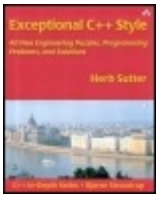


Exceptional C++ Style 40 New Engineering Puzzles, Programming Problems, and Solutions

By Herb Sutter

Start Reading

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

- Table of Contents

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.

Yes, I think Budapest is a great city—so rich with history, so replete with metaphor.

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: `vector`s 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.

First, the substantive problem is that this assertion might fail. Why might it fail? Because the call to `reserve` will guarantee that the `vector`'s `capacity` is at least 2—but it might also be greater than 2. Indeed it is likely to be greater because a `vector`'s size must grow exponentially and so typical implementations might choose to always grow the internal buffer on exponentially increasing boundaries even when specific sizes are requested via `reserve`. So this comparison should instead test by using >=, not strict equality:

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

Second, the stylistic problem is that the assertion (even the corrected version) is redundant. Why? Because the standard already guarantees what is here being asserted. Why add needless clutter by testing for it explicitly? This doesn't make sense unless you suspect a bug in the standard library implementation you're using, in which case you have bigger problems.

```
v[0] = 1;
v[1] = 2;
```

Both of these lines are flat-out errors, but they might be hard-to-find flat-out errors because they'll likely "work" after a fashion on your implementation of the standard library.

There's a big difference between `size` (which goes with `resize`) and `capacity` (which goes with `reserve`):

- `size` tells you how many elements are currently actually present in the container, and `resize` adjusts the actual contents of the container to be the specified size by adding or removing elements at the end. Both functions are available for `list`, `vector`, and `deque`, but not other containers.

- `capacity` tells you how many elements have room before adding another would force the `vector` to allocate more space, and `reserve` grows (never shrinks) into a larger internal buffer if necessary to ensure at least the specified space is available. Both functions are available only for `vector`.

In this case, we used `v.reserve(2)` and therefore know that `v.capacity() >= 2`, and that's fine as far as it goes—but we never actually added any elements to `v`, so `v` is still empty! At this point `v` just happens to have room for two or more elements.

Guideline

Remember the difference between `size`/`resize` and `capacity`/`reserve`.

We can safely use `operator[]` (or `at`) only to modify elements that are actually present, which means that they count toward `size`. At first you might wonder why `operator[]` couldn't just be smart enough to add the element if it's not already there, but if `operator[]` were allowed to work this way, you could create a vector with "holes" in it! For example, consider:

```
vector<int> v;
v.reserve(100);
v[99] = 42; // an error, but for the sake of discussion let's say it was allowed.

// … then at this point what can we say about the values of v[0..98]?
```

Alas, because `operator[]` isn't required to perform range-checking, on most implementations the expression `v[0]` will simply return a reference to the not-yet-used space in the `vector`'s internal buffer where the first element would eventually go. Therefore, the statement `v[0] = 1;` will probably appear to work, kind of, sort of, in that if the program were to `cout << v[0]` now, the result would probably be 1, just as (mis)expected.

Again, whether this will actually happen on the implementation you're using isn't guaranteed; it's just one typical possibility. The standard puts no requirements on what the implementation should do with flat-out errors such as writing `v[0]` for an empty `vector v`, because the programmer is assumed to know enough not to write such things. And after all, if the programmer had wanted the library to perform bounds-checking, then presumably he would have written `v.at(0)`, right?

Of course, the assignments `v[0] = 1; v[1] = 2;` would have been fine if the earlier code had performed a `v.resize(2)` instead of just a `v.reserve(2)`—but it didn't, so they're not. Alternatively, it would have been fine to replace them with `v.push_back(1); v.push_back(2);` which is the always-safe way to tack elements onto the end of a container.

```
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
 cout << *i << endl;
```

```
}
```

First, note that this loop prints nothing because of course the `vector` is still empty. This might surprise the original programmer, until that programmer realizes that the earlier code never really added anything to the `vector`—it was just (dangerously) playing around with some of the reserved but still-officially-unused space hanging around inside the `vector`.

Having said that, there is no outright error in this loop, but there are several stylistic problems that I would comment on if I saw this code in a code review setting. Most are low-level comments:

1. Be as `const correct` as possible. The iterator is never used to modify the `vector`'s contents, so it should be a `const_iterator`.

2. Prefer comparing iterators with `!=`, not `<`. True, because `vector<int>::iterator` happens to be a random-access iterator (not necessarily an `int*`, of course!), there's no downside to using `<` in the comparison with `v.end()` as shown. But `<` only works with random-access iterators, whereas `!=` works with other iterator types too, so we should routinely use `!=` all the time unless we really need `<`. (Note that using `!=` also makes it that much easier to switch to using a different container in the future, if desired. For example, `std::list`'s iterators don't support `<` because they're only bidirectional iterators.)

3. Prefer using prefix `--` and `++`, instead of postfix. Get in the habit of by default writing `++i` instead of `i++` in loops unless you really need the old value of `i`. For example, postfix is natural and fine when you're writing something like `v[i++]` to access the `i`-th element and at the same time increment a loop counter.

4. Avoid needless recalculations. In this case, the value returned by `v.end` doesn't change during the loop, so instead of recalculating it on every iteration, it might be worthwhile to precalculate it before the loop begins.

Note: If your implementation's `vector<int>::iterator` is just an `int*`, and your implementation inlines `end` and does reasonable optimization, it's probable that there's zero overhead here anyway because the compiler will probably be able to see that the value returned by `end` doesn't change and that it can therefore safely hoist the code out of the loop. This is a pretty common case. However, if your implementation's `vector<int>::iterator` is not an `int*` (for example, in most debugging implementations it would instead be of class type), the function(s) are not inlined, and/or the compiler doesn't perform the suggested optimizations, then hoisting the calculation code out of the loop yourself can make a performance difference.

