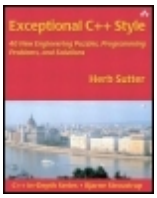
Exceptional C++ Style

40 New Engineering Puzzles, Programming Problems, and Solutions

Herb Sutter

C++ In-Depth Series • Bjarne Stroustrup



Exceptional C++ Style 40 New Engineering Puzzles, Programming Problems, and Solutions

By Herb Sutter

Publisher: Addison Wesley
Pub Date: August 02, 2004
ISBN: 0-201-76042-8
Pages: 352

- Table of Contents

Preface
Style or Substance?
The Exceptional Socrates
What I Assume You Know
How to Read This Book
##. The Topic of This Item
Acknowledgments
Generic Programming and the C++ Standard Library
Chapter 1. Uses and Abuses of vector
Solution
Chapter 2. The String Formatters of Manor Farm, Part 1: sprintf
Solution
Chapter 3. The String Formatters of Manor Farm, Part 2: Standard (or Blindingly Elegant) Alternatives
Solution
Chapter 4. Standard Library Member Functions
Solution
Chapter 5. Flavors of Genericity, Part 1: Covering the Basis [sic]
Solution
Chapter 6. Flavors of Genericity, Part 2: Generic Enough?
Solution
Chapter 7. Why Not Specialize Function Templates?
Solution
Chapter 8. Befriending Templates
Solution
Chapter 9. Export Restrictions, Part 1: Fundamentals
Solution
A Tale of Two Models
Illustrating the Issues
Export InAction [sic]
Issue the First: Source Exposure
Issue the Second: Dependencies and Build Times
Summary
Chapter 10. Export Restrictions, Part 2: Interactions, Usability Issues, and Guidelines
Solution
Exception Safety Issues and Techniques
Chapter 11. Try and Catch Me
Solution
Chapter 12. Exception Safety: Is It Worth It?
Solution
Chapter 13. A Pragmatic Look at Exception Specifications
Solution

Class Design, Inheritance, and Polymorphism
Chapter 14. Order, Order!
Solution
Chapter 15. Uses and Abuses of Access Rights
Solution
Chapter 16. (Mostly) Private
Solution
Chapter 17. Encapsulation
Solution
Chapter 18. Virtuality
Solution
Chapter 19. Enforcing Rules for Derived Classes
Solution
Memory and Resource Management
Chapter 20. Containers in Memory, Part 1: Levels of Memory Management
Solution
Chapter 21. Containers in Memory, Part 2: How Big Is It Really?
Solution
Chapter 22. To new, Perchance to throw, Part 1: The Many Faces of new
Solution
In-Place, Plain, and Nothrow new
Class-Specific new
A Name-Hiding Surprise
Summary
Chapter 23. To new, Perchance to throw, Part 2: Pragmatic Issues in Memory Management
Solution
Optimization and Efficiency
Chapter 24. Constant Optimization?
Solution
Chapter 25. inline Redux
Solution
Chapter 26. Data Formats and Efficiency, Part 1: When Compression Is the Name of the Game
Solution
Chapter 27. Data Formats and Efficiency, Part 2: (Even Less) Bit-Twiddling
Solution
Traps, Pitfalls, and Puzzlers
Chapter 28. Keywords That Aren't (or, Comments by Another Name)
Solution
Chapter 29. Is It Initialization?
Solution
Chapter 30. double or Nothing
Solution
Chapter 31. Amok Code
Solution
Chapter 32. Slight Typos? Graphic Language and Other Curiosities
Solution
Chapter 33. Operators, Operators Everywhere
Solution
Style Case Studies
Chapter 34. Index Tables
Solution
Chapter 35. Generic Callbacks
Solution
Chapter 36. Construction Unions
Solution

Chapter 37. Monoliths "Unstrung," Part 1: A Look at std::string
Solution
Summary
Chapter 38. Monoliths "Unstrung," Part 2: Refactoring std::string
Solution
Chapter 39. Monoliths "Unstrung," Part 3: std::string Diminishing
Solution
Chapter 40. Monoliths "Unstrung," Part 4: std::string Redux
Solution
Bibliography

Preface

The scene: Budapest. A hot summer evening. Looking across the Danube, with a view of the eastern bank.

In the cover photo showing this pastel-colored European scene, what's the first building that jumps out at you? Almost certainly it's the Parliament building on the left. The massive neo-Gothic building catches the eye with its graceful dome, thrusting spires, dozens of exterior statues and other ornate embellishments—and catches the eye all the more so because it stands in stark contrast to the more utilitarian buildings around it on the Danube waterfront.

Why the difference? For one thing, the Parliament building was completed in 1902; the other stark, utilitarian buildings largely date from Hungary's stark and utilitarian Communist era, between World War II and 1989.

"Aha," you might think, "that explains the difference. All very nice, of course, but what on earth does this have to do with Exceptional C++ Style?"

Certainly the expression of style has much to do with the philosophy and mindset that goes into it, and that is true whether we're talking about building architecture or software architecture. I feel certain that you have seen software designed on the scale and ornateness of the Parliament building; I feel equally sure that you have seen utilitarian blocky (or should that be "bloc-y"?) software buildings. On the extremes, I am just as convinced that you have seen many gilded lilies that err on the side of style, and many ugly ducklings that err on the side of pushing code out the door (and don't even turn out to be swans).

Style or Substance?

Which is better?

Don't be too sure you know the answer. For one thing, "better" is an unuseful term unless you define specific measures. Better for what? Better in which cases? For another, the answer is almost always a balance of the two and begins with: "It depends…"

This book is about finding that balance in many detailed aspects of software design and implementation in C++, and knowing your tools and materials well to know when they are appropriate.

Quick: Is the Parliament building a better building, crafted with better style, than the comparatively drab ones around it? It's easy to say yes unthinkingly—until you have to consider building and maintaining it:

- Construction. When it was completed in 1902, this was the largest Parliament building in the world. It also cost a horrendous amount of time, effort, and money to produce and was considered by many to be a "white elephant," which means a beautiful thing that comes at too high a cost. Consider: By comparison, how many of the ugly, drab, and perhaps outright boring concrete buildings could have been built for the same investment? And you work in an industry where the time-to-market pressure is far fiercer than the time pressure was in the age of this Parliament.
- Maintenance. Those of you familiar with the Parliament will note that in this picture the Parliament building was under renovation and had been in that state for a number of years, at a controversial and arguably ruinous cost. But there's more to the maintenance story than just this recent round of expensive renovations: Sadly, the beautiful sculptures you can see on the exterior of the building were made of the wrong materials, materials that were too soft. Soon after the building was originally completed, those sculptures became the subjects of a continual repair program under which they have been replaced with successively harder and more durable materials, and the heavy maintenance of the "bells and whistles" begun in the early 1900s has gone on continuously ever since—for the past century.

Likewise in software, it is important to find the right balance between construction cost and functionality, between elegance and maintainability, between the potential for growth and excessive ornateness.

We deal with these and similar tradeoffs every day as we go about software design and architecture in C++. Among the questions this book tackles are the following: Does making your code exception-safe make it better? If so, for what meanings of "better," and when might it not be better? That question is addressed specifically in this book. What about encapsulation; does it make your software better? Why? When doesn't it? If you're wondering, read on. Is inlining a good optimization, and when is it done? (Be very very careful when you answer this one.) What does C++'s export feature have in common with the Parliament building? What does std::string have in common with the monolithic architecture of the Danube waterfront in our idyllic little scene?

Finally, after considering many C++ techniques and features, at the end of this book we'll spend our last section looking at real examples of published code and see what the authors did well, what they did poorly, and what alternatives would perhaps have struck a better balance between workmanlike substance and exceptional C++ style.

What do I hope that this and the other Exceptional C++ books will help to do for you? I hope they will add perspective, add knowledge of details and interrelationships, and add understanding of how balances can be struck in your software's favor.

Please look one more time at the front cover photo, at the top right—that's it, right there. We should want to be in the balloon flying over the city, enjoying the full perspective of the whole view, seeing how style and substance coexist and interact and interrelate and intermingle, knowing how to make the tradeoffs and strike the right balances, each choice in its place in the integral and thriving whole.

The Exceptional Socrates

The Greek philosopher Socrates taught by asking his students questions—questions designed to guide them and help them draw conclusions from what they already knew and to show them how the things they were learning related to each other and to their existing knowledge. This method has become so famous that we now call it the Socratic method. From our point of view as students, Socrates' legendary approach involves us, makes us think, and helps us relate and apply what we already know to new information.

This book takes a page from Socrates, as did its predecessors, Exceptional C++ [Sutter00] and More Exceptional C++ [Sutter02]. It assumes you're involved in some aspect of writing production C++ software today, and it uses a question-answer format to teach how to make effective use of standard C++ and its standard library, with a particular focus on sound software engineering in modern C++. Many of the problems are drawn directly from experiences I and others have encountered while working with production C++ code. The goal of the questions is to help you draw conclusions from things you already know as well as things you've just learned, and to show how they interrelate. The puzzles will show how to reason about C++ design and programming issues—some of them common issues, some not so common; some of them plain issues, some more esoteric; and a couple because, well, just because they're fun.

This book is about all aspects of C++. I don't mean to say that it touches on every detail of C++—that would require many more pages—but rather that it draws from the wide palette of the C++ language and library features to show how apparently unrelated items can be used together to synthesize novel solutions to common problems. It also shows how apparently unrelated parts of the palette interrelate on their own, even when you don't want them to, and what to do about it. You will find material here about templates and namespaces, exceptions and inheritance, solid class design and design patterns, generic programming and macro magic—and not just as randomized tidbits, but as cohesive Items showing the interrelationships among all these parts of modern C++.

Exceptional C++ Style continues where Exceptional C++ and More Exceptional C++ left off. This book follows in the tradition of the first two: It delivers new material, organized in bite-sized Items and grouped into themed sections. Readers of the first book will find some familiar section themes, now including new material, such as exception safety, generic programming, and optimization and memory management techniques. The books overlap in structure and theme but not in content. This book continues the strong emphasis on generic programming and on using the C++ standard library effectively, including coverage of important template and generic programming techniques.

Versions of most Items originally appeared in magazine columns and on the Internet, particularly as print columns and articles I've written for C/C++ Users Journal, Dr. Dobb's Journal, the former C++ Report, and other publications, and also as Guru of the Week [GotW] issues #63 to #86. The material in this book has been significantly revised, expanded, corrected, and updated since those initial versions, and this book (along with its de rigueur errata list available at www.gotw.ca) should be treated as the current and authoritative version of that original material.

What I Assume You Know

I expect that you already know the basics of C++. If you don't, start with a good C++ introduction and overview. Good choices are a classic tome such as Bjarne Stroustrup's The C++ Programming Language [Stroustrup00] or Stan Lippman and Josée Lajoie's C++ Primer, Third Edition [Lippman98]. Next, be sure to pick up a style guide such as Scott Meyers' classic Effective C++ books [Meyers96, Meyers97]. I find the browser-based CD version [Meyers99] convenient and useful.

How to Read This Book

Each Item in this book is presented as a puzzle or problem, with an introductory header that resembles the following:

##. The Topic of This Item

Difficulty: #

A few words about what this Item will cover.

The topic tag and difficulty rating gives you a hint of what you're in for, and typically there are both introductory/review questions ("JG," a term for a new junior-grade military officer) leading to the main questions ("Guru"). Note that the difficulty rating is my subjective guess at how difficult I expect most people will find each problem, so you might well find that a "7" problem is easier for you than some "5" problem. Since writing Exceptional C++ [Sutter00] and More Exceptional C++ [Sutter02], I've regularly received e-mail saying that "Item #N is easier (or harder) than that!" It's common for different people to vote "easier!" and "harder!" for the same Item. Ratings are personal; any Item's actual difficulty for you depends on your knowledge and experience and could be easier or harder for someone else. In most cases, though, you should find the rating to be a reasonable guide to what to expect.

You might choose to read the whole book front to back; that's great, but you don't have to. You might decide to read all the Items in a section together because you're particularly interested in that section's topic; that's cool too. Except where there are what I call a "miniseries" of related problems which you'll see designated as Part 1, Part 2, and so on, the Items are pretty independent, and you should feel free to jump around, following the many cross-references among the Items in the book, as well as the many references to the first two Exceptional C++ books. The only guidance I'll offer is that the miniseries are designed to be read consecutively as a group; other than that, the choice is yours.

Unless I call something a complete program, it's probably not. Remember that the code examples are usually just snippets or partial programs and aren't expected to compile in isolation. You'll usually have to provide some obvious scaffolding to make a complete program out of the snippet shown.

Finally, a word about URLs: On the web, stuff moves. In particular, stuff I have no control over moves. That makes it a real pain to publish random web URLs in a print book lest they become out of date before the book makes it to the printer, never mind after it's been sitting on your desk for five years. When I reference other peoples' articles or web sites in this book, I do it via a URL on my own web site, www.gotw.ca, which I can control and which contains just a straight redirect to the real web page. Nearly all the other works I reference are listed in the Bibliography, and I've provided an online version with active hyperlinks. If you find that a link printed in this book no longer works, send me e-mail and tell me; I'll update that redirector to point to the new page's location (if I can find the page again) or to say that the page no longer exists (if I can't). Either way, this book's URLs will stay up to date despite the rigors of print media in an Internet world. Whew.

Acknowledgments

My thanks go first and most of all to my wife Tina for her enduring love and support, and to all my family for always being there, during this project and otherwise. Even when I had to "go dark" sometimes to crank out another few articles or edit another few Items, their patience knew no bounds. Without their patience and kindness this book would never have come to exist in its current form.

Our little puppy Frankie offered her own valuable contribution, namely wanting to play—even when I was working, thus forcing me to come up for air every once in a while. Frankie knows nothing whatever about software architecture or programming language design or even code micro-optimizations, but she's exuberantly happy anyway. Hmm.

Many thanks to series editor Bjarne Stroustrup, to editors Peter Gordon and Debbie Lafferty, and to Tyrrell Albaugh, Bernard Gaffney, Curt Johnson, Chanda Leary-Coutu, Charles Leddy, Malinda McCain, Chuti Prasertsith, and the rest of the Addison-Wesley team for their assistance and persistence during this project. It's hard to imagine a better bunch of people to work with, and their enthusiasm and cooperation has helped make this book everything I'd hoped it would become.

There is one other group of people who deserve thanks and credit, namely the many expert reviewers who generously offered their insightful comments and savage criticisms on all or part of this material exactly where needed. Their efforts have made the text you hold in your hands that much more complete, more readable, and more useful than it would otherwise have been. Special thanks for their technical feedback to series editor Bjarne Stroustrup, and to the following people who contributed comments on various parts of this material as it was developed: Dave Abrahams, Steve Adamczyk, Andrei Alexandrescu, Chuck Allison, Matt Austern, Joerg Barfurth, Pete Becker, Brandon Bray, Steve Dewhurst, Jonathan Caves, Peter Dimov, Javier Estrada, Attila Fehér, Marco Dalla Gasperina, Doug Gregor, Mark Hall, Kevlin Henney, Howard Hinnant, Cay Horstmann, Jim Hyslop, Mark E. Kaminsky, Dennis Mancl, Brian McNamara, Scott Meyers, Jeff Peil, John Potter, P. J. Plauger, Martin Sebor, James Slaughter, Nikolai Smirnov, John Spicer, Jan Christiaan van Winkel, Daveed Vandevoorde, and Bill Wade. The remaining errors, omissions, and shameless puns are mine, not theirs.

Herb Sutter
Seattle, May 2004

Generic Programming and the C++ Standard Library

One of C++'s most powerful features is its support for generic programming. This power is reflected directly in the flexibility of the C++ standard library, especially in its containers, iterators, and algorithms portion, originally known as the standard template library (STL).

Like More Exceptional C++ [Sutter02], this book opens with Items that focus our attention on some familiar parts of the STL, notably vector and string, as well as on some that might be less familiar. How can you avoid common gotchas when using the standard library's most basic container, vector ? How would you perform common C-style string manipulation in C++? What lessons, good and bad and down-right ugly, can we learn about library design from the STL itself?

After getting our feet wet with these opening looks into the predefined STL templates themselves, we'll delve into more general issues with templates and generic programming in C++. How can we avoid making our own templated code need-lessly (and quite unintentionally) nongeneric? Why is it actually a bad idea to specialize function templates, and what should we do instead? How can we correctly and portably do something as seemingly simple as grant friendship in the world of templates? And what's with this funny little export keyword, anyway?

This and more, as we delve into topics related to generic programming and the C++ standard library.

Chapter 1. Uses and Abuses of vector

Difficulty: 4

Almost everybody uses std::vector, and that's good. Unfortunately, many people misunderstand some of its semantics and end up unwittingly using it in surprising and dangerous ways. How many of the subtle problems illustrated in this Item might be lurking in your current program?

JG Question

1. Given a vector<int> v, what is the difference between the lines marked A and B?

```
void f(vector<int>& v) {
 v[0];       // A
 v.at(0);    // B
}
```

Guru Question

2. Consider the following code:

```
vector<int> v;

v.reserve(2);
assert(v.capacity() == 2);
v[0] = 1;
v[1] = 2;
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
 cout << *i << endl;
}

cout << v[0];
v.reserve(100);
assert(v.capacity() == 100);
cout << v[0];

v[2] = 3;
v[3] = 4;
// ...
v[99] = 100;
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
 cout << *i << endl;
}
```

Critique this code. Consider both style and correctness.

Solution

Accessing Vector Elements

1. Given a vector<int> v, what is the difference between the lines marked A and B?

```
// Example 1-1: [] vs. at
//
void f(vector<int>& v) {
 v[0];     // A
 v.at(0);  // B
}
```

In Example 1-1, if v is not empty then there is no difference between lines A and B. If v is empty, then line B is guaranteed to throw a std::out_of_range exception, but there's no telling what line A might do.

There are two ways to access contained elements within a vector. The first, vector<T>::at, is required to perform bounds-checking to ensure that the vector actually contains the requested element. It doesn't make sense to ask for, say, the 100th element in a vector that contains only 10 elements at the moment, and if you try to do such a thing, at will protest by throwing a std::out_of_range hissy fit (also known as an exception).

On the other hand, vector<T>::operator[] is allowed, but not required, to perform bounds-checking. There's not a breath of wording in the standard's specification for operator[] that says anything about bounds-checking, but neither is there any requirement that it have an exception specification, so your standard library implementer is free to add bounds-checking to operator[] too. If you use operator[] to ask for an element that's not in the vector, you're on your own, and the standard makes no guarantees about what will happen (although your standard library implementation's documentation might)—your program may crash immediately, the call to operator[] might throw an exception, or things might seem to work and occasionally and/or mysteriously fail.

