persistence during this project. They are a real pleasure to work with.

Inspiration for some of the "sound bites" came from many sources, including the playful style of [Cline99], the classic **import this** of [Peters99], and the legendary and eminently quotable Alan Perlis.

We especially want to thank the people whose technical feedback has helped to make many parts of this book better than they would otherwise have been. Series editor Bjarne Stroustrup's incisive comments from concept all the way through to the final draft were heavily influential and led to many improvements. We want to give special thanks to Dave Abrahams, Marshall Cline, Kevlin Henney, Howard Hinnant, Jim Hyslop, Nicolai Josuttis, Jon Kalb, Max Khesin, Stan Lippman, Scott Meyers, and Daveed Vandevoorde for their active participation in review cycles and detailed comments on several drafts of this material. Other valuable comments and feedback were contributed by Chuck Allison, Samir Bajaj, Marc Barbour, Damian Dechev, Steve Dewhurst, Peter Dimov, Alan Griffiths, Michi Henning, James Kanze, Matt Marcus, Petru Marginean, Robert C. "Uncle Bob" Martin, Jeff Peil, Peter Pirkelbauer, Vladimir Prus, Dan Saks, Luke Wagner, Matthew Wilson, and Leor Zolman.

As usual, the remaining errors, omissions, and shameless puns are ours, not theirs.

Herb?utter
Andrei?lexandrescu

Seattle,?eptember?004

Organizational and Policy Issues

If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.

Gerald Weinberg

In the grand tradition of C and C++, we count the zero-based way. The prime directive, Item 0, covers what we feel is the most basic advice about coding standards.

The rest of this introductory section goes on to target a small number of carefully selected basic issues that are mostly not directly about the code itself, but on essential tools and techniques for writing solid code.

Our vote for the most valuable Item in this section goes to Item 0: Don't sweat the small stuff. (Or: Know what not to standardize.)

0. Don't sweat the small stuff. (Or: Know what not to standardize.)

Summary

Say only what needs saying: Don't enforce personal tastes or obsolete practices.

Discussion

Issues that are really just personal taste and don't affect correctness or readability don't belong in a coding standard. Any professional programmer can easily read and write code that is formatted a little differently than they're used to.

Do use consistent formatting within each source file or even each project, because it's jarring to jump around among several styles in the same piece of code. But don't try to enforce consistent formatting across multiple projects or across a company.

Here are several common issues where the important thing is not to set a rule but just to be consistent with the style already in use within the file you're maintaining:

- *Don't specify how much to indent, but do indent to show structure:* Use any number of spaces you like to indent, but be consistent within at least each file.
- *Don't enforce a specific line length, but do keep line lengths readable:* Use any length of line you like, but don't be excessive. Studies show that up to ten-word text widths are optimal for eye tracking.
- *Don't overlegislate naming, but do use a consistent naming convention:* There are only two must-dos: a) never use "underhanded names," ones that begin with an underscore or that contain a double underscore; and b) always use `ONLY_UPPERCASE_NAMES` for macros and never think about writing a macro that is a common word or abbreviation (including common template parameters, such as `T` and `U` ; writing `#define T` ***`anything`*** is extremely disruptive). Otherwise, do use consistent and meaningful names and follow a file's or module's convention. (If you can't decide on your own naming convention, try this one: Name classes, functions, and `enum`s `LikeThis`; name variables `likeThis`; name private member variables `likeThis_`; and name macros `LIKE_THIS`.)
- *Don't prescribe commenting styles (except where tools extract certain styles into documentation), but do write useful comments:* Write code instead of comments where possible (e.g., see Item 16). Don't write comments that repeat the code; they get out of sync. Do write illuminating comments that explain approach and rationale.

Finally, don't try to enforce antiquated rules (see Examples 3 and 4) even if they once appeared in older coding standards.

Examples

Example 1: Brace placement. There is no readability difference among:

```
void using_k_and_r_style() {
```

References

[BoostLRG] ?[Brooks95

1. Compile cleanly at high warning levels

Summary

Take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds. Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.

Discussion

Your compiler is your friend. If it issues a warning for a certain construct, often there's a potential problem in your code.

Successful builds should be silent (warning-free). If they aren't, you'll quickly get into the habit of skimming the output, and you *will* miss real problems. (See Item 2.)

To get rid of a warning: a) understand it; and then b) rephrase your code to eliminate the warning and make it clearer to both humans and compilers that the code does what you intended.

Do this even when the program seemed to run correctly in the first place. Do this even when you are positive that the warning is benign. Even benign warnings can obscure later warnings pointing to real dangers.

Examples

Example 1: A third-party header file. A library header file that you cannot change could contain a construct that causes (probably benign) warnings. Then wrap the file with your own version that **`#include`**s the original header and selectively turns off the noisy warnings for that scope only, and then **`#include`** your wrapper throughout the rest of your project. Example (note that the warning control syntax will vary from compiler to compiler):

```
// File: myproj/my_lambda.h -- wraps Boost's lambda.hpp
//  Always include this file; don't use lambda.hpp directly.
//  NOTE: Our build now automatically checks "grep lambda.hpp <srcfile>".
// Boost.Lambda produces noisy compiler warnings that we know are innocuous.
// When they fix it we'll remove the pragmas below, but this header will
still exist.
//
#pragma warning(push)     // disable for this header only
 #pragma warning(disable:4512)
 #pragma warning(disable:4180)
 #include <boost/lambda/lambda.hpp>
#pragma warning(pop)      // restore original warning level
```

Example 2: "Unused function parameter." Check to make sure you really didn't mean to use the function parameter (e.g., it might be a placeholder for future expansion, or a required part of a standardized signature that your code has no use for). If it's not needed, simply delete the name of a function parameter:

Team LiB PREVIOUS NEXT

Exceptions

Sometimes, a compiler may emit a tedious or even spurious warning (i.e., one that is mere noise) but offer no way to turn it off, and it might be infeasible or unproductive busywork to rephrase the code to silence the warning. In these rare cases, as a team decision, avoid tediously working around a warning that is merely tedious: Disable that specific warning only, disable it as locally as possible, and write a clear comment documenting why it was necessary.

Team LiB PREVIOUS NEXT

References

[Meyers97

2. Use an automated build system

Summary

Push the (singular) button: Use a fully automatic ("one-action") build system that builds the whole project without user intervention.

Discussion

A one-action build process is essential. It must produce a dependable and repeatable translation of your source files into a deliverable package. There is a broad range of automated build tools available, and no excuse not to use one. Pick one. Use it.

We've seen organizations that neglect the "one-action" requirement. Some consider that a few mouse clicks here and there, running some utilities to register COM/CORBA servers, and copying some files by hand constitute a reasonable build process. But you don't have time and energy to waste on something a machine can do faster and better. You need a one-action build that is automated and dependable.

Successful builds should be silent, warning-free (see Item 1). The ideal build produces no noise and only one log message: "Build succeeded."

Have two build modes: Incremental and full. An incremental build rebuilds only what has changed since the last incremental or full build. Corollary: The second of two successive incremental builds should not write any output files; if it does, you probably have a dependency cycle (see Item 22), or your build system performs unnecessary operations (e.g., writes spurious temporary files just to discard them).

A project can have different forms of full build. Consider parameterizing your build by a number of essential features; likely candidates are target architecture, debug vs. release, and breadth (essential files vs. all files vs. full installer). One build setting can create the product's essential executables and libraries, another might also create ancillary files, and a full-fledged build might create an installer that comprises all your files, third-party redistributables, and installation code.

As projects grow over time, so does the cost of not having an automated build. If you don't use one from the start, you will waste time and resources. Worse still, by the time the need for an automated build becomes overwhelming, you will be under more pressure than at the start of the project.

Large projects might have a "build master" whose job is to care for the build system.

References

[Brooks95

3. Use a version control system

Summary

The palest of ink is better than the best memory (Chinese proverb): Use a version control system (VCS). Never keep files checked out for long periods. Check in frequently after your updated unit tests pass. Ensure that checked-in code does not break the build.

Discussion

Nearly all nontrivial projects need more than one developer and/or take more than a week of work. On such projects, you *will* need to compare historical versions of the same file to determine when (and/or by whom) changes were introduced. You *will* need to control and manage source changes.

When there are multiple developers, those developers will make changes in parallel, possibly to different parts of the same file at the same time. You need tools to automate checkout/versioning of file and, in some cases, merging of concurrent edits. A VCS automates and controls checkouts, versioning, and merging. A VCS will do it faster and more correctly than you could do it by hand. And you don't have time to fiddle with administriviayou have software to write.

Even a single developer has "oops!" and "huh?" moments, and needs to figure out when and why a bug or change was introduced. So will you. A VCS automatically tracks the history of each file and lets you "turn the clock back." The question isn't whether you will want to consult the history, but when.

Don't break the build. The code in the VCS must always build successfully.

The broad range of VCS offerings leaves no excuse not to use one. The least expensive and most popular is **cvs** (see References). It is a flexible tool, featuring TCP/IP access, optional enhanced security (by using the secure shell **ssh** protocol as a back-end), excellent administration through scripting, and even a graphical interface. Many other VCS products either treat **cvs** as a standard to emulate, or build new functionality on top of it.

Exceptions

A project with one programmer that takes about a week from start to finish probably can live without a VCS.

References

[BetterSCM] ?[Brooks95

4. Invest in code reviews

Summary

Re-view code: More eyes will help make more quality. Show your code, and read others'. You'll all learn and benefit.

Discussion

A good code review process benefits your team in many ways. It can:

- Increase code quality through beneficial peer pressure.
- Find bugs, non-portable code (if applicable), and potential scaling problems.
- Foster better design and implementation through cross-breeding of ideas.
- Bring newer teammates and beginners up to speed.
- Develop common values and a sense of community inside the team.
- Increase meritocracy, confidence, motivation, and professional pride.

Many shops neither reward quality code and quality teams nor invest time and money encouraging them. We hope we won't have to eat our words a couple of years from now, but we feel that the tide is slowly changing, due in part to an increased need for safe and secure software. Code reviews help foster exactly that, in addition to being an excellent (and free!) method of in-house training.

Even if your employer doesn't yet support a code reviewing process, do increase management awareness (hint: to start, show them this book) and do your best to make time and conduct reviews anyway. It is time well spent.

Make code reviews a routine part of your software development cycle. If you agree with your teammates on a reward system based on incentives (and perhaps disincentives), so much the better.

Without getting too formalistic, it's best to get code reviews in writinga simple e-mail can suffice. This makes it easier to track your own progress and avoid duplication.

When reviewing someone else's code, you might like to keep a checklist nearby for reference. We humbly suggest that one good list might be the table of contents of the book you are now reading. Enjoy!

In summary: We know we're preaching to the choir, but it had to be said. Your ego may hate a code review, but the little genius programmer inside of you loves it because it gets results and leads to better code and stronger applications.

References

[Constantine95

PREVIOUS NEXT

Design Style

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

Alan Perlis

But I also knew, and forgot, Hoare's dictum that premature optimization is the root of all evil in programming.

Donald Knuth, *The Errors of TeX* [Knuth89]

It's difficult to fully separate Design Style and Coding Style. We have tried to leave to the next section those Items that generally crop up when actually writing code.

This section focuses on principles and practices that apply more broadly than just to a particular class or function. A classic case in point is the balance among simplicity and clarity (Item 6), avoiding premature optimization (Item 8), and avoiding premature pessimization (Item 9). Those three Items apply, not just at the function-coding level, but to the larger areas of class and module design tradeoffs and to far-reaching application architecture decisions. (They also apply to all programmers. If you think otherwise, please reread the above Knuth quote and note its citation.)

Following that, many of the other Items in this and the following section deal with aspects of dependency managementa cornerstone of software engineering and a recurring theme throughout the book. Stop and think of some random good software engineering technique*any* good technique. Whichever one you picked, in one way or another it will be about reducing dependencies. Inheritance? Make code written to use the base class less dependent on the actual derived class. Minimize global variables? Reduce long-distance dependencies through widely visible data. Abstraction? Eliminate dependencies between code that manipulates concepts and code that implements them. Information hiding? Make client code less dependent on an entity's implementation details. An appropriate concern for dependency management is reflected in avoiding shared state (Item 10), applying information hiding (Item 11), and much more.

Our vote for the most valuable Item in this section goes to Item 6: Correctness, simplicity, and clarity come first. That they really, really must.

PREVIOUS NEXT

5. Give one entity one cohesive responsibility

Summary

Focus on one thing at a time: Prefer to give each entity (variable, class, function, namespace, module, library) one well-defined responsibility. As an entity grows, its scope of responsibility naturally increases, but its responsibility should not diverge.

Discussion

A good business idea, they say, can be explained in one sentence. Similarly, each program entity should have one clear purpose.

An entity with several disparate purposes is generally disproportionately harder to use, because it carries more than the sum of the intellectual overhead, complexity, and bugs of its parts. Such an entity is larger (often without good reason) and harder to use and reuse. Also, such an entity often offers crippled interfaces for any of its specific purposes because the partial overlap among various areas of functionality blurs the vision needed for crisply implementing each.

Entities with disparate responsibilities are typically hard to design and implement. "Multiple responsibilities" frequently implies "multiple personalities"a combinatorial number of possible behaviors and states. Prefer brief single-purpose functions (see also Item 39), small single-purpose classes, and cohesive modules with clean boundaries.

Prefer to build higher-level abstractions from smaller lower-level abstractions. Avoid collecting several low-level abstractions into a larger low-level conglomerate. Implementing a complex behavior out of several simple ones is easier than the reverse.

Examples

Example 1: ***`realloc`***. In Standard C, **`realloc`** is an infamous example of bad design. It has to do too many things: allocate memory if passed **`NULL`**, free it if passed a zero size, reallocate it in place if it can, or move memory around if it cannot. It is not easily extensible. It is widely viewed as a shortsighted design failure.

Example 2: ***`basic_string`***. In Standard C++, **`std::basic_string`** is an equally infamous example of monolithic class design. Too many "nice-to-have" features were added to a bloated class that tries to be a container but isn't quite, is undecided on iteration vs. indexing, and gratuitously duplicates many standard algorithms while leaving little space for extensibility. (See Item 44's Example.)

References

[Henney02a] ?[Henney02b] ?[McConnell93

6. Correctness, simplicity, and clarity come first

Summary

KISS (Keep It Simple Software): Correct is better than fast. Simple is better than complex. Clear is better than cute. Safe is better than insecure (see Items 83 and 99).

Discussion

It's hard to overstate the value of simple designs and clear code. Your code's maintainer will thank you for making it understandableand often that will be your future self, trying to remember what you were thinking six months ago. Hence such classic wisdom as:

Programs must be written for people to read, and only incidentally for machines to execute.

Harold Abelson and Gerald Jay Sussman

Write programs for people first, computers second.

Steve McConnell

The cheapest, fastest and most reliable components of a computer system are those that aren't there.

Gordon Bell

Those missing components are also the most accurate (they never make mistakes), the most secure (they can't be broken into), and the easiest to design, document, test and maintain. The importance of a simple design can't be overemphasized.

Jon Bentley

Many of the Items in this book naturally lead to designs and code that are easy to change, and clarity is the most desirable quality of easy-to-maintain, easy-to-refactor programs. What you can't comprehend, you can't change with confidence.

Probably the most common tension in this area is between code clarity and code optimization (see Items 7, 8, and 9). Whennot ifyou face the temptation to optimize prematurely for performance and thereby pessimize clarity, recall Item 8's point: It is far, far easier to make a correct program fast than it is to make a fast program correct.

Avoid the language's "dusty corners." Use the simplest techniques that are effective.

Examples

Example 1: Avoid gratuitous/clever operator overloading. One needlessly weird GUI library had users write `w + c;` to add a child control `c` to a widget `w`. (See Item 26.)

Example 2: Prefer using named variables, not temporaries, as constructor parameters. This avoids possible declaration ambiguities. It also often makes the purpose of your code clearer and thus is easier to maintain. It's also often safer (see Items 13 and 31).

References

[Abelson96] ?[Bentley00

7. Know when and how to code for scalability

Summary

Beware of explosive data growth: Without optimizing prematurely, keep an eye on asymptotic complexity. Algorithms that work on user data should take a predictable, and preferably no worse than linear, time with the amount of data processed. When optimization is provably necessary and important, and especially if it's because data volumes are growing, focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.

Discussion

This Item illustrates one significant balance point between Items 8 and 9, "don't optimize prematurely" and "don't pessimize prematurely." That makes this a tough Item to write, lest it be misconstrued as "premature optimization." It is not that.

Here's the background and motivation: Memory and disk capacity continue to grow exponentially; for example, from 1988 to 2004 disk capacity grew by about 112% per year (nearly 1,900-fold growth per decade), whereas even Moore's Law is just 59% per year (100-fold per decade). One clear consequence is that whatever your code does today it may be asked to do tomorrow against more data*much* more data. A bad (worse than linear) asymptotic behavior of an algorithm will sooner or later bring the most powerful system to its knees: Just throw enough data at it.

Defending against that likely future means we want to avoid "designing in" what will become performance pits in the face of larger files, larger databases, more pixels, more windows, more processes, more bits sent over the wire. One of the big success factors in future-proofing of the C++ standard library has been its performance complexity guarantees for the STL container operations and algorithms.

Here's the balance: It would clearly be wrong to optimize prematurely by using a less clear algorithm in anticipation of large data volumes that may never materialize. But it would equally clearly be wrong to pessimize prematurely by turning a blind eye to algorithmic complexity, a.k.a. "big-Oh" complexity, namely the cost of the computation as a function of the number of elements of data being worked on.

There are two parts to this advice. First, even before knowing whether data volumes will be large enough to be an issue for a particular computation, by default avoid using algorithms that work on user data (which could grow) but that don't scale well with data unless there is a clear clarity and readability benefit to using a less scalable algorithm (see Item 6). All too often we get surprised: We write ten pieces of code thinking they'll never have to operate on huge data sets, and then we'll turn out to be perfectly right nine of the ten times. The tenth time, we'll fall into a performance pitwe know it has happened to us, and we know it has happened or will happen to you. Sure, we go fix it and ship the fix to the customer, but it would be better to avoid such embarrassment and rework. So, all things being equal (including clarity and readability), do the following up front:

- *Use flexible, dynamically-allocated data and instead of fixed-size arrays:* Arrays "larger than the largest I'll ever need" are a terrible correctness and security fallacy. (See Item 77.) Arrays are acceptable when sizes really are fixed at compile time.
- *Know your algorithm's actual complexity:* Beware subtle traps like linear-seeming algorithms that actually call other linear operations, making the algorithm actually quadratic. (See Item 81 for an example.)
- *Prefer to use linear algorithms or faster wherever possible:* Constant-time complexity, such as **`push_back`** and hash table lookup, is perfect (see Items 76 and 80). O(log N) logarithmic complexity, such as **`set`**/**`map`** operations and **`lower_bound`** and **`upper_bound`** with random-access iterators, is good (see Items 76, 85, and 86). O(N) linear complexity, such as **`vector::insert`** and **`for_each`**, is acceptable (see Items 76, 81, and 84).
- *Try to avoid worse-than-linear algorithms where reasonable:* For example, by default spend some effort on finding a replacement if you're facing a O(N log N) or O(N^2) algorithm, so that your code won't fall into a disproportionately deep performance pit in the event that data volumes grow significantly. For example, this is a major reason why Item 81 advises to prefer range member functions (which are generally linear) over repeated calls of their single-element counterparts (which easily becomes quadratic as one linear operation invokes another linear operation; see Example 1 of Item 81).
- *Never use an exponential algorithm unless your back is against the wall and you really have no other option:* Search hard for an alternative before settling for an exponential algorithm, where even a modest increase in data volume means falling off a performance cliff.

References

[Bentley00

8. Don't optimize prematurely

Summary

Spur not a willing horse (Latin proverb): Premature optimization is as addictive as it is unproductive. The first rule of optimization is: Don't do it. The second rule of optimization (for experts only) is: Don't do it yet. Measure twice, optimize once.

Discussion

As [Stroustrup00

Examples

Example: An **`inline`** *irony.* Here is a simple demonstration of the hidden cost of a premature micro-optimization: Profilers are excellent at telling you, by function hit count, what functions you should have marked inline but didn't; profilers are terrible at telling you what functions you did mark inline but shouldn't have. Too many programmers "inline by default" in the name of optimization, nearly always trading higher coupling for at best dubious benefit. (This assumes that writing **`inline`** even matters on your compiler. See [Sutter00], [Sutter02], and [Sutter04].)

Exceptions

When writing libraries, it's harder to predict what operations will end up being used in performance-sensitive code. But even library authors run performance tests against a broad range of client code before committing to obfuscating optimizations.

References

[Bentley00

9. Don't pessimize prematurely

Summary

Easy on yourself, easy on the code: All other things being equal, notably code complexity and readability, certain efficient design patterns and coding idioms should just flow naturally from your fingertips and are no harder to write than the pessimized alternatives. This is not premature optimization; it is avoiding gratuitous pessimization.

Discussion

Avoiding premature optimization does not imply gratuitously hurting efficiency. By premature pessimization we mean writing such gratuitous potential inefficiencies as:

- Defining pass-by-value parameters when pass-by-reference is appropriate. (See Item 25.)
- Using postfix `++` when the prefix version is just as good. (See Item 28.)
- Using assignment inside constructors instead of the initializer list. (See Item 48.)

It is not a premature optimization to reduce spurious temporary copies of objects, especially in inner loops, when doing so doesn't impact code complexity. Item 18 encourages variables that are declared as locally as possible, but includes the exception that it can be sometimes beneficial to hoist a variable out of a loop. Most of the time that won't obfuscate the code's intent at all, and it can actually help clarify what work is done inside the loop and what c alculations are loop-invariant. And of course, prefer to use algorithms instead of explicit loops. (See Item 84.)

Two important ways of crafting programs that are simultaneously clear and efficient are to use abstractions (see Items 11 and 36) and libraries (see Item 84). For example, using the standard library's `vector`, `list`, `map`, `find`, `sort` and other facilities, which have been standardized and implemented by world-class experts, not only makes your code clearer and easier to understand, but it often makes it faster to boot.

Avoiding premature pessimization becomes particularly important when you are writing a libr ary. You typically can't know all contexts in which your library will be used, so you will want to strike a balance that leans more toward efficiency and reusability in mind, while at the same time not exaggerating efficiency for the benefit of a small fraction of potential callers. Drawing the line is your task, but as Item 7 shows, the bigger fish to focus on is scalability and not a little cycle-squeezing.

References

[Keffer95] pp.12-13 ?[Stroustrup00

10. Minimize global and shared data

Summary

Sharing causes contention: Avoid shared data, especially global data. Shared data increases coupling, which reduces maintainability and often performance.

Discussion

This statement is more general than Item 18's specific treatment.

Avoid data with external linkage at namespace scope or as static class members. These complicate program logic and cause tighter coupling between different (and, worse, distant) parts of the program. Shared data weakens unit testing because the correctness of a piece of code that uses shared data is conditioned on the history of changes to the data, and further conditions the functioning of acres of yet-unknown code that subsequently uses the data further.

Names of objects in the global namespace additionally pollute the global namespace.

If you must have global, namespace-scope, or static class objects, be sure to initialize such objects carefully. The order of initialization of such objects in different compilation units is undefined, and special techniques are needed to handle it correctly (see References). The order-of-initialization rules are subtle; prefer to avoid them, but if you do have to use them then know them well and use them with great care.

Objects that are at namespace scope, static members, or shared across threads or processes will reduce parallelism in multithreaded and multiprocessor environments and are a frequent source of performance and scalability bottlenecks. (See Item 7.) Strive for "shared-nothing;" prefer communication (e.g., message queues) over data sharing.

Prefer low coupling and minimized interactions between classes. (See [Cargill92].)

Exceptions

The program-wide facilities `cin`, `cout` , and `cerr` are special and are implemented specially. A factory has to maintain a registry of what function to call to create a given type, and there is typically one registry for the whole program (but preferably it should be internal to the factory rather than a shared global object; see Item 11).

Code that does share objects across threads should always serialize all access to those shared objects. (See Item 12 and [Sutter04c].)

References

[Cargill92] pp. 126.136, 169-173 ?[Dewhurst03

11. Hide information

Summary

Don't tell: Don't expose internal information from an entity that provides an abstraction.

Discussion

To minimize dependencies between calling code that manipulates an abstraction and the abstraction's implementation(s), data that is internal to the implementation must be hidden. Otherwise, calling code can accessor, worse, manipulatethat information, and the intended-to-be-internal information has leaked into the abstraction on which calling code depends. Expose an abstraction (preferably a domain abstraction where available, but at least a get/set abstraction) instead of data.

Information hiding improves a project's cost, schedule, and/or risk in two main ways:

- *It localizes changes:* Information hiding reduces the "ripple effect" scope of changes, and therefore their cost.
- *It strengthens invariants:* It limits the code responsible for maintaining (and, if it is buggy, possibly breaking) program invariants. (See Item 41.)

Don't expose data from any entity that provides an abstraction (see also Item 10). Data is just one possible incarnation of abstract, conceptual state. If you focus on concepts and not on their representations you can offer a suggestive interface and tweak implementation at willsuch as caching vs. computing on-the-fly or using various representations that optimize certain usage patterns (e.g., polar vs. Cartesian).

A common example is to never expose data members of class types by making them **`public`** (see Item 41) or by giving out pointers or handles to them (see Item 42), but this applies equally to larger entities such as libraries, which must likewise not expose internal information. Modules and libraries likewise prefer to provide interfaces that define abstractions and traffic in those, and thereby allow communication with calling code to be safer and less tightly coupled than is possible with data sharing.

Exceptions

Testing code often needs white-box access to the tested class or module.

Value aggregates ("C-style `struct`s") that simply bundle data without providing any abstraction do not need to hide their data; the data is the interface. (See Item 41.)

References

[Brooks95

12. Know when and how to code for concurrency

Team LiB ◀ PREVIOUS NEXT ▶

Summary

Thread *safely*: If your application uses multiple threads or processes, know how to minimize sharing objects where possible (see Item 10) and share the right ones safely.

Team LiB ◀ PREVIOUS NEXT ▶

Discussion

Threading is a huge domain. This Item exists because that domain is important and needs to be explicitly acknowledged, but one Item can't do it justice and we will only summarize a few essentials; see the References for many more details and techniques. Among the most important issues are to avoid deadlocks, livelocks, and malign race conditions (including corruption due to insufficient locking).

The C++ Standard says not one word about threads. Nevertheless, C++ is routinely and widely used to write solid multithreaded code. If your application shares data across threads, do so safely:

- *Consult your target platforms' documentation for local synchronization primitives:* Typical ones range from lightweight atomic integer operations to memory barriers to in-process and cross-process mutexes.

- *Prefer to wrap the platform's primitives in your own abstractions:* This is a good idea especially if you need cross-platform portability. Alternatively, you can use a library (e.g., pthreads [Butenhof97]) that does it for you.

- *Ensure that the types you are using are safe to use in a multithreaded program:* In particular, each type must at minimum:

    - *Guarantee that unshared objects are independent:* Two threads can freely use different objects without any special action on the caller's part.

    - *Document what the caller needs to do to use the same object of that type in different threads:* Many types will require you to serialize access to such shared objects, but some types do not; the latter typically either design away the locking requirement, or they do the locking internally themselves, in which case, you still need to be aware of the limits of what the internal locking granularity will do.

  Note that the above applies regardless of whether the type is some kind of string type, or an STL container like **`vector`**, or any other type. (We note that some authors have given advice that implies the standard containers are somehow special. They are not; a container is just another object.) In particular, if you want to use standard library components (e.g., **`string`**, containers) in a multithreaded program, consult your standard library implementation's documentation to see whether that is supported, as described earlier.

When authoring your own type that is intended to be usable in a multithreaded program, you must do the same two things: First, you must guarantee that different threads can use different objects of that type without locking (note: a type with modifiable static data typically can't guarantee this). Second, you must document what users need to do in order to safely use the same object in different threads; the fundamental design issue is how to distribute the responsibility of correct execution (race- and deadlock-free) between the class and its client. The main options are:

- *External locking: Callers are responsible for locking.* In this option, code that uses an object is responsible for knowing whether the object is shared across threads and, if so, for serializing all uses of the object. For example, string types typically use external locking (or immutability; see the third option on the next page).

- *Internal locking: Each object serializes all access to itself, typically by locking every public member function, so that callers may not need to serialize uses of the object.* For example, producer/consumer queues typically use internal locking, because their whole

References

[Alexandrescu02a] ?[Alexandrescu04] ?[Butenhof97] ?[Henney00] ?[Henney01] ?[Meyers04]

13. Ensure resources are owned by objects. Use explicit RAII and smart pointers

Summary

Don't saw by hand when you have power tools: C++'s "resource acquisition is initialization" (RAII) idiom is *the* power tool for correct resource handling. RAII allows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms. When allocating a raw resource, immediately pass it to an owning object. Never allocate more than one resource in a single statement.

Discussion

C++'s language-enforced constructor/destructor symmetry mirrors the symmetry inherent in resource acquire/release function pairs such as **fopen**/**fclose**, **lock** /**unlock**, and **new**/**delete** . This makes a stack-based (or reference-counted) object with a resource-acquiring constructor and a resource-releasing destructor an excellent tool for automating resource management and cleanup.

The automation is easy to implement, elegant, low-cost, and inherently error-safe. If you choose not to use it, you are choosing the nontrivial and attention-intensive task of pairing the calls correctly by hand, including in the presence of branched control flows and exceptions. Such C-style reliance on micromanaging resource deallocation is unacceptable when C++ provides direct automation via easy-to-use RAII.