5. Prefer `'\n'` to `endl`. Using `endl` forces the stream to flush its internal output buffers. If the stream is buffered and you don't really need a flush each time, just write a flush once at the end of the loop and your program will perform that much faster.

Finally, the last comment hits at a higher level:

6. Prefer reusing the standard library's `copy` and `for_each` instead of handcrafting your own loops, where using the standard facilities is clean and easy. Season to taste. I say "season to taste" because here's one of those places where taste and aesthetic judgment do matter. In simple cases, `copy` and `for_each` can and do improve readability over handcrafted loops. Beyond those simple cases, though, unless you have a nice expression template library available, code written using `for_each` can get unreadable pretty quickly because the code in the loop body has to be split off into functors. Sometimes that kind of factoring is still a good thing; other times it's merely obscure.

That's why your tastes may vary. Still, in this case I would be tempted to replace the loop with something like:

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
```

Besides, when you reuse `copy` like this, you can't get the `!=`, `++`, `end`, and `endl` parts wrong to begin with, because they're done for you. (Of course, this assumes that you don't want to flush the output stream after each `int` is output; if you do, there's no way to do it without writing your own loop instead of reusing `std::copy`.) Reuse, when applied well, not only makes code more readable but can also make it better by avoiding some opportunities for pitfalls.

You can take this a step further and write a container-based algorithm for copying—that is, an algorithm that operates on an entire container, not just an iterator range. Doing this would also automatically get the `const_iterator` part right. For example:

```
template<class Container, class OutputIterator>
OutputIterator copy(const Container& c, OutputIterator result) {
 return std::copy(c.begin(), c.end(), result);
}
```

Here we simply wrap `std::copy` for an entire container, and because the container is taken by `const&` the iterators will automatically be `const_iterator`s.

Guidelines

Be `const correct`. In particular, use `const_iterator` when you are not modifying the contents of a container.

Prefer comparing iterators with `!=`, not `<`.

Get in the habit of using the prefix forms of `--` and `++` by default, unless you really need the old value.

Practice reuse: Prefer reusing existing algorithms, particularly standard algorithms (e.g.,

`for_each`), instead of crafting your own loops.

Next, we encounter the code:

```
cout << v[0];
```

When the program performs `cout << v[0];` now, it will probably produce a 1. This is because the program scribbled over memory in a way that it shouldn't have, but that will probably not cause an immediate fault—more's the pity.

```
v.reserve(100);
assert(v.capacity() == 100);
```

Again, this assertion should use >= and then becomes redundant anyway, as before.

```
cout << v[0];
```

Surprise! This time, the `cout << v[0];` will probably produce a 0—the value 1 that we just set has mysteriously vanished!

Why? Assuming that the `reserve(100)` actually did trigger a reallocation of `v`'s internal buffer (i.e., if the initial call to `reserve(2)` didn't already raise the capacity to 100 or more), `v` would only copy over into the new buffer the elements it actually contains—and it doesn't actually think it contains any! The new buffer initially holds zeroes, and that's what stays there.

```
v[2] = 3;
v[3] = 4;
// …
v[99] = 100;
```

No doubt you are even now shaking your head sadly at this deplorable code. This is bad, bad, bad… but because `operator[]` isn't required to perform bounds-checking, on most implementations this will probably silently appear to work and won't cause an immediate exception or memory trap.

If instead the user had written:

```
v.at(2) = 3;
v.at(3) = 4;
// …
v.at(99) = 100;
```

then the problem would have become obvious, because the very first call would have thrown an `out_of_range` exception.

```
for(vector<int>::iterator i = v.begin(); i < v.end(); i++) {
 cout << *i << endl;
}
```

Again this prints nothing, and I'd consider replacing it with:

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
```

Notice again that this reuse automatically solves the `!=`, prefix `++`, `end`, and `endl` comments too—the opportunity to get them wrong never arises! Good reuse often makes code automatically faster and safer, too.

Summary

Know the difference between `size` and `capacity`. Know also the difference between `operator[]` and `at`, and always use the latter if you want bounds-checked access. Doing so can easily save you long hours of sweat and tears inside a debugger.

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

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

The code in Example 2-1 is just one example of how we might use `sprintf`. I'm going to use Example 2-1 as a motivating case for discussion, but don't get too tied to this simple `PrettyFormat` one-liner. Keep in mind the larger picture: We're interested in looking at how we would normally choose to format nonstring values as strings in the general case, perhaps in code that's more likely to change and grow over time than the simple case in Example 2-1.

I'm going to list the major issues involved by analyzing `sprintf` in more detail. `sprintf` has two major advantages and three distinct disadvantages. The two advantages are as follows:

Issue #1: Ease of use and clarity. Once you've learned the commonly used formatting flags and their combinations, using `sprintf` is succinct and obvious, not convoluted. It says directly and concisely what needs to be said. For this, the `printf` family is hard to beat in most text-formatting work. (True, most of us still have to look up the more rarely used formatting flags, but they are after all used rarely.)

Issue #2: Maximum efficiency (ability to directly use existing buffers). By using `sprintf` to put the result directly into an already-provided buffer, `PrettyFormat` gets the job done without needing to perform any dynamic memory allocations or other extra off-to-the-side work. It's given an already-allocated place to put the output and puts the result directly there.

Caveat lector: Of course, don't put too much weight on efficiency just yet; your application might well not notice the difference. Never optimize prematurely, but optimize only when timings show that you really need to do so. Write for clarity first and for speed later if necessary. In this case, never forget that the efficiency comes at the price of memory management encapsulation—Issue #2 is phrased here as "you get to do your own memory management," but the flip side is "you have to do your own memory management"!