Given that bounds-checking protects us against many common problems, why isn't operator[] required to perform bounds-checking? The short answer is: Efficiency. Always checking bounds would cause a (possibly slight) performance overhead on all programs, even ones that never violate bounds. The spirit of C++ includes the dictum that, by and large, you shouldn't have to pay for what you don't use, so bounds-checking isn't required for operator[]. In this case we have an additional reason to want the efficiency: vectors are intended to be used instead of built-in arrays, and so should be as efficient as built-in arrays, which don't do bounds-checking. If you want to be sure that bounds get checked, use at instead.

Size-ing Up Vector

Let's turn now to Example 1-2, which manipulates a vector<int> by using a few simple operations.

2. Consider the following code:

```
// Example 1-2: Some fun with vectors
//
vector<int> v;

v.reserve(2);
assert(v.capacity() == 2);
```

This assertion has two problems, one substantive and one stylistic.

Chapter 2. The String Formatters of Manor Farm, Part 1: sprintf

Difficulty: 3

In this Item and the next, an Orwellian look at the mysteries of sprintf and why the alternatives are always (yes, always) better.

JG Question

1. What is sprintf? Name as many standard alternatives to sprintf as you can.

Guru Question

2. What are the major strengths and weaknesses of sprintf? Be specific.

Solution

"All animals are equal, but some animals are more equal than others."

—George Orwell, Animal Farm

1. What is sprintf? Name as many standard alternatives to sprintf as you can.

Consider the following C code that uses sprintf to convert an integer value to a human-readable string representation, perhaps for output on a report or in a GUI window:

// Example 2-1: Stringizing some data in C, using sprintf.

```
// PrettyFormat takes an integer, and formats it into the provided output buffer.
// For formatting purposes, the result must be at least 4 characters wide.
//
void PrettyFormat(int i, char* buf) {
 // Here's the code, neat and simple:
 sprintf(buf, "%4d", i);
}
```

The $64,000 question is: How would you do this kind of thing in C++?

Well, all right, that's not quite the question because, after all, Example 2-1 is valid C++. The true $64,000 question is: Throwing off the shackles and limitations of the C standard [C99] on which the C++ standard [C++03] is based, if indeed they are shackles, isn't there a superior way to do this in C++ with its classes and templates and so forth?

That's where the question gets interesting, because Example 2-1 is the first of no fewer than four direct, distinct, and standard ways to accomplish this task. Each of the four ways offers a different tradeoff among clarity, type safety, run-time safety, and efficiency. Moreover, to paraphrase George Orwell's revisionist pigs, "all four choices are standard, but some are more standard than others"—and, to add insult to injury, not all of them are from the same standard. They are, in the order I'll discuss them:

- sprintf [C99, C++03]
- snprintf [C99]
- std::stringstream [C++03]
- std::strstream [C++03]

Finally, as though that's not enough, there's a fifth not-yet-standard-but-liable-to become-standard alternative for simple conversions that don't require special for matting:

- boost::lexical_cast [Boost]

Enough chat; let's dig in.

The Joys and Sorrows of sprintf

Chapter 3. The String Formatters of Manor Farm, Part 2: Standard (or Blindingly Elegant) Alternatives

Difficulty: 6

Our Orwellian look at the mysteries of sprintf concludes with a comparative analysis of snprintf, std::stringstream, std::strstream, and the nonstandard but blindingly elegant boost::lexical_cast.

Guru Question

1. For each of the following alternatives to sprintf, compare and contrast its strengths and weaknesses, using the analysis and example code from Item 2:
   a. snprintf
   b. std::stringstream
   c. std::strstream
   d. boost::lexical_cast

Solution

Alternative #1: snprintf

1. For each of the following alternatives to sprintf, compare and contrast its strengths and weaknesses, using the analysis and example code from Item 2:
   a. snprintf

Of the other choices, sprintf's closest relative is of course snprintf. snprintf adds only one new facility to sprintf, but it's an important one: the ability to specify the maximum length of the output buffer, thereby eliminating buffer overruns. Of course, if the buffer is too small, then the output will be truncated.

snprintf has long been a widely available nonstandard extension present on most major C implementations. With the advent of the C99 standard [C99], snprintf has "come out" and gone legit, now officially sanctioned as a standard facility. Until your own compiler is C99-compliant, though, you might have to use this under a vendor-specific extension name such as _snprintf.

Frankly, you should already have been using snprintf over sprintf anyway, even before snprintf was standard. Calls to length-unchecked functions such as sprintf are banned in most good coding standards, and for good reason. The use of unchecked sprintf calls has long been a notoriously common problem, causing program crashes in general[4] and security weaknesses in particular.[5]

[4] This is a real problem, and not just with sprintf() but with all length-unchecked calls in the standard C library. Try a Google search for "strcpy" and "buffer overflow" to see what new problems have come up this week.

[5] For example, for years it was fashionable for malicious web servers to crash web browsers by sending them very long URLs that were likely to be longer than the web browser's internal URL string buffer. Browsers that didn't check the length before copying into the fixed-length buffer ended up writing past the end of the buffer, usually overwriting data but in some cases overwriting code areas with malicious code that could then be executed. It's surprising just how much software out there was, and is, using unchecked calls.

With snprintf we can correctly write the length-checked version we were trying to create earlier:

```
// Example 3-1: Stringizing some data in C, using snprintf.
//
void PrettyFormat(int i, char* buf, int buflen) {
 // Here's the code, neat and simple and now a lot safer:
 snprintf(buf, buflen, "%4d", i);
}
```

Note that it's still possible for the caller to get the buffer length wrong. That means snprintf still isn't as 100% bulletproof for overflow safety as the later alternatives that encapsulate their own resource management, but it's certainly lots safer and deserves a "Yes" under the "Length-safe?" question. With sprintf we have no good way to avoid for certain the possibility of buffer overflow; with snprintf we can ensure it doesn't happen.

Note that some prestandard versions of snprintf behaved slightly differently. In particular, under one major implementation, if the output fills or would overflow the buffer, the buffer is not zero-terminated. On such environments, the function would need to be written slightly differently to account for the nonstandard behavior:

```
// Stringizing some data in C, using a not-quite-C99 _snprintf variant.
//
void PrettyFormat(int i, char* buf, int buflen) {
 // Here's the code, neat and simple and now a lot safer:
```

Chapter 4. Standard Library Member Functions

Difficulty: 5

Reuse is good, but can you always reuse the standard library with itself? Here is an example that might surprise you, where one feature of the standard library can be used portably with any of your code as much as you like, but it cannot be used portably with the standard library itself.

JG Question

1. What is std::mem_fun? When would you use it? Give an example.

Guru Question

2. Assuming a correct incantation in the indicated comment, is the following expression legal and portable C++? Why or why not?

```
std::mem_fun</*...*/>(&(std::vector<int>::clear))
```

Solution

Fun with mem_fun

1. What is std::mem_fun? When would you use it? Give an example.

The standard mem_fun adapter lets you use member functions with standard library algorithms and other code that normally deals with free functions.

For example, given:

```
class Employee {
public:
 int DoStandardRaise() { /*...*/ }
//...
};

int GiveStandardRaise(Employee& e) {
 return e.DoStandardRaise();
}

std::vector<Employee> emps;
```

We might be used to writing code like the following:

```
std::for_each(emps.begin(), emps.end(), &GiveStandardRaise);
```

But what if GiveStandardRaise didn't exist or for some other reason we needed to call the member function directly? Then we could write the following:

```
std::vector<Employee> emps;
std::for_each(emps.begin(), emps.end(),
        std::mem_fun_ref(&Employee::DoStandardRaise));
```

The _ref bit at the end of the name mem_fun_ref is a bit of an historical oddity. When writing code like this, you should just remember to say mem_fun_ref if the container is a plain old container of objects, because for_each will be operating on references to those objects, and to say mem_fun if it's a container of pointers to objects:

```
std::vector<Employee*> emp_ptrs;
std::for_each(emp_ptrs.begin(), emp_ptrs.end(),
        std::mem_fun(&Employee::DoStandardRaise));
```

You'll probably have noticed that, for clarity, I've been showing how to do this with functions that take no parameters. You can use the bind… helpers to deal with some functions that take an argument, and the principle is the same. Unfortunately you can't use this approach for functions that take two or more arguments. Still, it can be useful.

And that, in a nutshell, is mem_fun. This brings us to the awkward part:

Use mem_fun, Just Not with the Standard Library

2. Assuming a correct incantation in the indicated comment, is the following expression legal and portable C++? Why or why not?

Chapter 5. Flavors of Genericity, Part 1: Covering the Basis [sic]

Difficulty: 4

To get our feet wet before delving into Item 6, consider this simple example of flexible generic code in C++. The code examples in this Item and the next are taken from Exceptional C++ [Sutter00, page 42].

JG Question

1. "C++ templates provide compile-time polymorphism." Explain.

Guru Question

2. What are the semantics of the following function? Be as complete as you can, and be sure to explain why there are two template parameters and not just one.

```
template <class T1, class T2>
void construct(T1* p, const T2& value) {
 new (p) T1(value);
}
```

Solution

1. "C++ templates provide compile-time polymorphism." Explain.

When we think of polymorphism in an object-oriented world, we think of the kind of run-time polymorphism we get from using virtual functions. A base class establishes an interface "contract" as defined by a set of virtual functions, and derived classes may inherit from the base class and override the virtual functions in a way that preserves the contracted semantics. Then other code that expects to work on a Base object (and accepts the Base object by pointer or reference) can work equally well with a Derived object:

```
// Example 5-1(a): Ye olde garden-variety run-time polymorphism.
//
class Base {
public:
 virtual void f();
 virtual void g();
};
class Derived : public Base {
 // override f and/or g if desired
};

void h(Base& b) {
 b.f();
 b.g();
}

int main() {
  Derived d;
  h(d);
}
```

This is great stuff, and gives a lot of run-time flexibility. There are two main drawbacks of run-time polymorphism: First, the types must be related in a hierarchy derived from a common base class. Second, when the virtual functions are called in a tight loop you might notice some performance penalty because normally each call to a virtual function must be made through an extra pointer indirection, as the compiler figures out the Derived function you really mean to call.

If you know the types you're using at compile time, you can get around both of the drawbacks: You can use types that are not related by inheritance, as long as they provide the expected operations:

```
// Example 5-1(b): Ye newe Cvariety compile-time polymorphism. Powerful
// stuff. We're still finding out just what kinds of nifty things this makes possible.
//
class Xyzzy {
public:
 void f(bool someParm = true);
 void g();
 void GoToGazebo();

 // … more functions …

};

class Murgatroyd {
public:
 void f();
 void g(double two = 6.28, double e = 2.71828);
 int HeavensTo(const Z&) const;
```

Chapter 6. Flavors of Genericity, Part 2: Generic Enough?

Difficulty: 7

How generic is a generic function, really? The answer can depend as much on its implementation as on its interface, and a perfectly generalized interface can be hobbled by simple—and awkward-to-diagnose—programming lapses.

Guru Question

1. There is a subtle genericity trap in the following functions. What is it, and what's the best way to fix it?

```
template <class T>
void destroy(T* p) {
 p->~T();
}

template <class FwdIter>
void destroy(FwdIter first, FwdIter last) {
 while(first != last) {
  destroy(first);
  ++first;
 }
}
```

2. What are the semantics of the following function, including the requirements on T? Is it possible to remove any of those requirements? If so, demonstrate how, and argue whether doing so is a good idea or a bad idea.

```
template <class T>
void swap(T& a, T& b) {
 T temp(a); a = b; b = temp;
}
```

Solution

1. There is a subtle genericity trap in the following functions. What is it, and what's the best way to fix it?

```
// Example 6-1: destroy
//
template <class T>
void destroy(T* p) {
 p->~T();
}

template <class FwdIter>
void destroy(FwdIter first, FwdIter last) {
 while(first != last) {
  destroy(first);
  ++first;
 }
}
```

destroy destroys an object or a range of objects. The first version takes a single pointer and calls the pointed-at object's destructor. The second version takes an it erator range, and iteratively destroys the individual objects in the designated range.

Still, there's a subtle trap here. This didn't make a difference in any example where it first appeared in [Sutter00], but it's a little odd: The two-parameter destroy(FwdIter,FwdIter) version is templatized to take any generic iterator, and yet it calls the one-parameter destroy(T*) by passing it one of the iterators—which requires that FwdIter must be a plain old pointer! This needlessly loses some of the generality of templatizing on FwdIter.

Guideline

Remember that pointers (into an array) are always iterators, but iterators are not always pointers.

It also means you can get Really Obscure error messages when compiling code that tries to call destroy(FwdIter,FwdIter) with nonpointer iterators, because (at least one of) the actual failure(s) will be on the destroy(first) line inside the two-parameter version, which typically generates such useful messages as the following, taken from one popular compiler:

```
'void __cdecl destroy(template-parameter-1,template-parameter-1)' : expects 2
arguments - 1 provided
'void __cdecl destroy(template-parameter-1 *)' : could not deduce template
argument for 'template-parameter-1 *' from '[the iterator type I was using]'
```

These error messages aren't as bad as some I've seen, and with only a little bit of extra reading they do actually tell you (mostly) what's going on. The first message indicates that the compiler was trying to resolve the statement destroy(first); as a call to the two-parameter version; the second indicates an attempt instead to resolve it as a call to the one-parameter version. Both attempts failed, each for a different reason: The two-parameter version can take iterators but needs two of them, not just one, and the one-parameter version can take just one parameter but needs it to be a pointer. No dice.

Having said all that, in reality we'd almost never want to use destroy with anything but pointers in the first place just

Chapter 7. Why Not Specialize Function Templates?

Difficulty: 8

Although the title of this Item is a question, it could also be made into a statement: This Item is about when and why not to specialize templates.

JG Question

1. What two major kinds of templates are there in C++, and how can they be specialized?

Guru Question

2. In the following code, which version of f will be invoked by the last line? Why?

```
template<class T>
void f(T);

template<>
void f<int*>(int*);

template<class T>
void f(T*);

// ...

int *p;
f(p);                   // which of the f's is called here?
```

Solution

The Important Difference: Overloading vs. Specialization

It's important to make sure we have the terms down pat, so here's a quick review.

1. What two major kinds of templates are there in C++, and how can they be specialized?

In C++, there are class templates and function templates. These two kinds of templates don't work in exactly the same ways, and the most obvious difference is in overloading: Plain old C++ classes don't overload, so class templates don't overload either. On the other hand, plain old C++ functions having the same name do overload, so function templates are allowed to overload too. This is pretty natural. What we have so far is summarized in Example 7-1:

```
// Example 7-1: Class vs. function template, and overloading
//

// A class template
template<class T> class X { /*...*/ };                  // (a)

// A function template with two overloads
template<class T> void f(T);                            // (b)
template<class T> void f(int, T, double);               // (c)
```

These unspecialized templates are also called the primary templates.

Further, primary templates can be specialized. This is where class templates and function templates diverge further, in ways that will become important later in this Item. A class template can be partially specialized and/or fully specialized.[11] A function template can only be fully specialized, but because function templates can overload, we can get nearly the same effect via overloading that we could have achieved via partial specialization. The following code illustrates these differences:

[11] In standardese, a full specialization is called an "explicit specialization."

```
// Example 7-1, continued: Specializing templates
//

// A partial specialization of (a) for pointer types
template<class T> class X<T*> { /*...*/ };

// A full specialization of (a) for int
template<> class X<int> { /*...*/ };

// A separate primary template that overloads (b) and (c)—not a partial
// specialization of (b), because there's no such thing as a partial specialization
// of a function template!
template<class T> void f(T*);                           // (d)

// A full specialization of (b) for int
template<> void f<int>(int);                            // (e)

// A plain old function that happens to overload with (b), (c), and (d)
// —but not (e), which we'll discuss in a moment
void f(double);                                         // (f)
```

Chapter 8. Befriending Templates

Difficulty: 4

If you want to declare a function template specialization as a friend, how do you do it? According to the C++ standard, you can choose either of two legal syntaxes. According to real-world compilers, however, one of the syntaxes is widely unsupported; the other works on all current versions of popular compilers… except one.

Let's say we have a function template that does SomethingPrivate to the objects it operates on. In particular, consider the boost::checked_delete function template from [Boost], which deletes the object it's given—among other things, it invokes the object's destructor:

```
namespace boost {
 template<typename T> void checked_delete(T* x) {

 // … other stuff …

 delete x;
 }
}
```

Now, say you want to use this function template with a class where the operation in question (here the destructor) happens to be private:

```
class Test {
 ~Test() { }            // private!
};

Test* t = new Test;
boost::checked_delete(t);   // error: Test's destructor is private,
                // so checked_delete can't call it.
```

The solution is simple: Just make checked_delete a friend of Test. (The only other option is to give up and make Test's destructor public.) What could be easier?

And indeed, in the standard C++ language there are two legal and easy ways to do it. If only compilers would agree….

JG Question

1. Show the obvious standards-conforming syntax for declaring boost::checked_delete as a friend of Test.

Guru Question

2. Why is the obvious syntax unreliable in practice? Describe the more reliable alternative.

Solution

This Item exists as a reality check: Befriending a template in another namespace is easier said (in the standard) than done (using real-world compilers that don't quite get the standard right).

In sum, I have some good news, some bad news, and then some good news again:

- The Good News: There are two perfectly good standards-conforming ways to do it, and the syntax is natural and unsurprising.
- The Bad News: Neither standard syntax works on all current compilers. Even some of the strongest and most conformant compilers don't let you write one or both of the legal, sanctioned, standards-conforming and low-cholesterol methods that you should be able to use.
- The Good News (reprise): One of the perfectly good standards-conforming ways does work on every current compiler I tried except gcc.

Let's investigate.

The Original Attempt

1. Show the obvious standards-conforming syntax for declaring boost::checked_delete as a friend of Test.

This Item was prompted by a question on Usenet by Stephan Born, who wanted to do just that. His problem was that when he tried to write the friend declaration to make a specialization of boost::checked_delete a friend of his class Test, the code wouldn't work on his compiler.