Whenever you deal with a resource that needs paired acquire/release function calls, encapsulate that resource in an object that enforces pairing for you and performs the resource release in its destructor. For example, instead of calling a pair of **OpenPort**/ **ClosePort** nonmember functions directly, consider:

```
class Port {
public:
  Port( const string& destination );  // call OpenPort
  ~Port();                            // call ClosePort
```

Exceptions

Smart pointers can be overused. Raw pointers are fine in code where the pointed-to object is visible to only a restricted quantity of code (e.g., purely internal to a class, such as a `tree` class's internal node navigation pointers).

References

[Alexandrescu00c] ?[Cline99

Coding Style

One man's constant is another man's variable.

Alan Perlis

In this section, we tighten our focus from general design issues to issues that arise most often during actual coding.

The rules and guidelines in this section target coding practices that aren't specific to a particular language area (e.g., functions, classes, or namespaces) but that improve the quality of your code. Many of these idioms are about getting your compiler to help you, including the powerful tool of declarative `const` (Item 15) and internal `#include` guards (Item 24). Others will help you steer clear of land mines (including some outright undefined behavior) that your compiler can't always check for you, including avoiding macros (Item 16) and uninitialized variables (Item 19). All of them help to make your code more reliable.

Our vote for the most valuable Item in this section goes to Item 14: Prefer compile- and link-time errors to run-time errors.

14. Prefer compile- and link-time errors to run-time errors

Summary

Don't put off 'til run time what you can do at build time: Prefer to write code that uses the compiler to check for invariants during compilation, instead of checking them at run time. Run-time checks are control- and data-dependent, which means you'll seldom know whether they are exhaustive. In contrast, compile-time checking is not control- or data-dependent and typically offers higher degrees of confidence.

Discussion

The C++ language offers many opportunities to "accelerate" error detection by pushing it to compilation time. Exploiting these static checking capabilities offers you many advantages, including the following:

- *Static checks are data- and flow-independent:* Static checking offers guarantees that are independent of the program inputs or execution flow. In contrast, to make sure that your run-time checking is strong enough, you need to test it for a representative sample of all inputs. This is a daunting task for all but the most trivial systems.
- *Statically expressed models are stronger:* Oftentimes, a program that relies less on run-time checks and more on compile-time checks reflects a better design because the model the program creates is properly expressed using C++'s type system. This way, you and the compiler are partners having a consistent view of the program's invariants; run-time checks are often a fallback to do checking that could be done statically but cannot be expressed precisely in the language. (See Item 68.)
- *Static checks don't incur run-time overhead:* With static checks replacing dynamic checks, the resulting executable will be faster without sacrificing correctness.

One of C++'s most powerful static checking tools is its static type checking. The debate on w hether types should be checked statically (C++, Java, ML, Haskell) or dynamically (Smalltalk, Ruby, Python, Lisp) is open and lively. There is no clear winner in the general case, and there are languages and development styles that favor either kind of checking with reportedly good results. The static checking crowd argues that a large category of run-time error handling can be thus easily eliminated, resulting in stronger programs. On the other hand, the dynamic checking camp says that compilers can only check a fraction of potential bugs, so if you need to write unit tests anyway you might as well not bother with static checking at all and get a less restrictive programming environment.

One thing is clear: Within the context of the statically typed language C++, which provides strong static checking and little automatic run-time checking, programmers should definitely use the type system to their advantage wherever possible (see also Items 90 through 100). At the same time, run-time checks are sensible for data- and flow-dependent checking (e.g., array bounds checking or input data validation) (see Items 70 and 71).

Examples

There are several instances in which you can replace run-time checks with compile-time checks.

Example 1: Compile-time Boolean conditions. If you are testing for compile-time Boolean conditions such as `sizeof(int) >= 8`, use static assertions instead of run-time tests. (But see also Item 91.)

Example 2: Compile-time polymorphism. Consider replacing run-time polymorphism (virtual functions) with compile-time polymorphism (templates) when defining generic functions or types. The latter yields code that is better checked statically. (See also Item 64.)

Example 3: Enums. Consider defining `enum`s (or, better yet, full-fledged types) when you need to express symbolic constants or restricted integral values.

Example 4: Downcasting. If you frequently use `dynamic_cast` (or, worse, an unchecked `static_cast`) to perform downcasting, it can be a sign that your base classes offer too little functionality. Consider redesigning your interfaces so that your program can express computation in terms of the base class.

Exceptions

Some conditions cannot be checked at compile time and require run-time checks. For these, prefer to use assertions to detect internal programming errors (see Item 68) and follow the advice in the rest of the error handling section for other run-time errors such as data-dependent errors (see Items 69 through 75).

References

[Alexandrescu01

15. Use `const` proactively

Summary

`const` is your friend: Immutable values are easier to understand, track, and reason about, so prefer constants over variables wherever it is sensible and make **`const`** your default choice when you define a value: It's safe, it's checked at compile time (see Item 14), and it's integrated with C++'s type system. Don't cast away **`const`** except to call a const-incorrect function (see Item 94).

Discussion

Constants simplify code because you only have to look at where the constant is defined to know its value everywhere. Consider this code:

```
void Fun( vector<int>& v ) {
```

Examples

Example: Avoid **`const`** *pass-by-value function parameters in function declarations.* The following two declarations are exactly equivalent:

```
void Fun( int x );

void Fun( const int x );      // redeclares the same function: top-level
const is ignored
```

In the second declaration, the **`const`** is redundant. We recommend declaring functions without such top-level **`const`**s, so that readers of your header files won't get confused. However, the top-level **`const`** does make a difference in a function's *definition* and can be sensible there to catch unintended changes to the parameter:

```
void Fun( const int x ) {     // Fun's actual definition
```

References

[Allison98

16. Avoid macros

Summary

`TO_PUT_IT_BLUNTLY`: Macros are the bluntest instrument of C and C++'s abstraction facilities, ravenous wolves in functions' clothing, hard to tame, marching to their own beat all over your scopes. Avoid them.

Discussion

It's hard to find language that's colorful enough to describe macros, but we'll try. To quote from [Sutter04

Examples

Example: Passing a template instantiation to a macro. Macros barely understand C's parentheses and square brackets well enough to balance them. C++, however, defines a new parenthetical construct, namely the **<** and **>** used in templates. Macros can't pair those correctly, which means that in a macro invocation

```
MACRO( Foo<int, double> )
```

the macro thinks it is being passed two arguments, namely **Foo<int** and **double>**, when in fact the construct is one C++ entity.

Exceptions

Macros remain the only solution for a few important tasks, such as `#include` guards (see Item 24), `#ifdef` and `#if defined` for conditional compilation, and implementing `assert` (see Item 68).

For conditional compilation (e.g., system-dependent parts), avoid littering your code with `#ifdef`s. Instead, prefer to organize code such that the use of macros drives alternative implementations of one common interface, and then use the interface throughout.

You may want to use macros (cautiously) when the alternative is extreme copying and pasting snippets of code around.

We note that both [C99] and [Boost] include moderate and radical extensions, respectively, to the preprocessor.

References

[Boost] ?[C99] ?[Dewhurst03

17. Avoid magic numbers

Summary

Programming isn't magic, so don't incant it: Avoid spelling literal constants like **42** or **3.14159** in code. They are not self-explanatory and complicate maintenance by adding a hard-to-detect form of duplication. Use symbolic names and expressions instead, such as **width * aspectRatio**.

Discussion

Names add information and introduce a single point of maintenance; raw numbers duplicated throughout a program are anonymous and a maintenance hassle. Constants should be enumerators or `const` values, scoped and named appropriately.

One `42` may not be the same as another `42`. Worse, "in-head" computations made by the programmer (e.g., "this `84` comes from doubling the `42` used five lines ago") make it tedious and error-prone to later replace `42` with another constant.

Prefer replacing hardcoded strings with symbolic constants. Keeping strings separate from the code (e.g., in a dedicated `.cpp` or resource file) lets non-programmers review and update them, reduces duplication, and helps internationalization.

Examples

Example 1: Important domain-specific constants at namespace level.

```
const size_t PAGE_SIZE          = 8192,
             WORDS_PER_PAGE     = PAGE_SIZE / sizeof(int),
             INFO_BITS_PER_PAGE = 32 * CHAR_BIT;
```

Example 2: Class-specific constants. You can define static integral constants in the class definition; constants of other types need a separate definition or a short function.

```
// File widget.h
class Widget {
 static const int defaultWidth = 400;        // value provided in declaration
 static const double defaultPercent;         // value provided in definition
 static const char* Name() {return "Widget"; }
};

// File widget.cpp
const double Widget::defaultPercent = 66.67; // value provided in definition
const int Widget::defaultWidth;              // definition required
```

References

[Dewhurst03

18. Declare variables as locally as possible

Summary

Avoid scope bloat, as with requirements so too with variables): Variables introduce state, and you should have to deal with as little state as possible, with lifetimes as short as possible. This is a specific case of Item 10 that deserves its own treatment.

Discussion

Variables whose lifetimes are longer than necessary have several drawbacks:

- *They make the program harder to understand and maintain:* For example, should code update the module-wide `path` string if it only changes the current drive?
- *They pollute their context with their name:* As a direct consequence of this, namespace-level variables, which are the most visible of all, are also the worst (see Item 10).
- *They can't always be sensibly initialized:* Never declare a variable before you can initialize it sensibly. Uninitialized variables are a pervasive source of bugs in all C and C++ programs, and they require our proactive attention because they can't always be detected by compilers (see Item 19).

In particular, older versions of C before [C99] required variables to be defined only at the beginning of a scope; this style is obsolete in C++. A serious problem with this restriction is that at the beginning of the scope you often don't yet have enough information to initialize variables with pertinent information. This leaves you with two choiceseither initialize with some default blank value (e.g., zero), which is usually wasteful and can lead to errors if the variable ends up being used before it has a useful state, or leave them uninitialized, which is dangerous. An uninitialized variable of user-defined types will self-initialize to some blank value.

The cure is simple: Define each variable as locally as you can, which is usually exactly the point where you also have enough data to initialize it and immediately before its first use.

Exceptions

It can sometimes be beneficial to hoist a variable out of a loop. (See Item 9.)

Because constants don't add state, this Item does not apply to them. (See Item 17.)

References

[Dewhurst03

19. Always initialize variables

PREVIOUS NEXT

Summary

Start with a clean slate: Uninitialized variables are a common source of bugs in C and C++ programs. Avoid such bugs by being disciplined about cleaning memory before you use it; initialize variables upon definition.

PREVIOUS NEXT

Discussion

In the low-level efficiency tradition of C and C++ alike, the compiler is often not required to initialize variables unless you do it explicitly (e.g., local variables, forgotten members omitted from constructor initializer lists). Do it explicitly.

There are few reasons to ever leave a variable uninitialized. None is serious enough to justify the hazard of undefined behavior.

If you've used a procedural language (e.g., Pascal, C, Fortran, or Cobol) you might be used to defining variables in separation from the code that uses them, and then assigning them values later when they're about to be used. This approach is obsolete and not recommended (see Item 18).

A common misconception about uninitialized variables is that they will crash the program, so that those few uninitialized variables lying around here and there will be quickly revealed by simple testing. On the contrary, programs with uninitialized variables can run flawlessly for years if the bits in the memory happen to match the program's needs. Later, a call from a different context, a recompilation, or some change in another part of the program will cause failures ranging from inexplicable behavior to intermittent crashes.

Examples

Example 1: Using a default initial value or **`?:`** *to reduce mixing data flow with control flow*.

```
// Not recommended: Doesn't initialize variable
int speedupFactor;
if( condition )
  speedupFactor = 2;
else
  speedupFactor = -1;

// Better: Initializes variable
int speedupFactor = -1;
if( condition )
  speedupFactor = 2;
// Better: Initializes variable
int speedupFactor = condition ? 2 : -1;
```

The better alternatives nicely leave no gap between definition and initialization.

Example 2: Replacing a complicated computational flow with a function. Sometimes a value is computed in a way that is best encapsulated in a function (see Item 11):

```
// Not recommended: Doesn't initialize variable
int speedupFactor;

if( condition ) {
```

Exceptions

Input buffers and `volatile` data that is directly written by hardware or other processes does not need to be initialized by the program.

PREVIOUS NEXT

References

[Dewhurst03

20. Avoid long functions. Avoid deep nesting

Summary

Short is better than long, flat is better than deep: Excessively long functions and nested code blocks are often caused by failing to give one function one cohesive responsibility (see Item 5), and both are usually solved by better refactoring.

Discussion

Every function should be a coherent unit of work bearing a suggestive name (see Item 5 and the Discussion in Item 70). When a function instead tries to merge such small conceptual elem ents inside a long function body, it ends up doing too much.

Excessive straight-line function length and excessive block nesting depth (e.g., **`if`**, **`for`**, **`while`**, and **`try`** blocks) are twin culprits that make functions more difficult to understand and maintain, and often needlessly so.

Each level of nesting adds intellectual overhead when reading code because you need to maintain a mental stack (e.g., enter conditional, enter loop, enter **`TRy`**

Exceptions

A function might be legitimately long and/or deep when its functionality can't be reasonably refactored into independent subtasks because every potential refactoring would require passing many local variables and context (rendering the result less readable rather than more readable). But if several such potential functions take similar arguments, they might be candidates for becoming members of a new class.

References

[Piwowarski82] ?[Miller56]

21. Avoid initialization dependencies across compilation units

Summary

Keep (initialization) order: Namespace-level objects in different compilation units should never depend on each other for initialization, because their initialization order is undefined. Doing otherwise causes headaches ranging from mysterious crashes when you make small changes in your project to severe non-portability even to new releases of the same compiler.

Discussion

When you define two namespace-level objects in different compilation units, which object's constructor is called first is not defined. Often (but not always) your tools might happen to initialize them in the order in which the compilation units' object files are linked, but this assumption is usually not reliable; even when it does hold, you don't want the correctness of your code to subtly depend on your makefile or project file. (For more on the evils of order dependencies, see also Item 59.)

Therefore, inside the initialization code of any namespace-level object, you can't assume that any other object defined in a different compilation unit has already been initialized. These considerations apply to dynamically initialized variables of primitive types, such as a namespace-level `bool reg_success = LibRegister("mylib");`

Note that, even before they are ever constructed using a constructor, namespace-level objects are statically initialized with all zeroes (as opposed to, say, automatic objects that initially contain garbage). Paradoxically, this zero-initialization can make bugs harder to detect, because instead of crashing your program swiftly the static zero-initialization gives your yet-uninitialized object an appearance of legitimacy. You'd think that that string is empty, that pointer is null, and that integer is zero, when in fact no code of yours has bothered to initialize them yet.

To avoid this problem, avoid namespace-level variables wherever possible; they are dangerous (see Item 10). When you do need such a variable that might depend upon another, consider the Singleton design pattern; used carefully, it might avoid implicit dependencies by ensuring that an object is initialized upon first access. Still, Singleton is a global variable in sheep's clothing (see again Item 10), and is broken by mutual or cyclic dependencies (again, zero-initialization only adds to the confusion).

References

[Dewhurst03

22. Minimize definitional dependencies. Avoid cyclic dependencies

Summary

Don't be over-dependent: Don't `#include` a definition when a forward declaration will do.

Don't be co-dependent: Cyclic dependencies occur when two modules depend directly or indirectly on one another. A module is a cohesive unit of release (see page 103); modules that are interdependent are not really individual modules, but super-glued together into what's really a larger module, a larger unit of release. Thus, cyclic dependencies work against modularity and are a bane of large projects. Avoid them.

Discussion

Prefer forward declarations except where you really need a type's definition. You need a full definition of a class `C`in two main cases:

- *When you need to know the size of a* `C` *object:* For example, when allocating a `C` on the stack or as a directly-held member of another type.
- *When you need to name or call a member of* `C`*:* For example, when calling a member function.

In keeping with this book's charter, we'll set aside from the start those cyclic dependencies that cause compile-time errors; you've already fixed them by following good advice present in the literature and Item 1. Let's focus on cyclic dependencies that remain in compilable code, see how they trouble your code's quality, and what steps need be taken to avoid them.

In general, dependencies and their cycles should be thought of at module level. A module is a cohesive collection of classes and functions released together (see Item 5 and page 103). In its simplest form, a cyclic dependency has two classes that directly depend upon each other:

```
class Child;                    // breaks the dependency cycle

class Parent {
```

Exceptions

Cycles among classes are not necessarily badas long as the classes are considered part of

References

[*Alexandrescu01*

23. Make header files self-sufficient

Summary

Behave responsibly: Ensure that each header you write is compilable standalone, by having it include any headers its contents depend upon.

Discussion

If one header file won't work unless the file that includes it also includes another header, that's gauche and puts unnecessary burden on that header file's users.

Years ago, some experts advised that headers should not include other headers because of the cost of opening and parsing a guarded header multiple times. Fortunately, this is largely obsolete: Many modern C++ compilers recognize header guards automatically (see Item 24) and don't even open the same header twice. Some also offer precompiled headers, which help to ensure that often-used, seldom-changed headers will not be parsed often.

But don't include headers that you *don't* need; they just create stray dependencies.

Consider this technique to help enforce header self-sufficiency: In your build, compile each header in isolation and validate that there are no errors or warnings.

Examples

Some subtler issues arise in connection with templates.

Example 1: Dependent names. Templates are compiled at the point where they are defined, except that any dependent names or types are not compiled until the point where the template is instantiated. This means that a **`template<class T> class Widget`** with a **`std::deque<T>`** member does not incur a compile-time error even when **`<deque>`** is not included, as long as nobody instantiates **`Widget`**. Given that **`Widget`** exists in order to be instantiated, its header clearly should **`#include <deque>`**.

Example 2: Member function templates, and member functions of templates, are instantiated only if used. Suppose that **`Widget`** doesn't have a member of type **`std::deque<T>`**, but **`Widget`**'s **`transmogrify`** member function uses a **`deque`**. Then **`Widget`**'s callers can instantiate and use **`Widget`** just fine even if no one includes **`<deque>`**, as long as they don't use **`TRansmogrify`**. By default, the **`Widget`** header should still **`#include <deque>`** because it is necessary for at least some callers of **`Widget`**. In rare cases where an expensive header is being included for few rarely used functions of a template, consider refactoring those functions as nonmembers supplied in a separate header that does include the expensive one. (See Item 44.)

References

[Lakos96

24. Always write internal `#include` guards. Never write external `#include` guards

Summary

Wear head(er) protection: Prevent unintended multiple inclusions by using `#include` guards with unique names for all of your header files.

Discussion

Each header file should be guarded by an internal **`#include`** guard to avoid redefinitions in case it is included multiple times. For example, a header file **`foo.h`** should follow the general form:

```
#ifndef FOO_H_INCLUDED_
#define FOO_H_INCLUDED_
```

Exceptions

In very rare cases, a header file may be intended to be included multiple times.

References

[C++03

Functions and Operators

If you have a procedure with ten parameters, you probably missed some.

Alan Perlis

Functions, including overloaded operators, are the fundamental units of work. As we will see later on in the section on Error Handling and Exceptions (and particularly in Item 70), this has a direct effect on how we reason about the correctness and safety of our code.

But first, let's consider some fundamental mechanics for writing functions, including operators. In particular, we'll focus on their parameters, their semantics, and their overloading.

Our vote for the most valuable Item in this section goes to Item 26: Preserve natural semantics for overloaded operators.

25. Take parameters appropriately by value, (smart) pointer, or reference

Summary

Parameterize well: Distinguish among input, output, and input/output parameters, and between value and reference parameters. Take them appropriately.

Discussion

Choosing well among values, references, and pointers for parameters is good habit that maximizes both safety and efficiency.

Although efficiency should not be our primary up-front concern (see Item 8), neither should we write needlessly inefficient code when all other things, including clarity, are equal (see Item 9).

Prefer to follow these guidelines for choosing how to take parameters. For input-only parameters:

- Always `const`-qualify all pointers or references to input-only parameters.
- Prefer taking inputs of primitive types (e.g., `char`, `float`) and value objects that are cheap to copy (e.g., `Point`, `complex<float>`) by value.
- Prefer taking inputs of other user-defined types by reference to `const`.
- Consider pass-by-value instead of reference if the function requires a copy of its argument. This is conceptually identical to taking a reference to `const` plus doing a copy, and it can help compiler to better optimize away temporaries.

For output or input/output parameters:

- Prefer passing by (smart) pointer if the argument is optional (so callers can pass null as a "not available" or "don't care" value) or if the function stores a copy of the pointer or otherwise manipulates ownership of the argument.
- Prefer passing by reference if the argument is required and the function won't store a pointer to it or otherwise affect its ownership. This states that the argument is required and makes the caller responsible for providing a valid object.

Don't use C-style varargs (see Item 98).

References

[Alexandrescu03a] ?[Cline99

26. Preserve natural semantics for overloaded operators

Summary

Programmers hate surprises: Overload operators only for good reason, and preserve natural semantics; if that's difficult, you might be misusing operator overloading.

Discussion

Although anyone would agree (we hope) that one should not implement subtraction in an `operator+` implementation, other cases can be subtle. For example, does your `Tensor` class's `operator*` mean the scalar product or the vector product? Does `operator+=( Tensor& t, unsigned u )` add `u` to each of `t`'s elements, or will it resize `t`? In such ambiguous or counterintuitive cases, prefer using named functions instead of fostering cryptic code.

For value types (but not all types; see Item 32): "When in doubt, do as the `int`s do." [Meyers96] Mimicking the behavior of and relationships among operators on built-in types ensures that you don't surprise anyone. If your semantics of choice are likely to raise eyebrows, maybe operator overloading is not a good idea.

Programmers expect operators to come in bundles. If the expression `a @ b` is well formed for some operator `@` you define (possibly after conversions), ask: Can the caller also write `b @ a` without surprises? Can the caller write `a @= b`? (See Item 27.) If the operator has an inverse (e.g., `+` and `-`, or `*` and `/`), are both supported?

Named functions are less likely to have such assumed relationships, and therefore should be preferred for clearer code if there can be any doubt about semantics.

Exceptions

There are highly specialized libraries (e.g., parser generators and regular expression engines) that define domain-specific conventions for operators that are very different from their C++ meanings (e.g., a regular expression engine might use `operator*` to express "zero or more"). Prefer instead to find an alternative to unusual operator overloading (e.g., [C++TR104] regular expressions use strings, so that `*` can be used naturally without overloading operato rs). If after careful thought you choose to use operators anyway, make sure you define a coherent framework for your conventions and that you don't step on the toes of any built-in operator.

References

[Cline99

None 27. Prefer the canonical forms of arithmetic and assignment operators

Summary

If you `a+b`, also `a+=b`: When defining binary arithmetic operators, provide their assignment versions as well, and write to minimize duplication and maximize efficiency.

Discussion

In general, for some binary operator `@` (be it `+`, `-`, `*`, and so on), you should define its assignment version such that `a @= b` and `a = a @ b` have the same meaning (other than that the first form might be more efficient and only evaluates `a` once). The canonical way of achieving this goal is to define `@` in terms of `@=`, as follows:

```
T& T::operator@=( const T& ) {
```

Examples

Example: An implementation of **`+=`** *for strings.* When concatenating strings, it is useful to know the length in advance so as to allocate memory only once.

```
String& String::operator+=( const String& rhs ) {
```

Exceptions

In some cases (e.g., `operator*=` on complex numbers), an operator might mutate its left-hand side so significantly that it can be more advantageous to implement `operator*=` in terms of `operator*` rather than the reverse.

References

[Alexandrescu03a] ?[Cline99

28. Prefer the canonical form of `++` and `--`. Prefer calling the prefix forms

Summary

If you `++c`, also `c++`: The increment and decrement operators are tricky because each has pre- and postfix forms, with slightly different semantics. Define `operator++` and `operator--` such that they mimic the behavior of their built-in counterparts. Prefer to call the prefix v ersions if you don't need the original value.

◂ PREVIOUS NEXT ▸

Discussion

An ancient joke about C++ was that the language is called C++ and not ++C because the language is improved (incremented), but many people still use it as C (the previous value). Fortunately, the joke is now obsolete, but it's a helpful illustration for understanding the difference between the two operator forms.

For **++** and **--**, the postfix forms return the original value, whereas the prefix forms return the new value. Prefer to implement the postfix form in terms of the prefix form. The canonical form is:

```
T& T::operator++() {          T& T::operator--() {     // the prefix form:
 // perform increment          // perform decrement    //   - do the work
 return *this;                 return *this;           //   - always return
*this;
}                             }

T T::operator++(int) {        T T::operator--(int) {   // the postfix form:
 T old( *this );               T old( *this );         //   - remember old value
 ++*this;                      --*this;     //   - call the prefix version
 return old;                   return old;             //   - return the old
value
}                             }
```

In calling code, prefer using the prefix form unless you actually need the original value returned by the postfix version. The prefix form is semantically equivalent, just as much typing, and often slightly more efficient by creating one less object. This is not premature optimization; it is avoiding premature pessimization (see Item 9).

◂ PREVIOUS NEXT ▸

Exceptions

Expression template frameworks preserve the semantics via different means.

◀ PREVIOUS NEXT ▶

References

[Cline99

29. Consider overloading to avoid implicit type conversions

Summary

Do not multiply objects beyond necessity (Occam's Razor): Implicit type conversions provide syntactic convenience (but see Item 40). But when the work of creating temporary objects is unnecessary and optimization is appropriate (see Item 8), you can provide overloaded functions with signatures that match common argument types exactly and won't cause conversions.

Discussion

If you're in the office and run out of paper, what do you do? Of course, you walk to your trusty photocopier and make several copies of a white sheet of paper.

As silly as it sounds, this is often what implicit conversions do: unnecessarily go through the trouble of creating temporaries, just to perform some trivial operation on them and toss them away (see Item 40). A common example is string comparison:

```
class String {
```

References

[Meyers96

30. Avoid overloading `&&`, `||`, or `,` (comma)

Summary

Wisdom means knowing when to refrain: The built-in `&&`, `||` , and `,` (comma) enjoy special treatment from the compiler. If you overload them, they become ordinary functions with very different semantics (you *will* violate Items 26 and 31), and this is a sure way to introduce

Discussion

The primary reason not to overload **`operator&&`**, **`operator||`** , or **`operator,`** (comma) is that you *cannot* implement the full semantics of the built-in operators in these three cases, and programmers commonly expect those semantics. In particular, the built-in versions evaluate left-to-right, and for **`&&`** and **`||`** also use short-circuit evaluation.

The built-in versions of **`&&`** and **`||`** first evaluate their left-hand expression, and if that fully determines the result (**`false`** for **`&&`**, **`true`** for **`||`**) then the right-hand expression doesn't need to be evaluatedand is guaranteed not to be. We all get so used to this handy feature that we routinely allow the correctness of the right-hand side depend on the success of the left-hand side:

```
Employee* e = TryToGetEmployee();
if( e && e->Manager() )
```

Examples

*Example: Initialization library with overloaded **`operator,`** for sequence initialization.* One library helpfully tried to make it easier to add multiple values to a container in one shot by overloading the comma. For example, to append to a **`vector<string> letters`**:

```
set_cont(letters) += "a", "b";        // problematic
```

That's fine until the day the caller writes:

```
set_cont(letters) += getstr(), getstr(); // order unspecified when using
overloaded ","
```

If **`getstr`** gets user console input, for example, and the user enters the strings **`"c"`** and **`"d"`** in that order, the strings can actually be applied in either order. That's a surprise, because this is not a problem for the built-in sequencing **`operator,`**:

```
string s = getstr(), getstr();            // order well-specified using
built-in ","
```

Exceptions

An exception is expression template libraries, which by design capture all operators.

References

[Dewhurst03

31. Don't write code that depends on the order of evaluation of function arguments

Summary

Keep (evaluation) order: The order in which arguments of a function are evaluated is unspecified, so don't rely on a specific ordering.

Discussion

In the early days of C, processor registers were a precious resource, and compilers were hard pressed to allocate them efficiently for complex expressions in high-level languages. To allow generation of faster code, the creators of C gave the register allocator an extra degree of freedom: When calling a function, the order of evaluation of its arguments was left unspecified. That motivation is arguably less strong with today's processors, but the fact remains that the order of evaluation is unspecified in C++ and, as it turns out, varies widely across compilers. (See also Item 30.)

This can cause big trouble to the unwary. Consider this code:

```
void Transmogrify( int, int );

int count = 5;
Transmogrify( ++count, ++count );         // order of evaluation unknown
```

All we can say for certain is that **`count`** will be **`7`** as soon as **`TRansmogrify`**'s body is enteredbut we can't say which of its arguments is **`6`** and which is **`7`**. This uncertainty applies to much less obvious cases, such as functions that modify their argument (or some global state) as a side effect:

```
int Bump( int& x ) {return ++x; }
Transmogrify( Bump(count), Bump(count) );  // still unknown
```

Per Item 10, avoid global and shared variables in the first place. But even if you avoid them, others' code might not. For example, some standard functions do have side effects (e.g., **`strtok`**, and the various overloads of **`operator<<`** that take an **`ostream`**).

The cure is simpleuse named objects to enforce order of evaluation. (See Item 13.)

```
int bumped = ++count;
Transmogrify( bumped, ++count );         // ok
```

References

[Alexandrescu00c] ?[Cline99

Class Design and Inheritance

The most important single aspect of software development is to be clear about what you are trying to build.

Bjarne Stroustrup

What kinds of classes does your team prefer to design and build? Why?

Interestingly, most of the Items in this section are motivated primarily or exclusively by dependency management. For example, inheritance is the second-strongest relationship you can express in C++, second only to **`friend`**; it should come as no surprise, then, that it's important to use such a powerful tool judiciously, correctly, and well.