Alas, as most `sprintf` users know, the story doesn't end quite there. `sprintf` also has these significant drawbacks:

Issue #3: Length safety. Using `sprintf` is a common source of buffer overrun errors if the destination buffer doesn't happen to be big enough for the whole output.[1] For example, consider this calling code:

[1] A common beginner's error is to rely on the width specifier, here `4`, which doesn't work, because the width specifier dictates a minimum width, not a maximum width.

```
char smallBuf[5];
int value = 42;
PrettyFormat(value, buf);          // er, well, sort of okay
assert(value == 42);
```

In this case, the value `42` happens to be small enough so that the five-byte result `"42\0"` happens to fit into `smallBuf`. But the day the code changes to:

```
char smallBuf[5];
int value = 12108642;
PrettyFormat(value, buf);          // oops
assert(value == 12108642);         // likely to fail
```

we'll start scribbling past the end of `smallBuf`, which might be into the bytes of `value` itself if the compiler chose a memory layout that put `value` immediately after `smallBuf` in memory.

We can't easily make Example 2-1 much safer, though. True, we could change Example 2-1 to take the length of the buffer and then check `sprintf`'s return value, which will tell after the fact how many bytes `sprintf` ended up writing. This gives us something like:

```
// BAD: A not-at-all-improved PrettyFormat.
//
void PrettyFormat(int i, char* buf, int buflen) {
 if(buflen <= sprintf(buf, "%4d", i)) {         // this is no better

   // …and now what? by the time the problem is detected here,
   // we've already corrupted whatever we were going to corrupt
 }
}
```

That's no solution at all. By the time the error is detected, the overrun has already occurred, we'll already have scribbled on someone else's bytes, and in bad cases our execution might never even get to the error-reporting code.[2]

[2] Note that in some cases you can mitigate the buffer length problem, at least in theory, by creating your own formats at run-time. I say "in theory" because this is usually impractical; the code is always obscure and often fragile. As Bjarne Stroustrup puts it in [Stroustrup99], speaking of a similar case:

> The expert-level alternative is not one I'd care to explain to novices:
>
> ```
> char fmt[10];
>
> // create a format string: plain %s can overflow
> psprintf(fmt,"%%%ds",max-1);
> ```

```
// read at most max-1 characters into name
scanf(fmt,name);
```

Issue #4: Type safety. For `sprintf`, type errors are run-time errors, not compile-time errors, and they might not even manifest right away. The `printf` family uses C's variable argument lists, and C compilers generally don't check the parameter types for such lists.[3] Nearly every C programmer has had the joy of finding out in subtle and not-so-subtle ways that they got the format specifier wrong, and all too often such errors are found only after a pressure-filled late-night debugging session spent trying to duplicate a mysterious crash reported by a key customer.

[3] Using `lint`-like tools will help to catch this kind of error.

Granted, the code in Example 2-1 is so trivial that it's likely easy enough to maintain now when we know we're just throwing a single `int` at `sprintf`, but even so it's not hard to go wrong if your finger happens to hit something other than `d` by mistake.

For example, `c` happens to be right next to `d` on most keyboards; if we'd simply mistyped the `sprintf` call as

```
sprintf(buf, "%4c", i);              // oops
```

then we'd probably see the mistake quite quickly when the output is some character instead of a number, because `sprintf` will silently reinterpret the first byte of `i` as a `char` value. Alternatively, `s` is also right next to `d`, and if we'd mistyped it as

```
sprintf(buf, "%4s", i);              // oops again
```

then we'd probably also catch the error quite quickly because the program is likely to crash immediately or at least intermittently. In this case `sprintf` will silently reinterpret the integer as a pointer to `char` and then happily attempt to follow that pointer into some random region of memory.

But here's a more subtle one: What if we'd instead mistyped `d` as `ld`?

```
sprintf(buf, "%4ld", i);             // a subtler error
```

In this case, the format string is telling `sprintf` to expect a `long int`, not just an `int`, as the first piece of data to be formatted. This too is bad C code, but the trouble is that not only won't this be a compile-time error, but it might not even be a run-time error right away. On many popular platforms, the result will still be the same as before. Why? Because on many popular platforms `int`s happen to

have the same size and layout as `long`s. You might not notice this error until you port the code to a platform where `int` isn't the same size as `long`, and even then it might not always produce incorrect output or immediate crashes.

Finally, consider a related issue.

Issue #5: Templatability. It's very hard to use `sprintf` in a template. Consider:

```
template<typename T>
void PrettyFormat(T value, char* buf) {
 sprintf(buf, "%/*what goes here?*/", value);
}
```

The best (worst?) you could do is declare the primary template and then provide specializations for all the types that are compatible with `sprintf`:

```
// BAD: A kludgy templated PrettyFormat.
//
template<typename T>
void PrettyFormat(T value, char* buf);  // note: primary template is not defined

template<> void PrettyFormat<int>(int value, char* buf) {
 sprintf(buf, "%d", value);
}
template<> void PrettyFormat<char>(char value, char* buf) {
 sprintf(buf, "%c", value);
}

// … etc., ugh …
```

In summary, here's `sprintf`:

	Sprintf
Standard?	Yes: [C90], [C++03], [C99]
Easy to use, good code clarity?	Yes
Efficient, no extra allocation?	Yes
Length-safe?	No
Type-safe?	No
Usable in template?	No

The other solutions we'll consider in the next Item choose different tradeoffs among these

considerations.

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 weakne 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 `sprintf`, but it's an important one: the ability to specify the maximum length of the output buffer, there 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 implemen With the advent of the C99 standard [C99], `snprintf` has "come out" and gone legit, now officially sa as a standard facility. Until your own compiler is C99-compliant, though, you might have to use this un vendor-specific extension name such as `_snprintf`.