Here's his original code:

```
// Example 8-1: One way to grant friendship
//
class Test {
 ~Test() { }
 friend void boost::checked_delete(Test* x);
};
```

Alas, not only does this code not work on the poster's compiler, it in fact fails on quite a few compilers. In brief, Example 8-1's friend declaration has the following characteristics:

- It's legal according to the standard but relies on a dark corner of the language.
- It's rejected by many current compilers, including very good ones.
- It's easily fixed to not rely on dark corners and work on all but one current compiler (gcc).

I am about to delve into explaining the four ways that the C++ language lets you declare friends. It's easy. I'm also going to have some fun showing you what real compilers do, and then finish with a guideline for how to write the most portable code.

Chapter 9. Export Restrictions, Part 1: Fundamentals

Difficulty: 7

The scoop on export—what some people think it does, what it actually might do, and why it's the most widely ignored major feature in the C++ standard.

JG Question

1. What is meant by the "inclusion model" for templates?

Guru Question

2. What is meant by the "separation model" for templates?
3. What are some of the major drawbacks to the inclusion model for:
   a. normal functions?
   b. templates?
4. How can the drawbacks in Question 3 be helped by the standard C++ separation model for:
   a. normal functions?
   b. templates?

Solution

The standard C++ template export feature is widely misunderstood, with more restrictions and consequences than most people at first realize. This Item and the next take a closer look at our experience to date with export.

As of this writing there is still exactly one commercially available compiler that supports the export feature. The Comeau[13] compiler, built on the Edison Design Group (EDG)[14] front-end C++ language implementation, which was the first (and so far only) C++ implementation to add support for export, was released in 2002. There is still little experience with using export on real-world projects, although that will hopefully change if export-capable implementations become more widely available and used. But there are things that we do know and that the original implementers have learned.

[13] See www.comeaucomputing.com.

[14] See www.edg.com.

Here's what this Item and the next cover:

- What export is, and how it's intended to be used.
- The problems export is widely assumed to address and why it does not in fact address them the way most people think.
- The current state of export, including what our implementation experience to date has been.
- The (often nonobvious) ways that export changes the fundamental meaning of other apparently unrelated parts of the C++ language.
- Some advice on how to use export effectively if and when you do happen to acquire an export-capable compiler.

A Tale of Two Models

The C++ standard supports two distinct template source code organization models: the inclusion model that we've been using for years, and the separation model that is relatively new.

1. What is meant by the "inclusion model" for templates?

In the inclusion model, template code is as good as all inline from a source perspective (though the template doesn't have to be actually inline): The template's full source code must be visible to any code that uses the template. This is called the inclusion model because we basically have to #include all template definitions right there in the template's header file.[15]

[15] Or the equivalent, such as stripping the definitions out into a separate .cpp file but having the template's .h header file #include the .cpp definition file, which amounts to the same thing.

If you know today's C++ templates, you know the inclusion model. It's the only template source model that has received any real press over the past ten years because it's the only model that has been available on standard C++ compilers until now. All the templates you're likely to have ever seen over the years in C++ books and articles up to the time of this writing fall into this category.

2. What is meant by the "separation model" for templates?

On the other hand, the separation model is intended to allow "separate" compilation of templates. (The "separate" is in quotation marks for a reason.) In the separation model, template definitions do not need to be visible to callers. It's tempting to add "just like plain functions," but that's actually incorrectit's a similar mental picture, but the effects are significantly different, as we shall see when we get to the surprises. The separation model is relatively newit was added to the standard in the mid-1990s, but the first commercial implementation, by EDG, didn't appear until the summer of 2002.[16]

[16] Note that Cfront had some similar functionality a decade earlier. But Cfront's implementation was slow, and it was based on a "works most of the time" heuristic such that, when Cfront users encountered template-related build problems, a common first step to get rid of the problem was to blow away the cache of instantiated templates and reinstantiate everything from scratch.

Bear with me as I risk delving too deeply into compilerese for one paragraph: A subtle but important distinction to keep in mind is that the inclusion and separation models really are different source code organization models. That is, they're about how you can choose to arrange and organize your source code. They are not different instantiation models; that is, a compiler does essentially the same work to instantiate templates under either source model, inclusion or export. This is important because this is part of the underlying reason why export's limitations, which we'll get to in a moment, surprise many people, especially that using export is unlikely to improve build times to the degree that separate compilation for functions routinely does. For example, under either source model, the compiler can still perform optimizations such as relying on (rather than enforcing) the One Definition Rule (ODR) to only instantiate each unique combination of template parameters once, no matter how often and widely that combination is used throughout your project. Such optimizations and instantiation policies are available to compiler writers regardless of whether the inclusion or separation model is being used to physically organize the template's source code; although it's true that the separation model allows the optimizations, so does the inclusion model.

Illustrating the Issues

3. What are some of the major drawbacks to the inclusion model for:
   a. normal functions?
   b. templates?

To illustrate, let's look at some code.

We'll look at a function template under both the inclusion and separation models, but for comparison purposes I'm also going to show a plain old function under the usual inline and out-of-line separately-compiled models. This will help to highlight the differences between today's usual function separate compilation and export's "separate" template compilation. The two are not the same, even though the terms commonly used to describe them look the same, and that's why I put "separate" in quotes for the latter.

Consider the following code, a plain old inline function and an inclusion-model function template:

```
// Example 9-3(a): A garden-variety inline function
//

// --- file f.h, shipped to user ---
namespace MyLib {
 inline void f(int) {
 // natty and quite dazzling implementation, the product of many years of work;
 // uses some other helper classes and functions
   }
}
```

The following inclusion-model template demonstrates the parallel case for templates:

```
// Example 9-3(b): An innocent and happy little template, uses the inclusion model
//
// --- file g.h, shipped to user ---
namespace MyLib {
 template<typename T>
 void g(T&) {
 // avant-garde, truly stellar implementation, the product of many years of work;
 // uses some other helper classes and functions—the functions aren't necessarily
 // declared "inline", but the body's code is all here in the same file just the same
 }
}
```

In both cases, the Example 9-3 code harbors issues familiar to C++ programmers:

- Source exposure for the definitions: The whole world can see the perhaps-proprietary definitions for f and g. In itself, that might or might not be such a bad thing; more on that later.
- Source dependencies: All callers of f and g depend on the respective bodies' internal details, so every time the body changes, all its callers have to recompile. Also, if either f's or g's body uses any other types not already mentioned in their respective declarations, then all their respective callers will need to see those types' full

Export InAction [sic]

Can we solve, or at least mitigate, these problems?

4. How can the drawbacks in Question 3 be helped by the standard C++ separation model for:
   a. normal functions?

For the function, the answer is an easy "of course," because of separate compilation:

```
// Example 9-4(a): A garden-variety separately compiled function
//
// --- file f.h, shipped to user ---
namespace MyLib {
 void f(int);                    // MYOB
}
// --- file f.cpp, optionally shipped ---
namespace MyLib {
 void f(int) {
  // natty and quite dazzling implementation, the product of many years of work;
  // uses some other helper classes and functions—and is now separately compiled
  }
}
```

Unsurprisingly, this solves both problems, at least in the case of f. (The same idea can be applied to whole classes using the Pimpl Idiom; see Exceptional C++ [Sutter00].)

- No source exposure for the definition: We can still ship the implementation's source code if we want to, but we don't have to. Note that many popular libraries, even closely guarded proprietary ones, ship source code anyway (possibly at extra cost) because users demand it for debuggability and other reasons.
- No source dependencies: Callers no longer depend on f's internal details, so every time the body changes, all its callers only have to relink. This frequently makes builds an order of magnitude or more faster. Similarly, usually to somewhat less dramatic effect on build times, f's callers no longer depend on types used only in the body of f.

That's all well and good for the function, but we already knew all that. We've been doing this since C, and since before C (which is a very very long time ago).

The real question is: What about the template?

   a. templates?

The idea behind export is to get something like this effect for templates. One might naïvely expect the following code to get the same advantages as the code in Example 9-4(a). One would be wrong, but one would still be in good company because this has surprised a lot of people, including world-class experts. Consider:

```
// Example 9-4(b): A more independent little template?
//
// --- file g.h, shipped to user ---
namespace MyLib {
```

Issue the First: Source Exposure

The first problem is unsolved: Source exposure for the definition remains.

Nothing in the C++ standard says or implies that you won't have to ship full source code for g anyway just because you wrote the export keyword. Indeed, in the only existing implementation of export, the compiler requires that the template's full definition be shipped—the full source code.[17] One reason is that a C++ compiler still needs the exported template definition's full definition context when instantiating the template elsewhere as it's used. For just one example why, consider what the C++ standard says about what happens when instantiating a template:

[17] "But couldn't we ship encrypted source code?" is a common question. The answer is that any encryption that a program can undo without user intervention (say to enter a password each time) is easily breakable. Also, several companies have already tried "encrypting" or otherwise obfuscating source code before, for a variety of purposes including protecting inclusion-model templates in C++; those attempts have been widely abandoned because the practice annoys customers, doesn't really protect the source code well, and the source code rarely needs such protection in the first place because there are other and better ways to protect intellectual property claims—obfuscation comes to the same end here.

"[Dependent] names are unbound and are looked up at the point of the template instantiation in both the context of the template definition and the context of the point of instantiation."

—[C++03] §14.6.2

A dependent name is a name that depends on the type of a template parameter; most useful templates mention dependent names. At the point of instantiation or a use of the template, dependent names must be looked up in two places. They must be looked up in the instantiation context; that's easy, because that's where the compiler is already working. But they must also be looked up in the definition context, and there's the rub, because that includes knowing not only the template's full definition but also the context of that definition inside the file containing the definition, including what other relevant function signatures are in scope and so forth, so that overload resolution and other work can be performed.

Think about Example 9-4(b) from the compiler's point of view: Your library has an exported function template g with its definition nicely ensconced away outside the header. Well and good. The library gets shipped. A year later, one fine sunny day, it's used in some customer's translation unit h.cpp where he decides to instantiate g<CustType> for a CustType that he just wrote that morning… what does the compiler have to do to generate object code? It has to look, among other places, at g's definition, at your implementation file. And there's the rub… export does not eliminate such dependencies on the template's definition, it merely hides them.

Exported templates are not truly "separately compiled" in the usual sense we mean when we apply that term to functions. Exported templates cannot in general be separately compiled to object code in advance of use; for one thing, until the exact point of use, we can't even know the actual types the template will be instantiated with. So exported templates are at best "separately partly compiled" or "separately parsed." The template's definition needs to be actually compiled with each instantiation. (There is a similarity here to Java and .NET libraries where the bytecode or IL can be reversed to reveal something very like the source code.)

Guideline

Remember that export doesn't imply true separate compilation of templates like we have for functions.

Issue the Second: Dependencies and Build Times

The second problem is likewise unresolved: Dependencies are hidden, but remain.

Every time the template's body changes, the compiler has to reinstantiate all the uses of the template. During that process, the translation units that use g are still processed together with all of g's internals, including the definition of g and the types used only in the body of g.

The template code still has to be compiled in full later, when each instantiation context is known. Here is the key concept to remember:

Guideline

Remember that export only hides dependencies; it doesn't eliminate them.

It's true that callers no longer visibly depend on g's internal details, inasmuch as g's definition is no longer openly brought into the caller's translation unit via #included code; the dependency can be said to be hidden at the human-reading-the-source-code level.

But that's not the whole story, because we're talking compilation-the-compiler-must-perform dependencies here, not human-reading-the-code-while-sipping-a-latte dependencies, and compilation dependencies on the template definitions still exist. True, the compiler might not have to go recompile every translation unit that uses the template; but it must go away and recompile at least enough of the other translation units that use the template so that all the combinations of template parameter types on which the template is ever used get reinstantiated from scratch. The compiler can't just go relink object code that is truly separately compiled.

Note that compilers could be made smart enough to handle inclusion-model templates the same way—namely, not rebuilding all files that use the template but only enough of them to cover all the instantiations—if the code is organized as shown in Example 9-4(b) but with export removed and a new line #include "g.cpp" added to g.h. The idea is that the compiler would rely on the One Definition Rule rather than enforcing it; i.e., it would assume that the other instantiations with the same parameters must be identical, rather than actually performing all the instantiations and then checking whether they are really identical.

Further, remember that many templates use other templates, and therefore the compiler next performs a cascading recompilation of those templates (and their translation units) too, and then of whatever templates those templates use, and so on recursively, until there are no more cascading instantiations to be done. (If, at this point in our discussion, you are glad that you personally don't have to implement export, that's a normal reaction.)

Even with export, it is not the case that all callers of a changed exported template "just have to relink." Unlike the situation with true separate function compilation where builds will speed dramatically, it is unknown as of this writing whether export-ized builds will in general be the same speed, faster, or slower in common real-world use.

Summary

So far, we've looked at the motivation behind export and why it's not truly "separate" compilation for templates in the same way we have separate compilation for nontemplates. Many people think that export means that template libraries can be shipped without full definitions and/or that build speeds will be faster. Neither outcome is promised by export. The community's experience to date is that source or its direct equivalent must still be shipped and that build speeds are expected to be the same or slower, rarely faster, principally because dependencies, though masked, still exist, and the compiler might still have to do the same amount of work (or more) in common cases.

In the next Item, we'll see why export complicates the C++ language and makes it trickier to use, including that export actually changes the fundamental meaning of parts of the rest of the language in surprising ways that it is not clear were foreseen. We'll also see some initial advice on how to use export effectively if you happen to acquire an export-capable compiler.

Chapter 10. Export Restrictions, Part 2: Interactions, Usability Issues, and Guidelines

Difficulty: 9

How export interacts with existing C++ language features, and the first guidelines on how to use it safely.

JG Question

1. When was export set in its current form in the C++ standard? When was it first implemented?

Guru Question

2. In what ways does export change the meaning of other C++ language features? Briefly explain the interactions.
3. How does export affect the programmer?
4. What real and potential benefits does export have?

Solution

This is the second of a two-part miniseries. In the previous Item, we covered the following:

- What export is and how it's intended to be used. We looked at an analysis of the similarities and differences between the "inclusion" and "export" template source code organization models, and why they're not parallel to the differences between inline and separately compiled functions.
- The problems export is widely assumed to address, and why it does not infact address them the way most people think.

Widespread expectations notwithstanding, export is not about truly "separate" compilation for templates in the same way we have true separate compilation for nontemplates. Many people expect that export means that template libraries can be shipped without full source code definitions (or their direct equivalent), and/or that build speeds will be faster. Neither outcome is promised by export.

The community's most informed experience to date is that full source or its direct equivalent must still be shipped and that it is yet unknown whether build speeds will be better, worse, or just about the same in common real-world usage. Why? Principally this is because dependencies, though masked, still exist, and the compiler still has to do at least the same amount of work in common cases. In short, it's a mistake (albeit a natural one) to think that export gives true separate compilation for templates in the sense that the template author need only ship declaration headers and object code. Rather, what is exported is similar to Java and .NET libraries where the bytecode or IL can be reversed to reveal something very like the source; it is not traditional object code.

This time, I'll cover:

- The current state of export, including what our implementation experience to date has been.
- The (often nonobvious) ways that export changes the fundamental meaning of other apparently unrelated parts of the C++ language.
- Some advice on using export effectively if and when you do happen to acquire an export-capable compiler.

But first, consider a little history.

Historical Perspective: 1988-1996

1. When was export set in its current form in the C++ standard? When was it first implemented?

The answers are 1996 and 2002, respectively. (If you feel that the date of a feature's first implementation usually ought to precede the date of the feature's standardization, well, you aren't alone, but this isn't the only example where the C++ standard has taken this kind of inventive approach with novelties.)

Given this, and given also that there are some valid criticisms of export, it might be tempting to start casting derisive stones and sharp remarks at the people who came up with what we might view as a misfeature. It would also be ungracious and unkind, and could possibly smack of armchair-quarterbacking. This "backgrounder" part of the Item exists for balance, because on the export issue it's been easy for people to go to extremes in both directions, pro and con export.

Exception Safety Issues and Techniques

Exception handling is a fundamental error reporting mechanism in modern languages, including C++. In Exceptional C++ [Sutter00] and More Exceptional C++ [Sutter02] we considered in detail many issues related to defining what exception safety is, how to go about writing exception-safe code, and language issues and interactions to be aware of.

In this section, we continue to build on that material by turning our attention to some specific exception-related language features. We begin by answering some perennial questions: Is exception safety all about writing TRy and catch in the right places? If not, then what? And what kinds of things should you consider when developing an exception safety policy for your software?

Delving beyond that, it's worth spending an entire Item to lay out reasons why writing exception-safe code is, well, just plain good for you, because doing that promotes programming styles that lead to more robust and more maintainable code in general, quite apart from their benefits in the presence of exceptions. But there is a limit to goodness and to "if some is good, then more is better" thinking, and that limit is hit well and hard when we get to exception specifications: Why are they in the language? Why are they well motivated in principle? And why, despite all that, should you stop using them in your programs?

This and more, as we dip our cups and drink again from the font of today's most current exceptional community wisdom.

Chapter 11. Try and Catch Me

Difficulty: 3

Is exception safety all about writing TRy and catch in the right places? If not, then what? And what kinds of things should you consider when developing an exception safety policy for your software?

JG Question

1. What is a try-block?

Guru Question

2. "Writing exception-safe code is fundamentally about writing try and catch in the correct places." Discuss.
3. When should try and catch be used? When should they not be used? Express the answer as a good coding standard guideline.

Solution

Playing catch

1. What is a try-block?

A try-block is a block of code (compound statement) whose execution will be attempted, followed by a series of one or more handler blocks that can be entered to catch an exception of the appropriate type if one is emitted from the attempted code. For example:

```
// Example 11-1: A try-block example
//
try {
 if(some_condition)
  throw string("this is a string");
 else if(some_other_condition)
  throw 42;
}
catch(const string&) {
 // do something if a string was thrown
}
catch(...) {
 // do something if anything else was thrown
}
```

In Example 11-1, the attempted code might throw a string, an integer, or nothing at all.

There's More to Life Than Playing catch

2. "Writing exception-safe code is fundamentally about writing TRy and catch in the correct places." Discuss.

Put bluntly, such a statement reflects a fundamental misunderstanding of exception safety. Exceptions are just another form of error reporting, and we certainly know that writing error-safe code is not just about where to check return codes and handle error conditions.

Actually, it turns out that exception safety is rarely about writing TRy and catch—and the more rarely the better. Also, never forget that exception safety affects a piece of code's design; it is never just an afterthought that can be retrofitted with a few extra catch statements as if for seasoning.