In this section, we focus on the key issues in class design, from minimalism to abstraction, from composition to inheritance, from virtual to nonvirtual, from **`public`** to **`private`**, from **`new`** to **`delete`**: How to get them right, how not to get them wrong, how to avoid the subtle pitfalls, and especially how to manage dependencies.

In the section after this one, we'll narrow our focus specifically to the Big Four special member functions: Default construction, copy construction, copy assign ment, and destruction.

Our vote for the most valuable Item in this section goes to Item 33: Prefer minimal classes to monolithic classes.

32. Be clear what kind of class you're writing

Summary

Know thyself: There are different kinds of classes. Know which kind you are writing.

Discussion

Different kinds of classes serve different purposes, and so follow different rules. Value classes (e.g., **`std::pair`**, **`std::vector`**) are modeled after built-in types. A value class:

- Has a public destructor, copy constructor, and assignment with value semantics.
- Has no virtual functions (including the destructor).
- Is intended to be used as a concrete class, not as a base class (see Item 35).
- Is usually instantiated on the stack or as a directly held member of another class.

Base classes are the building blocks of class hierarchies. A base class:

- Has a destructor that is public and virtual or else protected and nonvirtual (see Item 50), and a nonpublic copy constructor and assignment operator (see Item 53).
- Establishes interfaces through virtual functions.
- Is usually instantiated dynamically on the heap and used via a (smart) pointer.

Loosely, traits classes are templates that carry information about types. A traits class:

- Contains only **`typedef`**s and static functions. It has no modifiable state or virtuals.
- Is not usually instantiated (construction can normally be disabled).

Policy classes (normally templates) are fragments of pluggable behavior. A policy class:

- May or may not have state or virtual functions.
- Is not usually instantiated standalone, but only as a base or member.

Exception classes exhibit an unusual mix of value and reference semantics: They are thrown by value but should be caught by reference (see Item 73). An exception class:

- Has a public destructor and no-fail constructors (especially a no-fail copy constructor; throwing from an exception's copy constructor will abort your program).
- Has virtual functions, and often implements cloning (see Item 54) and visitation.
- Preferably derives virtually from **`std::exception`**.

Ancillary classes typically support specific idioms (e.g., RAII; see Item 13). They should be easy to use correctly and hard to use incorrectly (e.g., see Item 53).

References

[Abrahams01b] ?[Alexandrescu00a] ?[Alexandrescu00b] ?[Alexandrescu01

33. Prefer minimal classes to monolithic classes

Summary

Divide and conquer: Small classes are easier to write, get right, test, and use. They are also more likely to be usable in a variety of situations. Prefer such small classes that embody simple concepts instead of kitchen-sink classes that try to implement many and/or complex concepts (see Items 5 and 6).

Discussion

Designing fancy large classes is a typical mistake when starting object-oriented design. The prospect of having a class that offers complete and complex functionality in one shot can be quite alluring. However, designing smaller, minimal classes that can be easily combined is an approach that is more successful in practice for systems of any size, for many reasons:

- A minimal class embodies one concept at the right level of granularity. A monolithic class is likely to embody several separate concepts, and using one implies dragging the intellectual overhead of all others. (See Items 5 and 11)
- A minimal class is easier to comprehend, and more likely to be used and reused.
- A minimal class is easier to deploy. A monolithic class must often be deployed as a bulky indivisible unit. For example, a monolithic `Matrix` class might attempt to implement and deploy exotic functionality such as computing the eigenvalues of a matrixeven when the majority of clients just want simple linear algebra. A better packaging would implement various functional areas as nonmember functions operating on a minimal `Matrix` type. Then the functional areas can be tested and deployed in separation to the callers who need them. (See Item 44)
- Monolithic classes dilute encapsulation. When a class has many member functions that don't need to be members but areand therefore have gratuitous visibility to the class's private implementationthen the class's private data members become nearly as bad as public variables.
- Monolithic classes usually result from an attempt to predict and deliver a "complete" solution to a problem; in practice, they virtually never succeed. There's always something more that people wantand something less, for that matter.
- Monolithic classes are harder to make correct and error-safe because they often tackle multiple responsibilities. (See Items 5 and 44)

References

[Cargill92] pp. 85-86, 152, 174-177 ?[Lakos96

34. Prefer composition to inheritance

Summary

Avoid inheritance taxes: Inheritance is the second-tightest coupling relationship in C++, second only to friendship. Tight coupling is undesirable and should be avoided where possible. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design.

Discussion

Inheritance is often overused, even by experienced developers. A sound rule of software engineering is to minimize coupling: If a relationship can be expressed in more than one way, use the weakest relationship that's practical.

Given that inheritance is nearly the strongest relationship we can express in C++, second only to friendship, it's only really appropriate when there is no equivalent weaker alternative. If you can express a class relationship using composition alone, you should prefer that.

In this context, "composition" means simply embedding a member variable of a type within another type. This way, you can hold and use the object in ways that allow you control over the strength of the coupling.

Composition has important advantages over inheritance:

- *Greater flexibility without affecting calling code:* A private data member is under your control. You can switch from holding it by value to holding by (smart) pointer or Pimpl (see Item 43) without breaking client code; you would only need to change the implementations of the class's own member functions that use it. If you decide you need different functionality, you can easily change the type of the member or the manner of holding it while keeping the class's public interface consistent. In contrast, if you begin with a public inheritance relationship, it is likely that clients have already come to depend on the inheritance; you have therefore committed your class to it and cannot easily change your base class decision later on. (See Item 37.)

- *Greater compile-time insulation, shorter compile times:* Holding an object by pointer (preferably a smart pointer), rather than as a direct member or base class, can also allow you to reduce header dependencies because declaring a pointer to an object doesn't require that object's full class definition. By contrast, inheritance always requires the full definition of the base class to be visible. A common technique is to aggregate all private members behind a single opaque pointer, called a Pimpl (see Item 43).

- *Less weirdness:* Inheriting from a type can cause name lookup to pull in functions and function templates defined in the same namespace as that type. This is very subtle and hard to debug. (See also Item 58)

- *Wider applicability:* Some classes were not designed to be bases in the first place (and see Item 35). Most classes, however, can fulfill the role of a member.

- *Great robustness and safety:* The tighter coupling of inheritance makes it more difficult to write error-safe code. (See [Sutter02

Exceptions

Do use public inheritance to model substitutability. (See Item 37.)

Even if you don't need to provide a substitutability relationship to all callers, you do need *nonpublic* inheritance if you need any of the following, in rough order from most common (the first two points) to exceedingly rare (the rest):

- If you need to override a virtual function.
- If you need access to a protected member.
- If you need to construct the used object before, or destroy it after, a base class.
- If you need to worry about virtual base classes.
- If you *know* you benefit from the empty base class optimization, including that it matters in this case and that your target compiler(s) actually perform it in this case. (See Item 8)
- If you need controlled polymorphism. That is, if you need a substitutability relationship, but that relationship should be visible only to selected code (via friendship).

References

[Cargill92] pp. 49-65, 101-105 ?[Cline99

35. Avoid inheriting from classes that were not designed to be base classes

Summary

Some people don't want to have kids: Classes meant to be used standalone obey a different blueprint than base classes (see Item 32). Using a standalone class as a base is a serious design error and should be avoided. To add behavior, prefer to add nonmember functions instead of member functions (see Item 44). To add state, prefer composition instead of inheritance (see Item 34). Avoid inheriting from concrete base classes.

Discussion

Using inheritance when it is not needed betrays a misplaced faith in the power of object orientation. In C++, you need to do specific things when defining base classes (see also Items 32, 50, and 54), and very different and often contrary things when designing standalone classes. Inheriting from a standalone class opens your code to a host of problems, very few of which will be ever flagged by your compiler.

Beginners sometimes derive from value classes, such as a `string` class (the standard one or another), to "add more functionality." However, defining free (nonmember) functions is vastly superior to creating `super_string` for the following reasons:

- Nonmember functions work well within existing code that already manipulates `string`s. If instead you supply a `super_string`, you force changes throughout your code base to change types and function signatures to `super_string`.
- Interface functions that take a `string` now need to: a) stay away from `super_string` 's added functionality (unuseful); b) copy their argument to a `super_string` (wasteful); or c) cast the `string` reference to a `super_string` reference (awkward and potentially illegal).
- `super_string`'s member functions don't have any more access to `string`'s internals than nonmember functions because `string` probably doesn't have `protected` members (remember, it wasn't meant to be derived from in the first place).
- If `super_string` hides some of `string`'s functions (and redefining a nonvirtual function in a derived class is not overriding, it's just hiding), that could cause widespread confusion in code that manipulates `string`s that started their life converted automatically from `super_string`s.

So, prefer to add functionality via new nonmember functions (see Item 44). To avoid name lookup problems, make sure you put them in the same namespace as the type they are meant to extend (see Item 57). Some people dislike nonmember functions because the invocation syntax is `Fun(str)` instead of `str.Fun()`, but this is just a matter of syntactic habit and familiarity. (Then there's the sound bite, attributed to the legendary Alan Perlis, that too much syntactic sugar causes cancer of the semicolon.)

What if `super_string` wants to inherit from `string` to add more *state*, such as encoding or a cached word count? Public inheritance is still not recommended, because `string` is not protected against slicing (see Item 54) and so any copying of a `super_string` to a `string` will silently chop off all of the carefully maintained extra state.

Finally, inheriting from a class with a public nonvirtual destructor risks littering your code with undefined behavior by `delete`-ing pointers to `string` that actually point to `super_string` objects (see Item 50). This undefined behavior might seem to be tolerated by your compiler and memory allocator, but it leaves you in the dark swamp of silent errors, memory leaks, heap corruptions, and porting nightmares.

Examples

Example: Composition instead of public or private inheritance. What if you do need a **`localized_string`** that is "almost like **`string`**, but with some more state and functions and some tweaks to existing **`string`** functions," and many functions' implementations will be unchanged? Then implement it in terms of **`string`** safely by using containment instead of inheritance (which prevents slicing and undefined polymorphic deletion), and add passthrough functions to make unchanged functions visible:

```
class localized_string {
public:
```

References

[Dewhurst03

36. Prefer providing abstract interfaces

Summary

Love abstract art: Abstract interfaces help you focus on getting an abstraction right without muddling it with implementation or state management details. Prefer to design hierarchies that implement abstract interfaces that model abstract concepts.

Discussion

Prefer to define and inherit from abstract interfaces. An abstract interface is an abstract class made up entirely of (pure) virtual functions and having no state (member data) and usually no member function implementations. Note that avoiding state in abstract interfaces simplifies the entire hierarchy design (see [Meyers96] for examples).

Prefer to follow the Dependency Inversion Principle (DIP; see [Martin96a] and [Martin00]). The DIP states that:

- High-level modules should not depend upon low-level modules. Rather, both should depend upon abstractions.
- Abstractions should not depend upon details. Rather, details should depend upon abstractions.

Respecting the DIP implies that hierarchies should be rooted in abstract classes, not concrete classes. (See Item 35.) The abstract base classes must worry about defining functionality, not about implementing it. Put another way: Push policy up and implementation down.

The Dependency Inversion Principle has three fundamental design benefits:

- *Improved robustness:* The less stable parts of a system (implementations) depend on more stable parts (abstractions). A robust design is one in which changes have localized effect. In a fragile system, on the other hand, a small change ripples in unfortunate ways through unexpected parts of the system. This is exactly what happens with designs that have concrete base classes.
- *Greater flexibility:* Designs based on abstract interfaces are generally more flexible. If the abstractions are properly modeled, it is easy to devise new implementations for new requirements. On the contrary, a design that depends on many concrete details is rigid, in that new requirements lead to core changes.
- *Good modularity:* A design relying on abstractions has good modularity because its dependencies are simple: Highly changeable parts depend on stable parts, not vice versa. At the other extreme, a design that has interfaces mixed with implementation details is likely to sport intricate webs of dependency that make it hard to reapply as a unit to plug into another system.

The related Law of Second Chances states: "The most important thing to get right is the interface. Everything else can be fixed later. Get the interface wrong, and you may never be allowed to fix it." [Sutter04]

Typically, choose a public virtual destructor to enable polymorphic deletion (per Item 50), unless you use an object broker such as COM or CORBA that uses an alternate memory management mechanism.

Be wary about using multiple inheritance of classes that are not abstract interfaces. Designs that use multiple inheritance can be very expressive, but are harder to get right and easier to get wrong. In particular, state management is particularly hard in designs using multiple inheritance.

As noted in Item 34), inheriting from a type can also cause name lookup coupling: subtly pulling in functions from the namespace of that type. (See also Item 58.)

Examples

Example: Backup program.

Exceptions

The empty base optimization is one instance when inheritance (preferably nonpublic) is used for purely optimization purposes. (But see Item 8.)

It would appear that policy-based designs have a high-level component depend on implementation details (the policies). However, that is only a use of static polymorphism. The abstract interfaces are there, except that they are implicit, not explicitly stated via pure virtual functions.

References

[Alexandrescu01] ?[Cargill92] pp. 12-15, 215-218 ?[Cline99

37. Public inheritance is substitutability. Inherit, not to reuse, but to be reused

Summary

Know what: Public inheritance allows a pointer or reference to the base class to actually refer to an object of some derived class, without destroying code correctness and without needing to change existing code.

Know why: Don't inherit publicly to reuse code (that exists in the base class); inherit publicly in order to be reused (by existing code that already uses base objects polymorphically).

Discussion

Despite two decades of object-oriented design knowledge, the purpose and practice of public inheritance are still frequently misunderstood, and many uses of inheritance are flawed.

Public inheritance must always model "is-a" ("works-like-a") according to the Liskov Substitution Principle (see [Liskov88]): All base contracts must be fulfilled, and so all overrides of virtual member functions must require no more and promise no less than their base versions if they are to successfully fulfill the base's contract. Code using a pointer or reference to a `Base` must behave correctly even when that pointer or reference actually points to a `Derived`.

Misuse of inheritance destroys correctness. Incorrectly implemented inheritance most typically goes astray by failing to obey the explicit or implicit contract that the base class establishes. Such contracts can be subtle, and when they cannot be expressed directly in code the programmer must take extra care. (Some patterns help to declare more intent in code; see Item 39)

To distill a frequently cited example: Consider that two classes `Square` and `Rectangle` each have virtual functions for setting their height and width. Then `Square` cannot correctly inherit from `Rectangle`, because code that uses a modifiable `Rectangle` will assume that `SetWidth` does not change the height (whether `Rectangle` explicitly documents that contract or not), whereas `Square::SetWidth` cannot preserve that contract and its own squareness invariant at the same time. But `Rectangle` cannot correctly inherit from `Square` either, if clients of `Square` assume for example that a `Square`'s area is its width squared, or if they rely on some other property that doesn't hold for `Rectangle`s.

The "is-a" description of public inheritance is misunderstood when people use it to draw irr elevant real-world analogies: A square "is-a" rectangle (mathematically) but a `Square` is not a `Rectangle` (behaviorally). Consequently, instead of "is-a," we prefer to say "works-like-a" (or, if you prefer, "usable-as-a") to make the description less prone to misunderstanding.

Public inheritance is indeed about reuse, but not the way many programmers seem to think. As already pointed out, the purpose of public inheritance is to implement substitutability (see [Liskov88]). The purpose of public inheritance is *not* for the derived class to reuse base class code to implement itself in terms of the base class's code. Such an is-implemented-in-terms-of relationship can be entirely proper, but should be modeled by c ompositionor, in special cases only, by nonpublic inheritance (see Item 34).

Put another way: When dynamic polymorphism is correct and appropriate, composition is selfish; inheritance is generous.

A new derived class is a new special case of an existing general abstraction. Existing (dynamically) polymorphic code that uses a `Base&` or `Base*` by calling `Base`'s virtual functions should be able to seamlessly use objects of `MyNewDerivedType` that inherits from `Base`. The new derived type adds new functionality to the existing code, which does not need to be changed but can seamlessly increase its functionality when new derived objects are plugged in.

New requirements should naturally be met by new code; new requirements should not cause rework on existing code. (See Item 36)

Before object orientation, it has always been easy for new code to call existing code. Public inheritance specifically makes it easier for existing code to seamlessly and safely call new code. (So do templates, which provide static polymorphism that can blend well with dynamic polymorphism; see Item 64)

Exceptions

Policy classes and mixins add behavior by public inheritance, but this is not abusing public inheritance to model is-implemented-in-terms-of.

References

[Cargill92] pp. 19-20 ?[Cline99

38. Practice safe overriding

Summary

Override responsibly: When overriding a virtual function, preserve substitutability; in particular, observe the function's pre- and post-conditions in the base class. Don't change default arguments of virtual functions. Prefer explicitly redeclaring overrides as `virtual`. Beware of hiding overloads in the base class.

Discussion

Although derived classes usually add more state (i.e., data members), they model *subsets*, not supersets, of their base classes. In correct inheritance, a derived class models a special case of a more general base concept (see Item 37).

This has direct consequences for correct overriding: Respecting the inclusion relationship implies substitutabilityoperations that apply to entire sets should apply to any of their subsets as well. After the base class guarantees the preconditions and postconditions of an operation, any derived class must respect those guarantees. An override can ask for *less* and provide *more*, but it must never require more or promise less because that would break the contract that was promised to calling code.

Defining a derived override that can fail (e.g., throws an exception; see Item 70) is correct only if the base class did not advertise that the operation always succeeds. For example, say **`Employee`** offers a virtual member function **`GetBuilding`** intended to return an encoding of the building where the **`Employee`** works. What if we want to write a **`RemoteContractor`** derived class that overrides **`GetBuilding`** to sometimes throw an exception or return a null building encoding? That is valid only if **`Employee`**'s documentation specifies that **`GetBuilding`** might fail and **`RemoteContractor`** reports the failure in an **`Employee`**-documented way.

When overriding, never change default arguments. They are not part of the function's signature, and client code will unwittingly pass different arguments to the function, depending on what node of the hierarchy they have access to. Consider:

```
class Base {
```

Examples

Example: ***`Ostrich`*****.** If class **`Bird`** defines the virtual function **`Fly`** and you derive a new class **`Ostrich`** (a notoriously flightless bird) from **`Bird`**, how do you implement **`Ostrich::Fly`**? The answer is, "It depends." If **`Bird::Fly`** guarantees success (i.e., provides the no-fail guarantee; see Item 71) because flying is an essential part of the **`Bird`** model, then **`Ostrich`** is not an adequate implementation of that model.

References

[Dewhurst03

39. Consider making virtual functions nonpublic, and public functions nonvirtual

Summary

In base classes with a high cost of change (particularly ones in libraries and frameworks): Prefer to make public functions nonvirtual. Prefer to make virtual functions private, or protected if derived classes need to be able to call the base versions. (Note that this advice does not apply to destructors; see Item 50.)

Discussion

Most of us have learned through bitter experience to make class members private by default unless we really need to expose them. That's just good encapsulation. This wisdom is applied most frequently to data members (see Item 41), but it applies equally to all members, including virtual functions.

Particularly in OO hierarchies that are expensive to change, prefer full abstraction: Prefer to make public functions nonvirtual, and prefer to make virtual functions private (or protected if derived classes need to be able to call the base versions). This is the Nonvirtual Interface (NVI) pattern. (NVI is similar to other patterns, notably Template Method [Gamma95], but has a distinct motivation and purpose.)

A public virtual function inherently has two different and competing responsibilities, aimed at two different and competing audiences:

- *It specifies interface:* Being public, it is directly part of the interface the class presents to the rest of the world.
- *It specifies implementation detail:* Being virtual, it provides a hook for derived classes to replace the base implementation of that function (if any); it is a customization point.

Because these two responsibilities have different motives and audiences, they can be (and frequently are) in conflict, and then one function by definition cannot fulfill both responsibilities perfectly. That a public virtual function inherently has two significantly different jobs and two competing audiences is a sign that it's not separating concerns wellincluding that it is inherently violating Items 5 and 11and that we should consider a different approach.

By separating public functions from virtual functions, we achieve the following significant benefits:

- *Each interface can take its natural shape:* When we separate the public interface from the customization interface, each can easily take the form it naturally wants to take instead of trying to find a compromise that forces them to look identical. Often, the two interfaces want different numbers of functions and/or different parameters; for example, outside callers may call a single public `Process` function that performs a logical unit of work, whereas customizers may prefer to override only certain parts of the processing, which is naturally modeled by independently overridable virtual functions (e.g., `DoProcessPhase1`, `DoProcessPhase2`) so that derived classes aren't forced to override everything. (This latter example specifically is the Template Method pattern arrived at from a different route.)
- *The base class is in control:* The base class is now in complete control of its interface and policy and can enforce interface preconditions and postconditions (see Items 14 and 68), insert instrumentation, and do any similar work all in a single convenient reusable placethe nonvirtual interface function. This "prefactoring" for separation thus promotes good class design.
- *The base class is robust in the face of change:* We are free to change our minds later and add pre- and postcondition checking, or separate processing into more steps, or refactor, or implement a fuller interface/implementation separation using the Pimpl idiom (see Item 43), or make other modifications to the base class's customizability, without affecting any of the code that uses or inherits from the class. Note that it is much easier to start with NVI (even if the public function is just a one-line passthrough to the virtual function) and then later add checking or instrumentation, because that can then be done without breaking any code that uses or inherits from the class. It is much harder to start with a public virtual function and later have to break it apart, which necessarily breaks either the code that uses the class or the code that inherits from the class, depending on whether we choose to keep the original function as the virtual function or as the public function, respectively.

See also Item 54.

Exceptions

NVI does not apply to destructors because of their special order of execution (see Item 50).

NVI does not directly support covariant return types for callers. If you need covariance that is visible to calling code without using `dynamic_cast` downcasts (see also Item 93), it's easier to make the virtual function public.

References

[Allison98

40. Avoid providing implicit conversions

Summary

Not all change is progress: Implicit conversions can often do more damage than good. Think twice before providing implicit conversions to and from the types you define, and prefer to rely on explicit conversions (`explicit` constructors and named conversion functions).

Discussion

Implicit conversions have two main problems:

- They can fire in the most unexpected places.
- They don't always mesh well with the rest of the language.

Implicitly converting constructors (constructors that can be called with one argument and are not declared **`explicit`**) interact poorly with overloading and foster invisible temporary objects that pop up all over. Conversions defined as member functions of the form **`operator T`** (where **`T`** is a type) are no betterthey interact poorly with implicit constructors and can allow all sorts of nonsensical code to compile. Examples are embarrassingly numerous (see References). We mention only two (see Examples).

In C++, a conversion sequence can include at most one user-defined conversion. However, when built-in conversions are added to the mix, the results can be extremely confusing. The solution is simple:

- By default, write **`explicit`** on single-argument constructors (see also Item 54):
-
- `class Widget {`

Examples

Example 1: Overloading. Say you have a **`Widget::Widget( unsigned int )`** that can be invoked implicitly, and a **`Display`** function overloaded for **`Widget`**s and **`double`**s. Consider the following overload resolution surprise:

```
void Display( double );                // displays a double
void Display( const Widget& );         // displays a Widget

Display( 5 );                          // oops: creates and displays a Widget
```

Example 2: Errors that work. Say you provide **`operator const char*`** for a **`String`** class:

```
class String {
```

Exceptions

When used sparingly and with care, implicit conversions can make calling code short and intuitive. The standard **std::string** defines an implicit constructor that takes a **const char*** . This works fine because the designers took some precautions:

- There is no automatic conversion *to* **const char***; that conversion is provided through two named functions, **c_str** and **data**.
- All comparison operators defined for **std::string** (e.g., **==**, **!=**, **<**) are overloaded to compare a **const char*** and a **std::string** in any order (see Item 29). This avoids the creation of hidden temporary variables.

Even so, there can still be some weirdness with overloaded functions:

```
void Display( int );
void Display( std::string );

Display( NULL );                     // calls Display(int)
```

This result might be surprising. (Incidentally, if it did call **Display(std::string)**, the code would have exhibited undefined behavior because it's illegal to construct a **std::string** from a null pointer, but its constructor isn't required to check for the null.)

References

[Dewhurst03

41. Make data members private, except in behaviorless aggregates (C-style `struct`s)

Summary

They're none of your caller's business: Keep data members private. Only in the case of simple C-style `struct` types that aggregate a bunch of values but don't pretend to encapsulate or provide behavior, make all data members public. Avoid mixes of public and nonpublic data, which almost always signal a muddled design.

Discussion

Information hiding is key to good software engineering (see Item 11). Prefer making all data members private; private data is the best means that a class can use to preserve its invariants now, and to keep preserving them in the face of future changes.

Public data is bad if a class models an abstraction and must therefore maintain invariants. Having public data means that part of your class's state can vary uncontrollably, unpredictably, and asynchronously with the rest of its state. It means that an abstraction is sharing responsibility for maintaining one or more invariants with the unbounded set of all code that uses the abstraction, and that is obviously, fundamentally, and indefensibly flawed. Reject such designs outright.

Protected data has all the drawbacks of public data, because having protected data still means that an abstraction is sharing the responsibility of maintaining some invariant with an unbounded set of codein this case, with the unbounded set of current and future derived classes. Further, any code can read and modify protected data as easily as public data by deriving a new class and using that to get at the data.

Mixing public and nonpublic data members in the same class is confusing and inconsistent. Private data demonstrates that you have invariants and some intent to preserve them; mixing it with public data means a failure to decide clearly whether the class really is supposed to be an abstraction or not.

Nonprivate data members are almost always inferior to even simple passthrough get/set functions, which allow for robust versioning.

Consider hiding a class's private members using the Pimpl idiom. (See Item 43)

Examples

Example 1: Proper encapsulation. Most classes (e.g., `Matrix`, `File`, `Date`, `BankAccount`, `Security`) should have all private data members and expose adequate interfaces. Allowing calling code to manipulate their internals directly would directly work against the abstraction they provide and the invariants they must sustain.

A `Node` aggregate, as commonly used in the implementation of a `List` class, typically contains some data and two pointers to `Node:next_` and `prev_.Node` 's members don't need to be hidden from `List`. But now consider Example 3.

Example 2: `treeNode`. Consider a `TRee<T>` container implemented in terms of `treeNode<T>`, an aggregate used within `TRee` that holds previous//pointers and a `T` object payload. `treeNode`'s members can all be public because they don't need to be hidden from `TRee`, which directly manipulates them. But `tree` should hide `treeNode` altogether (e.g., as a private nested class, or defined only in `tree`'s implementation file), because it is an internal detail of `tree` that callers shouldn't depend on or manipulate. Finally, `TRee` does *not* hide the contained `T` objects, because the payload is the caller's responsibility; containers use the iterator abstraction to expose the contained objects while hiding internal structures.

Example 3: Getters and setters. If there is no better domain abstraction available, public and protected data members (e.g., `color`) can at least be made private and hidden behind get and set functions (e.g., `GetColor`, `SetColor`); these provide a minimal abstraction and robust versioning.

Using functions raises the level of discourse about "color" from that of a concrete state to that of an abstract state that we are free to implement as we want: We can change to an internal color encoding other than `int`, add code to update the display when changing color, add instrumentation, and make many other changes without breaking calling code. At worst, callers just recompile (i.e., we preserve source-level compatibility); at best, they don't have to recompile or relink at all (if the change also preserves binary compatibility). Neither source nor binary compatibility is possible for such changes if the starting design has a public `color` member variable to which calling code becomes tightly coupled.

Exceptions

Get/set functions are useful, but a class consisting mostly of gets/sets is probably poorly designed: Decide whether it wants to provide an abstraction or be a `struct`.

Value aggregates (also called "C-style `struct`s") simply keep a bunch of data together but do not actually add significant behavior or attempt to model an abstraction and enforce invariants; they are not meant to be abstractions. Their data members should all be public because the data members *are* the interface. For example, `std::pair<T,U>` is used by the standard containers to aggregate two otherwise unrelated elements of type `T` and `U`, and `pair` itself doesn't add behavior or invariants.

References

[Dewhurst03

42. Don't give away your internals

Summary

Don't volunteer too much: Avoid returning handles to internal data managed by your class, so clients won't uncontrollably modify state that your object thinks it owns.

Discussion

Consider:

```
class Socket {
public:
```

Exceptions

Sometimes classes must provide access to internal handles for compatibility reasons, such as interfacing with legacy code or other systems. For example, **std::basic_string** offers access to its internal handle via the **data** and the **c_str** member functions for compatibility with functions that expect C-style pointersbut that presumably do not store those pointers or try to write through them! Such backdoor access functions are a necessary evil that should be used rarely and cautiously, and the conditions under which the handle remains valid must be carefully documented.

References

[C++03

43. Pimpl judiciously

Summary

Overcome the language's separation anxiety: C++ makes private members inaccessible, but not invisible. Where the benefits warrant it, consider making private members truly invisible using the Pimpl idiom to implement compiler firewalls and increase information hiding. (See Items 11 and 41.)

Discussion

When it makes sense to create a "compiler firewall" that completely insulates calling code from a class's private parts, use the Pimpl idiom: Hide them behind an opaque pointer (a pointer, preferably an appropriate smart pointer, to a class that is declared but not yet defined). For example:

```
class Map {
```

Exceptions

Only add complexity, including Pimpls, when you know you benefit from the extra level of indirection. (See Items 6 and 8.)

References

[Coplien92

44. Prefer writing nonmember nonfriend functions

◂ PREVIOUS NEXT ▸

Summary

Avoid membership fees: Where possible, prefer making functions nonmember nonfriends.