Frankly, you should already have been using `snprintf` over `sprintf` anyway, even before `snprintf` standard. Calls to length-unchecked functions such as `sprintf` are banned in most good coding standa for good reason. The use of unchecked `sprintf` calls has long been a notoriously common problem, c 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 b sending them very long URLs that were likely to be longer than the web browser's internal URL s 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 ar with malicious code that could then be executed. It's surprising just how much software out there 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 bulletproof for overflow safety as the later alternatives that encapsulate their own resource manageme certainly lots safer and deserves a "Yes" under the "Length-safe?" question. With `sprintf` we have no 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 implementation, if the output fills or would overflow the buffer, the buffer is not zero-terminated. On s environments, the function would need to be written slightly differently to account for the nonstandard

```
// 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:
 if(buflen > 0) {
   _snprintf(buf, buflen-1, "%4d", i);
   buf[buflen-1] = '\0';
 }
}
```

In every other way, `sprintf` and `snprintf` are the same. In summary, here's how `snprintf` compares `sprintf`:

	Snprintf	**sprintf**
Standard?	Yes: [C99] only, but will likely also be in C++0x	Yes: [C90], [C++03], [C99]
Easy to use, good code clarity?	Yes	Yes
Efficient, no extra allocation?	Yes	Yes
Length-safe?	Yes	No
Type-safe?	No	No
Usable in template?	No	No

From this, we derive the following advice:

Guideline

Never use `sprintf`.

If you do decide to use C `stdio` facilities, always use length-checked calls such as `snprintf` even if t
only available as a nonstandard extension on your current compiler. There's no drawback, and there's
benefit, to using `snprintf` instead.

When I've presented this material at conferences, at first I was shocked to discover that typically only
percent of a given class has heard of `snprintf`. But, nearly every time, there's one person in the audie
immediately puts up his hand to describe how, on his current project, they'd recently discovered a few
overrun bugs, globally replaced `sprintf` with `snprintf` throughout the project, and found during testi
only were those bugs gone but suddenly several other mysterious bugs had also disappeared—bugs tha
reported for years but that the team hadn't been able to diagnose and that had just been festering in the

As I was saying, ahem: Never use `sprintf`.

Alternative #2: std::stringstream

b. `std::stringstream`

The most common facility in C++ for stringizing data is the `stringstream` family. Here's what Examp
would look like, using an `ostringstream` instead of `sprintf`:

```
// Example 3-2: Stringizing some data in C++, using ostringstream.
//
void PrettyFormat(int i, string& s) {
 // Not quite as neat and simple:
 ostringstream temp;
 temp << setw(4) << i;
 s = temp.str();
}
```

Using `stringstream` exchanges the advantages and disadvantages of `sprintf`. Where `sprintf` shine
`stringstream` does less well:

Issue #1: Ease of use and clarity. Not only has one line of code turned into three, but we've also neede
introduce a temporary variable. This version of the code is superior in several ways, but code clarity
them. It's not that the manipulators are hard to learn—they're as easy to learn as the `sprintf` formattin
flags—but that they're generally more clumsy and verbose. I find that code littered with long names su
`setprecision(9)` and `<< setw(14)` all over the place is a bear to read (compared to, say, `%14.9`), e
all the manipulators are arranged reasonably well in columns.

Issue #2: Efficiency (ability to directly use existing buffers). A `stringstream` does its work in an ad
buffer off to the side and so will usually have to perform extra allocations for that working buffer and
other helper objects it uses. I tried the Example 3-2 code on two popular current compilers and instru
`::operator new` to count the allocations being performed. One platform performed two dynamic me
allocations, and the other performed three.

Where `sprintf` breaks down, however, `stringstream` glitters:

Issue #3: Length safety. The `stringstream`'s internal `basic_stringbuf` buffer automatically grows a to fit the value being stored.

Issue #4: Type safety. Using `operator<<` and overload resolution always gets the types right, even for defined types that provide their own stream insertion operators. No more obscure run-time errors beca type mismatches.

Issue #5: Templatability. Now that the right `operator<<` is automatically called, it's trivial to generali `PrettyFormat` to operate on arbitrary data types:

```
template<typename T>
void PrettyFormat(T value, string& s) {
 ostringstream temp;
 temp << setw(4) << value;
 s = temp.str();
}
```

In summary, here's how `stringstream` compares to `sprintf`:

	stringstream	**sprintf**
Standard?	Yes: [C++03]	Yes: [C90], [C++03], [C99]
Easy to use, good code clarity?	No	Yes
Efficient, no extra allocation?	No	Yes
Length-safe?	Yes	No
Type-safe?	Yes	No
Usable in template?	Yes	No

Alternative #3: std::strstream

c. `std::strstream`

Fairly or not, `strstream` is something of a persecuted pariah. Because it has been deprecated in the [C standard, the top C++ books at best cover it briefly (see [Josuttis99] page 649), mostly ignore it (see [Stroustrup00]), or even explicitly state they won't cover it because of its official second-string status [Langer00] page 587). Although deprecated because the standards committee felt it was superseded by

stringstream, which better encapsulates memory management, strstream is still an official part of standard that conforming C++ implementers must provide.[6]

[6] What does "deprecated" mean, in theory and in practice? When it comes to standards, "deprec denotes a feature that the committee warns you might disappear anytime in the future, possibly as as the next revision of the standard. To deprecate a feature amounts to "normative discouragemen the strongest thing the committee can do to discourage you from using a feature without actually ta the feature away from you immediately. In practice, it's hard to remove even the worst deprecated features because, once the feature appears in a standard, people write code that depends on the feature—and every standards body is loath to break backward compatibility. Even when a feature removed, implementers often continue to supply it because they, too, are loath to break backward compatibility. Oftentimes, deprecated features never do disappear from the standard. Standard Fo for example, still has features that have been deprecated for decades.