There are three major considerations when writing exception-safe code:

1. Where and when should I throw? This consideration is about writing throw in the right places. In particular, we need to answer:
   - What code should throw? That is, what errors will we choose to report by throwing an exception instead of by returning a failure value or using some other method?
   - What code shouldn't throw? In particular, what code should provide the no-fail guarantee? (See Item 12 and [Sutter99].)
2.

Chapter 12. Exception Safety: Is It Worth It?

Difficulty: 7

Is it worth the effort to write exception-safe code? This should no longer be a seriously disputed and debated point… but sometimes it still is.

Guru Question

1. Recap: Briefly define the Abrahams exception safety guarantees (basic, strong, and nofail).
2. When is it worth it to write code that meets:
   a. the basic guarantee?
   b. the strong guarantee?
   c. the nofail guarantee?

Solution

The Abrahams Guarantees

1. Recap: Briefly define the Abrahams exception safety guarantees (basic, strong, and nofail).

The basic guarantee says that failed operations might alter program state, but no leaks occur and affected objects/modules are still destructible and usable, in a consistent (but not necessarily predictable) state.

The strong guarantee involves transactional commit/rollback semantics: Failed operations guarantee that program state is unchanged with respect to the objects operated upon. This means no side effects that affect the objects, including the validity or contents of related helper objects such as iterators pointing into containers being manipulated.

Finally, the nofail guarantee says that failure simply will not be allowed to happen. In terms of exceptions, the operation will not throw an exception. (Abrahams and others, including the earlier Exceptional C++ books, originally called the nothrow guarantee. I have switched to calling it the nofail guarantee because these guarantees apply equally to all error handling, whether using exceptions or some other mechanism such as error codes.)

When Are Stronger Guarantees Worthwhile?

2. When is it worth it to write code that meets:

a. the basic guarantee?

b. the strong guarantee?

c. the nofail guarantee?

It is always worth it to write code that meets at least one of these guarantees. There are several good reasons:

1. Exceptions happen. (To paraphrase a popular saying.) They just do. The standard library emits them. The language emits them. We have to code for them. Fortunately, it's not that big a deal, because we now know how to do it. It does require adopting a few habits, however, and following them diligently—but then so did learning to program with error codes.

The big thorny problem is, as it ever was, the general issue of error handling. The detail of how to report errors, using return codes or exceptions, is almost entirely a syntactic detail where the main differences are in the semantics of how the reporting is done, so each approach requires its own style.

2. Writing exception-safe code is good for you. Exception-safe code and good code go hand in hand. The same techniques that have been popularized to help us write exception-safe code are, pretty much without exception, things we usually ought to be doing anyway. That is, exception-safety techniques are good for your code in and of themselves, even if exception safety weren't a consideration.

To see this in action, consider the major techniques I and others have written about to make exception safety easier:

- Use "resource acquisition is initialization" (RAII) to manage resource ownership. Using resource-owning

Chapter 13. A Pragmatic Look at Exception Specifications

Difficulty: 6

Now that the community has gained experience with exception specifications, it's time to reflect on when and how they should best be used. This Item considers the usefulness, or lack thereof, of exception specifications and how the answers can vary across real-world compilers.

JG Questions

1. What happens when an exception specification is violated? Why? Discuss the basic rationale for this C++ feature.
2. For each of the following functions, describe what exceptions the function could throw.

```
int Func();
int Gunc() throw();
int Hunc() throw(A,B);
```

Guru Question

3. Is an exception specification part of the function's type? Explain.
4. What are exception specifications, and what do they do? Be precise.
5. When is it worth it to write an exception specification on a function? Why would you choose to write one, or why not?

Solution

As we consider work now underway on the new C++ standard, C++0x, it's a good time to take stock of what we're doing with, and have learned from, our experience with the current standard [C++03]. The vast majority of standard C++'s features are good, and they get the lion's share of the print because there's not much point harping on the weaker features. Rather, the weaker and less useful features more often just get ignored and atrophy from disuse until many people forget they're even there (not always a bad thing). That's why you've seen relatively few articles about obscure features such as valarray, bitset, locales, and the legal expression 5[a] (although a version of the last one does show up in another Item later in this book)—and the same is true, we will find, for exception specifications.

Let's now take a closer look at the state of our experience with standard C++ exception specifications.

Moving Violations

1. What happens when an exception specification is violated? Why? Discuss the basic rationale for this C++ feature.

The idea of exception specifications is to do a run-time check that guarantees that only exceptions of certain types will be emitted from a function (or that none will be emitted at all). For example, the following function's exception specification guarantees that f will emit only exceptions of type A or B:

```
int f() throw(A, B);
```

If an exception would be emitted that's not on the invited-guests list, the function unexpected will be called. For example:

```
// Example 13-1
//
int f() throw(A, B) {          // A and B are unrelated to C
 throw C();                    // will call unexpected
}
```

You can register your own handler for the unexpected-exception case by using the standard set_unexpected function. Your replacement handler must take no parameters and it must have a void return type. For example:

```
void MyUnexpectedHandler() { /*...*/ }

std::set_unexpected(&MyUnexpectedHandler);
```

The remaining question is, what can your unexpected handler do? The one thing it can't do is return via a usual function return. There are two things it may do:

- It could decide to translate the exception into something that's allowed by that exception specification, by throwing its own exception that does satisfy the exception specification list that caused it to be called. Then stack unwinding would resume from where it had left off.
- It could call terminate, which ends the program. (The terminate function can itself be replaced, but any replacement must likewise also always end the program.)

The Story So Far

Class Design, Inheritance, and Polymorphism

In addition to the generic paradigm, C++ equally supports object-oriented design and programming. This section turns the spotlight on this more traditional area, with particular attention to the way C++ exposes OO features.

To get started, we'll consider a real-world example of code containing a subtle flaw, and we'll use it as a springboard to review basic object construction and teardown ordering. Then it's on to an in-depth foray into the world of writing robust code, which touches on the issue of code security: First, what parts of a class are accessible from various other code and, in particular, what ways are there for "leaking" the private parts of a class, intentionally or otherwise? What is encapsulation, and how does it relate to the choices we can and should make about member accessibility? Finally, how can we make our classes safer for versioning and for ensuring that base class contracts are easy to maintain correctly and won't be subverted accidentally?or otherwisein derived classes, which would otherwise lead to broken contracts and even security holes?

This and more, as we begin our foray into the world of objects.

Chapter 14. Order, Order!

Difficulty: 2

Programmers learning C++ often come up with interesting misconceptions of what can and can't be done in C++. In this example, contributed by Jan Christiaan van Winkel, a student makes a basic mistake—but one that many compilers let pass with no warnings at all.

JG Question

1. The following code was actually written by a student taking a C++ course, and the compiler the student was using issued no warnings about it. Indeed, several popular compilers issue no warnings for this code. What's wrong with it, and why?

```
#include <string>
using namespace std;

class A {
public:
 A(const string& s) { /* ... */ }
 string f() { return "hello, world"; }
};

class B : public A {
public:
 B() : A(s = f()) {}
private:
 string s;
};

int main() {
B b;
}
```

Guru Question

2. When you create a C++ object of class type, in what order are its various parts initialized? Be as specific and complete as you can. Demonstrate by showing the order of initialization of the various parts of an X object, using the following example.

```
class B1 { };
class V1 : public B1 { };
class D1 : virtual public V1 { };

class B2 { };
class B3 { };
class V2 : public B1, public B2 { };
class D2 : public B3, virtual public V2 { };

class M1 { };
class M2 { };
```

Solution

1. […] What's wrong with [this code], and why?

```
// Example 14-1
//

// …

B() : A(s = f()) {}

// …
```

This line harbors a couple of related problems, both associated with object lifetime and the use of objects before they exist. Note that the expression s = f() appears as the argument to the A base subobject constructor and hence will be executed before the A base subobject (or, for that matter, any part of the B object) is constructed.

First, this line of code tries to use the A base subobject before it exists. This particular student's compiler did not flag the (ab)use of A::f in that the member function f is being called on an A subobject that hasn't yet been constructed. Granted, the compiler is not required to diagnose such an error, but this is the kind of thing standards folks call "a quality of implementation issue"—something that a compiler is not required to do but that better compilers could be nice enough to do.

Second, this line then merrily tries to use the s member subobject before it exists, namely by calling the member function operator= on a string member subobject that hasn't yet been constructed.

2. When you create a C++ object of class type, in what order are its various parts initialized? Be as specific and complete as you can.

The following set of rules is applied recursively:

- First, the most derived class's constructor calls the constructors of the virtual base class subobjects. Virtual base classes are initialized in depth-first, left-to-right order.
- Next, direct base class subobjects are constructed in the order they are declared in the class definition.
- Next, (nonstatic) member subobjects are constructed in the order they were declared in the class definition.
- Finally, the body of the constructor is executed.

For example, consider the following code. Whether the inheritance is public, protected, or private doesn't affect initialization order, so I'm showing all inheritance as public.

Demonstrate by showing the order of initialization of the various parts of an X object, using the following example.

```
// Example 14-2
//
class B1 { };
class V1 : public B1 { };
```

Chapter 15. Uses and Abuses of Access Rights

Difficulty: 6

Who really has access to your class's internals? This Item is about forgers, cheats, pickpockets, and thieves and how to recognize and avoid them.

JG Question

1. What code can access the following parts of a class?
   a. public
   b. protected
   c. private

Guru Question

2. Consider the following header file:

```
// File x.h
//
class X {
public:
 X() : private_(1) { /*...*/ }

 template<class T>
 void f(const T& t) { /*...*/ }

 int Value() { return private_; }

 // ...

private:
 int private_;
};
```

   Demonstrate:
   a. a non-standards-conforming and non-portable hack; and
   b. a fully standards-conforming and portable technique

Solution

This Item is about forgers, cheats, pickpockets, and thieves.

1.

What code can access the following parts of a class?

In short:

a.

public

Public members can be accessed by any code.

b.

protected

Protected members can be accessed by the class's own member functions and friends and by the member functions and friends of derived classes.

c.

private

Private members can be accessed by the class's own member functions and friends only.

That's the usual answer, and it's true as far as it goes. In this Item, we consider a special case where this answer doesn't, well, quite go far enough, because C++ sometimes provides a way that makes it legal (if not moral) to subvert access to a class's private members.

4.

Consider the following header file: […] Demonstrate:

a.

a non-standards-conforming and non-portable hack; and

b.

a fully standards-conforming and portable technique

for any calling code to get direct access to this class's private_ member.

There's a strange and perverse fascination that makes people stare at car wrecks, oncoming headlights, and evil code hackery, so we might as well begin with a visit to a tragic "hit" scene in (a) and get it out of the way.

For a non-standards-conforming and non-portable hack, several ideas come to mind. Here are three of the more infamous offenders:

Criminal #1: The Forger

The Forger's hack of choice is to duplicate a forged class definition to make it say what he wants it to say. For example:

```
// Example 15-1: Lies and forgery
//
class X {
 // instead of including x.h, manually (and illegally) duplicates X's
 // definition, and adds a line such as:
```

Chapter 16. (Mostly) Private

Difficulty: 5

In C++, to what extent are the private parts of a class really truly private? In this Item, we see how private names are definitely not accessible from outside nonfriend code, and yet they do leak out of a class in small ways—some of which are well known, others of which aren't, and one of which can even be done as a coldly calculated, deliberate act.

Guru Question

1. Quick—assuming that the Twice functions are defined in another translation unit that is included in the link, should the following C++ program compile and run correctly? If no, why not? If yes, what is the output?

```
// Twice(x) returns 2*x
//
class Calc {
public:
 double                Twice(double d);
private:
 int                   Twice(int i);
 std::complex<float>   Twice(std::complex<float> c);
};

int main() {
 Calc c;
 return c.Twice(21);
}
```

Solution

"So, just how private is private, Private?"[20]

[20] O. B. Scure, C. Heap, and F. Ictional. Military Sounding (Reference, 2003).

At the heart of the solution lies this question: In C++, to what extent are the private parts of a class, such as the Question's Twice, really truly private? In this Item, we see how private names are definitely not accessible from outside nonfriend code, and yet they can and do leak out of a class in small ways—some of which are wellknown, others of which aren't, and two of which can even be done as a coldly calculated, deliberate act.

The Basic Story: Accessibility

The fundamental thing to recognize is this: Like public and protected, private is an access specifier. That is, it controls what other code might have access to the member's name—and that's all. Quoting from the C++ standard [C++03], the opening words of clause 11 state:

A member of a class can be

- private; that is, its name can be used only by members and friends of the class in which it is declared.
- protected; that is, its name can be used only by members and friends of the class in which it is declared, and by members and friends of classes derived from this class (see class.protected).
- public; that is, its name can be used anywhere without access restriction.

This is pretty basic stuff, but for completeness let's look at a simple example that makes it clear that access is indeed well controlled and there's no standards-conforming way around this. Example 16-1 demonstrates that nonfriend code outside the class can never get to a private member function by name either directly (by explicit call) or indirectly (via a function pointer), because the function name can't be used at all, not even to take the function's address:[21]

[21] Undefined hacks like trying to #define private public are nonstandard, deplorable, and reportedly punishable by law in 42 states. See Item 15 for more on that and similar misbegotten practices.

```
// Example 16-1: I can't get No::Satisfaction
//
class No {
private:
 virtual void Satisfaction() { }
};

int main() {
 No no;
 no.Satisfaction();          // error

 typedef void (No::*PMember)();
 PMember p = &No::Satisfaction;  // error
 return (no.*p)();           // nice try…
}
```

Note that this covers the case of virtual functions too. A private member that is a virtual function can be overridden by any derived class, but it can't be accessed by the derived class. That is, the derived class can override any virtual function with its own function of the same name, but the derived class cannot call or otherwise use the name of a base

Chapter 17. Encapsulation

Difficulty: 4

What exactly is encapsulation as it applies to C++ programming? What does proper encapsulation and access control mean for member data—should it ever be public or protected? This Item focuses on alternative answers to these questions, and shows how those answers can increase either the robustness or the fragility of your code.

JG Question

1. What does "encapsulation" mean? How important is it to object-oriented design and programming?

Guru Question

2. Under what circumstances, if any, should nonstatic class data members be made public, protected, and private? Express your answer as a coding guideline.
3. The std::pair class template uses public data members because it is not an encapsulated class but only a simple way of grouping data. Imagine a class template that is like std::pair but that additionally provides a deleted flag, which can be set and queried but cannot be unset. Clearly the flag itself must be private so as to prevent users from unsetting it directly. If we choose to keep the other data members public, as they are in std::pair, we end up with something like the following:
[View full width]

```
template<class T, class U>
class Couple {
public:
 // The main data members are public…
 T first;
 U second;

 // … but there is still classlike machinery and private
➥ implementation.
 Couple()                : deleted_(false) {}
 void MarkDeleted()      { deleted_ = true; }
 bool IsDeleted()        { return deleted_; }

private:
 bool deleted_;
};
```

Should the other data members still be public, as shown? Why or why not? If so, is this a good example of why mixing public and private data in the same class might sometimes be good design?

Solution

1. What does "encapsulation" mean?

According to Webster's Third New International Dictionary:

en-cap-su-late vt: to surround, encase, or protect in or as if in a capsule

Encapsulation in programming has precisely the same sense: To protect the internal implementation of a class by hiding those internals behind a surrounding and encasing interface visible to the outside world.

The definition of the word "capsule," in turn, gives good guidance as to what makes a good class interface:

cap-sule [F, fr. L /capsula/ small box, dim. of /capsa/ chest, case]

1a: a membrane or saclike structure enclosing a part or organ …

2 : a closed container bearing spores or seeds …

4a: a gelatin shell enclosing medicine …

5 : a metal seal …

6 : … envelope surrounding certain microscopic organisms …

9 : a small pressurized compartment for an aviator or astronaut …

Note the recurring theme in the words:

- Surround, encase, enclose, envelope

A good class interface hides the class's internals, presenting a "face" to the outside world that is separate and distinct from the internals. Because a capsule surrounds exactly one cohesive group of subobjects, its interface should likewise be cohesive—its parts should be directly related.

The outer surface of a bacterium's capsule contains its means for sensing, touching, and interacting with the outside world, and the outside world with it. (Those means would be a lot less useful if they were inside.)

- Closed, seal

A good class interface is complete and does not expose any internals. The interface acts as a hermetic seal and often acts as a code firewall (at compile time, at run-time, or both) whereby outside code cannot depend on class internals, and so changes to class internals cause no impact on outside code.

A bacterium whose capsule isn't closed won't live long; its internals will quickly escape, and the organism will die.

Chapter 18. Virtuality

Difficulty: 7

In this Item, we delve into old questions with new and/or improved answers. The up-to-date answers to two recurring questions about virtual functions lead directly to four guidelines targeting robust class design. These guidelines answer these questions: why should interfaces be nonvirtual? why should virtuals be private? and what's with the old and hoary advice about base class destructors, and is the old hoary advice really true?

JG Question

1. What is the "common advice" about base class destructors?

Guru Question

2. When should virtual functions be public, protected, or private? Justify your answer, and demonstrate existing practice.

Solution

In this Item, I want to present up-to-date answers to two recurring questions about virtual functions. These answers then lead directly to four class design guidelines.

The questions are old, but people keep asking them, and some of the answers have changed over time as we've gained experience with modern C++.

The Common Advice About Base Class Destructors

1. What is the "common advice" about base class destructors?

I know you've heard the question "Should base class destructors be virtual?"

Sigh. I wish this were only a frequently asked question. Alas, it's more often a frequently debated question. If I had a penny for every time I've seen this debate, I could buy a cup of coffee. Not just any old coffee, mind you—I could buy a genuine Starbucks Venti Extra Toffee Nut latte (my current favorite). Maybe even two of them, if I was willing to throw in a dime of my own.

The usual answer to this question is: "Huh? Of course base class destructors should always be virtual!" This answer is wrong, and the C++ standard library itself contains counterexamples refuting it, but it's right often enough to give the illusion of correctness.

We'll get back to this in a moment, but it's only the second of two questions about virtual function accessibility. Let's start with the more general one.

Virtual Question #1: Publicity vs. Privacy?

A general question we need to consider is this:

2. When should virtual functions be public, protected, or private? Justify your answer, and demonstrate existing practice.

The short answer is: rarely if ever, sometimes, and by default, respectively—the same answer we've already learned for other kinds of class members.