◂ PREVIOUS NEXT ▸

Discussion

Nonmember nonfriend functions improve encapsulation by minimizing dependencies: The body of the function cannot come to depend on the nonpublic members of the class (see Item 11). They also break apart monolithic classes to liberate separable functionality, further reducing coupling (see Item 33). They improve genericity, because it's hard to write templates that don't know whether or not an operation is a member for a given type (see Item 67).

Use this algorithm to determine whether a function should be a member and/or friend:

```
// If you have no choice then you have no choice; make it a member if it
must be:

If the function is one of the operators =, ->, [], or (), which must be members:

    Make it a member.

// If it can be a nonmember nonfriend, or benefits from being a nonmember
friend, do it:

Else if: a) the function needs a different type as its left-hand argument (as do
operators >> or <<, for example); or b) it needs type conversions on its
leftmost argument; or c) it can be implemented using the class's public
interface alone:

    Make it a nonmember (and friend if needed in cases a) and b) ).

    If it needs to behave virtually:

        Add a virtual member function to provide the
        virtual behavior, and implement the nonmember in
        terms of that.

Else: Make it a member.
```

Examples

Example: ***`basic_string`*****.** The standard **`basic_string`** is a needlessly monolithic class with 103 member functionsof which 71 could be written as nonmember nonfriends without loss of efficiency. Many of them duplicate functionality already available as algorithms, or are themselves algorithms that would be useful more widely if they weren't buried inside **`basic_string`**. (See Items 5 and 32, and [Sutter04].)

References

[*Lakos96*

45. Always provide `new` and `delete` together

Summary

They're a package deal: Every class-specific overload `void* operator new(`***`parms`***`)` must be accompanied by a corresponding overload `void operator delete(void*, `***`parms`***`)`, where ***`parms`*** is a list of extra parameter types (of which the first is always `std::size_t`). The same goes for the array forms `new[]` and `delete[]`.

Discussion

You rarely need to provide a custom **`new`** or **`delete`**, but if you need one you usually need both. If you de fine a class-specific **`T::operator new`** to do some special allocation, it's very likely you need to define a class-specific **`T::operator delete`** as well to do the corresponding special deallocation.

That much may be somewhat basic, but there is a subtler reason for this Item: The compiler might be yearning for an overload of **`T::operator delete`** even when you never actually invoke it. That's why you always need to provide **`operator new`** and **`operator delete`** (and **`operator new[]`** and **`operator delete[]`**) in pairs.

Say you define a class with customized allocation:

```
class T {
```

Exceptions

The in-place form of **`operator new`**

```
void* T::operator new(size_t, void* p) { return p; }
```

does not need a corresponding **`operator delete`** because there is no real allocation going on. All compilers we tested issue no spurious warnings concerning the absence of **`void T::operator delete(void*, size_t, void*)`**.

References

[C++03

46. If you provide any class-specific `new`, provide all of the standard forms (plain, in-place, and `nothrow`)

Summary

Don't hide good `news`: If a class defines any overload of `operator new`, it should provide overloads of all three of plain, in-place, and non-throwing `operator new`. If you don't, they'll be hidden and unavailable to users of your class.

Discussion

You rarely need to provide a custom **`new`** or **`delete`**, but if you need them you usually don't want to hide the built-in signatures.

In C++, after you define a name in a scope (e.g., in a class scope), it will hide the same name in all enclosing scopes (e.g., in base classes or enclosing namespaces), and overloading never happens across scopes. And when said name is **`operator new`**, you need to be extra cautious lest you make life hard for your class's clients.

Say you define a class-specific **`operator new`**:

```
class C {
```

References

[*Dewhurst03*

Construction, Destruction, and Copying

Just because the standard provides a cliff in front of you, you are not necessarily required to jump off it.

Norman Diamond

There is enough to be said about the Big Four special member functions that you will probably be unsurprised to see that they rate having their own section. Herein we collect knowledge and best practices related to default construction, copy construction, copy assignment, and destruction.

One of the reasons you need to be careful with these functions is that, given half a chance, the compiler will write them for you. Another reason is that C++ by default treats classes as value-like types, but not all types are value-like (see Item 32). Know when to write (or disable) these special member functions explicitly, and follow the rules and guidelines in this section to help ensure that you get correct, extensible, and error-safe code.

Our vote for the most valuable Item in this section goes to Item 51: Destructors, deallocation, and **`swap`** never fail.

47. Define and initialize member variables in the same order

Summary

Agree with your compiler: Member variables are always initialized in the order they are declared in the class definition; the order in which you write them in the constructor initialization list is ignored. Make sure the constructor code doesn't confusingly specify a different order.

Discussion

Consider the code below:

```
class Employee {
 string email_, firstName_, lastName_;

public:
 Employee( const char* firstName, const char* lastName )
  : firstName_(firstName), lastName_(lastName)
  , email_(firstName_ + "." + lastName_ + "@acme.com") {}
};
```

This code harbors a bug that's as pernicious as it is hard to spot. Because **`email_`** is declared before **`first_`** and **`last_`** in the class definition, it will be initialized first and will attempt to use the other not-yet-initialized fields. Worse, if the constructor's definition is in a separate file, the long-distance influence that the order of member variable declarations has over the constructor's correctness will be even harder to spot.

The reason for this language design decision is to ensure there is a unique order to destroy members; otherwise, the destructor would have to destroy objects in different orders, depending on the constructor that built the object. The bookkeeping overhead for that was deemed unacceptable.

The solution is to always write member initializers in the same order in which the members are declared. That way, any illegal dependency will become obvious. Better yet, strive to not have one member's initialization depend on other members.

Many compilers (but not all) will issue a warning if you break this rule.

◀ PREVIOUS NEXT ▶

References

[Cline99

48. Prefer initialization to assignment in constructors

Summary

Set once, use everywhere: In constructors, using initialization instead of assignment to set member variables prevents needless run-time work and takes the same amount of typing.

Discussion

Constructors generate initialization code under the covers. Consider:

```
class A {
  string s1_, s2_;

public:
  A() {s1_ = "Hello, "; s2_ = "world"; }
};
```

In reality, constructor's code is generated as if you wrote:

```
A() : s1_(), s2_() {s1 = "Hello, "; s2 = "world"; }
```

That is, the objects not explicitly initialized by you are automatically initialized using their default constructors, and then assigned to using their assignment operators. Most often, the assignment operator of a nontrivial object needs to do slightly more than a constructor because it needs to deal with an already-constructed object.

Say what you mean: Initialize member variables in the initializer list, with code that better expresses intent and in addition, as a perk, is usually smaller and faster:

```
A() : s1_("Hello, "), s2_("world") {}
```

This isn't premature optimization; it's avoiding premature pessimization. (See Item 9)

Exceptions

Always perform unmanaged resource acquisition, such as a `new` expression whose result is not immediately passed to a smart pointer constructor, in the constructor body and not in initializer lists (see [Sutter02]). Of course, it's better to not have such unsafe unowned resources in the first place (see Item 13).

References

[Dewhurst03

49. Avoid calling virtual functions in constructors and destructors

Team LiB ◂ PREVIOUS NEXT ▸

Summary

Virtual functions only "virtually" always behave virtually: Inside constructors and destructors, they don't. Worse, any direct or indirect call to an unimplemented *pure virtual* function from a constructor or destructor results in undefined behavior. If your design wants virtual dispatch into a derived class from a base class constructor or destructor, you need other techniques such as post-constructors.

Team LiB ◂ PREVIOUS NEXT ▸

Discussion

In C++, a complete object is constructed one base class at a time.

Say we have a base class `B` and a class `D` derived from `B`. When constructing a `D` object, while executing `B`'s constructor, the dynamic type of the object under construction is `B`. In particular, a call to a virtual function `B::Fun` will hit `B`'s definition of `Fun`, regardless of whether `D` overrides it or not; and that's a good thing, because calling a `D` member function when the `D` object's members haven't even been initialized yet would lead to chaos. Only after the construction of `B` has completed is `D`'s constructor body executed and its identity as a `D` established. As a rule of thumb, keep in mind that during `B`'s construction there is no way to tell whether the `B` is a standalone object or a base part of some other further-derived object; virtually-acting virtual functions would be such a "way."

To add insult to injury, a call from a constructor to a pure virtual function that isn't defined at all has, well, undefined behavior. Such code is therefore not only confusing, but it is also more fragile in face of maintenance.

On the other hand, some designs ask for "post-construction," that is, a virtual function that must be invoked right after the full object has been constructed. Some techniques for this are shown in the References. Here's a non-exhaustive list of options:

- *Pass the buck:* Just document that user code must call the post-initialization function right after constructing an object.
- *Post-initialize lazily:* Do it during the first call of a member function. A Boolean flag in the base class tells whether or not post-construction has taken place yet.
- *Use virtual base class semantics:* Language rules dictate that the constructor most-derived class decides which base constructor will be invoked; you can use that to your advantage. (See [Taligent94].)
- *Use a factory function:* This way, you can easily force a mandatory invocation of a post-constructor function. (See Examples.)

No post-construction technique is perfect. The worst dodge the whole issue by simply asking the caller to invoke the post-constructor manually. Even the best require a different syntax for constructing objects (easy to check at compile time) and/or cooperation from derived class authors (impossible to check at compile time).

Examples

Example: Using a factory function to insert a post-constructor call. Consider:

```
class B {                                    // hierarchy root
protected:
  B() {/* ... */ }

  virtual void PostInitialize() {/* ... */ }  // called right after
construction

public:
  template<class T>
  static shared_ptr<T> Create() {          // interface for creating objects
    shared_ptr<T> p( new T );
    p->PostInitialize();
    return p;
  }
};

class D : public B {/* ... */ };              // some derived class

shared_ptr<D> p = D::Create<D>();          // creating a D object
```

This rather fragile design sports the following tradeoffs:

- Derived classes such as **D** must not expose a public constructor. Otherwise, **D**'s users could create **D** objects that don't invoke **PostInitialize**.
- Allocation is limited to **operator new.B** can, however, override **new** (see Items 45 and 46).
- **D** must define a constructor with the same parameters that **B** selected. Defining several overloads of **Create** can assuage this problem, however; and the overloads can even be templated on the argument types.
- If the requirements above are met, the design guarantees that **PostInitialize** has been called for any fully constructed **B**-derived object. **PostInitialize** doesn't need to be virtual; it can, however, invoke virtual functions freely.

◀ PREVIOUS NEXT ▶

References

[Alexandrescu01

50. Make base class destructors public and virtual, or protected and nonvirtual

Summary

To delete, or not to delete; that is the question: If deletion through a pointer to a base **`Base`** should be allowed, then **`Base`**'s destructor must be public and virtual. Otherwise, it should be protected and nonvirtual.

Discussion

This simple guideline illustrates a subtle issue and reflects modern uses of inheritance and object-oriented design principles.

For a base class `Base`, calling code might try to `delete` derived objects through pointers to `Base`. If `Base`'s destructor is public and nonvirtual (the default), it can be accidentally called on a pointer that actually points to a derived object, in which case the behavior of the attempted deletion is undefined. This state of affairs has led older coding standards to impose a blanket requirement that all base class destructors must be virtual. This is overkill (even if it is the common case); instead, the rule should be to make base class destructors virtual if and only if they are public.

To write a base class is to define an abstraction (see Items 35 through 37). Recall that for each member function participating in that abstraction, you need to decide:

- Whether it should behave virtually or not.
- Whether it should be publicly available to all callers using a pointer to `Base` or else be a hidden internal implementation detail.

As described in Item 39, for a normal member function, the choice is between allowing it to be called via a `Base*` nonvirtually (but possibly with virtual behavior if it invokes virtual functions, such as in the NVI or Template Method patterns), virtually, or not at all. The NVI pattern is a technique to avoid public virtual functions.

Destruction can be viewed as just another operation, albeit with special semantics that make nonvirtual calls dangerous or wrong. For a base class destructor, therefore, the choice is between allowing it to be called via a `Base*` virtually or not at all; "nonvirtually" is not an option. Hence, a base class destructor is virtual if it can be called (i.e., is public), and nonvirtual otherwise.

Note that the NVI pattern cannot be applied to the destructor because constructors and destructors cannot make deep virtual calls. (See Items 39 and 55.)

Corollary: Always write a destructor for a base class, because the implicitly generated one is public and nonvirtual.

Examples

Either clients should be able to `delete` polymorphically using a pointer to `Base`, or they shouldn't. Each alternative implies a specific design:

- *Example 1: Base classes with polymorphic deletion.* If polymorphic deletion should be allowed, the destructor must be public (else calling code can't call it) and it must be virtual (else calling it results in undefined behavior).

- *Example 2: Base classes without polymorphic deletion.* If polymorphic deletion shouldn't be allowed, the destructor must be nonpublic (so that calling code can't call it) and should be nonvirtual (because it needn't be virtual).

Policy classes are frequently used as base classes for convenience, not for polymorphic behavior. It is recommended to make their destructors protected and nonvirtual.

Exceptions

Some component architectures (e.g., COM and CORBA) don't use a standard deletion mechanism, and foster different protocols for object disposal. Follow the local patterns and idioms, and adapt this guideline as appropriate.

Consider also this rare case:

- **B** is both a base class and a concrete class that can be instantiated by itself (and so the destructor must be public for **B** objects to be created and destroyed).
- Yet **B** also has no virtual functions and is not meant to be used polymorphically (and so although the destructor is public it does not need to be virtual).

Then, even though the destructor has to be public, there can be great pressure to not make it virtual because as the first virtual function it would incur all the run-time type overhead when the added functionality should never be needed.

In this rare case, you could make the destructor public and nonvirtual but clearly document that further-derived objects must not be used polymorphically as **B**'s. This is what was done with **std::unary_function**.

In general, however, avoid concrete base classes (see Item 35). For example, **unary_function** is a bundle-of-typedefs that was never intended to be instantiated standalone. It really makes no sense to give it a public destructor; a better design would be to follow this Item's advice and give it a protected nonvirtual destructor.

References

[Cargill92] pp. 77-79, 207 ?[Cline99

51. Destructors, deallocation, and `swap` never fail

Summary

Everything they attempt shall succeed: Never allow an error to be reported from a destructor, a resource deallocation function (e.g., `operator delete`), or a swap function. Specifically, types whose destructors may throw an exception are flatly forbidden from use with the C++ standard library.

Discussion

These are key functions that must not fail because they are necessary for the two key operations in transactional programming: to back out work if problems are encountered during processing, and to commit work if no problems occur. If there's no way to safely back out using no-fail operations, then no-fail rollback is impossible to implement. If there's no way to safely commit state changes using a no-fail operation (notably, but not limited to, swap), then no-fail commit is impossible to implement.

Consider the following advice and requirements found in the C++ Standard:

> *If a destructor called during stack unwinding exits with an exception,* **`terminate`** *is called (15.5.1). So destructors should generally catch exceptions and not let them propagate out of the destructor.*
>
> [C++03

References

[C++03

52. Copy and destroy consistently

Summary

What you create, also clean up: If you define any of the copy constructor, copy assignment operator, or destructor, you might need to define one or both of the others.

Discussion

If you need to define any of these three functions, it means you need it to do more than its default behaviorand the three are asymmetrically interrelated. Here's how:

- *If you write/disable either the copy constructor or the copy assignment operator, you probably need to do the same for the other:* If one does "special" work, probably so should the other because the two functions should have similar effects. (See Item 53, which expands on this point in isolation.)
- *If you explicitly write the copying functions, you probably need to write the destructor:* If the "special" work in the copy constructor is to allocate or duplicate some resource (e.g., memory, file, socket), you need to deallocate it in the destructor.
- *If you explicitly write the destructor, you probably need to explicitly write or disable copying:* If you have to write a nontrivial destructor, it's often because you need to manually release a resource that the object held. If so, it is likely that those resources require careful duplication, and then you need to pay attention to the way objects are copied and assigned, or disable copying completely.

In many cases, holding properly encapsulated resources using RAII "owning" objects can eliminate the need to write these operations yourself. (See Item 13.)

Prefer compiler generated special members; only these can be classified as "trivial," and at least one major STL vendor heavily optimizes for classes having trivial special members. This is likely to become common practice.

Exceptions

When any of the special functions are declared only to make them private or virtual, but without special semantics, it doesn't imply that the others are needed.

In rare cases, classes that have members of strange types (e.g., references, **`std::auto_ptr`** s) are an exception because they have peculiar copy semantics. In a class holding a reference or an **`auto_ptr`**, you likely need to write the copy constructor and the assignment operator, but the default destructor already does the right thing. (Note that using a reference or **`auto_ptr`** member is almost always wrong.)

References

[Cline99

53. Explicitly enable or disable copying

Summary

Copy consciously: Knowingly choose among using the compiler-generated copy constructor and assignment operator, writing your own versions, or explicitly disabling both if copying should not be allowed.

Discussion

A common mistake (and not only among beginners) is to forget to think about copy and assignment semantics when defining a class. This happens often with small helper classes such as those meant for RAII support (see Item 13).

Ensure that your class provides sensible copying, or none at all. The choices are:

- *Explicitly disable both:* If copying doesn't make sense for your type, disable both copy construction and copy assignment by declaring them as private unimplemented functions:
-
- `class T {`

References

[Dewhurst03

54. Avoid slicing. Consider `Clone` instead of copying in base classes

Summary

Sliced bread is good; sliced objects aren't: Object slicing is automatic, invisible, and likely to bring wonderful polymorphic designs to a screeching halt. In base classes, consider disabling the copy constructor and copy assignment operator, and instead supplying a virtual `Clone` member function if clients need to make polymorphic (complete, deep) copies.

Discussion

When you build class hierarchies, usually the intent is to get polymorphic behavior. You want objects, once created, to preserve their type and identity. This goal conflicts with C++'s usual object copy semantics because the copy constructor is not virtual and cannot be made virtual. Consider:

```
class B {
```

Exceptions

Some designs might require that the copy constructors of base classes are left public (e.g., when part of your hierarchy is a third-party library). In that case, prefer passing by (smart) pointer to passing by reference; as Item 25 shows, passing by pointer is much less vulnerable to slicing and unwanted temporary construction.

References

[Dewhurst03

55. Prefer the canonical form of assignment

Summary

Your assignment: When implementing `operator=`, prefer the canonical formnonvirtual and with a specific signature.

Discussion

Prefer to declare copy assignment for a type `T` with one of the following signatures (see [Stroustrup00] and [Alexandrescu03a]):

```
T& operator=( const T& );          // classic
T& operator=( T );                 // potentially optimizer-friendly (see
Item 27)
```

Settle for the second version if you would need to make a copy of the argument inside your operator anyway, such as the **`swap`**-based idiom featured in Item 56.

Avoid making any assignment operator virtual (see [Meyers96

References

[Alexandrescu03a] ?[Cargill92] pp41-42, 95 ?[Cline99

56. Whenever it makes sense, provide a no-fail `swap` (and provide it correctly)

Summary

`swap` is both a lightweight and a workhorse: Consider providing a `swap` function to efficiently and infallibly swap the internals of this object with another's. Such a function can be handy for implementing a number of idioms, from smoothly moving objects around to implementing assignment easily to providing a guaranteed commit function that enables strongly error-safe calling code. (See also Item 51.)

Discussion

A `swap` function typically looks like this, where `U` is some user-defined type:

```
class T {
```

Exceptions

Swapping is valuable for classes with value semantics. It is less often useful for base classes because you use those classes through pointers anyway. (See Items 32 and 54.)

References

[C++03

Namespaces and Modules

Systems have sub-systems and sub-systems have sub-systems and so on ad infinitumwhich is why we're always starting over.

Alan Perlis

The namespace is an important tool for managing names and reducing name collisions. So is a module, which is additionally an important tool for managing releases and versioning. We define a module as any cohesive unit of release (see Item 5) maintained by the same person or team; typically, a module is also consistently compiled with the same compiler and switch settings. Modules exist at many levels of granularity that range widely in size; a module can be as small as a single object file that delivers a single class, to a single shared or dynamic library generated from multiple source files whose contents form a subsystem inside a larger application or are released independently, to as large as a huge library composed of many smaller modules (e.g., shared libraries, DLLs, or other libraries) and containing thousands of types. Even though such entities as shared libraries and dynamic libraries are not mentioned directly in the C++ Standard, C++ programmers routinely build and use libraries, and good modularization is a fundamental part of successful dependency management (see for example Item 11).

It's hard to imagine a program of any significant size that doesn't use both namespaces and modules. In this section, we cover basic guidelines on using these two related management and bundling tools with a view to how they interact well or badly with the rest of the C++ language and run-time environment. These rules and guidelines show how to maximize the "well" and avoid the "badly."

Our vote for the most valuable Item in this section goes to Item 58: Keep types and functions in separate namespaces unless they're specifically intended to work together.

57. Keep a type and its nonmember function interface in the same namespace

Summary

Nonmembers are functions too: Nonmember functions that are designed to be part of the interface of a class `X` (notably operators and helper functions) must be defined in the same namespace as the `X` in order to be called correctly.

Discussion

Both public member functions and nonmember functions form part of the public interface of a class. The Interface Principle states: For a class **X**, all functions (including nonmember functions) that both "mention" **X** and are "supplied with" **X** in the same namespace are logically part of **X**, because they form part of **X**'s interface. (See Item 44 and [Sutter00].)

The C++ language is explicitly designed to enforce the Interface Principle. The reason why argument-dependent lookup (ADL, also known as Koenig lookup) was added to the language was to ensure that code that uses an object **x** of type **X** can use its nonmember function interface (e.g., **cout << x**, which invokes the nonmember **operator<<** for **X**) as easily as it can use member functions (e.g., **x.f()**, which requires no special lookup because **f** is clearly to be looked up in the scope of **X**). ADL ensures that nonmember functions that take **X** objects and that are supplied with **X**'s definition can participate as first-class members of **X**'s interface, just like **X**'s direct member functions naturally do.

In particular, the primary motivating example for ADL was the case where **X** is **std::string** (see [Sutter00]).

Consider a class **X**, defined in namespace **N**:

```
class X {
public:
  void f();
};

X operator+( const X&, const X& );
```

Callers will typically want to write code like this, where **x1**, **x2** , and **x3** are objects of type **X**:

```
x3 = x1 + x2;
```

If the **operator+** is declared in the same namespace as **X**, there's no problem, and such code always just works, because the supplied **operator+** will be looked up using ADL.

If the **operator+** is not declared in the same namespace as **X**, the caller's code fails to work. The caller has two workarounds to make it work. The first is to use explicit qualification:

```
x3 = N::operator+( x1, x2 );
```

This is deplorable and shameful because it requires the user to give up natural operator syntax, which is the point of operator overloading in the first place. The only other option is to write a **using** statement:

```
using N::operator+;
// or: using namespace N;
x3 = x1 + x2;
```

Writing either of the indicated **using** statements is perfectly acceptable (see Item 59), but the caller doesn't have to jump through these hoops when the author of **X** does the right thing and puts **operator+** for **X** objects into the same namespace as **X**.

For the flip side of this issue, see Item 58).

Examples

Example 1: Operators. The streaming `operator<<` and `operator>>` for objects of some class type `X` are perhaps the most compelling examples of functions that are clearly part of the interface of the class `X`, but which are always nonmember functions (this is of necessity, because the left-hand argument is a stream, not an `X`). The same argument applies to other nonmember operators on `X` objects. Make sure that your operators appear in the same namespace as the class on which they operate. When you have the option, prefer making operators and all other functions nonmember nonfriends (see Item 44).

Example 2: Other functions. If the author of `X` supplies named helper functions that take `X` objects, they should be supplied in the same namespace, otherwise calling code that uses `X` objects will not be able to use the named functions without explicit qualification or a `using` statement.

PREVIOUS NEXT

References

[Stroustrup00] ?§

58. Keep types and functions in separate namespaces unless they're specifically intended to work together

Summary

Help prevent name lookup accidents: Isolate types from unintentional argument-dependent lookup (ADL, also known as Koenig lookup), and encourage intentional ADL, by putting them in their own namespaces (along with their directly related nonmember functions; see Item 57). Avoid putting a type into the same namespace as a templated function or operator.

Discussion

By following this advice, you will avoid having to track down hard-to-diagnose errors in your code and avoid having to know about excessively subtle language details that you, well, should never have to know about.

Consider this actual example that was posted publicly to a newsgroup:

```
#include <vector>

namespace N {
  struct X {};

  template<typename T>
  int* operator+( T , unsigned ) {/* do something */}
}

int main() {
  std::vector<N::X> v(5);
  v[0] ;
}
```

The statement **`v[0];`** compiles on some standard library implementations but not on others. To make a very long story moderately less long (take a deep breath): The exceedingly subtle problem is that inside most implementations of **`vector<T>::operator[]`** lurks code like **`v.begin() + n`**, and the name lookup for that **`operator+`** function *might* reach out into the namespace (here **`N`**) of the type that **`vector`** is instantiated with (here **`X`**). Whether it reaches out into **`N`** like that depends on how **`vector<T>::iterator`** happens to be defined in that release of that standard library implementationbut if it does look into **`N`**, then it will find **`N::operator+`**. Finally, depending on the types involved, the compiler might just discover that **`N::operator+`** is a better match than the **`std::operator+`** for **`vector<T>::iterator`**s that was provided (and intended to be called) in that standard library implementation. (One way that the standard library implementation could protect itself from this is to not write code like **`v.begin() + n`** in that way, which injects an unintentional point of customization: Either arrange for **`v.begin()`**'s type to not depend in any way on the template parameter, or rewrite the call to **`operator+`** as a qualified call. See Item 65)

In short, you'll almost certainly never figure out what's going on from the error messageif you're lucky enough to get an error message, that is, because you might happen to hit the worst of all possible worlds where **`N::operator+`** is chosen but unfortunately turns out to be compilable, although completely unintended and wildly wrong.

If you think you haven't been bit by this, just think back: Can you remember a time when you wrote code that used the standard library (for example) and got mysterious and incom prehensible compiler errors? And you kept slightly rearranging your code and recompiling, and rearranging some more and compiling some more, until the mysterious compile errors went away, and then you happily continued onwith at best a faint nagging curiosity about why the compiler didn't like the only-ever-so-slightly different arrangement of the code you wrote at first? We've all had those days, and the odds are decent that the mystery culprit was some form of the aforementioned problem, where ADL pulled in names from other namespaces inappropriately just because types from those namespaces were being used nearby.

This problem is not unique to uses of the standard library. It can and does happen in C++ with the use of any type that is defined in the same namespace as functionsespecially templated functions, and most especially operatorsthat aren't specifically related to that type. Don't do that.

Bottom line: You shouldn't have to know this stuff. The easiest way to avoid this whole category of problems is to in general avoid putting nonmember functions that are not part of the interface of a type **`X`** into the same namespace as **`X`**, and especially never ever put templated functions or operators into the same namespace as a user-defined type.

Note: Yes, the C++ standard library puts algorithms and other function templates, such as

References

[Stroustrup00

59. Don't write namespace `using`s in a header file or before an `#include`

Summary

Namespace `using`s are for your convenience, not for you to inflict on others: Never write a `using` declaration or a `using` directive before an `#include` directive.

Corollary: In header files, don't write namespace-level `using` directives or `using` declarations; instead, explicitly namespace-qualify all names. (The second rule follows from the first, because headers can never know what other header `#include`s might appear after them.)

Discussion

In short: You can and should use namespace **`using`** declarations and directives liberally *in your implementation files after* ***`#include`*** *directives* and feel good about it. Despite repeated assertions to the contrary, namespace **`using`** declarations and directives are not evil and they do not defeat the purpose of namespaces. Rather, they are what make namespaces usable.

Namespaces deliver the powerful advantage of unambiguous name management. Most of the time, different programmers don't choose the very same name for a type or function; but on the off chance that they do so, and that the two pieces of code end up being used together, having those names in separate namespaces prevents them from colliding. (We don't, after all, want the global namespace pollution experienced by default in languages like C.) In the rare case when there is such an actual ambiguity, calling code can explicitly qualify a name to say which one it wants. But the vast majority of the time there is no ambiguity: And that is why namespace **`using`** declarations and directives are what make namespaces usable, because they greatly reduce code clutter by freeing you from having to tediously qualify every name every time (which would be onerous, and frankly people just won't put up with it) and still letting you qualify names only in those rare cases when you need to resolve an actual ambiguity.

But **`using`** declarations and directives are for your coding convenience, and you shouldn't use them in a way that affects someone else's code. In particular, don't write them anywhere they could be followed by someone else's code: Specifically, don't write them in header files (which are meant to be included in an unbounded number of implementation files, and you shouldn't mess with the meaning of that other code) or before an **`#include`** (you really don't want to mess with the meaning of code in someone else's header).

Most people understand viscerally why a **`using`** directive (e.g., **`using namespace A;`**) causes pollution when it can affect code that follows and that isn't aware of it: Because it imports one namespace wholesale into another, including even those names that haven't been seen yet, it's fairly obvious that it can easily change the meaning of code that follows.

But here's the common trap: Many people think that **`using`** *declarations* issued at namespace level (for example, **`using N::Widget;`**) are safe. They are not. They are at least as dangerous, and in a subtler and more insidious way. Consider:

```
// snippet 1
namespace A {
 int f(double);
}

// snippet 2
namespace B {
  using A::f;
  void g();
}

// snippet 3
namespace A {
 int f(int);
}

// snippet 4
void B::g() {
  f(1);                         // which overload is called?
}
```

The dangerous thing happening here is that the **`using`** declaration *takes a snapshot* of whatever entities named **`f`** in namespace **`A`** have been seen by the time the **`using`** declaration is encountered. So, from within **`B`**, which overloads are visible depends on where these code snippets exist and in what order they are combined. (At this point, your internal "but order dependencies are evil!" klaxon should be blaring.) The second overload, **`f(int)`**, would be a

Exceptions

Migrating a large project from an old, pre-ANSI/ISO implementation of the standard library (one that puts all of its symbols in the global namespace) to an updated one (where most everything is in namespace **`std`**) might force you to carefully put a **`using`** directive in a header file. The way to do that is described in [Sutter02].