Because strstream is still standard, it deserves mention here too for completeness. It also happens to useful mix of strengths. Here's what Example 3-1 might look like using strstream:

```
// Example 3-3: Stringizing some data in C++, using ostrstream.
//
void PrettyFormat(int i, char* buf, int buflen) {
 // Not too bad, just don't forget ends:
 ostrstream temp(buf, buflen);
 temp << setw(4) << i << ends;
}
```

Issue #1: Ease of use and clarity. strstream comes in slightly behind stringstream for ease of use a clarity. Both require a temporary object to be constructed. With strstream you have to remember to t ends to terminate the string, which is somewhere between distasteful and dangerous. If you forget to d are in danger of overrunning the end of the buffer when reading it afterward if you're relying on its bei terminated by a null character; even sprintf isn't this fragile and always tacks on the null. But at leas strstream in the manner shown in Example 3-3 doesn't require calling a .str function to extract the r the end. (Of course, alternatively, if you let strstream create its own buffer, the memory is only partly encapsulated; you will need not only a .str call at the end to get the result out, but also a .freeze(fa the strstreambuf won't free the memory.)

Issue #2: Efficiency (ability to directly use existing buffers). By constructing the ostrstream object w pointer to an existing buffer, we don't need to perform any extra allocations at all; the ostrstream wil result directly in the output buffer. This is an important divergence from stringstream, which offers comparable facility for placing the result directly in an existing destination buffer, thereby avoiding ex allocation.[7] Of course, ostrstream can alternatively use its own dynamically allocated buffer if you have one handy already; just use ostrstream's default constructor instead.[8] Indeed, strstream is th of the options covered here that gives you this choice.

[7] stringstream does offer a constructor that takes a string&, but it simply takes a copy of the

`string`'s contents instead of directly using the supplied `string` as its work area.

[8] In Table 3-1's performance measurements, `strstream` shows unexpectedly poorly on two platforms, Borland C++ 5.5.1 and Visual C++ 7. The reason appears to be that on those implementations for some reason some allocations are always performed on each call to Example 3's `PrettyFormat()` (although both implementations still actually do perform fewer allocations given an existing buffer to work with, as is done in Example 3-3, than when the `strstream` has to make its own buffer). The other environments, as expected, perform no allocations.

Issue #3: Length safety. As used in Example 3-3, the `ostrstream`'s internal `strstreambuf` buffer auto checks its length to make sure it doesn't write beyond the end of the supplied buffer. If instead we had default-constructed `ostrstream`, its internal `strstreambuf` buffer would automatically grow as need the value being stored.

Issue #4: Type safety. Fully type-safe, just like `stringstream`.

Issue #5: Templatability. Fully templatable, just like `stringstream`. For example:

```
template<typename T>
void PrettyFormat(T value, char* buf, int buflen) {
 ostrstream temp(buf, buflen);
 temp << setw(4) << value << ends;
}
```

In summary, here's how `strstream` compares to `sprintf`:

	strstream	**sprintf**
Standard?	Yes: [C++03], but deprecated	Yes: [C90], [C++03], [C99]
Easy to use, good code clarity?	No	Yes
Efficient, no extra allocation?	Yes	Yes
Length-safe?	Yes	No
Type-safe?	Yes	No
Usable in template?	Yes	No

It's, um, slightly embarrassing that the deprecated facility shows so strongly in this side-by-side compa that's how life goes sometimes.

Alternative #4: boost::lexical_cast

d. `boost::lexical_cast`

If you haven't yet discovered [Boost], my advice is to discover it. It's a public library of C++ facilities principally by C++ standards committee members. Not only is it good peer-reviewed code written by and in the style of the C++ standard library, but these facilities also are explicitly intended as potential candidates for inclusion in the next C++ standard and are therefore worth getting to know. Besides, you freely use them today.

One of the facilities provided in the Boost libraries is `boost::lexical_cast`, which is a handy wrap around `stringstream`. Boost also includes other more ornate and heavyweight approaches that likew internal streams and provide for more `sprintf`-like formatting options, notably `boost::format`, but l the Boost code as written by Kevlin Henney is so concise and elegant, I can present it here in its entire workarounds for older compilers)… so I will, even though it's not a standardized facility:

```
template<typename Target, typename Source>
Target lexical_cast(Source arg) {
 std::stringstream interpreter;
 Target result;
 if(!(interpreter << arg) || !(interpreter >> result) || !(interpreter >> std::ws
 throw bad_lexical_cast();

 return result;
}
```

Note that `lexical_cast` is not intended to be a direct competitor for the more general string formatter Instead, `lexical_cast` is for converting data from one streamable type to another, and it competes mo with C's `atoi` et al. conversion functions as well as with the nonstandard but commonly available `ito` functions. It's close enough to our topic, however, that it definitely would be an omission not to mentio

Here's what Example 3-1 would look like using `lexical_cast`, minus the at-least-four-character requ

```
// Example 3-4: Stringizing some data in C++, using boost::lexical_cast.
//
void PrettyFormat(int i, string& s) {
 // Perhaps the neatest and simplest yet, if it's all you need:
 s = lexical_cast<string>(i);
}
```

Issue #1: Ease of use and clarity. This code embodies the most direct expression of intent of any of the examples.

Issue #2: Efficiency (ability to directly use existing buffers). Because `lexical_cast` uses `stringstr` no surprise that it needs at least as many allocations as `stringstream`. On one of the platforms I tried 3-4 performed one more allocation than the plain `stringstream` version presented in Example 3-2; o platform, it performed no additional allocations over the plain `stringstream` version.

Like `stringstream`, in terms of length safety, type safety, and templatability, `lexical_cast` shows ve strongly.

In summary, here's how `lexical_cast` compares to `sprintf`:

	lexical_cast	**sprintf**
Standard?	No (perhaps a potential candidate for C++0x)	Yes: [C90], [C++03], [C99]
Easy to use, good code clarity?	Yes	Yes
Efficient, no extra allocation?	No	Yes
Length-safe?	Yes	No
Type-safe?	Yes	No
Usable in template?	Yes	No

Summary

There are issues we've not considered in detail. For example, all the string formatting herein has been narrow `char`-based strings, not wide strings. We've also focused on the ability to gain efficiency by us existing buffers directly in the case of `sprintf`, `snprintf`, and `strstream`, but the flip side to "you g your own memory management" is "you have to do your own memory management," and the better enc of memory management offered by `stringstream`, `strstream`, and `lexical_cast` might matter to yo typo, `strstream` is in both lists; it depends on how you want to use it.)