Most of us have learned through bitter experience to make all class members private by default unless we really need to expose them. That's just good encapsulation. Certainly we've long ago learned that data members should always be private (except only in the case of C-style data structs, which are merely convenient groupings of data and are not intended to encapsulate anything; see Item 17). The same also goes for member functions, so I propose the following guidelines, which could be summarized as a statement about the benefits of privatization, at least as it applies to code.

Guideline

Prefer to make interfaces nonvirtual.

Note: I'm saying "prefer," not "always."

Interestingly, the C++ standard library already overwhelmingly follows this guideline. Not counting destructors (which are discussed separately later on under Guideline #4) and not double-counting the same virtual function twice when it

Chapter 19. Enforcing Rules for Derived Classes

Difficulty: 5

Too many times, just being at the top of the (inheritance) world doesn't mean that you can save programmers of derived classes from simple mistakes. But sometimes you can! This Item is about safe design of base classes, so that derived class writers have a more difficult time going wrong.

JG Questions

1. When are the following functions implicitly declared and implicitly defined for a class, and with what semantics? Be specific, and describe the circumstances under which the implicitly defined versions cause the program to be illegal (not well-formed).
   a. default constructor
   b. copy constructor
   c. copy assignment operator
   d. destructor
2. What functions are implicitly declared and implicitly defined for the following class X? With what signatures?

```
class X {
 auto_ptr<int> i_;
};
```

Guru Question

3. Say that you have a base class that requires that all derived classes not use one or more of the implicitly declared and defined functions. For example:
   [View full width]

```
class Count {
public:
 // The Author of Count hereby documents that derived classes shall
 // inherit virtually, and that all their constructors shall call
➥ Count's
 // special-purpose constructor only.
 //
 Count(/* special parameters */);
```

Solution

Implicitly Generated Functions (or, What the Compiler Does for/to You)

In C++, four class member functions can be implicitly generated by the compiler: the default constructor, the copy constructor, the copy assignment operator, and the destructor.

The reason for this is a combination of convenience and backward compatibility with C. Recall that C-style structs are just classes consisting of only public data members; in particular, they don't have any (explicitly defined) member functions, and yet you do have to be able to create, copy, and destroy them. To make this happen, the C++ language automatically generates the appropriate functions (or some appropriate subset thereof) to do the appropriate things if you don't define appropriate operations yourself.

This Item is about what all those "appropriate" words mean:

1. When are the following functions implicitly declared and implicitly defined for a class, and with what semantics? Be specific, and describe the circumstances under which the implicitly defined versions cause the program to be illegal (not well-formed).

In short, an implicitly declared function is only implicitly defined when you actually try to call it. For example, an implicitly declared default constructor is only implicitly defined when you try to create an object using no constructor parameters.

Why is it useful to distinguish between when the function is implicitly declared and when it's implicitly defined? Because it's possible that the function might never be called, and if it's never called, then the program is still legal even if the function's implicit definition would have been illegal.

For convenience, throughout this Item unless otherwise noted, "member" means "nonstatic class data member." I'll also say "implicitly generated" as a catchall for "implicitly declared and defined."

Exception Specifications of Implicitly Declared Functions

In all four of the cases where a function can be implicitly declared, the compiler will make its exception specification just loose enough to allow all exceptions that could be allowed by the functions the implicit definition would call. For example:

```
// Example 19-1(a)
//
class C // ...
{
 // ...
};
```

Because no constructors are explicitly declared, the implicitly generated default constructor has the semantics of invoking all base and member default constructors. Therefore the exception specification of C's implicitly generated default constructor must allow any exception that any base or member default constructor might emit. If any base class or member of C has a default constructor with no exception specification, the implicitly declared function can throw anything:

```
// public:
inline C::C();        // can throw anything
```

If every base class or member of C has a default constructor with an explicit exception specification, the implicitly

Memory and Resource Management

If there's one issue dear to the heart of C and C++ programmers, it's memory and resource management. One of C++'s greatest strengths compared to other languages is the power it gives the programmer to control and manage memory and other resources, particularly to selectively automate memory management using the standard containers.

How well do you understand the real memory cost of using the different standard containers? Can you state with certainty that a list containing 1,000 objects will consume less total memory space than, say, a set of 1,000 of the same type of object? Then, touching back on the exceptions front: Does using the nothrow form of new help to make code more exception-safe? And finally, on many popular real-world platforms, when can it make sense not to worry about new failure (gasp!), and why?

Chapter 20. Containers in Memory, Part 1: Levels of Memory Management

Difficulty: 3

Memory management on modern operating systems can be complex and sophisticated in its own right, but it's only one layer of memory management that matters to C++ programs. The standard library provides and enables several further levels, any and all of which can matter a lot to your program.

JG Question

1. What are memory managers (also known as memory allocators), and what is their basic function? Briefly describe two of the major dynamic memory management strategies in C++.

Guru Question

2. In the context of the C++ standard library and the typical environments in which implementations of that library are used, what different levels of memory management exist? What can be said about their relationship with each other, how they interact, and how they share responsibilities?

Solution

The question this pair of Items will drive at is Item 21, Question 2: How much memory do the various standard containers use to store the same number of objects of the same type T?

To answer this question, however, we have to take a little journey through the land of data structures after first passing through the outskirts of the swamp of dynamic memory management. In particular, we have to consider two major items:

- the internal data structures used by containers like vector, deque, list, set/multiset, and map/multimap; and
- the way dynamic memory allocation works.

Let's begin with a recap of dynamic memory allocation and then work our way back up to what it means for the standard library.

Memory Managers and Their Strategies: A Brief Survey

1. What are memory managers (also known as memory allocators), and what is their basic function? Briefly describe two of the major dynamic memory management strategies in C++.

To understand the total memory cost of using various containers, it's important to understand the basics of how the underlying dynamic memory allocation works—after all, the container has to get its memory from some memory manager somewhere, and that manager in turn has to figure out how to parcel out available memory by applying some memory management strategy.

Here, in brief, are two popular memory management strategies in C++. Further details are beyond the scope of this Item; consult your favorite operating systems text for more information:

- General-purpose allocation can provide any size of memory block that a caller might request (the request size, or block size). General-purpose allocation is very flexible but has several drawbacks, two of which are: a) performance, because it has to do more work; and b) fragmentation, because as blocks are allocated and freed we can end up with lots of little noncontiguous areas of unallocated memory.
- Fixed-size allocation always returns a block of the same fixed size. This is obviously less flexible than general-purpose allocation, but it can be done much faster and doesn't result in the same kind of fragmentation.

A third major strategy, garbage-collected allocation, is not fully compatible with C and C++ pointers, malloc, and new and, as such, that strategy isn't directly relevant for the purposes of this discussion. Garbage-collected heaps are increasingly popular, however, and coming to C++, only not with pointers and new; I plan to write more about that topic in a future book. (For details about garbage collection in general and for C++ in particular, see [C++CLI04] and [Jones96].)

In practice, you'll often see combinations of these strategies. For example, perhaps your memory manager uses a general-purpose allocation scheme for all requests over some size S and as an optimization provides a fixed-size allocation scheme for all requests up to size S. It's usually unwieldy to have a separate arena for requests of size 1, another for requests of size 2, and so on; what normally happens is that the manager has a separate arena for requests of multiples of a certain size, say 16 bytes. If you request 16 bytes, great, you only use 16 bytes; if you request 17 bytes, the request is allocated from the 32-byte arena, and 15 bytes are wasted. This is a source of possible

Chapter 21. Containers in Memory, Part 2: How Big Is It Really?

Difficulty: 3

When you ask for memory, what do you really know about what you get—and what it will actually cost? How much memory do the standard containers use—in theory, in reality, and in the code you'll be writing this afternoon?

JG Question

1. When you ask for n bytes of memory using new or malloc, do you actually use up n bytes of memory? Explain why or why not.

Guru Question

2. How much memory do the various standard containers use to store the same number of elements of the same type T?

Solution

"I'll Take 'Operator New' for 200 Bytes, Alex"

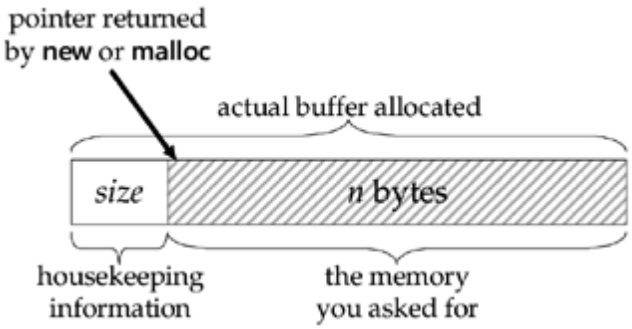
1.

When you ask for n bytes of memory using new or malloc, do you actually use up n bytes of memory? Explain why or why not.

When you ask for n bytes of memory using new or malloc, you actually use up at least n bytes of memory, because typically the memory manager must add some overhead to your request. Two common considerations that affect this overhead are housekeeping overhead and chunk size overhead.

Consider first the housekeeping overhead: In a general-purpose (i.e., not fixed-size) allocation scheme, the memory manager will have to somehow remember how big each block is so that it later knows how much memory to release when you call delete or free. Typically the manager remembers the block size by storing that value at the beginning of the actual block it allocates and then giving you a pointer to "your" memory that's offset past the housekeeping information (see Figure 21-1). Of course, this means it has to allocate extra space for that value, which could be a number as big as the largest possible valid allocation and so is typically the same size as a pointer. When freeing the block, the memory manager will just take the pointer you give it, subtract the number of housekeeping bytes and read the size, and then perform the deallocation.

Figure 21-1. Typical memory allocation using a general-purpose allocator for a request size of n bytes

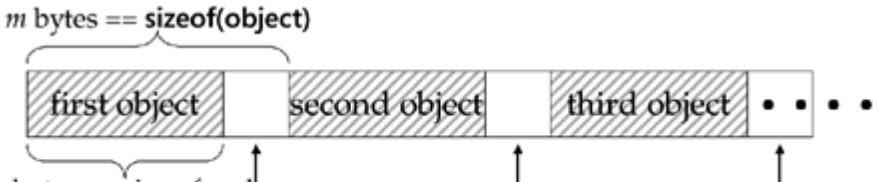


Of course, fixed-size allocation schemes (i.e., ones that return blocks of a given known size) don't need to store such overhead because they always know how big the block will be.

Next, let's look at the chunk size overhead: Even when you don't need to store extra information, a memory manager will often reserve more bytes than you asked for, because memory is often allocated in certain-sized chunks.

For one thing, some platforms require certain types of data to appear on certain byte boundaries (e.g., some require pointers to be stored on 4-byte boundaries) and either break or perform more slowly if they don't. This is called alignment, and it calls for extra padding within, and possibly at the end of, the object's data. Even plain old built-in C-style arrays are affected by this need for alignment because it contributes to sizeof(struct). See Figure 21-2, where I distinguish between internal padding bytes and at-the-end padding bytes, although both contribute to sizeof(struct).

Figure 21-2. What "contiguous" means—typical in-memory layout of an array of n-byte objects that require m-byte alignment (note: sizeof(object) == m)



Chapter 22. To new, Perchance to tHRow, Part 1: The Many Faces of new

Difficulty: 4

Any class that provides its own class-specific operator new, or operator new[], should also provide corresponding class-specific versions of plain new, in-place new, and nothrow new. Doing otherwise can cause needless problems for people trying to use your class.

JG Question

1. What are the three forms of new provided in the C++ standard?

Guru Question

2. What is class-specific new, and how do you make use of it? Describe any areas where you need to take particular care when providing your own class-specific new and delete.
3. In the following code, which operator new is invoked for each of the lines numbered 1 through 4?

```
      class Base {
public:
 static void* operator new(std::size_t, const FastMemory&);
};

class Derived : public Base {
 // ...
};

Derived* p1 = new Derived;                          // 1

Derived* p2 = new (std::nothrow) Derived;           // 2

void* p3 = /* some valid memory that's big enough for a Derived */ ;
new (p3) Derived;                                   // 3

Derived* p4 = new (FastMemory()) Derived;           // 4
```

Solution

In this Item and the next, I want to state and justify just two main pieces of advice:

- Any class that provides its own class-specific operator new or operator new[] should also provide corresponding class-specific versions of plain new, inplace new, and nothrow new. Doing otherwise can cause needless problems for people trying to use your class.
- Avoid using new(nothrow), and make sure that when you're checking for new failure, you're really checking what you think you're checking.

Some of this advice might be surprising, so let's examine the reasons and rationale that lead to it. This Item focuses on the first point; in the next we'll consider the second. For simplicity, I'm not going to mention the array forms of new specifically; what's said about the single-object forms applies correspondingly to the array forms.

In-Place, Plain, and Nothrow new

1. What are the three forms of new provided in the C++ standard?

The C++ standard provides three forms of new and allows any number of additional overloads.

One useful form is in-place new, which constructs an object at an existing memory address without allocating new space. For example:

```
// Example 22-1(a): Using in-place new, an "explicit constructor call"
//
void* p = ::operator new(sizeof(T)); // grab a sufficient amount of raw memory

new (p) T;                           // construct the T at address p, probably
                // calls ::operator new(std::size_t, void*) throw()
```

The standard also supplies "plain old new," which doesn't take any special additional parameters, and nothrow new, which does. Here's a complete list of the operator new overloads supplied in standard C++:

```
// The standard-provided overloads of operator new
// (there are also corresponding ones for array new[]):
//
void* ::operator new(std::size_t size) throw(std::bad_alloc);
        // usual plain old boring new
        // usage: new T


void* ::operator new(std::size_t size, const std::nothrow_t&) throw();
       // nothrow new — usage: new (std::nothrow) T

void* ::operator new(std::size_t size, void* ptr) throw();
       // in-place (or "put-it-there") new — usage: new (ptr) T
```

Programs are permitted to replace all but the last form with their own versions. All these standard functions live in global scope, not in namespace std. In brief, Table 22-1 summarizes the major characteristics of the standard versions of new.

Table 22-1. Comparison of the standard versions of new

Standard version of new	Additional parameter	Performs allocation	Can fail [27]	Throws	Replaceable
Plain	None	Yes	Yes (throws)	std::bad_alloc	Yes
Nothrow	std::nothrow_t	Yes	Yes (returns null)	No	Yes
In-Place	void*	No	No	No	No

[27] After operator new is done, the object's constructor will be invoked and of course that constructor operation might still fail, but we're not worried about that here. Here we're analyzing specifically whether or not operator new itself can fail.

Class-Specific new

2.

What is class-specific new, and how do you make use of it? Describe any areas where you need to take particular care when providing your own class-specific new and delete.

Besides letting programs replace some of the global operators new, C++ also lets classes provide their own class-specific versions. When reading Examples 22-1(a) and 22-1(b), did you notice the word "probably" in two of the comments? They were:

```
new (p) T;                          // construct the T at address p, probably
                    // calls ::operator new(std::size_t, void*) throw()

new (std::nothrow) T;       // probably calls the standard or some user-supplied
                    // operator ::new(std::size_t,
                    //              const std::nothrow_t&) throw()
```

Why only "probably"? Because the operators invoked might not necessarily be the ones at global scope, but might be class-specific ones. To understand this clearly, notice two interesting interactions between class-specific new and global new:

- Although you can't directly replace the standard global in-place new, you can write your own class-specific in-place new that gets used instead for that class.
- You can add class-specific nothrow new with or without replacing the global one.

So in these two code lines, it's possible that T (or one of T's base classes) provides its own versions of one or both operators new being invoked here, and if so those are the ones that will get used.

Here is a simple example of providing class-specific new, where we just provide our own versions of all three global flavors:

```
// Example 22-2: Sample class-specific versions of new
//
class X {
public:
 static void* operator new(std::size_t) throw();              // 1
 static void* operator new(std::size_t,
                           const std::nothrow_t&) throw();    // 2
 static void* operator new(std::size_t, void*) throw();       // 3
};

X* p1 = new X;                                                // calls 1

X* p2 = new (std::nothrow) X;                                 // calls 2

void* p3 = /* some valid memory that's big enough for an X */ ;
new (p3) X;                                                   // calls 3 (!)
```

I put an exclamation point after the third call to again draw attention to the funky fact that you can provide a class-specific version of in-place new even though you can't replace the global one.

A Name-Hiding Surprise

This, finally, brings us to the reason I've introduced all this machinery in the first place, namely the name-hiding problem:

3.

In the following code, which operator new is invoked for each of the lines numbered 1 through 4?

```
// Example 22-3: Name-hiding "news"
//
class Base {
public:
 static void* operator new(std::size_t, const FastMemory&);
};

class Derived : public Base {
 // …
};

 Derived* p1 = new Derived;                    // 1

 Derived* p2 = new (std::nothrow) Derived;     // 2

 void* p3 = /* some valid memory that's big enough for a Derived */ ;
 new (p3) Derived;                             // 3

 Derived* p4 = new (FastMemory()) Derived;     // 4
```

Most of us are familiar with the name-hiding problem in other contexts, such as a name in a derived class hiding one in the base class, but it's worth remembering that name hiding can crop up for operator new too.

Remember how name lookup works: In brief, the compiler starts in the current scope (here, in Derived's scope) and looks for the desired name (here, operator new); if no instances of the name are found, it moves outward to the next enclosing scope (in Base 's and then global scope) and repeats. Once it finds a scope containing at least one instance of the name (in this case, Base's scope), it stops looking and works only with the matches it has found, which means that further outer scopes (in this case, the global scope) are not considered and any functions in them are hidden; instead, the compiler looks at all the instances of the name it has found, selects a function using overload resolution, and, finally, checks access rules to determine whether the selected function can be called. The outer scopes are ignored even if none of the overloads found has a compatible signature, meaning that none of them could possibly be the right one; the outer scopes are also ignored even if the signature-compatible function that's selected isn't accessible. That's why name hiding works the way it does in C++. (For more details about name lookup and name hiding, see Item 30 in Exceptional C++ [Sutter00].)

What this means is that if a class C, or any of its base classes, contains a class-specific operator new with any signature, that function will hide all of the global ones and you won't be able to write normal new-expressions for C that intend to use the global versions. Here's what this means in the context of Example 22-3:

```
Derived* p1 = new Derived;                    // error: no match

Derived* p2 = new (std::nothrow) Derived;     // error: no match

void* p3 = /* some valid memory that's big enough for a Derived */ ;
new (p3) Derived;                             // error: no match

Derived* p4 = new (FastMemory()) Derived;     // calls Base::operator new()
```

Summary

If you provide any class-specific new, then:

- Always also provide class-specific plain (no-extra-parameters) new.
- Always also provide class-specific in-place new.
- Consider also providing class-specific nothrow new in case some users of your class do want to invoke it; otherwise, it will be hidden by other class-specific overloads of new.