References

[Stroustrup00

60. Avoid allocating and deallocating memory in different modules

Summary

Put things back where you found them: Allocating memory in one module and deallocating it in a different module makes your program fragile by creating a subtle long-distance dependency between those modules. They must be compiled with the same compiler version and same flags (notably debug vs. `NDEBUG`) and the same standard library implementation, and in practice the module allocating the memory had better still be loaded when the deallocation happens.

Discussion

Library writers want to improve the quality of their libraries, and as a direct consequence the internal data structures and algorithms used by the standard memory allocator can significantly vary from one version to the next. Furthermore, various compiler switches (e.g., turning debugging facilities on and off) can change the inner workings of the memory allocator significantly.

Therefore, make very few assumptions about deallocation functions (e.g., `::operator delete` or `std::free`) when you cross module boundariesespecially boundaries of modules that you can't guarantee will be compiled with the same C++ compiler and the same build options. Often, it is the case that various modules are in the same makefile and compiled with the same options, but comfort often leads to forgetfulness. Especially when it comes to dynamically linked libraries, large projects distributed across large teams, or the challenging "hot swapping" of modules, you should pay maximum attention to allocate and deallocate within the same module or subsystem.

A good way to ensure that deletion is performed by the appropriate function is to use the `shared_ptr` facility (see [C++TR104]). `shared_ptr` is a reference-counted smart pointer that can capture its "deleter" at construction time. The deleter is a function object (or a straight pointer to function) that performs deallocation. Because the said function object, or pointer to function, is part of the `shared_ptr` object's state, the module allocating the object can at the same time specify the deallocation function, and that function will be called correctly even if the point of deallocation is somewhere within another moduleadmittedly at a slight cost. (Correctness is worth the cost; see also Items 5, 6, and 8.) Of course, the original module must remain loaded.

References

[C++TR104]

61. Don't define entities with linkage in a header file

Summary

Repetition causes bloat: Entities with linkage, including namespace-level variables or functions, have memory allocated for them. Defining such entities in header files results in either link-time errors or memory waste. Put all entities with linkage in implementation files.

Discussion

While starting to use C++, we all learn quite quickly that header files like

```
// avoid defining entities with external linkage in a header
int fudgeFactor;
string hello("Hello, world!");
void foo() {
```

Exceptions

The following entities with external linkage can go in header files:

- *Inline functions:* These have external linkage, but the linker is guaranteed not to reject multiple copies. Other than that, they behave exactly like regular functions. In particular, the address of an inline function is guaranteed to be unique throughout a program.
- *Function templates:* Similar to inline functions, instantiations behave just like regular functions except that duplicate copies are acceptable (and had better be identical). Of course, a good compilation framework eliminates the unnecessary copies.
- *Static data members of class templates:* These can be particularly rough on the linker, but that's not your problemyou just define them in your header file and let your compiler and linker deal with it.

Also, a global data initialization technique known as "Schwarz counters" or "nifty counters" prescribes planting static (or unnamed-namespace) data in a header file. Jerry Schwarz made the technique popular by using it to initialize the standard I/O streams **`cin`**, **`cout`**, **`cerr`**, and **`clog`**.

◂ PREVIOUS NEXT ▸

References

[Dewhurst03

62. Don't allow exceptions to propagate across module boundaries

Summary

Don't throw stones into your neighbor's garden: There is no ubiquitous binary standard for C++ exception handling. Don't allow exceptions to propagate between two pieces of code unless you control the compiler and compiler options used to build both sides; otherwise, the modules might not support compatible implementations for exception propagation. Typically, this boils down to: Don't let exceptions propagate across module/subsystem boundaries.

Discussion

The C++ Standard does not specify the way exception propagation has to be implemented, and there isn't even a *de facto* standard respected by most systems. The mechanics of exception propagation vary not only with the operating system and compiler used, but also with the compiler options used to build each module of your application with a given compiler on a given operating system. Therefore, an application must prevent exception handling incompatibilities by shielding the boundaries of each of its major modules, meaning each unit for which the developer can ensure that the same compiler and options are consistently used.

At a minimum, your application must have catch-all

References

[Stroustrup00

Chapter 63. Use sufficiently portable types in a module's interface

Summary

Take extra care when living on the edge (of a module): Don't allow a type to appear in a module's external interface unless you can ensure that all clients understand the type correctly. Use the highest level of abstraction that clients can understand.

Discussion

The more widely distributed your library, and the less control you have over the build environment of all of its clients, the fewer the types that the library can reliably use in its external interface. Interfacing across modules involves binary data exchange. Alas, C++ doesn't specify standard binary interfaces; widely distributed libraries in particular might need to rely on built-in types like **`int`** and **`char`** to interface with the outer world. Even compiling the same type using different build options on the same compiler can cause binary-incompatible versions of the type.

Typically, either you control the compiler and options used to build the module and all its clients, and you can use any typeor you don't, and you can use only platform-provided types and C++ built-in types (even then, document the size and representation you expect for the latter). In particular, never mention standard library types in the interface of a module unless all other modules that use it will be compiled at the same time and with the same standard library source image.

There is a tradeoff between the problems of using types that can't be correctly understood by all clients, and the problems of using a low level of abstraction. Abstraction is important; if some clients understand only low-level types, and you must therefore use those, consider also supplying alternate operations that use higher-level types. Consider a **`SummarizeFile`** function that takes the file to be processed as a parameter. There are three common options for the parameter: It can be a **`char*`** that points to a C-style string containing the file's name; a **`string`** that containers the file's name; or an **`istream`** or a custom **`File`** object. Each of these choices is a tradeoff:

- *Option 1:* ***`char*`*.** Clearly the **`char*`** type is accessible to the widest audience of clients. Unfortunately, it is also the lowest-level option; in particular, it is less robust (e.g., the caller and callee must explicitly decide who allocates the memory and who deallocates it), more open to errors (e.g., the file might not exist), and less secure (e.g., to classic buffer overrun attacks).

- *Option 2:* ***`string`*.** The **`string`** type is accessible to the more restricted audience of clients that are written in C++ and compiled using the same standard library implementation, the same compiler, and compatible compiler settings. In exchange, it is more robust (e.g., callers and callees can be less explicit about memory management; but see Item 60) and more secure (e.g., **`string`** grows its buffer as needed, and is not inherently as susceptible to buffer overrun attacks). But this option is still relatively low-level, and thus open to errors that have to be checked for explicitly (e.g., the file might not exist).

- *Option 3:* ***`istream`* *or* *`File`*.** If you're going to jump to class types anyway, thereby requiring clients to be written in C++ using the same compiler and switches, use a strong abstraction: An **`istream`** (or custom **`File`** object that wraps **`istream`** to avoid a direct dependency on one standard library implementation) raises the level of abstraction and makes the API much more robust. The function knows that it's getting a **`File`** or a suitable input stream, does not need to manage memory for string filenames, and is immune to many accidental and deliberate errors possible with the other options. Few checks remain: The **`File`** must be open, and the contents must be in the right format, but that's about all that can go wrong.

Even when you choose to use a lower-level abstraction in a module's external interface, always use the highest level of abstraction internally and translate to the lower-level abstraction at the module's boundary. For example, if you will have non-C++ clients, you might use opaque **`void*`** or **`int`** handles to client code, but still use objects internally, and cast only at the module's interface to translate between the two.

Examples

Example: Using **`std::string`** *in a module interface.* Say a module wants to provide this API:

```
std::string Translate( const std::string& );
```

For libraries used internally in one team or company, this is usually fine. But if you need to dynamically link this module together with a caller that has a different implementation of **`std::string`** (sporting a different memory layout), strange things will happen because the client and the module can't understand each others' strings.

We have seen developers try to get around this by wrapping **`std::string`** with their own **`CustomString`**, only to be shocked when they continue to encounter the very same problem because they don't control the build process of all callers.

One solution is to rely on portable (probably built-in) types, either instead of or in addition to the function that takes a **`string`**. For example:

```
void Translate( const char* src, char* dest, size_t destSize );
```

Using a lower-level abstraction is more portable, but always adds complexity; e.g., here, both the caller and the callee have to explicitly deal with possible truncation if the buffer is not big enough. (Note that this version uses a caller-allocated buffer to avoid the pitfall of allocating and deallocating memory in different modules; see Item 60.)

References

[McConnell93

Templates and Genericity

Your quote here.

Bjarne Stroustrup, [Stroustrup00

64. Blend static and dynamic polymorphism judiciously

Team LiB ◂ PREVIOUS NEXT ▸

Summary

So much more than a mere sum of parts: Static and dynamic polymorphism are complementary. Understand their tradeoffs, use each for what it's best at, and mix them to get the best of both worlds.

Team LiB ◂ PREVIOUS NEXT ▸

Discussion

Dynamic polymorphism comes in the form of classes with virtual functions and instances manipulated indirectly (through pointers or references). Static polymorphism involves template classes and template functions.

Polymorphism means that a given value can have more than one type, and a given function can accept arguments of types other than the exact types of its parameters. "Polymorphism is a way of gaining some of the freedom of dynamic type checking without giving up the benefits of static type checking." [Webber03]

The strength of polymorphism is that the same piece of code can operate on different types, even types that were not known at the time the code was written. Such "post-hoc applicability" is the cornerstone of polymorphism because it amplifies the usefulness and reusability of code (see Item 37). (Contrast that with monomorphic code that rigidly operates only on the concrete types it was meant to work with.)

Dynamic polymorphism via public inheritance lets a value have more than one type. For example, a `Derived* p` can be viewed as a pointer not only to a `Derived`, but to an object of any type `Base` that's a direct or indirect base of `Derived` (the *subsumption* property). Dynamic polymorphism is also referred to as *inclusion polymorphism* because the set modeled by `Base` includes the specializations modeled by `Derived`.

Due to its characteristics, dynamic polymorphism in C++ is best at:

- *Uniform manipulation based on superset/subset relationships:* Different classes that hold a superset/subset (base/derived) relationship can be treated uniformly. A function that works on `Employee` objects works also on `Secretary` objects.

- *Static type checking:* All types are checked statically in C++.

- *Dynamic binding and separate compilation:* Code that uses classes in a hierarchy can be compiled apart from the code of the entire hierarchy. This is possible because of the indirection that pointers provide (both to objects and to functions).

- *Binary interfacing:* Modules can be linked either statically or dynamically, as long as the linked modules lay out the virtual tables the same way.

Static polymorphism via templates also lets a value have more than one type. Inside a `template<class T> void f( T t ) {`

References

[Alexandrescu01

65. Customize intentionally and explicitly

Summary

Intentional is better than accidental, and explicit is better than implicit: When writing a template, provide points of customization knowingly and correctly, and document them clearly. When using a template, know how the template intends for you to customize it for use with your type, and customize it appropriately.

Discussion

A common pitfall when writing template libraries is providing *unintentional* points of customizationthat is, points where a caller's code can get looked up and used inside your template, but you didn't mean for a caller's code to get involved. It's easy to do: Just call another function or operator the normal way (unqualified), and if one of its arguments happens to be of a template parameter type (or a related type) then ADL will pick it up. Examples abound: See Item 58 for an example.

Instead, be intentional: Know the three major ways to provide points of customization in a template, decide which one you want to use at a given point in your template, and code it correctly. Then, check to verify that that you didn't accidentally also code a customization hook in places where you didn't mean to.

The first way to provide a point of customization is the usual "implicit interface" (see Item 64) approach where your template simply relies on a type's having an appropriate member with a given name:

```
// Option 1: Provide a point of customization by requiring T to provide
"foo-ability"
// as a member function with a given name, signature, and semantics.

template<typename T>
void Sample1( T t ) {
 t.foo();                    // foo is a point of customization
 typename T::value_type x; // another example: providing a point of custom-
}                           //   ization to look up a type (usually via typedef)
```

To implement Option 1, the author of **`Sample1`** must:

- *Call the function with member notation:* Just use the natural member syntax.
- *Document the point of customization:* The type must provide an accessible member function **`foo`** that can be called with given arguments (here, none).

The second option is to use the "implicit interface" method, but with a nonmember function that is looked up via argument-dependent lookup (i.e., it is expected to be in the namespace of the type with which the template is instantiated); this is a major motivation for the language's ADL feature (see Item 57). Your template is relying on a type's having an appropriate *non*member with a given name:

```
// Option 2: Provide a point of customization by requiring T to provide "foo
-ability"
// as a nonmember function, typically looked up by ADL, with a given name,
signature,
// and semantics. (This is the only option that doesn't also work to look up
a type.)

template<typename T>
void Sample2( T t ) {
  foo( t );              // foo is a point of customization

  cout << t;             // another example: operator<< with operator nota-
}                        //   tion is the same kind of point of customization
```

To implement Option 2, the author of **`Sample2`** must:

- *Call the function with* unqualified *nonmember notation (including natural operator notation in the case of operators) and ensure the template itself doesn't have a member function with the same name:* It is essential for the template *not* to qualify the call to **`foo`** (e.g., don't write **`SomeNamespace::foo( t )`**) or to have its own member function of the same name, because either of those would turn off ADL and

References

[Stroustrup00

66. Don't specialize function templates

Summary

Specialization is good only when it can be done correctly: When extending someone else's function template (including `std::swap`), avoid trying to write a specialization; instead, write an overload of the function template, and put it in the namespace of the type(s) the overload is designed to be used for. (See Item 57.) When you write your own function template, avoid encouraging direct specialization of the function template itself.

Discussion

It's okay to overload function templates. Overload resolution considers all primary templates equally, and that's why it works as you would naturally expect from your experience with normal C++ function overloading: Whatever templates are visible are considered for overload resolution, and the compiler simply picks the best match.

Unfortunately, it's a lot less intuitive to specialize function templates. There are two basic reasons:

- *You can't specialize function templates partially, only totally:* Code that looks like partial specialization is really just overloading instead.
- *Function template specializations never participate in overloading:* Therefore, any specializations you write will not affect which template gets used, and this runs counter to what most people would intuitively expect. After all, if you had written a nontemplate function with the identical signature instead of a function template specialization, the nontemplate function would always be selected because it's always considered to be a better match than a template.

If you're writing a function template, prefer to write it as a single function template that should never be specialized or overloaded, and implement the function template entirely in terms of a class template. This is the proverbial extra level of indirection that steers you well clear of the limitations and dark corners of function templates. This way, programmers using your template will be able to partially specialize and explicitly specialize the class template to their heart's content without affecting the expected operation of the function template. This nicely avoids both the limitation that function templates can't be partially specialized, as well as the sometimes surprising effect that function template specializations don't overload. Problem solved.

If you're using someone else's plain old function template that doesn't use this technique (i.e., a function template that is not implemented in terms of a class template), and you want to write your own special-case version that should participate in overloading, don't write it as a specialization; just make it an overloaded nontemplate function. (See also Items 57 and 58.)

Examples

Example: ***`std::swap`***. The basic swap template swaps two values **`a`** and **`b`** by creating a **`temp`** copy of **`a`**, assigning **`a = b`**, and assigning **`b = temp`**. How can you extend it for your own types? For example, let's say you have your own type **`Widget`** in your own namespace **`N`**:

```
namespace N {
 class Widget {
```

References

[Austern99

67. Don't write unintentionally nongeneric code

Summary

Commit to abstractions, not to details: Use the most generic and abstract means to implement a piece of functionality.

Discussion

When writing code, use the most abstract means available that get the job done. Think in terms of what operations put the least "capabilities strain" on the interfaces they operate on. This habit makes your code more generic, and therefore more reusable and more resilient to changes in its surroundings.

On the contrary, code that gratuitously commits to details is rigid and fragile:

- *Use* **`!=`** *instead of* **`<`** *to compare iterators:* Using **`!=`** is more general and so applies to a larger class of objects; using **`<`** asks for ordering, and only random-access iterators can implement **`operator<`**. If you use **`operator!=`** your code will "port" easily to other kinds of iterators, such as forward and bidirectional iterators.
- *Prefer iteration to indexed access:* Most containers don't support indexed access; for example, **`list`** can't implement it efficiently. But all containers support iterators. Iteration is a better approach because it is more general, and it can still be used in conjunction with indexed access as needed.
- *Use* ***`empty()`*** *instead of* ***`size() == 0`:*** "Empty/non-empty" is a more primitive concept than "exact size." For example, you might not know the size of a stream but can always talk about emptiness, and the same applies to input iterators. Some containers, such as **`list`**, naturally implement **`empty`** more efficiently than **`size`**.
- *Use the highest class in the hierarchy that offers the functionality you need:* When programming with dynamic polymorphic classes, don't depend on details you don't need and that tie you to specific derived classes.
- *Write const-correct code (see Item 15):* Taking **`const&`** parameters puts less strain on the caller because **`const&`** covers constant and non-constant objects alike.

Exceptions

In some cases, indexing instead of iteration enables compilers to do better optimization. Before you're tempted to make use of this, make sure you really need it and that your compiler really does it. (See Item 8).

References

[*Koenig97*

Error Handling and Exceptions

Error handling is a difficult task for which the programmer needs all the help that can be provided.

Bjarne Stroustrup, [Stroustrup94

68. Assert liberally to document internal assumptions and invariants

Summary

Be assertive! Use `assert` or an equivalent liberally to document assumptions internal to a module (i.e., where the caller and callee are maintained by the same person or team) that must always be true and otherwise represent programming errors (e.g., violations of a function's postconditions detected by the caller of the function). (See also Item 70.) Ensure that assertions don't perform side effects.

Discussion

It's hard enough to find an error in your code when you're looking for it; it's even harder when you've assumed your code is error-free.

Steve McConnell

It is hard to overestimate the power of assertions. The `assert` macro and alternatives such as compile-time (and, less preferably, run-time) assertion templates are invaluable tools for detecting and debugging programming errors during a project's development. Of all such tools, they arguably have the best complexity/effectiveness ratio. The success of a project can be conditioned at least in part by the effectiveness with which developers use assertions in their code.

Assertions commonly generate code in debug mode only (when the `NDEBUG` macro is not defined), so they can be made "free" in release builds. Be generous with what you check. Never write expressions with side effects in `assert` statements. In release mode, when the `NDEBUG` macro is defined, `assert`s don't generate any code at all:

```
assert( ++i < limit );              // bad: i is incremented in debug mode
only
```

According to information theory, the quantity of information in an event is inversely proportional to the likelihood of that event happening. Thus, the less likely some `assert` is to fire, the more information it will bring to you when it does fire.

Avoid `assert(false)`, and prefer `assert( !"informational message" )`. Most compilers will helpfully emit the string in their error output. Also consider adding `&& "informational message"` to more complex assertions, especially instead of a comment.

Consider defining your own `assert`. The standard `assert` macro unceremoniously aborts the program with a message to the standard output. Your environment likely offers enhanced debugging capabilities; for example, it might allow starting an interactive debugger automatically. If so, you may want to define your own `MYASSERT` macro and use it. It can also be useful to retain most assertions even in release builds (prefer not to disable checks for performance reasons unless there's a proven need; see Item 8), and there is real benefit to having an assertion facility that can distinguish between "levels" of assertions, some of which stay on in release mode.

Assertions often check conditions that could be verified at compile time if the language were expressive enough. For example, your whole design might rely on every `Employee` having a nonzero `id_`. Ideally, the compiler would analyze `Employee`'s constructor and members and prove by static analysis that, indeed, that condition is always true. Absent such omniscience, you can issue an `assert( id_ != 0 )` inside the implementation of `Employee` whenever you need to make sure an `Employee` is sane:

```
unsigned int Employee::GetID() {
    assert( id_ != 0 && "Employee ID is invalid (must be nonzero)" );
    return id_;
}
```

Don't use assertions to report run-time errors (see Items 70 and 72). For example, don't use `assert` to make sure that `malloc` worked, that a window creation succeeded, or that a thread was started. You can, however, use `assert` to make sure that APIs work as documented. For example, if you call some API function that is documented to always return a positive value, but you suspect it might have a bug, plant an `assert` after the call to validate its postcondition.

It is not recommended to throw an exception instead of asserting, even though the standard `std::logic_error` exception class was originally designed for this purpose. The primary disadvantage of using an exception to report a programming error is that you don't really want stack unwinding to occuryou want the debugger to launch on the exact line where the

◀ PREVIOUS NEXT ▶

Examples

Example: Do **`assert`** *on basic assumptions.* We all have war stories about assertions that "couldn't possibly fire," but did. Time and again, it's: "This value is certainly positive!" "That pointer is obviously not null!" Do recheck tautologies: Software development is complex, change is the norm, and anything can happen in a program that is changing. Assertions verify that what you believe is "obviously true" today actually stays true. Do **`assert`** on tautologies that the type system cannot enforce:

```
string Date::DayOfWeek() const {
  assert( day_ > 0 && day_ <= 31 );                // invariant checks
  assert( month_ > 0 && month_ <= 12 );
```

References

[*Abrahams01b*] ?[*Alexandrescu03b*] ?[*Alexandrescu03c*] ?[*Allison98*

69. Establish a rational error handling policy, and follow it strictly

Summary

Consciously specify, and conscientiously apply, what so many projects leave to ad-hoc (mis)judgment: Develop a practical, consistent, and rational error handling policy early in design, and then stick to it. Ensure that it includes:

- *Identification:* What conditions are errors.
- *Severity:* How important or urgent each error is.
- *Detection:* Which code is responsible for detecting the error.
- *Propagation:* What mechanisms are used to report and propagate error notifications in each module.
- *Handling:* What code is responsible for doing something about the error.
- *Reporting:* How the error will be logged or users notified.

Change error handling mechanisms only on module boundaries.

Discussion

From this Item onward, this section focuses on dealing with run-time errors that are not due to faulty coding internal to a module or subsystem. (As Item 68 covers separately, prefer to use assertions to flag internal programming errors, ones that are just outright coding errors on some programmer's part.)

Determine an overall error reporting and handling policy for your application and for each module or subsystem, and stick to it. Include a policy for at least each of the following points.

Universally:

- *Error identification:* For each entity (e.g., each function, each class, each module), document the entity's internal and external invariants.

For each function:

- *Error identification:* For each function, document its preconditions and postconditions, the invariants it shares responsibility for maintaining, and the error-safety guarantee it supports. (See Items 70 and 71.) Note that destructors and deallocation functions in particular must always be written to support the no-fail guarantee, because otherwise it's often impossible to reliably and safely perform cleanup (see Item 51).

For each error (see the definition of "error" in Item 70):

- *Error severity and categorization:* For each error, identify a severity level. Preferably provide a way to fine-tune diagnostics for particular error categories and levels to facilitate remote user assistance.
- *Error detection:* For each error, document which code is responsible for detecting which error, following the advice of Item 70.
- *Error handling:* For each error, identify the code that is responsible for handling the error, following the advice in Item 74.
- *Error reporting:* For each error, identify appropriate reporting method(s). These commonly include recording the error in disk file logs, printed logs, electronic dump transmissions, or possibly inconvenient and annoying pager calls in the case of severe errors.

For each module:

- *Error propagation:* For each module (note: each module, not each error), identify which coding mechanism will be used to propagate errors (e.g., C++ exceptions, COM exceptions, CORBA exceptions, return codes).

We emphasize that error handling strategies should change only on module boundaries (see Items 62 and 63). Each module should consistently use a single error handling strategy and mechanism internally (e.g., modules written in C++ should use exceptions internally; see Item 72) and consistently use a single, possibly different, error handling strategy and mechanism in its interface (e.g., the module might present a flat C API to accommodate callers that could be written in various languages, or a COM wrapper that presents COM exceptions).

All functions that are entry points into the module are directly responsible for translating from the internal to the external strategy if they are different. For example, in a module that uses C++ exceptions internally but presents a C API boundary, all C APIs must

References

[Abrahams01b] ?[Allison98

70. Distinguish between errors and non-errors

Summary

A breach of contract is an error: A function is a unit of work. Thus, failures should be viewed as errors or otherwise based on their impact on functions. Within a function `f`, a failure is an error if and only if it violates one of `f`'s preconditions or prevents `f` from meeting any of its callees' preconditions, achieving any of `f`'s own postconditions, or reestablishing any invariant that `f` shares responsibility for maintaining.

In particular, here we exclude internal programming errors (i.e., where the caller and callee are the responsibility of the same person or team, such as inside a module), which are a separate category normally dealt with using assertions (see Item 68).

Discussion

It is crucial to crisply distinguish between errors and non-errors in terms of their effects on functions, especially for the purpose of defining safety guarantees (see Item 71). The key words in this Item are *precondition, postcondition*, and *invariant*.

The function is the basic unit of work, no matter whether you are programming C++ in a structured style, an OO style, or a generic style. A function makes assumptions about its starting state (documented as its preconditions, which the caller is responsible for fulfilling and the callee is responsible for validating) and performs one or more actions (documented as its results or postconditions, which the function as callee is responsible for fulfilling). A function may share responsibility for maintaining one or more invariants. In particular, a nonprivate mutating member function is by definition a unit of work on its object, and must take the object from one valid invariant-preserving state to another; during the body of the member function, the object's invariants can be (and nearly always must be) broken, and that is fine and normal as long as they are reestablished by the end of the member function. Higher-level functions compose lower-level functions into larger units of work.

An error is any failure that prevents a function from succeeding. There are three kinds:

- *Violation of, or failure to achieve, a precondition:* The function detects a violation of one its own preconditions (e.g., a parameter or state restriction), or encounters a condition that prevents it from meeting a precondition of another essential function that must be called.

- *Failure to achieve a postcondition:* The function encounters a condition that prevents it from establishing one of its own postconditions. If the function has a return value, producing a valid return value object is a postcondition.

- *Failure to reestablish an invariant:* The function encounters a condition that prevents it from reestablishing an invariant that it is responsible for maintaining. This is a special kind of postcondition that applies particularly to member functions; an essential postcondition of every nonprivate member function is that it must reestablish its class's invariants. (See [Stroustrup00

Examples

Example 1: ***`std::string::insert`*** *(precondition error).* When trying to insert a new character into a **`string`** at a specific position **`pos`**, a caller should check for invalid values of **`pos`** that would violate a documented parameter requirement; for example, **`pos > size()`**. The **`insert`** function can't perform its work successfully if it doesn't have a valid starting point.

Example 2: ***`std::string::append`*** *(postcondition error).* When appending a character to a **`string`**, failure to allocate a new buffer if the existing one is full prevents the operation from performing its documented function and achieving its documented postconditions, and is therefore an error.

Example 3: Inability to produce a return value (postcondition error). For a value-returning function, producing a valid return object is a postcondition. If the return value can't be correctly created (e.g., if the function returns a **`double`**, but there exists no **`double`** value with the mathematical properties requested of a result), that is an error.

Example 4: ***`std::string::find_first_of`*** *(not an error in the context of* ***`string`****).* When searching for a character in a **`string`**, failure to find the character is a legitimate outcome and not an error. At least, it is not an error as far the general-purpose **`string`** class is concerned; if the owner of the given **`string`** assumed the character would be present and its absence is thus an error according to a higher-level invariant, that higher-level calling code would then appropriately report an error with respect to its invariant.

Example 5: Different error conditions in the same function. In spite of the increased reliability of disks nowadays, writing to disk has traditionally remained subject to expected errors. If you design a **`File`** class, in the same function **`File::Write( const char* buffer, size_t size )`**, which requires that **`buffer`** is non-null and that the file be opened for writing, you might decide to do the following:

- *If* ***`buffer`*** *is* ***`NULL`*****:** Report an error on the precondition violation.
- *If the* ***`File`*** *is read-only:* Report an error on the precondition violation.
- *If the write does not succeed:* Report an error on the postcondition violation, because the function cannot do the work it promises to do.

Example 6: Different status for the same condition. The same condition can be a valid precondition for one function and not for another; the choice depends on the function's author, who is specifying his interface's semantics. In particular, **`std::vector`** provides two ways to perform indexed access: **`operator[]`**, which is not bounds-checked, and **`at`**, which is. Both require a precondition that the argument is not out of range. Because **`operator[]`** is not required to validate its argument or to be safe to call with an invalid argument, it must document that the caller has sole responsibility for ensuring that the argument is in the valid range; this function is inherently less safe. On the other hand, **`at`** is documented to behave safely even in the presence of an invalid argument, and to report an error (by throwing **`std::out_of_range`**) if the argument is found to be out of range.

References

[Abrahams01b] ?[Meyer00] ?[Stroustrup00

71. Design and write error-safe code

Summary

Promise, but don't punish: In each function, give the strongest safety guarantee that won't penalize callers who don't need it. Always give at least the basic guarantee.

Ensure that errors always leave your program in a valid state. This is the basic guarantee. Beware of invariant-destroying errors (including but not limited to leaks), which are just plain bugs.

Prefer to additionally guarantee that the final state is either the original state (if there was an error the operation was rolled back) or the intended target state (if there was no error the operation was committed). This is the strong guarantee.

Prefer to additionally guarantee that the operation can never fail at all. Although this is not possible for most functions, it is required for functions like destructors and deallocation functions. This is the no-fail guarantee.

Discussion

The basic, strong, and no-fail (then known as nothrow) guarantees were originally described in [Abrahams96] and publicized in [GotW], [Stroustrup00

Examples

Example 1: Retry after failure. If your program includes a command for saving data to a file and the write fails, make sure you revert to a state where the caller can retry the operation. In particular, don't release any data structure before data has been safely flushed to disk. For example, one text editor we know of didn't allow changing the file name to save to after a write error, which is a suboptimal state for further recovery.