There are also other nonstandard alternatives we've not considered in detail. I chose to show Boost's `lexical_cast` because of its elegant simplicity, but even in Boost there are more complete and more heavyweight options, notably `boost::format`, which provides more automation to support `sprintf`-l formatting on top of a similar approach to the `stringstream` and `strstream` techniques described he

Putting it all together, we get the side-by-side comparison summarized in Table 3-1. Given the conside we're using to judge the relative merits of each solution, there is no clear unique one-size-fits-all winn situations.

Table 3-1. C and C++ string formatting alternatives

	sprintf	**snprintf**	**stringstream**	**strstream**	**boost::lexical_cast**
Standard? (? = n/a)					

[C90]	Yes	No (common extension)	[illegible]	[illegible]	[illegible]
[C++03]	Yes	No (common extension)	Yes	Yes, but deprecated	No
[C99]	Yes	Yes	[illegible]	[illegible]	[illegible]
C++0x (speculation)	Yes	Likely	Yes	Likely (probably still deprecated)	Possible
	Usability Factors				
Easy to use, good code clarity?	Yes	Yes	No	No	Yes
Efficient, no extra allocation?	Yes	Yes	No	Yes	No
Length-safe?	No	Yes	Yes	Yes	Yes
Type-safe?	No	No	Yes	Yes	Yes
Usable in template?	No	No	Yes	Yes	Yes
	Sample Timings, Normalized to sprintf [9]				
Borland C++ 5.5.1 / Windows	1.0	1.0	12.6	8.1	19.7
Gnu g++ 2.95.2 / Cygwin + Windows	1.0	[illegible]	[illegible]	2.0	[illegible]
Microsoft VC7 / Windows	1.0	1.0	13.2	9.0	19.2
Rogue Wave 2.1.1 / SunPro 5.3 / SunOS 5.7	1.0	1.1	8.7	4.7	16.5
Rogue Wave	1.0	1.0	7.9	3.9	9.9

2.2.1 / HP aCC 3.30 / HP-UX 11.00		

[9] Sample results reporting the average of three runs each of which performs 1,000,000 calls to t corresponding Example code. Results may vary with other compiler versions and switch settings

On the basis of Table 3-1, we can justify the following guidelines, also summarized in Table 3-2:

- If all you're doing is converting a value to a `string` (or, for that matter, to anything else!): Prefer `boost::lexical_cast` by default.
- For simple formatting, or where you need wide string support or templatability: Prefer using `stringstream` or `strstream`; the code will be more verbose and harder to grasp than it would b `snprintf`, but for simple formatting it won't be too bad.
- For more complex formatting, and where you don't need wide string support or templatability: Pr `snprintf`. Just because it's C doesn't mean it's off limits to C++ programmers!
- Only if actual performance measurements show that any of the better alternatives is really a bottle specific point in your code: In those isolated cases only, instead consider using whichever one of alternatives `strstream` or `snprintf` makes sense.
- Never use `sprintf`.

Table 3-2. Guideline summary

	By default, where efficiency isn't an issue:	**In places where efficiency has measurably become an issue, really, your profiler told you so:**
If all you're doing is converting to a string representation:	`Boost::lexical_cast`	`std::strstream` or `snprintf`
For simple formatting, or where you need wide strings or templatability:	`std::stringstream` or `std::strstream`	`std::strstream` or `snprintf`
For more	`snprintf`	`snprintf`

complex formatting, if you don't need wide strings or templatability:

Finally, a last word about the pariah `strstream`: It offers an interesting combination of features, not th which being that it's the only option that allows you to choose whether to do your own memory manage let the object (partly) encapsulate it. Its lone technical drawback is that of being more fragile to use be the `ends` issue and the memory management approach; its only other drawback is social stigma, becaus shunted aside and doesn't get invited to parties much anymore, and you should be aware that there's a s possibility that both the standards committee and your compiler/library vendor might really take it awa you at some time in the future.

It's a bit strange to see a deprecated feature showing so well. Although a particular animal might have merits, even in the standard some animals are more equal than others.

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 oth
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 f
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 li
you should just remember to say `mem_fun_ref` if the container is a plain old container of objects, beca
`for_each` will be operating on references to those objects, and to say `mem_fun` if it's a container of pc
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 parameters. You can use the `bind...` helpers to deal with some functions that take an argument, and the is the same. Unfortunately you can't use this approach for functions that take two or more arguments. St 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 p C++? Why or why not?

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

First, note that no "incantation" should be necessary. I deliberately wrote the question this way becaus this writing some popular compilers cannot correctly deduce the template parameters. For such compi depending on your implementation of the standard library, you would have to write something like:

```
std::mem_fun<void, std::vector<int, std::allocator<int> > >(&(std::vector<int>::
```

Over time, this limitation will go away and compilers will be able to let you reliably omit the templat parameters.

You might wonder why I wrote "depending on your implementation of the standard library." After all, signature of `std::vector<int>::clear` is that it takes no parameters and returns `void`, right? The st tells us so, doesn't it?

Wrong (maybe), and that gets us to the crux of the problem.

The standard library specification deliberately gives some leeway to implementers when it comes to r functions. Specifically:

- A member function signature with default parameters might be replaced by "two or more member signatures with the equivalent behavior."
- A member function signature might have additional defaulted parameters.

Aye, and there, in the second item, is the rub: Those pesky "might-be-there-or-might-not," "now-you-se now-you-don't" extra parameter critters—for short, let's call them "peekaboo" parameters—are what c problem in this case.