In the next Item, we'll delve deeper into the question of what operator new failures mean, and how best to detect and handle them. Along the way, we'll see why it can be a good idea to avoid using new(nothrow)—perhaps most surprisingly, we'll also see that, on certain popular real-world platforms, memory allocation failures usually don't even manifest in the way the standard says they must! Stay tuned.

Chapter 23. To new, Perchance to tHRow, Part 2: Pragmatic Issues in Memory Management

Difficulty: 5

Avoid using new(nothrow), and make sure that when you're checking for new failure, you're really checking what you think you're checking.

JG Question

1. Explain the two main ways that the standard forms of new report an error if there isn't enough memory available.

Guru Question

2. Does using the nothrow form of new help us make our code more exception-safe? Justify your answer.
3. Describe real-world circumstances, either within standard C++ or outside strictly standard C++, where it might actually be impossible or useless to try to check for memory exhaustion.

Solution

In the previous Item, I illustrated and justified the following coding guideline:

- Any class that provides its own class-specific operator new or operator new[] should also provide corresponding class-specific versions of plain new, in-place new, and nothrow new. Doing otherwise can cause needless problems for people trying to use your class.

This time, we'll delve deeper into the question of what operator new failures mean, and how best to detect and handle them:

- Avoid using new(nothrow), and make sure that when you're checking for new failure, you're really checking what you think you're checking.

The first part might be mildly surprising advice, but it's the latter that is likely to raise even more eyebrows—because on certain popular real-world platforms, memory allocation failures usually don't even manifest in the way the standard says they must.

Again, for simplicity I'm not going to mention the array forms of new specifically. What's said about the single-object forms applies correspondingly to the array forms.

Exceptions, Errors, and new(nothrow)

First, a recap:

1. Explain the two main ways that the standard forms of new report an error if there isn't enough memory available.

Whereas most forms of new report failure by throwing a bad_alloc exception, nothrow new reports failure the time-honored malloc way, namely by returning a null pointer. This guarantees that nothrow new will never throw, as indeed its name implies.

The question is whether this really buys us anything: Some people have had the mistaken idea that nothrow new enhances exception safety because it prevents some exceptions from occurring. So here's the $64,000 motivating question: Does using nothrow new enhance program safety in general or exception safety in particular?

2. Does using the nothrow form of new help us make our code more exception-safe? Justify your answer.

The (possibly surprising) answer is: No, not really. Error reporting and error handling are orthogonal issues. "It's just syntax after all."[28]

[28] Can be sung to the tune of "It's a Small World (After All)."

The choice between throwing bad_alloc and returning null is just a choice between two equivalent ways of reporting an error. Therefore, detecting and handling the failure is just a choice between checking for an exception and checking for the null.

To a calling program that checks for the error, the difference is just syntactic. That means that it can be written with exactly equivalent program safety and exception safety, in either case—the syntax by which the error happens to be detected isn't relevant to safety, because it is just syntax and leads to only minor variations in the calling function's structure (e.g., something like if (null) { HandleError(); throw MyOwnException(); } vs. something like catch(bad_alloc) { HandleError(); throw MyOwnException(); }). Neither way of new error reporting provides

Optimization and Efficiency

We often want or need to make some part of our programs more efficient, and there are a lot of faces to optimization.

From time to time, it has been suggested that using the keyword const can help the compiler optimize code. Is that true or not? Why? Moving beyond const, other programmers like to rely on using the inline keyword for other kinds of optimizations. Does writing inline affect the performance of a program? If so, when, and which way? When can inlining be done, both under programmer control and otherwise?

Finally, we conclude this section with a brain teaser demonstrating how often there's no substitute for knowledge of the application domain to eke out the best possible efficiency, and we'll get an opportunity to bash out some honest-to-goodness low-level bit-twiddling and -fiddling code.

Chapter 24. Constant Optimization?

Difficulty: 3

Does const correctness help the compiler optimize code? A typical programmer's reaction is that, yes, they suppose it probably does. Which brings us to the interesting thing…

JG Question

1. Consider the following code:

```
const Y& f(const X& x) {
// … do something with x and find a Y object …
 return someY;
}
```

Does declaring the parameter and/or the return value as const help the compiler generate more optimal code or otherwise improve its code generation? Why or why not?

Guru Question

2. In general, explain why or why not the presence or absence of const can help the compiler enhance the code it generates.
3. Consider the following code:

```
void f(const Z z) {
 // ...
}
```

The questions:

a. Under what circumstances and for what kinds of class Z could this particular const qualification help generate different and better code? Discuss.

b. For the kinds of circumstances mentioned in (a), are we talking about a compiler optimization or some other kind of optimization? Explain.

c. What's a better way to get the same effect?

Solution

const: Not the Little Optimizer One Might Think

1. Consider the following code:

```
// Example 24-1
//
const Y& f(const X& x) {

 // … do something with x and find a Y object …

 return someY;

}
```

Does declaring the parameter and/or the return value as const help the compiler generate more optimal code or otherwise improve its code generation?

In short, no, it probably doesn't.

Why or why not?

What could the compiler do better? Could it avoid a copy of the parameter or the return value? No, because the parameter is already passed by reference, and the return is already by reference. Could it put a copy of x or someY into read-only memory? No, because both x and someY live outside its scope and come from and/or are given to the outside world. Even if someY is dynamically allocated on the fly within f itself, it and its ownership are given up to the caller.

But what about possible optimizations of code that appears inside the body of f? Because of the const, could the compiler somehow improve the code it generates for the body of f? This leads into the second and more general question:

2. In general, explain why or why not the presence or absence of const can help the compiler enhance the code it generates.

Referring again to Example 24-1, of course the usual reason that the parameter's constness doesn't usually let the compiler do fancy things still applies here: Even when you call a const member function, the compiler can't assume that the bits of object x or object someY won't be changed. Further, there are additional problems (unless the compiler performs global optimization): The compiler also may not know for sure whether any other code might have a non-const reference that aliases the same object as x and/or someY, and whether any such non-const references to the same object might get used incidentally during the execution of f; and the compiler might not even know whether the real objects, to which x and someY are merely references, were actually declared const in the first place.

Just because x and someY are declared const doesn't necessarily mean that their bits are physically const. Why not? Because either class might have mutable members— or their moral equivalent, namely const_casts inside member functions. Indeed, the code inside f itself might perform const_casts or C-style casts that turn the const declarations into lies.

There is one case where saying const can really mean something, and that is when objects are made const at the point where they are defined. In that case, the compiler can often successfully put such "really const" objects into read-only memory, especially if they are PODs whose memory images can be created at compile time and therefore can be stored right inside the program's executable image itself. Such objects are colloquially called "ROM-able."

Chapter 25. inline Redux

Difficulty: 7

Quick: When is inlining performed? And is it possible to write a function that can be guar-anteed never to be inlined? In this Item, we'll consider the many and varied opportunities for inlining, including many that are likely to surprise you. The answers, stated simply, are, "It's never too late, and nothing is impossible…"

JG Question

1. What is inlining?

Guru Question

2. When is inlining performed? Is it at:
   a. coding time?
   b. compile time?
   c. link time?
   d. application installation time?
   e. run time?
   f. some other time?
3. For extra marks: What kinds of functions are guaranteed never to be inlined?

Solution

Which answer did you pick for Question 2? If you picked (a) or (b), you're not alone. Those are the most common answers, and you might have thought of Item 8 of More Exceptional C++ [Sutter02], where I gave some detailed discussion about the C++ inline keyword. If that were all there was to it, we could stop right here, declare a spontaneous editorial holiday, and take the rest of this Item off. But we won't do that this time, because it turns out that there is more to say. Much more.

The reason I'm including this Item is to show why the most accurate answer to the primary question is "any or all of the above," and the general answer to the for-extra-marks Question 3 is "none." Wonder why? Read on.

Brief Recap: Inlining

1. What is inlining?

If you've already read Item 8 of More Exceptional C++ [Sutter02], the next few sections are review and you can safely skim them while skipping ahead to the discussion of part (c).

In short, "inlining" means replacing a function call with an "in-place" expansion of the function's body. For example, consider the following code:

```
// Example 25-1
//
double Square(double x) { return x * x; }

int main() {
 double d = Square(3.14159 * 2.71828);
}
```

The idea of inlining the function call is to treat this program (at least conceptually) as though it were written instead as something like this:

```
int main() {
 const double __temp = 3.14159 * 2.71828;
 double d = __temp * __temp;
}
```

This inlining eliminates the overhead of performing the function call, namely of pushing the parameter onto the stack and then having the CPU jump elsewhere in memory to execute the function's code, thus losing some locality of reference. This inlining is also not the same thing as treating Square as a macro, because an inlined function call is still a function call and its arguments are evaluated only once. With a macro they could be evaluated multiple times, such as in a macro like #define SquareMacro(x) ((x)*(x)), where the call SquareMacro(3.14159 * 2.71828) would expand to 3.14159 * 2.71828 * 3.14159 * 2.71828 (that's multiplying pi by e twice, not once).

There's a special case worth mentioning: recursive calls, where a function calls itself either directly or indirectly. Although these calls are often not inlineable, in some cases a compiler can still inline some recursion in much the same way that it can partly unroll some loops.

Incidentally, did you notice that this Example 25-1 illustrates inlining but does not use the inline keyword? That's intentional. We'll come back to this interesting twist several times as we consider the central question:

2. When is inlining performed? Is it at:
   a.

Chapter 26. Data Formats and Efficiency, Part 1: When Compression Is the Name of the Game

Difficulty: 4

How good are you at choosing highly compact and memory-efficient data formats? How good are you at writing bit-twiddling code? This Item and the next give you ample opportunity to exercise both skills as we consider efficient representations of chess games and a BitBuffer to hold them.

Background: I assume you know the basics of chess.

JG Question

1. Which of these standard containers uses the least memory to store the same number of objects of the same type T: deque, list, set, or vector? Explain.

Guru Question

2. You are creating a worldwide chess server that stores all games ever played on it. Because the database can get very large, you want to represent each game using as few bytes as possible. For this problem, consider only the actual game moves and ignore extra information such as the players' names and comments.

   For each of the following data sizes, demonstrate a format for representing a chess game that requires the indicated amount of data to store each half-move (a half-move is one move played by one side). For this question, assume 8 bits per byte.

   a. 128 bytes per half-move

   b. 32 bytes per half-move

   c. minimum 4 bytes and maximum 8 bytes per half-move

   d. minimum 2 bytes and maximum 4 bytes per half-move

   e. 12 bits per half-move

Solution

1. Which of these standard containers uses the least memory to store the same number of objects of the same type T: deque, list, set, or vector? Explain.

Recall Items 20 and 21, which cover the underlying memory footprints and structures of the various standard containers. Each kind of container chooses a different space/performance tradeoff:

- A vector<T> internally stores a contiguous array of T objects and so has no perelement overhead at all.
- A deque<T> can be thought of as a vector<T> whose internal storage is broken up into chunks. A deque<T> stores chunks, or "pages," of objects. This requires storing one extra pointer of management information per page, which usually works out to a fraction of a bit per element. There's no other per-element overhead, because deque<T> doesn't store any extra pointers or other information for individual T objects.
- A list<T> is a doubly linked list of nodes that hold T elements. This means that for each T element, list<T> also stores two pointers, which point to the previous and next nodes in the list. Every time we insert a new T element, we also create two more pointers, so a list<T> requires two pointers' worth of overhead per element.
- Finally, a set<T> (and, for that matter, a multiset<T>, map<Key,T>, or mul-timap<Key,T>) also stores nodes that hold T (or pair<const Key,T>) elements. The usual implementation of a set is as a tree with three extra pointers per node. Often people see this and think, "Why three pointers? Aren't two enough, one for the left child and one for the right child?" Because it must be possible to efficiently iterate over the set, we also need an "up" pointer to the parent node; otherwise, determining the next element starting from some arbitrary iterator can't be done efficiently. (Besides trees, other internal implementations of set are possible; for example, an alternating skip list can be used, although this still requires at least three pointers per element in the set (see [Marrie00]).

Part of choosing an efficient in-memory representation of data is choosing the right (read: most space-and time-efficient) container that supports the functionality you need. But that's not the end of it by any means: Another big part of choosing an efficient in-memory representation of data is determining how to represent the data that will be put into those containers. This question brings us to the meat of this Item.

Different Ways to Represent Data

The point of Question 2 is to demonstrate that there can be a plethora of ways to represent the same information:

2. You are creating a worldwide chess server that stores all games ever played on it. Because the database can get very large, you want to represent each game using as few bytes as possible. For this problem, consider only the actual game moves and ignore extra information such as the players' names and comments.

The rest of this Item uses the following standard terms and abbreviations:

K	King
Q	Queen

Chapter 27. Data Formats and Efficiency, Part 2: (Even Less) Bit-Twiddling

Difficulty: 8

Time to consider even more highly compact and memory-efficient data formats and get down to writing some bit-twiddling code.

Guru Question

1. To implement solution Item 26-2(e), you decide to create the following class that manages a buffer of bits. Implement it portably so that it will work correctly on all conforming C++ compilers regardless of platform.

```
class BitBuffer {
public:
 // … add other functions as needed …

 // Append num bits starting with the first bit of p.
 //
 void Append(unsigned char* p, size_t num);

 // Query #bits in use (initially zero).
 //
 size_t Size() const;

 // Get num bits starting with the start-th bit,
 // and store the result starting with the first
 // bit of p.
 //
 void Get(size_t start, size_t num, unsigned char* dest) const;

private:
 // … add details here …
};
```

2. Is it possible to store a chess game using fewer than 12 bits per half-move? If so, demonstrate how. If not, why not?

Solution

BitBuffer, the Binary Slayer

1.

To implement solution Item 26-2(e), you decide to create the following class that manages a buffer of bits. Implement it portably so that it will work correctly on all conforming C++ compilers regardless of platform.

First, note that the directive "assume 8 bits per byte "applied only to the previous Item—it does not apply here. We need a solution that will compile and run correctly on any conforming C++ implementation, no matter what kind of underlying platform it's running on.

The required interface boiled down to:

```
class BitBuffer {
public:
 void Append(unsigned char* p, size_t num);

 size_t Size() const;

 void Get(size_t start, size_t num, unsigned char* dest) const;

 // …

};
```

You might wonder why the BitBuffer interface was specified in terms of pointers to unsigned char. First off, there's no such thing as a pointer to a bit in standard C++, so that's out. Second, the C++ standard guarantees that operations on unsigned types (including unsigned char, unsigned short, unsigned int, and unsigned long) won't run afoul of "Hey, you didn't initialize that byte!" or "Hey, that's not a valid value!" messages from your compiler. As Bjarne Stroustrup writes in [Stroustrup00]:

The unsigned integer types are ideal for uses that treat storage as a bit array.

So compilers are required to treat unsigned char (and the other unsigned integer types) as raw bits of storage—which is just what we want. There are other approaches, but this is a reasonable one that lets us exercise our bit-fiddling coding skills, which happens to be a major goal of this Item.

The main question in implementing BitBuffer is: What internal representation should we use? I'll consider two major alternatives.

Attempt #1: Bit-Fiddling into an unsigned char Buffer

The first idea is to implement the BitBuffer via an internal big block of unsigned chars, and fiddle the bits ourselves when we put them in and take them out. We could let BitBuffer have a member of type unsigned char* that points to the buffer, but let's at least use a vector<unsigned char> so that we don't have to worry as much about basic memory management.

Do you think that sounds pretty easy? If you do, and you haven't yet tried to implement (and test!) it, take an hour or three and try it now. I bet you'll find it's not as simple as you think.

I'm not entirely ashamed to report that this version took me quite a bit of effort to write. Just drafting the initial version of the code took me more programming effort than I expected, and then it took a lot of debugging effort to find and fix all the bugs. I didn't keep track of the development effort, but in retrospect I estimate it took me several score compiles, including several to add reporting cout statements to analyze intermediate values and see where things were

Traps, Pitfalls, and Puzzlers

Before we get to our concluding section demonstrating several Style Case Studies in depth, let's pause to consider a few other C++ issues or just puzzling situations.

Why does C++, like most programming languages, reserve the names of keywords, so that, for example, you can't have a variable named class? Why does a line of code that looks like it ought to be doing some real work actually turn out to do nothing at all, with the compiler emitting not even a single machine instruction for it as though the line didn't even exist? On the other hand, why does code that looks just down-right wrong actually compile and run just fine, legally and reliably? When you're floating in a sea of numbers, why is it good to double-check your work?

Finally, with apologies to Dylan (the troubadour, not the language): How many times must the cannonballs fly before we learn macros don't care? The answer, my friend, is blowin' in the wind, the answer is blowin' in the wind….

Chapter 28. Keywords That Aren't (or, Comments by Another Name)

Difficulty: 3

All keywords are equal (to the parser), but some are more equal than others. In this Item, we see why reserving keywords is important, because keywords are important and special. But we'll also see two keywords that have absolutely no semantic impact on a C++ program.

JG Question

1. Why do most programming languages have reserved keywords, words that programs are not allowed to use?

Guru Question

2. How does adding the keyword auto alter the semantics of a C++ program?
3. How does adding the keyword register alter the semantics of a C++ program?

Solution

Why Have Keywords?

It's important for the C++ language to have keywords that are reserved to the language itself and that can't be used as the names of such things as types or functions or variables.

1. Why do most programming languages have reserved keywords, words that programs are not allowed to use?

If there weren't such reserved words, it would be easy to write programs that are impossible to compile because they're undecidably ambiguous. Consider the unbelievably simple conditional code in Example 28-1(a):

```
// Example 28-1(a): A legal C++ program.
//
int main() {
 if(true);              // 1: OK
 if(42);                // 2: OK
}
```

Lines 1 and 2 each test a condition; if the condition evaluates to true (and both do), an empty statement is executed.

Granted, that might not be the most thrilling code the planet has ever seen. I fervently hope it is not even the most thrilling code you have personally written in the past week. But it is legal C++, and it's the simplest code I can think of that illustrates the problems we would have if C++ allowed keywords to be used as identifiers.