Example 2: Skins. If you write a skinnable application, don't destroy the existing skin before attempting to load a new one. If loading the new skin fails, your application might end up in an unusable state.

Example 3: `std::vector::insert`. Because a `vector<T>`'s internal storage is contiguous, inserting an element into the middle requires shuffling some existing values over by one position to make room for the new element. The shuffling is done using `T::T(const T&)` and `T::operator=`, and if either of those two operations can fail (by throwing an exception), the only way to make `insert` provide the strong guarantee is to make a complete copy of the container, perform the operation on the copy, and, if successful, swap the original's and copy's state using the no-fail `vector<T>::swap`.

But if that were done every time by `insert` itself, every caller of `vector::insert` would always incur the space and performance penalty of making a complete copy of the container, whether it needed the strong guarantee or not. That is needlessly expensive. Instead, those callers who do want the strong guarantee can do the work themselves, and are given sufficient tools for doing so. (In the best case: Arrange for the contained type to not throw from its copy constructor or copy assignment operators. In the worst case: Take a copy, `insert` into the copy, and `swap` the copy with the original when successful.)

Example 4: Unlaunching a satellite. Consider a function `f` that as part of its work launches a satellite, and the `LaunchSatellite` function it uses provides the strong or no-fail guarantee. If `f` can perform all of its work that could fail before launching the satellite, `f` can be coded to provide the strong guarantee. But if `f` must perform other operations that might fail after having already performed the launch, it cannot provide the strong guarantee because it cannot bring the satellite back. (At any rate, such an `f` probably ought to be split into two functions, because a single function should probably not be trying to do multiple pieces of work of such significance; see Item 5.)

PREVIOUS NEXT

References

[Abrahams96] ?[Abrahams01b] ?[Alexandrescu03d] ?[Josuttis99

72. Prefer to use exceptions to report errors

PREVIOUS NEXT

Summary

When harmed, take exception: Prefer using exceptions over error codes to report errors. Use status codes (e.g., return codes, `errno`) for errors when exceptions cannot be used (see Item 62), and for conditions that are not errors. Use other methods, such as graceful or ungraceful termination, when recovery is impossible or not required.

PREVIOUS NEXT

Discussion

It's no coincidence that most modern languages created in the past 20 years use exceptions as their primary error reporting mechanism. Almost by definition, exceptions are for reporting *exceptions* to normal processingalso known as "errors," defined in Item 70 as violations of preconditions, postconditions, and invariants. Like all error reporting, exceptions should not arise during normal successful operation.

We will use the term "status codes" to cover all forms of reporting status via codes (including return codes, `errno`, a `GetLastError` function, and other strategies to return or retrieve codes), and "error codes" specifically for status codes that signify errors. In C++, reporting errors via exceptions has clear advantages over reporting them via error codes, all of which make your code more robust:

- *Exceptions can't be silently ignored:* The most terrible weakness of error codes is that they are ignored by default; to pay the slightest attention to an error code, you have to explicitly write code to accept the error and respond to it. It is common for programmers to accidentally (or lazily) fail to pay attention to error codes. This makes code reviews harder. Exceptions can't be silently ignored; to ignore an exception, you must explicitly `catch` it (even if only with

Examples

Example 1: Constructors (invariant error). If a constructor cannot successfully create an object of its type, which is the same as saying that it cannot establish all of the new object's invariants, it should throw an exception. Conversely, an exception thrown from a constructor always means that the object's construction failed and the object's lifetime never began; the language enforces this.

Example 2: Successful recursive tree search. When searching a tree using a recursive algorithm, it can be tempting to return the resultand conveniently unwind the search stackby throwing the result as an exception. But don't do it: An exception means an error, and finding the result is not an error (see [Stroustrup00]). (Note that, of course, failing to find the result is also not an error in the context of the search function; see the **`find_first_of`** example in Item 70.)

See also Item 70's examples, replacing "report an error" with "throw an exception."

Exceptions

In rare cases, consider using an error code if you are certain that both of the following are true:

- *The benefits of exceptions don't apply:* For example, you know that nearly always the immediate caller must directly handle the error and so propagation should happen either never or nearly never. This is very rare because usually the callee does not have this much information about the characteristics of all of its callers).
- *The actual measured performance of throwing an exception over using an error code matters:* That is, the performance difference is measured empirically, so you're probably in an inner loop and throwing often; the latter is rare because it usually means the condition is not really an error after all, but let's assume that somehow it is.

In very rare cases, some hard real-time projects may find themselves under pressure to consider turning off exception handling altogether because their compiler's exception handling mechanism has worst-case time guarantees that make it difficult or impossible for key operations to meet deadline schedules. Of course, turning off exception handling means that the language and standard library will not report errors in a standard way (or in some cases at all; see your compiler's documentation) and the project's own error reporting mechanisms would have to be based on codes instead. It is difficult to exaggerate just how undesirable and how much of a last resort this should be; before making this choice, analyze in detail how you will report errors from constructors and operators and how that scheme will actually work on the compiler(s) you are using. If after serious and deep analysis you still feel that you are really forced to turn off exception handling, still don't do it project-wide: Do so in as few modules as possible, and facilitate this by trying to group such deadline-sensitive operations together into the same module as much as possible.

References

[Alexandrescu03d] ?[Allison98

73. Throw by value, catch by reference

Summary

Learn to `catch` properly: Throw exceptions by value (not pointer) and catch them by reference (usually to `const`). This is the combination that meshes best with exception semantics. When rethrowing the same exception, prefer just `throw;` to `throw e;`.

Discussion

When throwing an exception, throw an object by value. Avoid throwing a pointer, because if you throw a pointer, you need to deal with memory management issues: You can't throw a pointer to a stack-allocated value because the stack will be unwound before the pointer reaches the call site. You could throw a pointer to dynamically allocated memory (if the error you're reporting isn't "out of memory" to begin with), but you've put the burden on the catch site to deallocate the memory. If you feel you really must throw a pointer, consider throwing a value-like smart pointer such as a `shared_ptr<T>` instead of a plain `T*`.

Throwing by value enjoys the best of all worlds because the compiler itself takes responsibility for the intricate process of managing memory for the exception object. All you need to take care of is ensuring that you implement a non-throwing copy constructor for your exception classes (see Item 32).

Unless you are throwing a smart pointer, which already adds an indirection that preserves polymorphism, catching by reference is the only good way to go. Catching a plain value by value results in slicing at the catch site (see Item 54), which violently strips away the normally-vital polymorphic qualities of the exception object. Catching by reference preserves the polymorphism of the exception object.

When rethrowing an exception `e`, prefer writing just `tHRow;` instead of `throw e;` because the first form always preserves polymorphism of the rethrown object.

Examples

Example: Rethrowing a modified exception. Prefer to rethrow using **`throw;`**:

```
catch( MyException& e ) {                    // catch by reference to non-const
 e.AppendContext("Passed through here");    // modify
 throw;                                     // rethrow modified object
}
```

References

[Dewhurst03

74. Report, handle, and translate errors appropriately

Summary

Know when to say when: Report errors at the point they are detected and identified as errors. Handle or translate each error at the nearest level that can do it correctly.

Discussion

Report an error (e.g., write `throw`) wherever a function detects an error that it cannot resolve itself and that makes it impossible for the function to continue execution. (See Item 70)

Handle an error (e.g., write a `catch` that doesn't rethrow the same or another exception or emit another kind of error code) in the places that have sufficient knowledge to handle the error, including to enforce boundaries defined in the error policy (e.g., on `main` and thread mainlines; see Item 62) and to absorb errors in the bodies of destructors and deallocation operations.

Translate an error (e.g., write a `catch` that does rethrow a different exception or emits another kind of error code) in these circumstances:

- *To add higher-level semantic meaning:* For example, in a word processing application, `Document::Open` could accept a low-level unexpected-end-of-file error and translate it to a document-invalid-or-corrupt error, adding semantic information.

- *To change the error handling mechanism:* For example, in a module that uses exceptions internally but whose C API public boundary reports error codes, a boundary API would catch an exception and emit a corresponding error code that fulfills the module's contract and that the caller can understand.

Code should not accept an error if it doesn't have the context to do something useful about that error. If a function isn't going to handle (or translate, or deliberately absorb) an error itself, it should allow or enable the error to propagate up to a caller who can handle it.

Exceptions

It can occasionally be useful to accept and re-emit (e.g., `catch` and rethrow) the same error in order to add instrumentation, even though the error is not actually being handled.

References

[Stroustrup00

75. Avoid exception specifications

Summary

Take exception to these specifications: Don't write exception specifications on your functions unless you're forced to (because other code you can't change has already introduced them; see Exceptions.)

Discussion

In brief, don't bother with exception specifications. Even experts don't bother. The main problems with exception specifications are that they're only "sort of" part of the type system, they don't do what most people think, and you almost always don't want what they actually do.

Exception specifications aren't part of a function's type, except when they are. They form a shadow type system whereby writing an exception specification is variously:

- *Illegal:* In a typedef for a pointer to function.
- *Allowed:* In the identical code without the typedef.
- *Required:* In the declaration of a virtual function that overrides a base class virtual function that has an exception specification.
- *Implicit and automatic:* In the declaration of the constructors, assignment operators, and destructors when they are implicitly generated by the compiler.

A common but nevertheless incorrect belief is that exception specifications statically guarantee that functions will throw only listed exceptions (possibly none), and enable compiler optimizations based on that knowledge.

In fact, exception specifications actually do something slightly but fundamentally different: They cause the compiler to inject additional run-time overhead in the form of implicit `try`/ `catch` blocks around the function body to enforce via run-time checking that the function does in fact emit only listed exceptions (possibly none), unless the compiler can statically prove that the exception specification can never be violated in which case it is free to optimize the checking away. And exception specifications can both enable and prevent further compiler optimizations (besides the inherent overhead already described); for example, some compilers refuse to inline functions that have exception specifications.

Worst of all, however, is that exception specifications are a blunt instrument: When violated, by default they immediately terminate your program. You can register an `unexpected_handler`, but it's highly unlikely to help you much because you get exactly one global handler and the only way the handler could avoid immediately calling `terminate` would be to rethrow an exception that is permissiblebut because you have only one handler for your whole application, it's hard to see how it could do useful recovery or usefully know what exceptions might be legal without trivializing exception specifications altogether (e.g., following the discipline of having all exception specifications allow some general `UnknownException` eliminates any advantage that having an exception specification might have had in the first place).

You generally can't write useful exception specifications for function templates anyway, because you generally can't tell what exceptions the types they operate on might throw.

Paying a performance overhead in exchange for enforcements that are nearly always useless because they are fatal if they ever fire is an excellent example of a premature pessimization (see Item 9).

These is no easy fix for the problems described in this Item. In particular, the problems are not easily solved by switching to static checking. People often suggest switching from dynamically checked exception specifications to statically checked ones, as provided in Java and other languages. In short, that just trades one set of problems for another; users of languages with statically checked exception specifications seem to equally often suggest switching to dynamically checked ones.

Exceptions

If you have to override a base class virtual function that already has an exception specification (e.g., ahem, `std::exception::what`), and you don't have the ability to change the class to remove the exception specifications (or to convince the class's maintainer to remove them), you will have to write a compatible exception specification on your overriding function, and you should prefer to make it no less restrictive than the base version so as to minimize the frequency with which it might be violated:

```
class Base {
```

References

[BoostLRG] ?[Stroustrup00

STL: Containers

By default, use **`vector`** *when you need a container.*

Bjarne Stroustrup, [Stroustrup00

76. Use `vector` by default. Otherwise, choose an appropriate container

Summary

Using the "right container" is great: If you have a good reason to use a specific container type, use that container type knowing that you did the right thing.

So is using `vector`: Otherwise, write `vector` and keep going without breaking stride, also knowing you did the right thing.

Discussion

Here are three fundamental issues and related questions for programming in general, and for choosing a container in particular:

- *Write for correctness, simplicity, and clarity first (see Item 6):* Prefer to choose the container that will let you write the clearest code. Examples: If you need to insert at a specific position, use a sequence container (e.g., **`vector`**, **`list`**). If you need random-access iterators, use **`vector`**, **`deque`** , or **`string`**. If you need dictionary-like lookups like **`c[0] = 42;`**, use an associative container (e.g., **`set`**, **`map`**)but if you need an *ordered* associative collection, you can't use hash-based (nonstandard

Examples

Example: ***`vector`*** *for small lists.* Is it a common fallacy to use **`list`** just because "**`list`** is obviously the right type for list operations," such as insertion into the middle of a sequence. Using a **`vector`** for small lists is almost always superior to using **`list`**. Even though insertion in the middle of the sequence is a linear-time operation for **`vector`** and a constant-time operation for **`list`**, **`vector`** usually outperforms **`list`** when containers are relatively small because of its better constant factor, and **`list`**'s Big-Oh advantage doesn't kick in until data sizes get larger.

Here, use **`vector`** unless the data sizes warrant a different choice (see Item 7), or because the strong safety guarantee is essential and the contained type's copy constructor or copy assignment operator might fail (in which case, it can be important that **`list`** provides the strong guarantee for **`insert`** operations for collections of such types).

References

[Austern99

77. Use `vector` and `string` instead of arrays

Summary

Why juggle Ming vases? Avoid implementing array abstractions with C-style arrays, pointer arithmetic, and memory management primitives. Using `vector` or `string` not only makes your life easier, but also helps you write safer and more scalable software.

Discussion

Buffer overruns and security flaws are, hands down, a front-running scourge of today's software. Silly limitations due to fixed-length arrays are a major annoyance even when within the limits of correctness. Most of these are caused by using bare C-level facilitiessuch as built-in arrays, pointers and pointer arithmetic, and manual memory managementas a substitute for higher-level concepts such as buffers, vectors, or strings.

Here are some reasons to prefer the standard facilities over C-style arrays:

- *They manage their own memory automatically:* No more need for fixed buffers of "longer than any reasonable length" ("time bombs" is a more accurate description), or for frantic `realloc`'ing and pointer adjustment.
- *They have a rich interface:* You can implement complex functionality easily and expressively.
- *They are compatible with C's memory model:* `vector` and `string::c_str` can be passed to C APIsin both read and write modes, of course. In particular, `vector` is C++'s gateway to C and other languages. (See Items 76 and 78.)
- • *They offer extended checking:* The standard facilities can implement (in debug mode) iterators and indexing operators that expose a large category of memory errors. Many of today's standard library implementations offer such debugging facilitiesuse them! (Run, don't walk, to Item 83.)
- *They don't waste much efficiency for all that:* Truth be told, in release mode `vector` and `string` favor efficiency over safety when the two are in tension. Still, all in all, the standard facilities offer a much better platform for creating safe components than do bare arrays and pointers.
- *They foster optimizations:* Modern standard library implementations include optimizations that many of us mere mortals have never thought of.

An array can be acceptable when its size really is fixed at compile time (e.g., `float[3]` for a three-dimensional point; switching to four dimensions would be a redesign anyway).

References

[Alexandrescu01a] ?[Dewhurst03

78. Use `vector` (and `string::c_str`) to exchange data with non-C++ APIs

Summary

`vector` isn't lost in translation: `vector` and `string::c_str` are your gateway to communicate with non-C++ APIs. But don't assume iterators are pointers; to get the address of the element referred to by a `vector<T>::iterator iter`, use `&*iter`.

Discussion

`vector` (primarily) and `string::c_str` and `string::data` (secondarily) are the best way to communicate data with non-C++ APIs in general, and C libraries in particular.

`vector`'s storage is always contiguous, so accessing the address of its first element returns a pointer to its contents. Use `&*v.begin()`, `&v[0]` , or `&v.front()` to get a pointer to `v`'s first element. To get a pointer to a `vector`'s `n`-th element, prefer to do the arithmetic first and take the address later (e.g., `&v.begin()[n]` or `&v[n]`) instead of fetching a pointer to `front` and then doing pointer arithmetic (e.g., `(&v.front())[n]`). This is because in the former case you are giving a checked implementation a crack at verifying that you don't access an element outside `v`'s bounds (see Item 83).

Do not assume that `v.begin()` returns a pointer to the first element, or in general that `vector` iterators are pointers. Although some STL implementations do define `vector<T>::iterator`s as bare `T*` pointers, iterators can be, and increasingly are, full-blown types (see again Item 83).

Although most implementations use contiguous memory for `string`, that's not guaranteed, so never take the address of a character in a `string` and assume it points to contiguous memory. The good news is that `string::c_str` always returns a null-terminated C-style string. (`string::data` also returns a pointer to contiguous memory, except that it's not guaranteed to be zero-terminated.)

When you pass pointers into a `vector` `v`'s data, C code can both read from and write to `v`'s elements but must stay within `size`'s bounds. A well-designed C API takes a maximum number of objects (up to `v.size()`) or a past-the-end pointer (`&*v.begin()+v.size()`).

If you have a container of `T` objects other than `vector` or `string` and want to pass its c ontents to (or populate it from) a non-C++ API that expects a pointer to an array of `T` objects, copy the container's contents to (or from) a `vector<T>` that can directly communicate with the non-C++ API.

References

[Josuttis99

79. Store only values and smart pointers in containers

Summary

Store objects of value in containers: Containers assume they contain value-like types, including value types (held directly), smart pointers, and iterators.

◂ PREVIOUS NEXT ▸

Discussion

The most common use of containers is to store values held directly (e.g., `vector<int>`, `set<string>`). For containers of pointers: If the container owns the pointed-to objects, prefer a container of reference-counted smart pointers (e.g., `list<shared_ptr<Widget> >`); otherwise, a container of raw pointers (e.g., `list<Widget*>`) or of other pointer-like values such as iterators (e.g., `list< vector<Widget>::iterator >`) is fine.

◂ PREVIOUS NEXT ▸

Examples

Example 1: ***auto_ptr.*** Objects of **auto_ptr<T>** are not value-like because of their transfer-of-ownership copy semantics. Using a container of **auto_ptr**s (e.g., a **vector< auto_ptr<int> >**) should fail to compile. Even if it does compile, *never* write that; if you want to write it, you almost certainly want a container of **shared_ptr**s instead.

Example 2: Heterogeneous containers. To have a container store and own objects of different but related types, such as types derived from a common **Base** class, prefer **container< shared_ptr<Base> >**. An alternative is to store proxy objects whose nonvirtual functions pass through to corresponding virtual functions of the actual object.

Example 3: Containers of non-value types. To contain objects even though they are not copyable or otherwise not value-like (e.g., **DatabaseLock**s and **TcpConnection**s), prefer containing them indirectly via smart pointers (e.g., **container< shared_ptr\zwbo<DatabaseLock> >** and **container< shared_ptr\zwbo<TcpConnection> >**).

Example 4: Optional values. When you want a **map<Thing, Widget>**, but some **Thing**s have no associated **Widget**, prefer **map<Thing, shared_ptr<Widget> >**.

Example 5: Index containers. To have a main container hold the objects and access them using different sort orders without resorting the main container, you can set up secondary containers that "point into" the main one and sort the secondary containers in different ways using dereferenced compare predicates. But prefer a container of **MainContainer::iterator**s (which are value-like) instead of a container of pointers.

References

[Allison98

80. Prefer `push_back` to other ways of expanding a sequence

Summary

push_back all you can: If you don't need to care about the insert position, prefer using **push_back** to add an element to sequence. Other means can be both vastly slower and less clear.

Discussion

You can insert elements into sequences at different points using **`insert`**, and you can append elements to sequences in various ways including:

```
vector<int> vec;                    // vec is empty
vec.resize(vec.size() + 1, 1);      // vec contains { 1 }
vec.insert(vec.end(), 2);           // vec contains { 1, 2 }
vec.push_back(3);                   // vec contains { 1, 2, 3 }
```

Of all forms, **`push_back`** alone takes *amortized constant time.* The other forms' performance can be as bad as quadratic. Needless to say, beyond small data volumes that makes those alternatives potential scalability barriers. (See Item 7)

`push_back`'s magic is simple: It expands capacity exponentially, not by a fixed increment. Hence the number of reallocations and copies decreases rapidly with size. For a container populated using only **`push_back`** calls, on average each element has been copied only onceregardless of the final size of the container.

Of course, **`resize`** and **`insert`** could employ the same strategy, but that is dependent on the implementation; only **`push_back`** offers the guarantee.

Algorithms can't call **`push_back`** directly because they don't have access to the container. You can ask algorithms to use **`push_back`** anyway by using a **`back_inserter`**.

Exceptions

If you know you're adding a range, even at the end of a container, prefer to use a range insertion function (see Item 81).

Exponential growth is generous with memory allocation. To fine-tune growth, call **`reserve`** explicitly**`push_back`**, **`resize`** , and the like never trigger reallocation if they have enough space. To "right-size" a **`vector`**, use the shrink-to-fit idiom (see Item 82).

References

[Stroustrup00

81. Prefer range operations to single-element operations

Summary

Don't use oars when the wind is fair (based on a Latin proverb): When adding elements to sequence containers, prefer to use range operations (e.g., the form of `insert` that takes a pair of iterators) instead of a series of calls to the single-element form of the operation. Calling the range operation is generally easier to write, easier to read, and more efficient than an explicit loop. (See also Item 84.)

Discussion

The more context you can give a function, the better the chances that it can do something useful with the information. In particular, when you call a single function and pass it a pair of iterators **`first`** and **`last`** that delimit a range, it can perform optimizations based on knowing the number of objects that are going to be added, which it obtains by computing **`distance(first,last)`**.

The same applies to "repeat *n* times" operations, such as the **`vector`** constructor that takes a repeat count and a value.

Examples

Example 1: ***vector::insert.*** Let's say you want to add **n** elements into a **vector v**. Calling **v.insert(position,x)** repeatedly can cause multiple reallocations as **v** grows its storage to accommodate each new element. Worse, each single-element **insert** is a linear operation because it has to shuffle over enough elements to make room, and this makes inserting **n** elements with repeated calls to the single-element **insert** actually a quadratic operation! Of course, you could get around the multiple-reallocation problem by calling **reserve**, but that doesn't reduce the shuffling and the quadratic nature of the operation. It's faster and simpler to just say what you're doing: **v.insert(position,first,last)**, where **first** and **last** are iterators delimiting the range of elements to be added into **v**. (If **first** and **last** are input iterators, there's no way to determine the size of the range before actually traversing it, and therefore **v** might still need to perform multiple reallocations; but the range version is still likely to perform better than inserting elements individually.)

Example 2: Range construction and assignment. Calling a constructor (or **assign** function) that takes an iterator range typically performs better than calling the default constructor (or **clear**) followed by individual insertions into the container.

PREVIOUS NEXT

References

[Meyers01

82. Use the accepted idioms to really shrink capacity and really erase elements

Summary

Use a diet that works: To really shed excess capacity from a container, use the "swap trick." To really erase elements from a container, use the erase-remove idiom.

Discussion

Some containers (e.g., **`vector`**, **`string`**, **`deque`**) can end up carrying around extra capacity that's no longer needed. Although the C++ standard library containers provide no guaranteed way to trim excess capacity, the following "swap trick" idiom works in practice to get rid of excess capacity for such a **`c`** of type **`container`**:

```
container<T>( c ).swap( c ); // the shrink-to-fit idiom to shed excess
capacity
```

Or, to empty **`c`** out completely, clearing all contained elements and shedding all possible capacity, the idiom is:

```
container<T>().swap( c );   // the idiom to shed all contents and capacity
```

In related news, a common surprise for new STL programmers is that the **`remove`** algorithm doesn't really remove elements from a container. Of course, it can't; an algorithm just works on an iterator range, and you can't actually remove something from a container without calling a container member function, usually **`erase`**. All **`remove`** does is to shuffle values around so that the elements that shouldn't be "removed" get shuffled up toward the beginning of the range, and return an iterator to one past the end of the unremoved elements. To really get rid of them, the call to **`remove`** needs to be followed by a call to **`erase`** hence the "erase-remove" idiom. For example, to erase all elements equal to **`value`** from a container **`c`**, you can write:

```
c.erase( remove( c.begin(), c.end() ,value ) );
```

Prefer to use a member version of **`remove`** or **`remove_if`** on a container that has it.

Exceptions

The usual shrink-to-fit idiom won't work on copy-on-write implementations of `std::string`. What usually does work is to call `s.reserve(0)` or to fake the `string` out by writing `string(s.begin(), s.end()).swap(s);` to use the iterator range constructor. In practice, these work to shed excess capacity. (Better still, `std::string` implementations are abandoning copy-on-write, which is an outdated optimization; see [Sutter02].)

References

STL: Algorithms

Prefer algorithms to loops.

Bjarne Stroustrup, [Stroustrup00

83. Use a checked STL implementation

Summary

Safety first (see Item 6): Use a checked STL implementation, even if it's only available for one of your compiler platforms, and even if it's only used during pre-release testing.

Discussion

Just like pointer mistakes, iterator mistakes are far too easy to make and will usually silently compile but then crash (at best) or appear to work (at worst). Even though your compiler doesn't catch the mistakes, you don't have to rely on "correction by visual inspection," and shouldn't: Tools exist. Use them.

Some STL mistakes are distressingly common even for experienced programmers:

- *Using an invalidated or uninitialized iterator:* The former in particular is easy to do.
- *Passing an out-of-bounds index* : For example, accessing element 113 of a 100-element container.
- *Using an iterator "range" that isn't really a range:* Passing two iterators where the first doesn't precede the second, or that don't both refer into the same container.
- *Passing an invalid iterator position:* Calling a container member function that takes an iterator position, such as the position passed to **`insert`**, but passing an iterator that refers into a different container.
- *Using an invalid ordering:* Providing an invalid ordering rule for ordering an associative container or as a comparison criterion with the sorting algorithms. (See [Meyers01

Examples

Example 1: Using an invalid iterator. It's easy to forget when iterators are invalidated and use an invalid iterator (see Item 99). Consider this example adapted from [Meyers01] that inserts elements at the front of a **deque**:

```
deque<double>::iterator current = d.begin();

for( size_t i = 0; i < max; ++i )
  d.insert( current++, data[i] + 41 );        // do you see the bug?
```

Quick: Do you see the bug? You have three seconds.Ding! If you didn't get it in time, don't worry; it's a subtle and understandable mistake. A checked STL implementation will detect this error for you on the second loop iteration so that you don't need to rely on your unaided visual acuity. (For a fixed version of this code, and superior alternatives to such a naked loop, see Item 84.)

Example 2: Using an iterator range that isn't really a range. An iterator range is a pair of iterators **first** and **last** that refer to the first element and the one-past-the-end-th element of the range, respectively. It is required that **last** be reachable from **first** by repeated increments of **first**. There are two common ways to accidentally try to use an iterator range that isn't actually a range: The first way arises when the two iterators that delimit the range point into the same container, but the first iterator doesn't actually precede the second:

```
for_each( c.end(), c.begin(), Something );    // not always this obvious
```

On each iteration of its internal loop, **for_each** will compare the first iterator with the second for equality, and as long as they are not equal it will continue to increment the first iterator. Of course, no matter how many times you increment the first iterator, it will never equal the second, so the loop is essentially endless. In practice, this will, at best, fall off the end of the container **c** and crash immediately with a memory protection fault. At worst, it will just fall off the end into uncharted memory and possibly read or change values that aren't part of the container. It's not that much different in principle from our infamous and eminently attackable friend the buffer overrun.

The second common case arises when the iterators point into different containers:

```
for_each( c.begin(), d.end(), Something );    // not always this obvious
```

The results are similar. Because checked STL iterators remember the containers that they refer into, they can detect such run-time errors.

References

[Dinkumware-Safe] ?[Horstmann95] ?[Josuttis99

84. Prefer algorithm calls to handwritten loops

Summary

Use function objects judiciously: For very simple loops, handwritten loops can be the simplest and most efficient solution. But writing algorithm calls instead of handwritten loops can be more expressive and maintainable, less error-prone, and as efficient.

When calling algorithms, consider writing your own custom function object that encapsulates the logic you need. Avoid cobbling together parameter-binders and simple function objects (e.g., **`bind2nd`**, **`plus`**), which usually degrade clarity. Consider trying the [Boost] Lambda library, which automates the task of writing function objects.

Discussion

Programs that use the STL tend to have fewer explicit loops than non-STL programs, replacing low-level semantics-free loops with higher-level and better-defined abstract operations that convey greater semantic information. Prefer a "process this range" algorithmic mindset over "process each element" loopy thinking.

A primary benefit that algorithms and design patterns have in common is that they let us speak at a higher level of abstraction with a known vocabulary. These days, we don't say "let many objects watch one object and get automatic notifications when its state changes;" rather, we say just "Observer." Similarly, we say "Bridge," "Factory," and "Visitor." Our shared pattern vocabulary raises the level, effectiveness, and correctness of our discussion. With algorithms, we likewise don't say "perform an action on each element in a range and write the results somewhere;" rather, we say `transform`. Similarly, we say `for_each`, `replace_if` , and `partition`. Algorithms, like design patterns, are self-documenting. Naked `for` and `while` loops just don't `do` when it comes to imparting any inherent semantic information about the purpose of the loop; they force readers to inspect their loop bodies to decipher what's going on.

Algorithms are also more likely to be correct than loops. Handwritten loops easily make mistakes such as using invalidated iterators (see Items 83 and 99); algorithms come already debugged for iterator invalidation and other common errors.

Finally, algorithms are also often more efficient than naked loops (see [Sutter00] and [Meyers01]). They avoid needless minor inefficiencies, such as repeated evaluations of `container.end()`. Slightly more importantly, the standard algorithms you're using were implemented by the same people who implemented the standard containers you're using, and by dint of their inside knowledge those people can write algorithms that are more efficient than any version you would write. Most important of all, however, is that many algorithms have highly sophisticated implementations that we in-the-trenches programmers are unlikely ever to match in handwritten code (unless we don't need the full generality of everything a given algorithm does).