Now consider the following speculative code that just recently arrived in my office (with a quiet pop and a faint smell reminiscent of camphor mixed with sulfur) from an alternate universe in which C++ did not reserve keywords and people happily tried to use the keywords as identifiers too. For a moment, leave aside how a compiler might make sense of this, and consider instead the far simpler question: What do you as a human think the code in Example 28-1(b) ought to mean?

```
// Example 28-1(b): Not legal C++, but what if it were?
//
class if {              // Let's call the class "if" (not legal, but what if it were?)
public:
 if(bool) {}            // 3: Hmm… constructor?
};
```

What does line 3 mean? "Oh, that's easy," someone might say. "We know that a conditional statement couldn't possibly make sense there, plus a type name wouldn't appear by itself as the condition being tested, so clearly this has got to be a constructor. Hey, maybe letting users reuse names like if isn't so bad after all—after all, it's not hard to guess what they must mean!" Some language designers have been quick to go down this dirty little road… and it's a very short dirty little road as it turns out, because you don't get very far along it before falling face-first across situations like lines 4 and 5:

```
// Example 28-1(b), continued: Now let's go back to that code again…
//
int main() {
 if(true);              // 4: Hmm… what does this mean?
 if(42);                // 5: Hmm… and this?
}
```

In this alternate universe's code, what would lines 4 and 5 mean? Are they the same old plain-jane conditional

Chapter 29. Is It Initialization?

Difficulty: 3

Most people know the famous quote: "What if they gave a war and no one came?" This time, we consider the question: "What if we initialized an object and nothing happened?" As Scarlett might say in such a situation: "This isn't right, I do declare!"

Assume that the relevant standard headers are included and that the appropriate using-declarations are made.

JG Question

1. What does the following code do?

```
deque<string> coll1;

copy(istream_iterator<string>(cin),
     istream_iterator<string>(),
     back_inserter(coll1));
```

Guru Question

2. What does the following code do?
[View full width]

```
deque<string> coll2(coll1.begin(), coll1.end());

deque<string> coll3(istream_iterator<string>(cin),
➥ istream_iterator<string>());
```

3. What must be changed to make the code do what the programmer probably expected?

Solution

Basic Population Mechanics

1. What does the following code do?

```
// Example 29-1(a)
//
deque<string> coll1;

copy(istream_iterator<string>(cin),
    istream_iterator<string>(),
    back_inserter(coll1));
```

This code declares a deque of strings called coll1 that is initially empty. It then populates the container by copying every whitespace-delimited string in the standard input stream (cin) into the deque using deque::push_back, until there is no more input available.

The Example 29-1(a) code is equivalent to:

```
// Example 29-1(b): Equivalent to 29-1(a)
//
deque<string> coll1;

istream_iterator<string> first(cin), last;
copy(first, last, back_inserter(coll1));
```

The only difference is that in Example 29-1(a) the istream_iterator objects are created on the fly as unnamed temporary objects, so they are destroyed at the end of the copy call. In Example 29-1(b), the istream_iterator objects are named variables and survive the copy call; they won't be destroyed until the end of whatever scope surrounds the Example 29-1(b) code.

Interlude: Population Explosion

2. What does the following code do?

```
// Example 29-2(a): Declaring another deque
//
deque<string> coll2(coll1.begin(), coll1.end());
```

This code declares a second deque of strings called coll2, and populates it using initializers passed to the constructor. Here, it uses the deque constructor that takes a pair of iterators corresponding to a range from which the contents should be copied. In this case, we're initializing coll2 from an iterator range that happens to correspond to "everything that's in coll1."

The code so far in Example 29-2(a) is nearly equivalent to the following:

```
// Example 29-2(b): Almost the same as Example 29-2(a)
//

// extra step: call default constructor
deque<string> coll2;
```

Chapter 30. double or Nothing

Difficulty: 4

No, this Item isn't about gambling. It is, however, about a different kind of "float," so to speak, and lets you test your skills about basic floating-point operations in C and C++.

JG Question

1. What's the difference between float and double?

Guru Question

2. Say that the following program takes 1 second to run, which is not unusual for a modern desktop computer:

```
int main() {
  double x = 1e8;
  while(x > 0) {
   --x;
  }
}
```

How long would you expect it to take if you changed double to float? Why?

Solution

float and double in a Nutshell

1. What's the difference between float and double?

Quoting from the C++ standard:

> There are three floating point types: float, double, and long double. The type double provides at least as much precision as float, and the type long double provides at least as much precision as double. The set of values of the type float is a subset of the set of values of the type double; the set of values of the type double is a subset of the set of values of the type long double.

—[C++03, §3.9.1/8]

Now let's see how this definition, particularly the last sentence, can affect your code.

The Wheel of Time

2. Say that the following program takes 1 second to run, which is not unusual for a modern desktop computer:

```
int main() {
 double x = 1e8;
 while(x > 0) {
  --x;
 }
}
```

How long would you expect it to take if you changed double to float? Why?

It will probably take either about 1 second (on a particular implementation floats might be somewhat faster, as fast, or somewhat slower than doubles) or forever, depending on whether or not float can exactly represent all integer values from 0 to 1e8 inclusive.

The previous quote from the standard means that there might be values that can be represented by a double but that cannot be represented by a float. In particular, on some popular platforms and compilers, double can exactly represent all integer values in the range 0 to 1e8, inclusive, but float cannot.

What if float can't exactly represent all integer values from 0 to 1e8? Then the modified program will start counting down but will eventually reach a value N that can't be represented and for which N-1 == N (due to insufficient floating-point precision)… and then the loop will stay stuck on that value until the machine on which the program is running runs out of power (due to a local power outage or battery life limits), its operating system crashes (more common on some platforms than others), Sol turns out to be a variable star and scorches the inner planets, or the universe dies of heat death, whichever comes first.[38]

[38] Indeed, because the program keeps the computer running needlessly, it also needlessly increases the entropy of the universe, thereby theoretically hastening said heat death. In short, such a program is quite environmentally unfriendly and should be considered a threat to our species. Don't write code like this.

Of course, performing any kind of additional work, whether by humans or machines, also increases the entropy of the universe, thereby hastening heat death. This is a good argument to keep in mind for times when your employer requests extra overtime. End of spurious digression.

Chapter 31. Amok Code

Difficulty: 4

Sometimes life hands you some debugging situations that seem just plain deeply weird. Try this one on for size, and see if you can reason about possible causes for the problem.

Guru Question

1. One programmer has written the following code:

```
//--- file biology.h ---
//

// … appropriate includes and other stuff …
class Animal {
public:
 // Functions that operate on this object:
 //
 virtual int Eat (int) { /*…*/ }
 virtual int Excrete (int) { /*…*/ }
 virtual int Sleep (int) { /*…*/ }
 virtual int Wake (int) { /*…*/ }

 // Apparently for animals that were once married, and
 // sometimes dislike their ex-spouses, we provide:
 //
 int EatEx (Animal* a) { /*…*/ }
 int ExcreteEx (Animal* a) { /*…*/ }
 int SleepEx (Animal* a) { /*…*/ }
 int WakeEx (Animal* a) { /*…*/ }

 // …
};

// Concrete classes.
//
class Cat : public Animal { /*…*/ };
class Dog : public Animal { /*…*/ };
class Weevil : public Animal { /*…*/ };
   // … more cute critters …

// Convenient, if redundant, helper functions.
//
int Eat (Animal* a) { return a->Eat(1); }
int Excrete (Animal* a) { return a->Excrete(1); }
int Sleep (Animal* a) { return a->Sleep(1); }
int Wake (Animal* a) { return a->Wake(1); }
```

Unfortunately, the code fails to compile. The compiler rejects the definition of at least one of the …Ex functions with an error message saying the function has already been defined.

Solution

1. One programmer has written the following code:

[…]

What's going on?

In short, several things might be going on to cause these symptoms, but there's one outstanding possibility that would account for all the observed behavior: Yes, you guessed it: an ugly combination of macros running amok and a side dish of mixed intentional and unintentional overloading.

Motivation

Certain popular C++ programming environments give you macros that are deliberately designed to change function names. Usually they do this for "good" or "innocent" reasons, namely, backward and forward API compatibility. For example, if a Sleep function in one version of an operating system is replaced by a SleepEx, the vendor supplying the header in which the functions are declared might decide to "helpfully" provide a macro that automatically changes Sleep to SleepEx:

```
#define Sleep SleepEx
```

This is most definitely Not a Good Idea. Macros are the antithesis of encapsulation, because their actual range of effect cannot be controlled, not even by the macro writer.

Macros Don't Care

Macros are obnoxious, smelly, sheet-hogging bedfellows for several reasons, most of which are related to the fact that they are a glorified text-substitution facility whose effects are applied during preprocessing, before any C++ syntax and semantic rules can even begin to apply. The following are some of macros' less charming habits.

1. Macros change names— more often than not to harm, not protect, the innocent.

It is an understatement to say that this silent renaming can make debugging some what confusing. Such macro renaming means that your functions aren't actually called what you think they're called.

For example, consider our nonmember function Sleep:

```
int Sleep (Animal* a) { return a->Sleep(1); }
```

You won't find Sleep anywhere in the object code or the linker map file because there's really no Sleep function at all. It's really called SleepEx. At first, when you're wondering where your Sleep went, you might think, "Ah, maybe the compiler is automatically inlining Sleep for me," because that could explain why the short function doesn't seem to exist in the object code. If you jump to conclusions and fire off an angry email to your compiler vendor complaining about aggressive optimizations, though, you're blaming the wrong company (or, at least, the wrong department).

Some of you might already have encountered this unfortunate and demonstrably bad effect. If you're like me, which is to say easily irritated by compiler weirdnesses and not satisfied with simple explanations, your curiosity bump might get the better of you. Then, curious, you might fire up the debugger and deliberately step into the function… only to be taken to the correct source line where the phantom function (which still looks as though it has its original name, in the source code) lives, stepping into the phantom function that indeed works and is indeed getting called, but which by all

Chapter 32. Slight Typos? Graphic Language and Other Curiosities

Difficulty: 5

Sometimes even small and hard-to-see typos can accidentally have a significant effect on code. To illustrate how hard typos can be to see and how easy phantom typos are to see accidentally even when they're not there, consider these examples.

Attempt to answer the following questions without using a compiler.

Guru Question

1. What is the output of the following program on a standards-conforming C++ compiler?

```
// Example 32-1
//
#include <iostream>
#include <iomanip>

int main() {
 int x = 1;
 for(int i = 0; i < 100; ++i);
 // What will the next line do? Increment???????????/
 ++x;
 std::cout << x << std::endl;
}
```

2. How many distinct errors should be reported when compiling the following code on a conforming C++ compiler?

```
// Example 32-2
//
struct X {
 static bool f(int* p) {
  return p && 0[p] and not p[1:>>p[2];
 };
};
```

Solution

1. What is the output of the following program on a standards-conforming C++ compiler?

For Example 32-1, assuming that there is no invisible whitespace at the end of the comment line, the output is 1.

There are two tricks here, one obvious and one less so.

First, consider the for loop line:

```
for(int i = 0; i < 100; ++i);
```

There's a semicolon at the end, a "curiously recurring typo pattern" that (usually accidentally) makes the body of the for loop just the empty statement. Even though the following lines might be indented and might even have braces around them, they are not part of the body of the for loop.

This was a deliberate red herring—in this case, because of the next point, it doesn't matter that the for loop never repeats any statements, because there's no increment statement to be repeated at all (even though there appears to be one). This brings us to the second point:

Second, consider the comment line. Did you notice that it ends oddly, with a '/' character?

```
// What will the next line do? Increment???????????/
```

Nikolai Smirnov writes:

Probably, what's happened in the program is obvious for you but I lost a couple of days debugging a big program where I made a similar error. I put a comment line ending with a lot of question marks accidentally releasing the 'Shift' key at the end. The result is [an] unexpected trigraph sequence '??/' which was converted to '\' (phase 1) which was annihilated with the following '\n' (phase 2).

—N. Smirnov, private communication

The ??/ sequence is converted to '\' which, at the end of a line, is a line-splicing directive—surprise! In this case, it splices the following line ++x; to the end of the comment line and thus makes the increment part of the comment. The increment is never executed. (If you look closely at the question one more time, you'll see this subtle hint right in the original code. In this book I've been italicizing code comments, and because I'm a stickler for consistency to the point of being obsessive-compulsive, I couldn't resist correctly italicizing that line because it is after all part of a comment.)

Interestingly, if you look at the Gnu g++ documentation for the –WTRigraphs command-line switch, you will encounter the following incorrect generalization:

Warnings are not given for trigraphs within comments, as they do not affect the meaning of the program. [39]

[39] A Google search for "trigraphs within comments" yields this and several other interesting and/or amusing hits, not all of which are printable.

That might be true much of the time, but here we have a case in point—from real-world code, no less—where this expectation certainly does not hold.

2. How many distinct errors should be reported when compiling the following code on a conforming C++ compiler?

Chapter 33. Operators, Operators Everywhere

Difficulty: 4

How many operators can you put together, when you really put your mind to it? This Item takes a break from production coding to get some fun C++ exercise.

JG Question

1.

What is the greatest number of plus signs (+) that can appear consecutively, without intervening whitespace, in a valid C++ program?

Note: Of course, plus signs in comments, preprocessor directives and macros, and literals don't count. That would be too easy.

Guru Question

2.

Similarly, what is the greatest number of each of the following characters that can appear consecutively, without whitespace, outside comments in a valid C++ program?

a.

&

b.

<

c.

|

For example, for (a), the code if(a && b) trivially demonstrates two consecutive & characters in a valid C++ program. Try for more.

Solution

Background:

Who Is Max Munch, and What's He Doing in My C++ Compiler?

The "max munch" rule says that, when interpreting the characters of source code into tokens, the compiler does it greedily—it makes the longest tokens possible. Therefore >> is always interpreted as a single token, the stream extraction (right-shift) operator, and never as two individual > tokens even when the characters appear in a situation like this:

```
template<class T = X<Y>> ...
```

That's why such code has to be written with an extra space, as:

```
template<class T = X<Y> > ...
```

Similarly, >>> is always interpreted as >> followed by >, never as > followed by >>, and so on.

Some Fun with Operators

1. What is the greatest number of plus signs (+) that can appear consecutively, without intervening whitespace, in a valid C++ program?

   Note: Of course, plus signs in comments, preprocessor directives and macros, and literals don't count. That would be too easy.

It is possible to create a source file containing arbitrarily long sequences of consecutive + characters, up to a compiler's limits (such as the maximum source file line length the compiler can handle).

If the sequence has an even number of + characters, it will be interpreted as ++ ++ ++ ++ … ++, a sequence of two-character ++ tokens. To make this work and have well-defined semantics because of sequence points, all we need is a class with a user-defined prefix ++ operator that allows chaining. For example:

```
// Example 33-1(a)
//
class A {
public:
 A& operator++() { return *this; }
};
```

Now we can (if we're so inclined) write code like:

```
A a;
++++++a;       // meaning: ++ ++ ++ a;
```

which works out to:

```
a.operator++().operator++().operator++()
```

What if the sequence has an odd number of + characters? Then it will be interpreted as ++ ++ ++ ++ … ++ +, a series of two-character ++ tokens ending with a final single-character +. To make this work, we just need to

Style Case Studies

It might be cheeky to dissect published code. It might be cheeky, but it's fun.

This concluding section introduces a new theme: We will examine several pieces of real-world published code, critique it to illustrate proper design and coding style by demonstrating what the published code does well and does poorly, and use that information to develop an improved version. You might be amazed at just how much can be done even with code that has been written, vetted, and proofread by experts.

Enjoy! But, in the spirit of straws and rafters, keep also in the back of your mind how some of these same issues might just apply also to the code checked into your own source-control system.

Chapter 34. Index Tables

Difficulty: 5

Index tables are a genuinely useful idiom and a technique that's worth being aware of. But how can we implement the technique effectively… nay, even better than that, exceptionally?

JG Question

1. Who benefits from clear, understandable code?

Guru Question

2. The following code presents an interesting and genuinely useful idiom for creating index tables into existing containers. For a more detailed explanation, see the original article [Hicks00].

   Critique this code and identify:

   a. Mechanical errors, such as invalid syntax or nonportable conventions.

   b. Stylistic improvements that would improve code clarity, reusability, and maintainability.

```
// program sort_idxtbl(…) to make a permuted array of indices
#include <vector>
#include <algorith>

template <class RAIter>
struct sort_idxtbl_pair
{
 RAIter it;
 int i;

 bool operator<(const sort_idxtbl_pair& s)
  { return (*it) < (*(s.it)); }

 void set(const RAIter& _it, int _i) { it=_it; i=_i; }

 sort_idxtbl_pair() {}
};

template <class RAIter>
void sort_idxtbl(RAIter first, RAIter last, int* pidxtbl)
{
 int iDst = last-first;
 typedef std::vector< sort_idxtbl_pair<RAIter> > V;
 V v(iDst);
 int i=0;
 RAIter it = first;
 V::iterator vit = v.begin();
 for(i=0; it<last; it++, vit++, i++)
```

Solution

Clarity: A Short Sermon

1. Who benefits from clear, understandable code?

In short, just about everyone benefits.

First, clear code is easier to follow while debugging and, for that matter, is less likely to have as many bugs in the first place, so writing clean code makes your own life easier even in the very short term. (For a case in point, see the discussion surrounding Example 27-2 in Item 27.) Further, when you return to the code a month or a year later—as you surely will if the code is any good and is actually being used—it's much easier to pick it up again and understand what's going on. Most programmers find keeping full details of code in their heads difficult for even a few weeks, especially after having moved on to other work; after a few months or even a few years, it's too easy to go back to your own code and imagine it was written by a stranger—albeit a stranger who curiously happened to follow your personal coding style.

But enough about selfishness. Let's turn to altruism: Those who have to maintain your code also benefit from clarity and readability. After all, to maintain code well one must first grok the code. "To grok," as coined by Robert Heinlein, means to comprehend deeply and fully; in this case, that includes understanding the internal workings of the code itself, as well as its side effects and interactions with other sub-systems. It is altogether too easy to introduce new errors when changing code one does not fully understand. Code that is clear and understandable is easier to grok, and therefore, fixes to such code become less fragile, less risky, less likely to have un-intended side effects.

Most important, however, your end users benefit from clear and understandable code for all these reasons: Such code is likely to have had fewer initial bugs in the first place, and it's likely to have been maintained more correctly without as many new bugs being introduced.

Guideline

By default, prefer to write for clarity and correctness first.

Dissecting Index Tables

2. The following code presents an interesting and genuinely useful idiom for creating index tables into existing containers. For a more detailed explanation, see the original article [Hicks00].

   Critique this code and identify:

   a. Mechanical errors, such as invalid syntax or nonportable conventions.

   b. Stylistic improvements that would improve code clarity, reusability, and maintainability.

Again, let me repeat that which bears repeating: This code presents an interesting and genuinely useful idiom. I've frequently found it necessary to access the same container in different ways, such as using different sort orders. For this reason it can be useful indeed to have one principal container that holds the data (for example, a vector<Employee>) and secondary containers of iterators into the main container that support variant access methods (for example, a set<vector<Employee>::iterator, Funct> where Funct is a functor that compares Employee objects

Chapter 35. Generic Callbacks

Difficulty: 5

Part of the allure of generic code is its usability and reusability in as many kinds of situations as reasonably possible. How can the simple facility presented in the cited article be stylistically improved, and how can it be made more useful than it is and really qualify as generic and widely usable code?

JG Question

1. What qualities are desirable in designing and writing generic facilities? Explain.

Guru Question

2. The following code presents an interesting and genuinely useful idiom for wrapping callback functions. For a more detailed explanation, see the original article. [Kalev01]

   Critique this code and identify:

   a. Stylistic choices that could be improved to make the design better for more idiomatic C++ usage.

   b. Mechanical limitations that restrict the usefulness of the facility.

```
template < class T, void (T::*F)() >
class callback
{
public:
 callback(T& t) : object(t) {}          // assign actual object to T
 void execute() {(object.*F)();}        // launch callback function

private:
 T& object;
};
```

Solution

Generic Quality

1. What qualities are desirable in designing and writing generic facilities? Explain.

Generic code should above all be usable. That doesn't mean it has to include all options up to and including the kitchen sink. What it does mean is that generic code ought to make a reasonable and balanced effort to avoid at least three things:

1. Avoid undue type restrictions. For example, are you writing a generic container? Then it's perfectly reasonable to require that the contained type have, say, a copy constructor and a nonthrowing destructor. But what about a default constructor or an assignment operator? Many useful types that users might want to put into our container don't have a default constructor, and if our container uses it, then we've eliminated such a type from being used with our container. That's not very generic. (For a complete example, see Item 11 of Exceptional C++ [Sutter00].)
2. Avoid undue functional restrictions. If you're writing a facility that does X and Y, what if some user wants to do Z, and Z isn't so much different from Y? Sometimes you'll want to make your facility flexible enough to support Z; sometimes you won't. Part of good generic design is choosing the ways and means by which your facility can be customized or extended. That this is important in generic design should hardly be a surprise, though, because the same principle applies to object-oriented class design.

Policy-based design is one of several important techniques that allow "pluggable" behavior with generic code. For examples of policy-based design, see any of several chapters in [Alexandrescu01]; the SmartPtr and Singleton chapters are a good place to start.

This leads to a related issue:

3. Avoid unduly monolithic designs. This important issue doesn't arise as directly in our style example under consideration below, and it deserves some dedicated consideration in its own right, hence it gets not only its own Item, but its own concluding miniseries of Items: see Items 37 through 40.

In these three points, you'll note the recurring word "undue." That means just what it says: Good judgment is needed when deciding where to draw the line between failing to be sufficiently generic (the "I'm sure nobody would want to use it with anything but char" syndrome) on the one hand and overengineering (the "what if someday someone wants to use this toaster-oven LED display routine to control the booster cutoff on an interplanetary spacecraft?" misguided fantasy) on the other.

Dissecting Generic Callbacks

4. The following code presents an interesting and genuinely useful idiom for wrapping callback functions. For a more detailed explanation, see the original article. [Kalev01]

Here again is the code:

```
template < class T, void (T::*F)() >
class callback
{
public:
 callback(T& t) : object(t) {}       // assign actual object to T
```

Chapter 36. Construction Unions

Difficulty: 4

No, this Item isn't about organizing carpenters and bricklayers. Rather, it's about deciding between what's cool and what's uncool, good motivations gone astray, and the consequences of subversive activities carried on under the covers. It's about getting around the C++ rule of using constructed objects as members of unions.

JG Questions

1. What are unions, and what purpose do they serve?
2. What kinds of types cannot be used as members of unions? Why do these limitations exist? Explain.

Guru Question

3. The article [Manley02] cites the motivating case of writing a scripting language: Say that you want your language to support a single type for variables that at various times can hold an integer, a string, or a list. Creating a union { int i; list<int> l; string s; }; doesn't work for the reasons covered in Questions 1 and 2. The following code presents a workaround that attempts to support allowing any type to participate in a union. (For a more detailed explanation, see the original article.)

   Critique this code and identify:

   a. Mechanical errors, such as invalid syntax or nonportable conventions.

   b. Stylistic improvements that would improve code clarity, reusability, and maintainability.

```
#include <list>
#include <string>
#include <iostream>
using namespace std;

#define max(a,b) (a)>(b)?(a):(b)

typedef list<int> LIST;
typedef string STRING;

struct MYUNION {
  MYUNION() : currtype(NONE) {}
  ~MYUNION() {cleanup();}

  enum uniontype {NONE,_INT,_LIST,_STRING};
  uniontype currtype;
  inline int& getint();
```

Solution

Unions Redux

1. What are unions, and what purpose do they serve?

   Unions allow more than one object, of either class or built-in type, to occupy the same space in memory. For example:

```
// Example 36-1
//
union U {
 int i;
 float f;
};

U u;

u.i = 42;                          //ok, now i is active
std::cout << u.i << std::endl;

u.f = 3.14f;                       // ok, now f is active
std::cout << 2 * u.f << std::endl;
```

But only one of the types can be "active" at a time—after all, the storage can hold only one value at a time. Also, unions support only some kinds of types, which leads us into the next question:

2. What kinds of types cannot be used as members of unions? Why do these limitations exist? Explain.

From the C++ standard:

An object of a class with a non-trivial constructor, a non-trivial copy constructor, a non-trivial destructor, or a non-trivial copy assignment operator cannot be a member of a union, nor can an array of such objects.

In brief, for a class type to be usable in a union, it must meet all the following criteria:

- The only constructors, destructors, and copy assignment operators are the compiler-generated ones.
- There are no virtual functions or virtual base classes.
- Ditto for all of its base classes and nonstatic members (or arrays thereof).

That's all, but that sure eliminates a lot of types.

Unions were inherited from C. The C language has a strong tradition of efficiency and support for low-level close-to-the-metal programming, which has been compatibly preserved in C++; that's why C++ also has unions. On the other hand, the C language does not have any tradition of language support for an object model supporting class types with constructors and destructors and user-defined copying, which C++ definitely does; that's why C++ also has to define what, if any, uses of such newfangled types make sense with the "oldfangled" unions and do not violate the C++ object model including its object lifetime guarantees.

If C++'s restrictions on unions did not exist, Bad Things could happen. For example, consider what could happen if

Chapter 37. Monoliths "Unstrung," Part 1: A Look at std::string

Difficulty: 3

I've decided to conclude the Style Case Studies section somewhat impishly, with a miniseries targeting an example from the C++ standard library itself: std::string. We begin our critique with a review of an important design guideline and a contrasting overview of the standard string facility.

JG Question

1. What is a monolithic class, and why is it bad? Explain.

Guru Question

2. List all the member functions of std::basic_string.

Solution

Avoid Unduly Monolithic Designs

1. What is a monolithic class, and why is it bad? Explain.

The word "monolithic" is used to describe software that is a single, big, indivisible piece, like a monolith. The word "monolith" comes from the words "mono" (one) and "lith" (stone), whose vivid image of a single massive boulder well illustrates the heavyweight and indivisible nature of such code.

A single heavyweight facility that thinks it does everything is often a dead end. After all, a single big heavyweight facility doesn't necessarily do more—it often does less, because the more complex it is, the narrower its application and relevance is likely to be.

In particular, a class might fall into the "monolith trap" by trying to offer its functionality through member functions instead of nonmember functions, even when nonmember nonfriend functions would be possible and at least as good. This has at least two drawbacks:

- (Major) It isolates potentially independent functionality inside a class. The operation in question might otherwise be nice to use with other types, but because it's hardwired into a particular class, that won't be possible, whereas if it were exposed as a nonmember function template, it could be more widely usable.
- (Minor) It can discourage extending the class with new functionality. "But wait!" someone might say. "It doesn't matter whether the class's existing interface leans toward member or nonmember functions, because I can equally well extend it with my own new nonmember functions either way!" That's technically true and misses the salient point: If the class's built-in functionality is offered mainly (or only) via member functions and therefore presents that as the class's natural idiom, the extended nonmember functions cannot follow the original natural idiom and will always remain visibly second-class johnny-come-latelies. That the class presents its functions as members is a semantic statement to its users, who will be accustomed to the member syntax that extenders of the class cannot use. (Do we really want to go out of our way to make our users commonly have to look up which functions are members and which ones aren't? That already happens often enough even in better designs.)

Where it's practical, break generic components down into pieces.

Guidelines

Prefer "one class (or function), one responsibility."

Where possible, prefer writing functions as nonmember nonfriends.

The rest of this Item will go further toward justifying the latter guideline.

The Basics of Strings

2. List all the member functions of std::basic_string.

It's, well, a fairly long list.

Summary

Where it's practical, break generic components down into pieces:

- Prefer "one class (or function), one responsibility."
- Where possible, prefer writing functions as nonmember nonfriends.

Was the recitation of all those member functions worth it? With the drudge work now behind us, let's look back at what we've just traversed and use the information in the next Item.…

Chapter 38. Monoliths "Unstrung," Part 2: Refactoring std::string

Difficulty: 5

"All for one, and one for all" might work for musketeers, but it doesn't work nearly as well for class designers. Here's an example that is not altogether exemplary, and it illustrates just how badly you can go wrong when design turns into overdesign. The example is, unfortunately, taken from a standard library near you.

JG Question

1. Which member functions of std::string must be member functions? Why?

Guru Question

2. Which member functions of std::string should be members? Why?
3. Demonstrate why std::string's member functions at, clear, empty, and length can be provided as nonmember nonfriend functions without loss of generality or usability, and without impact on the rest of std::string's interface.

Solution

Membership Has Its Rewards—and Its Costs

Recall the advice from the last Item: Where it's practical, break generic components down into pieces. Prefer "one class (or function), one responsibility." Where possible, prefer writing functions as nonmember nonfriends.

For example, if you write a string class and make searching, pattern-matching, and tokenizing available as member functions, you've hardwired those facilities so that they can't be used with any other kind of sequence. (If this frank preamble is giving you an uncomfortable feeling about basic_string, well and good.) On the other hand, a facility that accomplishes the same goal but is composed of several parts that are also independently usable is often a better design. In this example, it's often best to separate the algorithm from the container, which is what the STL does most of the time.

I (in Items 31 through 34 of Exceptional C++ [Sutter00]] and Scott Meyers (in [Meyers00]) have written before on why some nonmember functions are a legitimate part of a type's interface and why nonmember nonfriends should be preferred—among other reasons, to improve encapsulation. For example, as Scott wrote in his opening words of the cited article:

I'll start with the punchline: If you're writing a function that can be implemented as either a member or as a non-friend non-member, you should prefer to implement it as a non-member function. That decision increases class encapsulation. When you think encapsulation, you should think non-member functions.

—[Meyers00]

So when we consider the functions that will operate on a basic_string (or any other class type), we want to make them nonmember nonfriends if reasonably possible. Hence, here are some questions to ask about the members of basic_string in particular:

- Always make it a member if it has to be one. Which operations must be members, either because the C++ language just says so (e.g., constructors) or because of functional reasons (e.g., they must be virtual)? If they have to be, then oh well, they just have to be; case closed.
- Prefer to make it a member if it needs access to internals. Which operations need access to internal data we would otherwise have to grant via friendship? These should normally be members. Note that there are some rare exceptions such as operations needing conversions on their left-hand arguments and some like operator<< whose signatures don't allow the *this reference to be their first parameters; even these can normally be nonfriends implemented in terms of (possibly virtual) members, but sometimes doing that is merely an exercise in contortionism and they're best and naturally expressed as friends.
- In all other cases, prefer to make it a nonmember nonfriend. Which operations can work equally well as nonmember nonfriends? These can, and therefore normally should, be nonmembers. This should be the default case to strive for.

It's important to ask these and related questions because we want to put as many functions into the last bucket as possible.

A word about efficiency: For each function, we'll consider whether it can be implemented as a nonmember nonfriend function as efficiently as a member function. Is this premature optimization (an evil I often rail against)? Not a bit of it. The primary reason why I'm going to consider efficiency is to demonstrate just how many of basic_string's member functions could be implemented "equally well as nonmember nonfriends." I want to specifically shut down any accusations that making them nonmember nonfriends is a potential efficiency hit, such as preventing an operation from being performed in constant time if the standard so requires. A secondary reason is that optimizing a general-purpose

Chapter 39. Monoliths "Unstrung," Part 3: std::string Diminishing

Difficulty: 5

The Slim-Fast diet continues, and std::string is indeed slimming fast.

JG Question

1. Can string::resize be a nonmember function? Explain.

Guru Question

2. Analyze the following member functions of std::string and demonstrate whether or not they could or should instead be nonmember functions. Justify your an-swers.
   a. assign and +=/append/push_back
   b. insert

Solution

More Operations That Can Be Nonmember Nonfriends

In this Item, we'll see that all the following functions can be implemented as non-member nonfriends:

- resize (2)
- assign (6)
- += (3)
- append (6)
- push_back
- insert (7—all but the three-parameter version)

Let's investigate.

resize

1. Can string::resize be a nonmember function? Explain.

Well, let's see:

```
void resize(size_type n, charT c);
void resize(size_type n);
```

Can each resize be a nonmember nonfriend? Sure it can, because it can be implemented in terms of basic_string's public interface without loss of efficiency. Indeed, the standard's own functional specifications express both versions in terms of the functions we've already considered. For example:

```
template<class charT, class traits, class Allocator>
void resize(basic_string<charT, traits, Allocator>& s,
            typename Allocator::size_type n, charT c)
{
 if(n > s.max_size()) throw length_error("won't fit");
 if(n <= s.size()) {
  basic_string<charT, traits, Allocator> temp(s, 0, n);
  s.swap(temp);
} else {
  s.append(n - s.size(), c);
 }
}

template<class charT, class traits, class Allocator>
void resize(basic_string<charT, traits, Allocator>& s,
            typename Allocator::size_type n)
{
```

Chapter 40. Monoliths "Unstrung," Part 4: std::string Redux

Difficulty: 6

In this Item, we enter the home stretch, and uncover a (more) modest and minimal edition of std::string.

JG Question

1. Can string::erase be a nonmember function? Explain.

Guru Question

2. Analyze the remaining member functions of std::string and demonstrate whether or not they can or should instead be nonmember functions. Justify your answers.
   a. replace
   b. copy and substr
   c. compare
   d. find family (find, find_*, and rfind)

Solution

Still More Operations That Can Be Nonmember Nonfriends

In this Item, we're in the home stretch, and the finish line is in sight. Only a few functions remain—and we'll see that all of them can be implemented as nonmember nonfriends:

- erase (2—all but the "iter, iter" version)
- replace (8—all but the "iter, iter, num, char" and templated versions)
- copy
- substr
- compare (5)
- find (4)
- rfind (4)
- find_first_of (4)
- first_last_of (4)
- find_first_not_of (4)
- find_last_not_of (4)

Coffee Break (Sort Of): Erasing erase

1. Can string::erase be a nonmember function? Explain.

Pressing onward from our last Item, next let's tackle erase: After talking about the total 30-count'em-30 flavors of assign, append, insert, and replace—and having dealt with 20 of the 30 already—you will be relieved to know that there are only three forms of erase and that only two of them belong in this section. After what we just went through for the others, this is like knocking off for a well-deserved coffee break…

The troika of erase members is a little interesting. At least one of these erase functions must be a member (or friend), there being no other way to erase efficiently using the other already-mentioned member functions alone. There are actually two "families" of erase functions:

```
// erase(pos, length)
```

Bibliography

Note: For browsing convenience, this bibliography is also available online at:

http://www.gotw.ca/publications/xc++s/bibliography.htm

Bold references (e.g., [Alexandrescu02]) are hyperlinks in the online bibliography.

[Alexandrescu01] A. Alexandrescu. Modern C++ Design (Addison-Wesley, 2001).

[Alexandrescu02] A. Alexandrescu. "Discriminated Unions (I)," "… (II)," and "… (III)" (C/C++ Users Journal, 20(4,6,8), April/June/August 2002).

[Arnold00] M. Arnold, S. Fink, D. Grove, M. Hind, P. F. Sweeney. "Adaptive Optimization in the Jalapeño JVM" (Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2000).

[Bentley00] J. Bentley. Programming Pearls, Second Edition (Addison-Wesley, 2000).

[Boost] C++ Boost (www.boost.org).

[BoostES] "Boost Library Requirements and Guidelines, Exception-specification rationale" (Boost web site).

[Cargill94] T. Cargill. "Exception Handling: A False Sense of Security" (C++ Report, 9(6), November-December 1994).

[C90] ISO/IEC 9899:1990(E), Programming LanguagesC (ISO C90 and ANSI C89 standard).

[C99] ISO/IEC 9899:1999(E), Programming LanguagesC (revised ISO and ANSI C99 standard).

[C++98] ISO/IEC 14882:1998(E), Programming LanguagesC++ (ISO and ANSI C++ standard).

[C++03] ISO/IEC 14882:2003(E), Programming LanguagesC++ (up-dated ISO and ANSI C++ standard including the contents of [C++98] plus errata corrections).

[C++CLI04] C++/CLI Language Specification, Working Draft 1.6 (Ecma International, August 2004).

[Cline99] M. Cline, G. Lomow, and M. Girou. C++ FAQs, Second Edition (Addison-Wesley, 1999).

[Coplien92] J. Coplien. Advanced C++ Programming Styles and Idioms (Addison-Wesley, 1992).

[Dewhurst02] S. Dewhurst. "C++ Hierarchy Design Idioms" (Software Development 2002 West conference talk, April 2002).

[Dewhurst03] S. Dewhurst. C++ Gotchas (Addison-Wesley, 2003).

[Ellis90] M. Ellis and B. Stroustrup. The Annotated C++ Reference Manual (Addison-Wesley, 1990).

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley, 1995).

[GotW] H. Sutter. Guru of the Week

[Hicks00] C. Hicks. "Creating an Index Table in STL" (C/C++ Users Journal, 18(8), August 2000).