In general, the more widely used a library is, the better debugged and more efficient it will be simply because it has so many users. You are unlikely to be using any other library that is as widely used as your standard library implementation. Use it and benefit. STL's algorithms are already written; why write them again?

Consider trying [Boost] lambda functions. Lambda functions are an important tool that solves the biggest drawback of algorithms, namely readability: They write the function objects for you, leaving the actual code in place at the call point. Without lambda functions, your choices are to either use function objects (but then even simple loop bodies live in a separate place away from the call point) or else use the standard binders and function objects such as `bind2nd` and `plus` (these are tedious and confusing to use, and hard to combine because fundamental operations like `compose` are not part of the standard; but do consider the [C++TR104] bind library).

Examples

Here are two examples adapted from [Meyers01]:

Example 1: Transforming a ***`deque`*****.** After working through several incorrect iterations that ran afoul of iterator invalidation issues (e.g., see Item 83), we finally come up with the following correct handwritten loop for adding **`41`** to every element of **`data`**, an array of **`double`**s, and placing the result at the beginning of **`d`**, a **`deque<double>`**:

```
deque<double>::iterator current = d.begin();

for( size_t i = 0; i < max; ++i ) {
  current = d.insert( current, data[i] + 41 );    // be careful to keep
```

Exceptions

When used with function objects, algorithm calls place the body of the loop away from the call site, which can make the loop harder to read. (Cobbling together simple objects with the standard and nonstandard binders isn't a realistic option.)

[Boost] lambda functions solve these two problems and work reliably on modern compilers, but they don't work on older compilers and they can generate dense error messages when they're coded incorrectly. Calling named functions, including member functions, still requires binder-like syntax.

References

[Allison98

85. Use the right STL search algorithm

Summary

Search "just enough"the right search may be STL (slower than light), but it'll still be pretty fast: This Item applies to searching for a particular value in a range, or for the location where it would be if it were in the range. To search an unsorted range, use **`find`**/**`find_if`** or **`count`**/ **`count_if`**. To search a sorted range, use **`lower_bound`**, **`upper_bound`**, **`equal_range`**, or (rarely) **`binary_search`**. (Despite its common name, **`binary_search`** is usually not the right choice.)

Discussion

For unsorted ranges, **`find`**/**`find_if`** and **`count`**/**`count_if`** can tell you in linear time whether and where, respectively, an element exists in a range. Note that **`find`**/**`find_if`** is typically more efficient because it can stop searching when a match is found.

For sorted ranges, prefer the four binary search algorithms, **`binary_search`**, **`lower_bound`**, **`upper_bound`**, and **`equal_range`**, which work in logarithmic time. Alas, despite its nice name, **`binary_search`** is nearly always useless because it only returns a **`bool`** indicating whether a match was found or not. Usually, you want **`lower_bound`** or **`upper_bound`**or **`equal_range`**, which gives you the results of both **`lower_bound`** and **`upper_bound`** (but not at twice the cost).

`lower_bound` returns an iterator pointing to the first match (if there is one) or the location where it would be (if there is not); the latter is useful to find the right place to insert new values into a sorted sequence. **`upper_bound`** returns an iterator pointing to one past the last match (if there is one), which is the location where the next equivalent element can be added; this is useful to find the right place to insert new values into a sorted sequence while keeping equivalent elements in the order in which they were inserted.

Prefer **`p = equal_range( first, last, value ); distance( p.first, p.second );`** as a faster version of **`count( first, last, value );`** for sorted ranges.

If you are searching an associative container, prefer using the member functions with the same names instead of the nonmember algorithms. The member versions are usually more efficient, including that the member version of **`count`** runs in logarithmic time (and so there's no motivation to replace a call the member **`count`** with a call to **`equal_range`** followed by **`distance`**, as there is with the nonmember **`count`**).

References

[Austern99

86. Use the right STL sort algorithm

Summary

Sort "just enough:" Understand what each of the sorting algorithms does, and use the cheapest algorithm that does what you need.

Discussion

You don't always need a full `sort`; you usually need less, and rarely you need more. In general order from cheapest to most expensive, your standard sorting algorithm options are: `partition`, `stable_partition`, `nth_element`, `partial_sort` (with its variant `partial_sort_copy`), `sort`, and `stable_sort`. Use the least expensive one that does the work you actually need; using a more powerful one is wasteful.

`partition`, `stable_partition` , and `nth_element` run in linear time, which is nice.

`nth_element`, `partial_sort`, `sort`, and `stable_sort` require random-access iterators. You can't use them if you have only bidirectional iterators (e.g., `list<T>::iterator`s). If you need these algorithms but you don't have random-access iterators, consider using the index container idiom: Create a container that does support random-access iterators (e.g., a `vector`) of iterators into the range you have, and then use the more powerful algorithm on that using a dereferencing version of your predicate (one that dereferences the iterators before doing its usual comparison).

Use the

Examples

Example 1: ***`partition`*.** Use `partition` to just divide the range into two groups: all elements that satisfy the predicate, followed by all elements that don't. This is all you need to answer questions like these:

- "Who are all the students with a grade of B+ or better?" For example, `partition( students.begin(), students.end(), GradeAtLeast("B+") );` does the work and returns an iterator to the first student whose grade is not at least B+.
- "What are all the products with weight less than 10kg?" For example, `partition( products.begin(), products.end(), WeightUnder(10) );` does the work and returns an iterator to the first product whose weight is 10kg or less.

Example 2: ***`nth_element`*.** Use `nth_element` to put a single element in the correct `n`-th position it would occupy if the range were completely sorted, and with all other elements correctly preceding or following that `n`-th element. This is all you need to answer questions like these:

- "Who are my top 20 salespeople?" For example, `nth_element( s.begin(), s.begin()+19, s.end(), SalesRating );` puts the 20 best elements at the front.
- "What is the item with the median level of quality in this production run?" That element would be in the middle position of a sorted range. To find it, do `nth_element( run.begin(), run.begin()+run.size()/2, run.end(), ItemQuality );`.
- "What is the item whose quality is at the 75th percentile?" That item would be 25% of the way through the sorted range. To find it, do `nth_element( run.begin(), run.begin()+run.size()*.25, run.end(), ItemQuality );`.

Example 3: ***`partial_sort`*.** `partial_sort` does the work of `nth_element`, plus the elements preceding the `n`-th are all in their correct sorted positions. Use `partial_sort` to answer questions similar to those `nth_element` answers, but where you need the elements that match to be sorted (and those that don't match don't need to be sorted). This is all you need to answer questions like, "Who are the first-, second-, and third-place winners?" For example, `partial_sort( contestants.begin(), contestants.begin()+3, contestants.end(), ScoreCompare );` puts the top three contestants, in order, at the front of the containerand no more.

Exceptions

Although `partial_sort` is usually faster than a full `sort` because it has to do less work, if you are going to be sorting most (or all) of the range, it can be slower than a full `sort`.

References

[Austern99

87. Make predicates pure functions

Summary

Predicate purity: A predicate is a function object that returns a yes/no answer, typically as a `bool` value. A function is pure in the mathematical sense if its result depends only on its arguments (note that this use of "pure" has nothing to do with pure virtual functions).

Don't allow predicates to hold or access state that affects the result of their `operator()`, including both member and global state. Prefer to make `operator()` a `const` member function for predicates (see Item 15).

Discussion

Algorithms make an unknowable number of copies of their predicates at unknowable times in an unknowable order, and then go on to pass those copies around while casually assuming that the copies are all equivalent.

That is why it's your responsibility to make sure that copies of predicates are indeed all equivalent, and that means that they must be pure functions: functions whose result is not affected by anything other than the arguments passed to **`operator()`**. Additionally, predicates must also consistently return the same result for the same set of arguments they are asked to evaluate.

Stateful predicates may seem useful, but they are explicitly *not* very useful with the C++ standard library and its algorithms, and that is intentional. In particular, stateful predicates can only be useful if:

- *The predicate is not copied:* The standard algorithms make no such guarantee, and in fact assume that predicates are safely copyable.
- *The predicate is applied in a documented deterministic order:* The standard algorithms generally make no guarantee about the order in which the predicate will be applied to the elements in the range. In the absence of guarantees about the order in which objects will be visited, operations like "flag the third element" (see Examples) make little sense, because which element will be visited "third" is not well-defined.

It is possible to work around the first point by writing a lightweight predicate that uses reference-counting techniques to share its deep state. That solves the predicate-copying problem too because the predicate can be safely copied without changing its semantics when it is applied to objects. (See [Sutter02].) It is not possible, however, to work around the second point.

Always declare a predicate type's **`operator()`** as a const member function so that the compiler can help you avoid this mistake by emitting an error if you try to change any data members that the predicate type may have. This won't prevent all abusesfor example, it won't flag accesses to global databut it will help the compiler to help you avoid at least the most common mistakes.

Examples

Example: ***`FlagNth`*.** Here is the classic example from [Sutter02], which is intended to remove the third element from a container **`v`**:

```
class FlagNth {
public:
  FlagNth( size_t n ) : current_(0), n_(n) {}

  // evaluate to true if and only if this is the n-th invocation
  template<typename T>
  bool operator()( const T& ) {return ++current_ == n_; }                  //
bad: non-const

private:
  size_t current_, n_;
};
```

References

[Austern99

88. Prefer function objects over functions as algorithm and comparer arguments

Summary

Objects plug in better than functions: Prefer passing function objects, not functions, to algorithms. Comparers for associative containers must be function objects. Function objects are adaptable and, counterintuitively, they typically produce faster code than functions.

Discussion

First, function objects are easy to make adaptable, and always should be (see Item 89). Even if you already have a function, sometimes you have to wrap it in **`ptr_fun`** or **`mem_fun`** anyway to add adaptability. For example, you have to do this in order to build up more complex expressions using binders (see also Item 84):

```
inline bool IsHeavy( const Thing& ) {
```

References

[*Austern99*

89. Write function objects correctly

Summary

Be cheap, be adaptable: Design function objects to be values that are cheap to copy. Where possible, make them adaptable by inheriting from **`unary_`**- or **`binary_function`**.

Discussion

Function objects are modeled on function pointers. Like function pointers, the convention is to pass them by value. All of the standard algorithms pass objects by value, and your algorithms should too. For example:

```
template<class InputIter, class Func>
Function for_each( InputIter first, InputIter last, Function f );
```

Therefore, function objects must be cheap to copy and monomorphic (immune to slicing, so avoid virtual functions; see Item 54). But large and/or polymorphic objects are useful, and using them is okay; just hide the size and richness using the Pimpl idiom (see Item 43), which leaves the outer class as the required cheap-to-copy monomorphic type that still accesses rich state. The outer class should:

- *Be adaptable:* Inherit from **`unary_function`** or **`binary_function`**. (See below.)
- *Have a Pimpl:* Store a pointer (e.g., **`shared_ptr`**) to a large/rich implementation.
- *Have the function call operator(s):* Pass these through to the implementation object.

That should be all that's needed in the outer class, other than possibly providing non-default versions of construction, assignment, and/or destruction.

Function objects should be adaptable. The standard binders and adapters rely on certain typedefs, which are provided most conveniently when your function object derives from **`unary_function`** or **`binary_function`**. Instantiate **`unary_function`** or **`binary_function`** with the same types as your **`operator()`** takes and returns, except that for each non-pointer type strip off any top-level **`const`**s and **`&`**s from the type.

Avoid providing multiple **`operator()`** functions, because that makes adaptability difficult. It's usually impossible to provide the right adaptability typedefs because the same typedef would have different values for different **`operator()`** functions.

Not all function objects are predicates. Predicates are a subset of function objects. (See Item 87.)

References

[Allison98

Type Safety

Trying to outsmart a compiler defeats much of the purpose of using one.

Brian Kernighan & P.J. Plauger

If you lie to the compiler, it will get its revenge.

Henry Spencer

There will always be things we wish to say in our programs that in all known languages can only be said poorly.

Alan Perlis

Last, but certainly not least, we will consider type correctnessa very important property of a program that you should strive to preserve at all times. Theoretically, a type-correct function can never access untyped memory or return forged values. Practically, if your code maintains type soundness, it avoids a large category of nasty errors ranging from nonportability to corrupting memory to creating bogus values to exhibiting undefined behavior.

The basic idea underpinning how to maintain type soundness is to always read bits in the format in which they were written. Sometimes, C++ makes it easy to break this rule, and the following Items detail how to avoid such mistakes.

Our vote for the most valuable Item in this section goes to Item 91: Rely on types, not on representations. The type system is your friend and your staunchest ally; enlist its help, and try never to abuse its trust.

90. Avoid type switching; prefer polymorphism

Summary

Switch off: Avoid switching on the type of an object to customize behavior. Use templates and virtual functions to let types (not their calling code) decide their behavior.

Discussion

Type switching to customize behavior is brittle, error-prone, unsafe, and a clear sign of attempting to write C or Fortran code in C++. It is a rigid technique that forces you to go back and do surgery on existing code whenever you want to add new features. It is also unsafe because the compiler will not tell you if you forget to modify all of the switches when you add a type.

Ideally, adding new features to a program equates to adding more new code (see Item 37). In reality, we know that that's not always trueoftentimes, in addition to writing new code, we need to go back and modify some existing code. Changing working code is undesirable and should be minimized, however, for two reasons: First, it might break existing functionality. Second, it doesn't scale well as the system grows and more features are added, because the number of "maintenance knots" that you need to go back and change increases as well. This observation led to the Open-Closed principle that states: An entity (e.g., class or module) should be open for extension but closed for modification. (See [Martin96c] and [Meyer00].)

How can we write code that can be easily extended without modifying it? Use polymorphism by writing code in terms of abstractions (see also Item 36) and then adding various implementations of those abstractions as you add functionality. Templates and virtual function calls form a dependency shield between the code using the abstractions and the code implementing them (see Item 64).

Of course, managing dependencies this way depends on finding the right abstractions. If the abstractions are imperfect, adding new functionality will require changing the interface (not just adding new implementations of the interface), which usually requires changes to existing code. But abstractions are called "abstractions" for a reasonthey are supposed to be much more stable than the "details," that is, the abstractions' possible implementations.

Contrast that with code that uses few or no abstractions, but instead traffics directly in concrete types and their specific operations. That code is "detailed" alreadyin fact, it is swimming in a sea of details, a sea in which it is doomed soon to drown.

Examples

Example: Drawing shapes. The classic example is drawing objects. A typical C-style, **`switch`** -on-type solution would define an enumerated member variable **`id_`** for each shape that stores the type of that shape: rectangle, circle, and so on. Drawing code looks up the type and performs specific tasks:

```
class Shape {
```

References

[Dewhurst03

91. Rely on types, not on representations

Summary

Don't try to X-ray objects (see Item 96): Don't make assumptions about how objects are exactly represented in memory. Instead, let types decide how their objects are written to and read from memory.

Discussion

The C++ Standard makes few guarantees about how types are represented in memory:

- Base two is guaranteed for integral numbers.
- Two's complement is guaranteed for negative integral numbers.
- Plain Old Data (POD) types have C-compatible layout: Member variables are stored in their order of declaration.
- **`int`** holds at least 16 bits.

In particular, the following may be common but are *not* guaranteed on all current architectures, and in particular are liable to be broken on newer architectures:

- **`int`** is not exactly 32 bits, nor any particular fixed size.
- Pointers and **`int`**s don't always have the same size and can't always be freely cast to one another.
- Class layout does not always store bases and members in declaration order.
- There can be gaps between the members of a class (even a POD) for alignment.
- **`offsetof`** only works for PODs, not all classes (but compilers might not emit errors).
- A class might have hidden fields.
- Pointers might not look like integers at all. If two pointers are ordered, and you cast them to integral values, the resulting values might not be ordered the same way.
- You can't portably assume anything about the placement in memory of automatic variables, or about the direction in which the stack grows.
- Pointers to functions might have a different size than **`void*`**, even though some APIs force you to assume that their sizes are the same.
- You can't always write just any object at just any memory address, even if you have enough space, due to alignment issues.

Just define types appropriately, then read and write data using those types instead of thinking bits and words and addresses. The C++ memory model ensures efficient execution without forcing you to rely on manipulating representation. So don't.

References

[Dewhurst03

92. Avoid using `reinterpret_cast`

Summary

Lies have short legs (German and Romanian proverb): Don't try to use `reinterpret_cast` to force the compiler to reinterpret the bits of an object of one type as being the bits of an object of a different type. That's the opposite of maintaining type safety, and `reinterpret_cast` isn't even guaranteed to do that or anything else in particular.

Discussion

Recall: *If you lie to the compiler, it will get its revenge.*

Henry Spencer

`reinterpret_cast` reflects the strongest assumption a programmer can make about object representation, namely that the programmer knows better than the compilerto the point of being determined to pick an argument with the compiler's carefully maintained type information. Compilers will shut up if you tell them to, but use of force should be a last resort. Avoid assuming how data is represented, because such assumptions dramatically affect the safety and reliability of your code.

Besides, the reality is that `reinterpret_cast`'s effects are worse than reinterpreting objects' bit patterns (which would be bad enough). Except that some conversions are guaranteed to be reversible, its effects are actually implementation-defined, so you don't know if it will do even that. It is unreliable and nonportable.

Exceptions

Some low-level system-specific programming might force you to use **`reinterpret_cast`** to stream bits in and out some port, or to transform integers in addresses. Use unsafe casting as rarely as you can and only in well-hidden functions that abstract it away, so as to make your code ready for porting with minimal changes. If you need to cast among unrelated pointer types, prefer casting via **`void*`** instead of using **`reinterpret_cast`** directly. That is, instead of

```
T1* p1 = ... ;
T2* p2 = reinterpret_cast<T2*>( p1 );
```

write

```
T1* p1 = ... ;
void* pV = p1;
T2* p2 = static_cast<T2*>( pV );
```

References

[C++03

93. Avoid using `static_cast` on pointers

Summary

Pointers to dynamic objects don't `static_cast`: Safe alternatives range from using `dynamic_cast` to refactoring to redesigning.

Discussion

Consider replacing uses of **static_cast** with its more powerful relative **dynamic_cast**, and then you won't have to remember when **static_cast** is safe and when it's dangerous. Although **dynamic_cast** can be slightly less efficient, it also detects illegal casting (and don't forget Item 8). Using **static_cast** instead of **dynamic_cast** is like eliminating the stairs night-light, risking a broken leg to save 90 cents a year.

Prefer to design away downcasting: Refactor or redesign your code so that it isn't needed. If you see that you are passing a **Base** to a function that really needs a **Derived** to do its work, examine the chain of calls to identify where the needed type information was lost; often, changing a couple of prototypes leads to an excellent solution that also clarifies the type information flow to you.

Excessive downcasts might indicate that the base class interface is too sparse. This can lead to designs that define more functionality in derived classes, and then downcast every time the extended interface is needed. The one good solution is to redesign the base interface to provide more functionality.

If, and only if, the overhead of the **dynamic_cast** actually matters (see Item 8), consider defining your own cast that uses **dynamic_cast** during debugging and **static_cast** in the "all speed no guarantees" mode (see [Stroustrup00]):

```
template<class To, class From> To checked_cast( From* from ) {
 assert( dynamic_cast<To>(from) == static_cast<To>(from) && "checked_cast
failed" );
 return static_cast<To>( from );
}

template<class To, class From> To checked_cast( From& from ) {
 assert( dynamic_cast<To>(from) == static_cast<To>(from) && "checked_cast
failed" );
 return static_cast<To>( from );
}
```

This little duo of functions (one each needed for pointers and references) simply tests whether the two casts agree. We leave it up to you to customize **checked_cast** for your needs, or to use one provided by a library.

References

[Dewhurst03

94. Avoid casting away `const`

Summary

Some fibs are punishable: Casting away `const` sometimes results in undefined behavior, and it is a staple of poor programming style even when legal.

Discussion

Once you go `const`, you (should) never go back. If you cast away `const` on an object whose original definition really did define it to be `const`, all bets are off and you are in undefined behavior land. For example, compilers can (and do) put constant data into ROM or write-protected RAM pages. Casting away `const` from such a truly `const` object is a punishable lie, and often manifests as a memory fault.

Even when it doesn't crash your program, casting away `const` is a broken promise and doesn't do what many expect. For example, this doesn't allocate a variable-sized array:

```
void Foolish( unsigned int n ) {
 const unsigned int size = 1;
 const_cast<unsigned int&>(size) = n;   // bad: don't do this
 char buffer[size];                       // will always have size 1
```

Exceptions

Casting away **`const`** can be necessary to call const-incorrect APIs (see Item 15). It is also useful when a function that must take and return the same kind of reference has **`const`** and non-**`const`** overloads, implemented by having one call the other:

```
const Object& f( const Object& );

Object& f( Object& obj ) {
  const Object& ref = obj;
  return const_cast<Object&>( foo(ref) );    // have to const_cast the return
type
}
```

References

[Dewhurst03

95. Don't use C-style casts

Team LiB

◀ PREVIOUS NEXT ▶

Summary

Age doesn't always imply wisdom: C-style casts have different (and often dangerous) semantics depending on context, all disguised behind a single syntax. Replacing C-style casts with C++-style casts helps guard against unexpected errors.

Team LiB

◀ PREVIOUS NEXT ▶

Discussion

One problem with C-style casts is that they provide one syntax to do subtly different things, depending on such vagaries as the files that you **#include**. The C++ casts, while retaining some of the dangers inherent in casts, avoid such ambiguities, clearly document their intent, are easy to search for, take longer to write (which makes one think twice)and don't silently inject evil **reinterpret_cast**s (see Item 92).

Consider the following code, where **Derived** inherits from **Base**:

```
extern void Fun( Derived* );

void Gun( Base* pb ) {
 // let's assume Gun knows for sure pb actually points to a Derived
 // and wants to forward to Fun
 Derived* pd = (Base*)pb;                     // bad: C-style cast
 Fun( pd );
}
```

If **Gun** has access to the definition of **Derived** (say by including **"derived.h"**), the compiler will have the necessary object layout information to make any needed pointer adjustments when casting from **Base** to **Derived**. But say **Gun**'s author forgets to **#include** the appropriate definition file, and **Gun** only sees a forward declaration of **class Derived;**. In that case, the compiler will just assume that **Base** and **Derived** are unrelated types, and will reinterpret the bits that form **Base*** as a **Derived***, without making any necessary adjustments dictated by object layout!

In short, if you forget to **#include** the definition, your code crashes mysteriously, even though it compiles without errors. Avoid the problem this way:

```
extern void Fun( Derived* );

void Gun( Base* pb ) {
 // if we know for sure that pb actually points to a Derived, use:
 Derived* pd = static_cast<Derived*>(pb);    // good: C++-style cast
 // otherwise: = dynamic_cast<Derived*>(pb); // good: C++-style cast
 Fun(pd);
}
```

Now, if the compiler doesn't have enough static information about the relationship between **Base** and **Derived**, it will issue an error instead of automatically performing a bitwise (and potentially lethal) **reinterpret_cast** (see Item 92).

The C++-style casts can protect the correctness of your code during system evolution as well. Say you have an **Employee**-rooted hierarchy and you need to define a unique employee ID for each **Employee**. You could define the ID to be a pointer to the **Employee** itself. Pointers uniquely identify the objects they point to and can be compared for equality, which is exactly what's needed. So you write:

```
typedef Employee* EmployeeID;

Employee& Fetch( EmployeeID id ) {
 return *id;
}
```

Say you code a fraction of the system with this design. Later on, it turns out that you need to save your records to a relational database. Clearly, saving pointers is not something that you'd want to do. So, you change the design such that each employee has a unique *integral* identifier. Then, the integral IDs can be persisted in the database, and a hash table maps the IDs to **Employee** objects. Now the **typedef** is:

PREVIOUS NEXT

References

[Dewhurst03

96. Don't `memcpy` or `memcmp` non-PODs

Summary

Don't try to X-ray objects (see Item 91): Don't use **`memcpy`** and **`memcmp`** to copy or compare anything more structured than raw memory.

Discussion

memcpy and **memcmp** violate the type system. Using **memcpy** to copy objects is like making money using a photocopier. Using **memcmp** to compare objects is like comparing leopards by counting their spots. The tools and methods might appear to do the job, but they are too coarse to do it acceptably.

C++ objects are all about information hiding (arguably the most profitable principle in software engineering; see Item 11): Objects hide data (see Item 41) and devise precise abstractions for copying that data through constructors and assignment operators (see Items 52 through 55). Bulldozing over all that with **memcpy** is a serious violation of information hiding, and often leads to memory and resource leaks (at best), crashes (worse), or undefined behavior (worst). For example:

```
{
  shared_ptr<int> p1( new int ), p2( new int );   // create two ints on the
heap
  memcpy( &p1, &p2, sizeof(p1) );                // bad: a heinous crime
}// memory leak: p2's int is never deleted
 // memory corruption: p1's int is deleted twice
```

Abusing **memcpy** can affect aspects as fundamental as the type and the identity of an object. Compilers often embed hidden data inside polymorphic objects (the so-called virtual table pointer, or **vptr**) that give the object its run-time identity. In the case of multiple inheritance, several such **vptr**s can coexist at various offsets inside the object, and most implementations add yet more internal pointers when using virtual inheritance. During normal use, the compiler takes care of managing all of these hidden fields; **memcpy** can only wreak havoc.

Similarly, **memcmp** is an inappropriate tool for comparing anything more elaborate than bits. Sometimes, it does too little (e.g., comparing C-style strings is not the same as comparing the pointers with which the strings are implemented). And sometimes, paradoxically, **memcmp** does too much (e.g., **memcmp** will needlessly compare bytes that are not part of an object's state, such as padding inserted by the compiler for alignment purposes). In both cases, the comparison's result will be wrong.

References

[Dewhurst03

97. Don't use unions to reinterpret representation

Summary

A deceit is still a lie: Unions can be abused into obtaining a "cast without a cast" by writing one member and reading another. This is more insidious and even less predictable than `reinterpret_cast` (see Item 92).

Discussion

Don't read a field of a **union** unless it was the field that was last written. Reading a different field of a **union** than the field that was last written has undefined behavior, and is even worse than doing a **reinterpret_cast** (see Item 92); at least with the latter the compiler has the fighting chance to warn and repel an "impossible reinterpretation" such as pointer to **char**. When abusing a **union**, no reinterpretation of bits will ever yield a compile-time error or a reliable result.

Consider this code that is intended to deposit a value of one type (**char***) and extract the bits of that value as a different type (**long**):

```
union {
 long intValue_;
 char* pointerValue_;
};

pointerValue_ = somePointer;
long int gotcha = intValue_;
```

This has two problems:

- *It assumes too much:* It assumes that **sizeof(long)** and **sizeof(char*)** are equal, and that the bit representations are identical. These things are not true on all implementations (see Item 91).
- *It obscures your intent for both human readers and compilers:* Playing **union** games makes it harder for compilers to catch genuine type errors, and for humans to spot logical errors, than even the infamous **reinterpret_cast** (see Item 92).

Exceptions

If two POD `struct`s are members of a `union` and start with the same field types, it is legal to write one such matching field and read another.

PREVIOUS NEXT

References

[Alexandrescu02b] ?[Stroustrup00

98. Don't use varargs (ellipsis)

Summary

Ellipses cause collapses: The ellipsis is a dangerous carryover from C. Avoid varargs, and use higher-level C++ constructs and libraries instead.

Discussion

Functions taking a variable number of arguments are a nice commodity, but C-style varargs aren't the way to get them. Varargs have many serious shortcomings:

- *Lack of type safety:* Essentially, the ellipsis tells the compiler: "Turn all checking off. I'll take over from here and start `reinterpret_cast`ing." (See Item 92.)
- *Tight coupling and required manual cooperation between caller and callee:* The language's type checking has been disabled, so the call site must use alternate ways to communicate the types of the arguments being passed. Such protocols (e.g., `printf`'s format string) are notoriously error-prone and unsafe because they cannot be fully checked or enforced by either the caller or the callee. (See Item 99.)
- *Undefined behavior for objects of class type:* Passing anything but primitive and POD (plain old data) types via varargs has undefined behavior in C++. Unfortunately, most compilers don't give a warning when you do it.
- *Unknown number of arguments:* For even the simplest variable-arguments function (e.g., `min`) with a variable number of arguments of known type (e.g., `int`), you still need to have a protocol for figuring out the number of arguments. (Ironically, this is a good thing because it further discourages using varargs.)

Avoid using varargs in your functions' signatures. Avoid calling functions with varargs in their own signatures, including legacy functions and standard C library functions such as `sprintf`. Admittedly, calls to `sprintf` can often look more compact and easier to read than equivalent calls using `stringstream` formatting and `operator<<`just like it's also admittedly easier to hop into a car without pesky safety belts and bumpers. The risks are just not worth it. `printf` -related vulnerabilities continue to be a serious security problem at the time of this writing (see [Cowan01]), and an entire subindustry of tools exists to help find such type errors (see [Tsai01]).

Prefer to use type-safe libraries that support variable arguments using other means. For example, the [Boost] `format` library uses advanced C++ features to combine safety with speed and convenience.

References

[Boost] ?[Cowan01] ?[Murray93

99. Don't use invalid objects. Don't use unsafe functions

Team LiB PREVIOUS NEXT

Summary

Don't use expired medicines: Both invalid objects and historical but unsafe functions wreak havoc on your program's health.

Team LiB PREVIOUS NEXT

Discussion

There are three major kinds of invalid objects:

- *Destroyed objects:* Typical examples are automatic objects that have gone out of scope and deleted heap-based objects. After you call an object's destructor, its lifetime is over and it is undefined and generally unsafe to do anything with it.
- *Semantically invalidated objects:* Typical examples include dangling pointers to deleted objects (e.g., a pointer `p` after a `delete p;`) and invalidated iterators (e.g., a `vector<T>::iterator i` after an insertion at the beginning of the container the iterator refers into). Note that dangling pointers and invalidated iterators are conceptually identical, and the latter often directly contain the former. It is generally undefined and unsafe to do anything except assign another valid value to an invalidated object (e.g., `p = new Object;` or `i = v.begin();`).
- *Objects that were never valid:* Examples include objects "obtained" by forging a pointer (using `reinterpret_cast`, see Item 92), or accessing past array boundaries.

Be aware of object lifetimes and validity. Never dereference an invalid iterator or pointer. Don't make assumptions about what `delete` does and doesn't do; freed memory is freed, and shouldn't be subsequently accessed under any circumstances. Don't try to play with object lifetime by calling the destructor manually (e.g., `obj.~T()`) and then calling placement `new`. (See Item 55.)

Don't use the unsafe C legacy: `strcpy`, `strncpy`, `sprintf`, or any other functions that do write to range-unchecked buffers, and/or do not check and correctly handle out-of-bounds errors. C's `strcpy` does not check limits, and [C99]'s `strncpy` checks limits but does not append a null if the limit is hit; both are crashes waiting to happen and security hazards. Use more modern, safer, and more flexible structures and functions, such as those found in the C++ standard library (see Item 77). They are not always perfectly safe (for the sake of efficiency), but they are much less prone to errors and can be better used to build safe code.

References

[C99] ?[Sutter00

100. Don't treat arrays polymorphically

Summary

Arrays are ill-adjusted: Treating arrays polymorphically is a gross type error that your compiler will probably remain silent about. Don't fall into the trap.

Discussion

Pointers serve two purposes at the same time: that of monikers (small identifiers of objects), and that of array iterators (they can walk through arrays of objects using pointer arithmetic). As monikers, it makes a lot of sense to treat a pointer to `Derived` as a pointer to `Base`. As soon as the array iteration part enters the stage, however, such substitutability breaks down because an array of `Derived` isn't the same as an array of `Base`. To illustrate: Mice and elephants are both mammals, but that doesn't mean a convoy of a thousand elephants would be as long as one of a thousand mice.

Size does matter. When substituting a pointer to `Derived` to a pointer to `Base`, the compiler knows exactly how to adjust the pointer (if necessary) because it has enough information about both classes. However, when doing pointer arithmetic on a pointer `p` to `Base`, the compiler computes `p[n]` as `*(p + n * sizeof(Base))`, thus assuming that the objects lying in memory are all `Base` objectsand not objects of some derived type that might have a different size. Imagine, now, just how easy it is to tromp all over of an array of `Derived` if you convert the pointer marking its start to `Base*` (with compiler's silent approval) and then perform pointer arithmetic on that pointer (while the compiler doesn't blink an eye either)!

Such accidents are an unfortunate interaction between substitutability, which dictates that pointers to derived classes should be usable as pointers to their bases, and C's legacy pointer arithmetic, which assumes pointers are monomorphic and uses solely static information to compute strides.

To store arrays of polymorphic objects, you need an array (or, better still, a real container; see Item 77) of pointers to the base class (e.g., plain pointers or, better still, `shared_ptr`s; see Item 79). Then each pointer in the array refers to a polymorphic object, likely an object allocated on the free store. Or, if you want to expose an interface to a container of polymorphic objects, you need to encapsulate the entire array and offer a polymorphic interface for iteration.

Incidentally, a good reason to prefer references to pointers in interfaces is to make it clear that you're talking about one object, as opposed to possibly an array of them.

References

[C++TR104] ?[Dewhurst03

Bibliography

Note: For browsing convenience, this bibliography is also available online at:

http://www.gotw.ca/publications/c++cs/bibliography.htm

The bold references (e.g., **[Abrahams96]**) are hyperlinks in the online bibliography.

[Abelson96] H. Abelson and G. J. Sussman . *Structure and Interpretation of Computer Programs (2nd Edition)* (MIT Press, 1996).

[Abrahams96] D. Abrahams . "Exception Safety in STLport" (STLport website, 1996).

[Abrahams01a] D. Abrahams . "Exception Safety in Generic Components," in M. Jazayeri, R. Loos, D. Musser (eds.), *Generic Programming: International Seminar on Generic Programming, Dagstuhl Castle, Germany, April/May 1998, Selected Papers*, Lecture Notes in Computer Science 1766 (Springer, 2001).

[Abrahams01b] D. Abrahams . "Error and Exception Handling" ([Boost] website, 2001).

[Alexandrescu00a] A. Alexandrescu . "Traits: The else-if-then of Types" (*C++ Report*, 12(4), April 2000).

[Alexandrescu00b] A. Alexandrescu . "Traits on Steroids" (*C++ Report*, 12(6), June 2000).

[Alexandrescu00c] A. Alexandrescu and P. Marginean . "Change the Way You Write Exception-Safe CodeForever" (*C/C++ Users Journal*, 18(12), December 2000).

[Alexandrescu01] A. Alexandrescu . *Modern C++ Design* (Addison-Wesley, 2001).

[Alexandrescu01a] A. Alexandrescu . "A Policy-Based basic_string Implementation" (*C/C++ Users Journal*, 19(6), June 2001).

[Alexandrescu02a] A. Alexandrescu . "Multithreading and the C++ Type System" (InformIT website, February 2002).

[Alexandrescu02b] A. Alexandrescu .

Summary of Summaries

Organizational and Policy Issues

0. Don't sweat the small stuff. (Or: Know what not to standardize.)

Say only what needs saying: Don't enforce personal tastes or obsolete practices.

1. Compile cleanly at high warning levels.

Take warnings to heart: Use your compiler's highest warning level. Require clean (warning-free) builds.

Understand all warnings. Eliminate warnings by changing your code, not by reducing the warning level.

2. Use an automated build system.

Push the (singular) button: Use a fully automatic ("one-action") build system that builds the whole project without user intervention.

3. Use a version control system.

The palest of ink is better than the best memory (Chinese proverb): Use a version control system (VCS).

Never keep files checked out for long periods. Check in frequently after your updated unit tests pass.

Ensure that checked-in code does not break the build.

4. Invest in code reviews.

Re-view code: More eyes will help make more quality. Show your code, and read others'. You'll all learn and benefit.

Design Style

5. Give one entity one cohesive responsibility.

> *Focus on one thing at a time: Prefer to give each entity (variable, class, function, namespace, module, library) one well-defined responsibility. As an entity grows, its scope of responsibility naturally increases, but its responsibility should not diverge.*

6. Correctness, simplicity, and clarity come first.

> *KISS (Keep It Simple Software): Correct is better than fast. Simple is better than complex. Clear is better than cute. Safe is better than insecure (see Items 83 and 99).*

7. Know when and how to code for scalability.

> *Beware of explosive data growth: Without optimizing prematurely, keep an eye on asymptotic complexity.*
>
> *Algorithms that work on user data should take a predictable, and preferably no worse than linear, timewith the amount of data processed. When optimization is provably necessary and important, and especially if it's because data volumes are growing, focus on improving big-Oh complexity rather than on micro-optimizations like saving that one extra addition.*

8. Don't optimize prematurely.

> *Spur not a willing horse (Latin proverb): Premature optimization is as addictive as it is unproductive.*
>
> *The first rule of optimization is: Don't do it. The second rule of optimization (for experts only) is: Don't do it yet. Measure twice, optimize once.*

9. Don't pessimize prematurely.

> *Easy on yourself, easy on the code: All other things being equal, notably code complexity and readability, certain efficient design patterns and coding idioms should just flow naturally from your fingertips and are no harder to write than the pessimized alternatives. This is not premature optimization; it is avoiding gratuitous pessimization.*

10. Minimize global and shared data.

> *Sharing causes contention: Avoid shared data, especially global data. Shared data increases coupling, which reduces maintainability and often performance.*

11. Hide information.

> *Don't tell: Don't expose internal information from an entity that provides an abstraction.*

12. Know when and how to code for concurrency.

> *Th$_{sa}$rea$_{fe}$d$_{ly}$: If your application uses multiple threads or processes, know how to minimize sharing objects where possible (see Item 10) and share the right ones safely.*

13. Ensure resources are owned by objects. Use explicit RAII and smart pointers.

> *Don't saw by hand when you have power tools: C++'s "resource acquisition is initialization" (RAII) idiom is the power tool for correct resource handling. RAIIallows the compiler to provide strong and automated guarantees that in other languages require fragile hand-coded idioms. When allocating a raw resou rce, immediately pass it to an owning object. Never allocate more than one resource in a single statement.*

Coding Style

14. Prefer compile- and link-time errors to run-time errors.

> *Don't put off 'til run time what you can do at build time: Prefer to write code that uses the compiler to check for invariants during compilation, instead of checking them at run time. Run-time checks are control- and data-dependent, which means you'll seldom know whether they are exhaustive. In contrast, compile-time checking is not control- or data-dependent and typically offers higher degrees of confidence.*

15. Use const proactively.

> **`const`** *is your friend: Immutable values are easier to understand, track, and reason about, so prefer constants over variables wherever it is sensible and make* **`const`** *your default choice when you define a value: It's safe, it's checked at compile time (see Item 14), and it's integrated with C++'s type system. Don't cast away* **`const`** *except to call a const-incorrect function (see Item 94).*

16. Avoid macros.

> **`TO_PUT_IT_BLUNTLY`***: Macros are the bluntest instrument of C and C++'s abstraction facilities, ravenous wolves in functions' clothing, hard to tame, marching to their own beat all over your scopes. Avoid them.*

17. Avoid magic numbers.

> *Programming isn't magic, so don't incant it: Avoid spelling literal constants like* **`42`** *or* **`3.14159`** *in code. They are not self-explanatory and complicate maintenance by adding a hard-to-detect form of duplication. Use symbolic names and expressions instead, such as* **`width * aspectRatio`**.

18. Declare variables as locally as possible.

> *Avoid scope bloat, as with requirements so too with variables): Variables introduce state, and you should have to deal with as little state as possible, with lifetimes as short as possible. This is a specific case of Item 10 that deserves its own treatment.*

19. Always initialize variables.

> *Start with a clean slate: Uninitialized variables are a common source of bugs in C and C++ programs. Avoid such bugs by being disciplined about cleaning memory before you use it; initialize variables upon definition.*

20. Avoid long functions. Avoid deep nesting.

> *Short is better than long, flat is better than deep: Excessively long functions and nested code blocks are often caused by failing to give one function one cohesive responsibility (see Item 5), and both are usually solved by better refactoring.*

21. Avoid initialization dependencies across compilation units.

> *Keep (initialization) order: Namespace-level objects in different compilation units should never depend on each other for initialization, because their initialization order is undefined. Doing otherwise causes headaches ranging from mysterious crashes when you make small changes in your project to severe nonportability even to new releases of the same compiler.*

22. Minimize definitional dependencies. Avoid cyclic dependencies.

> *Don't be over-dependent: Don't* **`#include`** *a definition when a forward declaration will do.*

Functions and Operators

25. Take parameters appropriately by value, (smart) pointer, or reference.

> *Parameterize well: Distinguish among input, output, and input/output parameters, and between value and reference parameters. Take them appropriately.*

26. Preserve natural semantics for overloaded operators.

> *Programmers hate surprises: Overload operators only for good reason, and preserve natural semantics; if that's difficult, you might be misusing operator overloading.*

27. Prefer the canonical forms of arithmetic and assignment operators.

> *If you* **`a+b`**, *also* **`a+=b`**: *When defining binary arithmetic operators, provide their assignment versions as well, and write to minimize duplication and maximize efficiency.*

28. Prefer the canonical form of ++ and --. Prefer calling the prefix forms.

> *If you* **`++c`**, *also* **`c++`**: *The increment and decrement operators are tricky because each has pre- and postfix forms, with slightly different semantics. Define* **`operator++`** *and* **`operator--`** *such that they mimic the behavior of their built-in counterparts. Prefer to call the prefix versions if you don't need the original value.*

29. Consider overloading to avoid implicit type conversions.

> *Do not multiply objects beyond necessity (Occam's Razor): Implicit type conversions provide syntactic convenience (but see Item 40). But when the work of creating temporary objects is unnecessary and optimization is appropriate (see Item 8), you can provide overloaded functions with signatures that match common argument types exactly and won't cause conversions.*

30. Avoid overloading &&, ||, or , (comma).

> *Wisdom means knowing when to refrain: The built-in* **`&&`**, **`||`** *, and* **`,`** *(comma) enjoy special treatment from the compiler. If you overload them, they become ordinary functions with very different semantics (you will violate Items 26 and 31), and this is a sure way to introduce subtle bugs and fragilities.*

Class Design and Inheritance

32. Be clear what kind of class you're writing.

> *Know thyself: There are different kinds of classes. Know which kind you are writing.*

33. Prefer minimal classes to monolithic classes.

> *Divide and conquer: Small classes are easier to write, get right, test, and use. They are also more likely to be usable in a variety of situations. Prefer such small classes that embody simple concepts instead of kitchen-sink classes that try to implement many and/or complex concepts (see Items 5 and 6).*

34. Prefer composition to inheritance.

> *Avoid inheritance taxes: Inheritance is the second-tightest coupling relationship in C++, second only to friendship. Tight coupling is undesirable and should be avoided where possible. Therefore, prefer composition to inheritance unless you know that the latter truly benefits your design.*

35. Avoid inheriting from classes that were not designed to be base classes.

> *Some people don't want to have kids: Classes meant to be used standalone obey a different blueprint than base classes (see Item 32). Using a standalone class as a base is a serious design error and should be avoided. To add behavior, prefer to add nonmember functions instead of member functions (see Item 44). To add state, prefer composition instead of inheritance (see Item 34). Avoid inheriting from concrete base classes.*

36. Prefer providing abstract interfaces.

> *Love abstract art: Abstract interfaces help you focus on getting an abstraction right without muddling it with implementation or state management details. Prefer to design hierarchies that implement abstract interfaces that model abstract concepts.*

37. Public inheritance is substitutability. Inherit, not to reuse, but to be reused.

> *Know what: Public inheritance allows a pointer or reference to the base class to actually refer to an object of some derived class, without destroying code correctness and without needing to change existing code. Know why: Don't inherit publicly to reuse code (that exists in the base class); inherit publicly in order to be reused (by existing code that already uses base objects polymorphically).*

38. Practice safe overriding.

> *Override responsibly: When overriding a virtual function, preserve substitutability; in particular, observe the function's pre- and post-conditions in the base class. Don't change default arguments of virtual functions. Prefer explicitly redeclaring overrides as* **`virtual`**. *Beware of hiding overloads in the base class.*

39. Consider making virtual functions nonpublic, and public functions nonvirtual.

> *In base classes with a high cost of change (particularly ones in libraries and frameworks): Prefer to make public functions nonvirtual. Prefer to make virtual functions private, or protected if derived classes need to be able to call the base versions. (Note that this advice does not apply to destructors; see Item 50.)*

40. Avoid providing implicit conversions.

> *Not all change is progress: Implicit conversions can often do more damage than good. Think twice before providing implicit conversions to and from the types you define, and prefer to rely on explicit conversions (*

Construction, Destruction, and Copying

47. Define and initialize member variables in the same order.

> *Agree with your compiler: Member variables are always initialized in the order they are declared in the class definition; the order in which you write them in the constructor initialization list is ignored. Make sure the constructor code doesn't confusingly specify a different order.*

48. Prefer initialization to assignment in constructors.

> *Set once, use everywhere: In constructors, using initialization instead of assignment to set member variables prevents needless run-time work and takes the same amount of typing.*

49. Avoid calling virtual functions in constructors and destructors.

> *Virtual functions only "virtually" always behave virtually: Inside constructors and destructors, they don't. Worse, any direct or indirect call to an unimplemented pure virtual function from a constructor or destructor results i n undefined behavior. If your design wants virtual dispatch into a derived class from a base class constructor or destructor, you need other techniques such as post-constructors.*

50. Make base class destructors public and virtual, or protected and nonvirtual.

> *To delete, or not to delete; that is the question: If deletion through a pointer to a base* `Base` *should be allowed, then* `Base`*'s destructor must be public and virtual. Otherwise, it should be protected and nonvirtual.*

51. Destructors, deallocation, and swap never fail.

> *Everything they attempt shall succeed: Never allow an error to be reported from a destructor, a resource deallocation function (e.g.,* `operator delete`*), or a swap function. Specifically, types whose destructors may throw an exception are flatly forbidden from use with the C++ standard library.*

52. Copy and destroy consistently.

> *What you create, also clean up: If you define any of the copy constructor, copy assignment operator, or destructor, you might need to define one or both of the others.*

53. Explicitly enable or disable copying.

> *Copy consciously: Knowingly choose among using the compiler-generated copy constructor and assignment operator, writing your own versions, or explicitly disabling both if copying should not be allowed.*

54. Avoid slicing. Consider Clone instead of copying in base classes.

> *Sliced bread is good; sliced objects aren't: Object slicing is automatic, invisible, and likely to bring wonderful polymorphic designs to a screeching halt. In base classes, consider disabling the copy constructor and copy assignment operator, and instead supplying a virtual* `Clone` *member function if clients need to make polymorphic (complete, deep) copies.*

55. Prefer the canonical form of assignment.

> *Your assignment: When implementing* `operator=`*, prefer the canonical formnonvirtual and with a specific signature.*

56. Whenever it makes sense, provide a no-fail swap (and provide it correctly).

> `swap` *is both a lightweight and a workhorse: Consider providing a* `swap` *function to efficiently and infallibly swap the internals of this object with another's.*

Namespaces and Modules

57. Keep a type and its nonmember function interface in the same namespace.

> *Nonmembers are functions too: Nonmember functions that are designed to be part of the interface of a class* `X` *(notably operators and helper functions) must be defined in the same namespace as the* `X` *in order to be called correctly.*

58. Keep types and functions in separate namespaces unless they're specifically intended to work together.

> *Help prevent name lookup accidents: Isolate types from unintentional argument-dependent lookup (ADL, also known as Koenig lookup), and encourage intentional ADL, by putting them in their own namespaces (along with their directly related nonmember functions; see Item 57). Avoid putting a type into the same namespace as a templated function or operator.*

59. Don't write namespace usings in a header file or before an #include.

> *Namespace* `using`*s are for your convenience, not for you to inflict on others: Never write a* `using` *declaration or a* `using` *directive before an* `#include` *directive.*
>
> *Corollary: In header files, don't write namespace-level* `using` *directives or* `using` *declarations; instead, explicitly namespace-qualify all names. (The second rule follows from the first, because headers can never know what other header* `#include`*s might appear after them.)*

60. Avoid allocating and deallocating memory in different modules.

> *Put things back where you found them: Allocating memory in one module and deallocating it in a different module makes your program fragile by creating a subtle long-distance dependency between those modules. They must be compiled with the same compiler version and same flags (notably debug vs.* `NDEBUG`*) and the same standard library implementation, and in practice the module allocating the memory had better still be loaded when the deallocation happens.*

61. Don't define entities with linkage in a header file.

> *Repetition causes bloat: Entities with linkage, including namespace-level variables or functions, have memory allocated for them. Defining such entities in header files results in either link-time errors or memory waste. Put all entities with linkage in implementation files.*

62. Don't allow exceptions to propagate across module boundaries.

> *Don't throw stones into your neighbor's garden: There is no ubiquitous binary standard for C++ exception handling. Don't allow exceptions to propagate between two pieces of code unless you control the compiler and compiler options used to build both sides; otherwise, the modules might not support compatible implementations for exception propagation. Typically, this boils down to: Don't let exceptions propagate across module/subsystem boundaries.*

63. Use sufficiently portable types in a module's interface.

> *Take extra care when living on the edge (of a module): Don't allow a type to appear in a module's external interface unless you can ensure that all clients understand the type correctly. Use the highest level of abstraction that clients can understand.*

Templates and Genericity

64. Blend static and dynamic polymorphism judiciously.

> *So much more than a mere sum of parts: Static and dynamic polymorphism are complementary. Understand their tradeoffs, use each for what it's best at, and mix them to get the best of both worlds.*

65. Customize intentionally and explicitly.

> *Intentional is better than accidental, and explicit is better than implicit: When writing a template, provide points of customization knowingly and correctly, and document them clearly. When using a template, know how the template intends for you to customize it for use with your type, and customize it appropriately.*

66. Don't specialize function templates.

> *Specialization is good only when it can be done correctly: When extending someone else's function template (including* **`std::swap`***), avoid trying to write a specialization; instead, write an overload of the function template, and put it in the namespace of the type(s) the overload is designed to be used for. (SeeItem 57.) When you write your own function template, avoid encouraging direct specialization of the function template itself.*

67. Don't write unintentionally nongeneric code.

> *Commit to abstractions, not to details: Use the most generic and abstract means to implement a piece of functionality.*

Error Handling and Exceptions

68. Assert liberally to document internal assumptions and invariants.

> *Be assertive! Use `assert` or an equivalent liberally to document assumptions internal to a module (i.e., where the caller and callee are maintained by the same person or team) that must always be true and otherwise represent programming errors (e.g., violations of a function's postconditions detected by the caller of the function). (See also Item 70.) Ensure that assertions don't perform side effects.*

69. Establish a rational error handling policy, and follow it strictly.

> *Consciously specify, and conscientiously apply, what so many projects leave to ad-hoc (mis)judgment: Develop a practical, consistent, and rational error handling policy early in design, and then stick to it. Ensure that it includes:*

- *Identification: What conditions are errors.*
- *Severity: How important or urgent each error is.*
- *Detection: Which code is responsible for detecting the error.*
- *Propagation: What mechanisms are used to report and propagate error notifications in each module.*
- *Handling: What code is responsible for doing something about the error.*
- *Reporting: How the error will be logged or users notified.*

> *Change error handling mechanisms only on module boundaries.*

70. Distinguish between errors and non-errors.

> *A breach of contract is an error: A function is a unit of work. Thus, failures should be viewed as errors or otherwise based on their impact on functions. Within a function `f`, a failure is an error if and only if it violates one of `f`'s preconditions or prevents `f` from meeting any of its callees' preconditions, achieving any of `f`'s own postconditions, or reestablishing any invariant that `f` shares responsibility for maintaining.*

> *In particular, here we exclude internal programming errors (i.e., where the caller and callee are the responsibility of the same person or team, such as inside a module), which are a separate category normally dealt with using assertions (see Item 68).*

71. Design and write error-safe code.

> *Promise, but don't punish: In each function, give the strongest safety guarantee that won't penalize callers who don't need it. Always give at least the basic guarantee.*

> *Ensure that errors always leave your program in a valid state. This is the basic guarantee. Beware of invariant-destroying errors (including but not limited to leaks), which are just plain bugs.*

> *Prefer to additionally guarantee that the final state is either the original state (if there was an error the operation was rolled back) or the intended target state (if there was no error the operation was committed). This is the strong guarantee.*

> *Prefer to additionally guarantee that the operation can never fail at all. Although this is not possible for most functions, it is required for functions like destructors and deallocation functions. This is the no-fail guarantee.*

72. Prefer to use exceptions to report errors.

STL: Containers

76. Use vector by default. Otherwise, choose an appropriate container.

> *Using the "right container" is great: If you have a good reason to use a specific container type, use that container type knowing that you did the right thing.*
>
> *So is using `vector`: Otherwise, write `vector` and keep going without breaking stride, also knowing you did the right thing.*

77. Use vector and string instead of arrays.

> *Why juggle Ming vases? Avoid implementing array abstractions with C-style arrays, pointer arithmetic, and memory management primitives. Using `vector` or `string` not only makes your life easier, but also helps you write safer and more scalable software.*

78. Use vector (and string::c_str) to exchange data with non-C++ APIs.

> *`vector` isn't lost in translation: `vector` and `string::c_str` are your gateway to communicate with non-C++ APIs. But don't assume iterators are pointers; to get the address of the element referred to by a `vector<T>::iterator iter`, use `&*iter`.*

79. Store only values and smart pointers in containers.

> *Store objects of value in containers: Containers assume they contain value-like types, including value types (held directly), smart pointers, and iterators.*

80. Prefer push_back to other ways of expanding a sequence.

> *`push_back` all you can: If you don't need to care about the insert position, prefer using `push_back` to add an element to sequence. Other means can be both vastly slower and less clear.*

81. Prefer range operations to single-element operations.

> *Don't use oars when the wind is fair (based on a Latin proverb): When adding elements to sequence containers, prefer to use range operations (e.g., the form of `insert` that takes a pair of iterators) instead of a series of calls to the single-element form of the operation. Calling the range operation is generally easier to write, easier to read, and more efficient than an explicit loop. (See alsoItem 84.)*

82. Use the accepted idioms to really shrink capacity and really erase elements.

> *Use a diet that works: To really shed excess capacity from a container, use the "swap trick." To really erase elements from a container, use the erase-remove idiom.*

STL: Algorithms

83. Use a checked STL implementation.

> *Safety first (see Item 6): Use a checked STL implementation, even if it's only available for one of your compiler platforms, and even if it's only used during pre-release testing.*

84. Prefer algorithm calls to handwritten loops.

> *Use function objects judiciously: For very simple loops, handwritten loops can be the simplest and most efficient solution. But writing algorithm calls instead of handwritten loops can be more expressive and maintainable, less error-prone, and as efficient.*
>
> *When calling algorithms, consider writing your own custom function object that encapsulates the logic you need. Avoid cobbling together parameter-binders and simple function objects (e.g., `bind2nd`, `plus`), which usually degrade clarity. Consider trying the [Boost] Lambda library, which automates the task of writing function objects.*

85. Use the right STL search algorithm.

> *Search "just enough"the right search may be STL (slower than light), but it'll still be pretty fast: This Item applies to searching for a particular value in a range, or for the location where it would be if it were in the range. To search an unsorted range, use `find`/`find_if` or `count`/`count_if`. To search a sorted range, use `lower_bound`, `upper_bound`, `equal_range`, or (rarely) `binary_search`. (Despite its common name, `binary_search` is usually not the right choice.)*

86. Use the right STL sort algorithm.

> *Sort "just enough:" Understand what each of the sorting algorithms does, and use the cheapest algorithm that does what you need.*

87. Make predicates pure functions.

> *Predicate purity: A predicate is a function object that returns a yes/no answer, typically as a `bool` value. A function is pure in the mathematical sense if its result depends only on its arguments (note that this use of "pure" has nothing to do with pure virtual functions).*
>
> *Don't allow predicates to hold or access state that affects the result of their `operator()`, including both member and global state. Prefer to make `operator()` a `const` member function for predicates (see Item 15).*

88. Prefer function objects over functions as algorithm and comparer arguments.

> *Objects plug in better than functions: Prefer passing function objects, not functions, to algorithms. Comparers for associative containers must be function objects. Function objects are adaptable and, counterintuitively, they typically produce faster code than functions.*

89. Write function objects correctly.

> *Be cheap, be adaptable: Design function objects to be values that are cheap to copy. Where possible, make them adaptable by inheriting from `unary_`- or `binary_function`.*

Type Safety

90. Avoid type switching; prefer polymorphism.

> *Switch off: Avoid switching on the type of an object to customize behavior. Use templates and virtual functions to let types (not their calling code) decide their behavior.*

91. Rely on types, not on representations.

> *Don't try to X-ray objects (see Item 96): Don't make assumptions about how objects are exactly represented in memory. Instead, let types decide how their objects are written to and read from memory.*

92. Avoid using reinterpret_cast.

> *Lies have short legs (German and Romanian proverb): Don't try to use* **`reinterpret_cast`** *to force the compiler to reinterpret the bits of an object of one type as being the bits of an object of a different type. That's the opposite of maintaining type safety, and* **`reinterpret_cast`** *isn't even guaranteed to do that oranything else in particular.*

93. Avoid using static_cast on pointers.

> *Pointers to dynamic objects don't* **`static_cast`***: Safe alternatives range from using* **`dynamic_cast`** *to refactoring to redesigning.*

94. Avoid casting away const.

> *Some fibs are punishable: Casting away* **`const`** *sometimes results in undefined behavior, and it is a staple of poor programming style even when legal.*

95. Don't use C-style casts.

> *Age doesn't always imply wisdom: C-style casts have different (and often dangerous) semantics depending on context, all disguised behind a single syntax. Replacing C-style casts with C++-style casts helps guard against unexpected errors.*

96. Don't memcpy or memcmp non-PODs.

> *Don't try to X-ray objects (see Item 91): Don't use* **`memcpy`** *and* **`memcmp`** *to copy or compare anything more structured than raw memory.*

97. Don't use unions to reinterpret representation.

> *A deceit is still a lie: Unions can be abused into obtaining a "cast without a cast" by writing one member and reading another. This is more insidious and even less predictable than* **`reinterpret_cast`** *(see Item 92).*

98. Don't use varargs (ellipsis).

> *Ellipses cause collapses: The ellipsis is a dangerous carryover from C. Avoid varargs, and use higher-level C++ constructs and libraries instead.*

99. Don't use invalid objects. Don't use unsafe functions.

> *Don't use expired medicines: Both invalid objects and historical but unsafe functions wreak havoc on your program's health.*

100. Don't treat arrays polymorphically.

> *Arrays are ill-adjusted: Treating arrays polymorphically is a gross type error that your compiler will probably remain silent about. Don't fall into the trap.*

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

[SYMBOL] [A] [B] [C] [D] [E] [F] [G] [H] [I] [J] [K] [L] [M] [N] [O] [P] [Q] [R] [S] [T] [U] [V] [W] [Z]