Much of the time, any extra implementation-specific defaulted peekaboo parameters just go unnoticed; example, when you call a member function you'll get the default values for the peekaboo parameters, s don't need to ever be aware that the library implementer has thrown a few extra parameters on the end member function's signature. Unfortunately, such possible extra parameters do become very noticeable need to be sure of the exact signature of the member function—such as when you're trying to use `mem_f` that this is true even if your compiler deduces template arguments correctly, because of two potential p

- If the member function in question actually takes a parameter and you didn't expect one, you need something like `std::bind2nd` to get rid of it. (For more about the standard binder helpers, see [Josuttis99].) Of course, now your code won't work on implementations that tack on an extra para a different type, or none at all—but, hey, your code wasn't portable anyway, right?

- If the member function in question actually has two or more parameters (even if they're defaulted) can't use it with `mem_fun` at all. Bummer—but again, your code wasn't portable anyway, right?

In practice, though, the problem might not be all that bad. I don't know whether library implementers w avail themselves of the leeway to add extra parameters, or intend to do so in the future. To the extent th don't do so, you won't encounter these difficulties much in practice.

Unfortunately, though, that's not quite the end of the story. Finally, let's consider a more general conseq this leeway.

Use Pointers to Member Functions, Just Not with the Standard Library

Alas, there's an even more basic issue: It is impossible to portably create a pointer to a standard libra member function, period.

After all, if you want to create a pointer to a function, member or not, you have to know the pointer's ty which means you have to know the function's signature.

Because the signatures of standard library member functions are impossible to know exactly—unless y in your library implementation's header files to look for any peekaboo parameters, and even then the a might change on a new release of the same library—the bottom line is that you can't reliably form poin standard library member functions and still have portable code.

Summary

It's a little odd that you can portably use a standard library facility, namely `mem_fun`, with most everyt except the standard library itself. It's equally odd that you can portably use a language feature, namely to member function, with everything except the language's own standard library.

Normally the implementation latitude for standard member functions is invisible, and if all you're doin calling those functions you'll never know the difference. But if you use pointers to member functions, c binders, be aware that they can't be used reliably with standard library member functions—even if it h work for you today on particular member functions in your particular standard library implementation.

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 polymor we get from using virtual functions. A base class establishes an interface "contract" as defined by a set virtual functions, and derived classes may inherit from the base class and override the virtual function way that preserves the contracted semantics. Then other code that expects to work on a `Base` object (a 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. Secc when the virtual functions are called in a tight loop you might notice some performance penalty becaus normally each call to a virtual function must be made through an extra pointer indirection, as the comp 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 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 poss
//
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;

 // … more functions …

};
template<class T>
void h(T& t) {
 t.f();
 t.g();
}

int main() {
 Xyzzy x;
 Murgatroyd m;

 h(x);
 h(m);
}
```

As long as both objects `x` and `m` are of a type that provides member functions `f` and `g` that can be called no parameters, `h` will work. In Example 5-1(b), both types actually have different signatures for `f` and they also provide additional functions beyond just those two, but `h` doesn't care. As long as `f` and `g` can called without parameters, the compiler will allow `h` to make the calls. Of course, when called, those functions should also do something that's sensible for `h`!

So templates provide powerful compile-time polymorphism. Misuse of templates can cause really hard read error messages on many compilers, but templates are also one of C++'s most powerful features.

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.

```
// Example 5-2(a): construct.
//
template <class T1, class T2>
void construct(T1* p, const T2& value) {
 new (p) T1(value);
}
```

`construct` constructs an object in a given memory location, using an initial value. The form of `new` used here is called placement `new`, and instead of allocating memory for the new object, it just puts it into the

memory pointed at by `p`. Any object `new`'d in this way should generally be destroyed by calling its dest explicitly (as shown in Item 6, Question 1), rather than by using a `delete` expression.

Why two template parameters? Isn't one sufficient to make a copy of the value object? Well, if `constr` had only one template parameter, you would need to state explicitly the type of that parameter when cc from an object of a different type:

```
// Example 5-2(b): A less functional construct, and why it's less functional.
//
template <class T1>
void construct(T1* p, const T1& value) {
 new (p) T1(value);
}

// Assume that both p1 and p2 point to raw memory.
//
void f(double* p1, Base* p2) {
 Base b;
 Derived d;

 construct(p1, 2.718);              // ok
 construct(p2, b);                  // ok

 construct(p1, 42);                 // error: is T1 double or int?
 construct<double>(p1, 42);         // ok

 construct(p2, d);                  // error: is T1 Base or Derived?
 construct<Base>(p2, d);            // ok
}
```

The reason the two cases noted as error are ambiguous is that the compiler doesn't know enough to dec the template parameter, and so the programmer is forced to nominate a template parameter explicitly. shouldn't we allow programmers to silently construct a `double` from an `int` value? Probably; the wors could happen is that we might lose some precision. Shouldn't we allow programmers to silently constr `Base` from a `Derived`? Possibly; if `Base` allows that, then slicing would occur but that can be a legitin choice of operation.

Assuming that we want to allow the programmer to be able to do such things without explicitly naming we need to use the originally presented version that has two independent template parameters.

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 because of the way the function is intended to be used, given that it turns things back into raw memory and all. Still, only a simple change is needed to let `FwdIter` be any iterator type, not just a pointer, so why not do it: Have `destroy(iter,iter)` call the destructor explicitly. In the two-parameter version of `destroy`, change:

```
destroy(first);
```

to:

```
destroy(&*first);
```

This will almost always work. Here we are dereferencing the iterator to get a direct reference to the contained object and then taking its address, which guarantees that we get the pointer that we want. In a little more detail: All standard-conforming iterators are required to supply an `operator*` that returns a true `T&`. This is one of the reasons why proxied containers are not supported by the C++ standard; for more information about this and related issues, see the discussion of the expression `&*t.begin()` in Item 39 of More Exceptional C++ [Sutter02]. (It is possible, if rare, to make `destroy(&*first);` fail to work: As Astute Reader Bill Wade pointed out, that formulation won't work if `T` overrides its `operator&` to return something besides the object's address, but that's pathological and I have never seen a defensible design that does so.)

What's the moral of the story? Beware subtle genericity drawbacks when implementing one generic function in terms of another. In this case, there was a subtle principal drawback: The two-parameter version wasn't as generic for iterators as we originally thought. There was also an even subtler secondary drawback: The quick fix of changing `destroy(first);` to `destroy(&*first);` introduced a new requirement on `T`, namely that it should provide an `operator&` with normal semantics—one that really returns a pointer to the object. Both traps were neatly avoided when we stopped implementing one function in terms of the other.

Note: I am not discouraging you from implementing templates with templates; I'm encouraging you to

be aware of the potential interactions. Clearly templates are often correctly implemented in terms of other templates. For example, programmers are commonly expected to specialize the `std::swap` template for their own types if they know a more efficient way of swapping two values than copying them around, and if you write a `sort` template then, `sort` should be implemented by calling `swap`; otherwise `sort` would never be able to pick up an optimized `swap` for the element type.

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.

```
// Example 6-2(a): swap.
//
template <class T>
void swap(T& a, T& b) {
 T temp(a); a = b; b = temp;
}
```

`swap` just exchanges two values by using the copy constructor and copy assignment operator. It therefore requires that `T` have a copy constructor and a copy assignment operator.

If that's all you said, give yourself half marks only. One of the important things to note about the semantics of any function is its exception-safety status, including what guarantees it provides. In this case, `swap` is not at all exception-safe if `T`'s copy assignment operator can throw an exception. In particular, if `T::operator=` can throw but is atomic (all-or-nothing), then if the second assignment fails, we exit via an exception but `a` has already been modified; if additionally `T::operator=` can throw but is not atomic, then it is possible for `swap` to exit via an exception, but both parameters might have been modified and one might now contain neither of the two values. Therefore this `swap` must be documented as follows:

- If `T::operator=` doesn't throw, `swap` gives the guarantee that the operation is all-or-nothing except for side effects of `T` operations (see also [Sutter99]).
- Otherwise, if `T::operator=` can throw:
    - If `T::operator=` is atomic, and `swap` exits by means of an exception, the first argument might or might not already have been modified.
    - Otherwise, if `T::operator=` isn't atomic, and `swap` exits by means of an exception, both arguments might or might not already have been modified, and one of them might contain neither of the original two values.

There are two ways to remove the requirement that `T` have an assignment operator, and the first additionally provides better exception safety.

1. Specialize or overload `swap`. Say that we have a class `MyClass` that uses the common idiom of providing a nonthrowing `Swap`. Then we can specialize standard functions for `MyClass` as

follows.

```
// Example 6-2(b): Specializing swap.
//
class MyClass {
public:
 void Swap(MyClass&) /* throw() */ ;
 // ...
};

namespace std {
 template<> swap<MyClass>(MyClass& a, MyClass& b) { // throw()
  a.Swap(b);
 }
}
```

Alternatively, we can overload standard functions for `MyClass` as follows:

```
// Example 6-2(c): Overloading swap.
//
class MyClass {
public:
 void Swap(MyClass&) /* throw() */ ;
 // ...
};

// NOTE: Not in namespace std.
swap(MyClass& a, MyClass& b) /* throw() */ {
 a.Swap(b);
}
```

Doing this is usually a good idea—even if `T` does have an assignment operator that would allow the original version to work!

For example, the standard library itself overloads[10] `swap` for `vector` so that calling `swap` actually invokes `vector::swap`. This makes a lot of sense, because `vector::swap` can be much more efficient by avoiding making any copies of the `vector`s' data at all. The primary template in Example 6-2(a) would create a complete new copy (`temp`) of one of the `vector`s, and then perform additional copying from one `vector` to the other, then perform additional copying from `temp` to the other `vector`, which results in a lot of `T` operations and has complexity O(N) where N is the combined size of the `vector`s being swapped. The specialized version will typically simply just assign a few pointers and integral types, and it runs in constant (and usually negligible) time. Zap, you're done.

[10] Not "specializes," because you can't partially specialize function templates. See Item 7 for more about function templates and specialization.

So, if you create a type that provides a `swap`-like operation, it's usually a good idea to specialize

`std::swap` (or provide your own overloaded `swap` in another name space) that's specific to your new type. It will usually be more efficient than a routine application of the primary `std::swap` template's brute-force procedure and will often improve `swap`'s own exception safety.

Guideline

Consider specializing `std::swap` for your own types when objects of your type have a way to exchange their values more efficiently than via brute-force assignment.

2. Destroy-and-reconstruct. The idea here is to write `swap` in terms of `T`'s copy constructor instead of its copy assignment operator, and of course this works only if `T` indeed has a copy constructor:

```
// Example 6-2(d): swap without assignment.
//
template <class T>
void swap(T& a, T& b) {
 if(&a != &b) {                      // note: this check is now necessary!
   T temp(a);

   destroy(&a);
   construct(&a, b);

   destroy(&b);
   construct(&b, temp);
 }
}
```

First, this is never appropriate if `T` copy assignment can throw, because then you get all the exception safety problems of the original version, only in spades. You can get into situations where the objects not only hold indeterminate values but no longer exist at all!

If we know that `T` copy assignment is guaranteed not to throw, though, this version does have the extra ability to deal with types that can't be assigned but can be copy constructed, and there are indeed many such types. Being able to swap such types is not necessarily a good thing, because if a type can't be assigned, it's probably set up that way for a good reason—for example, it likely doesn't have value semantics, and it might have `const` or reference members—and so providing a mechanism to imbue (or impose) value semantics might be misguided and produce surprising and incorrect results.

Worse still, this approach plays games with object lifetimes, and that's always questionable. Here by "plays games" I mean that it changes not only the values, but the very existence, of the operated-upon objects. Code using the Example 6-2(d) form of `swap` could easily produce surprising results when users forget about the unusual destruction semantics.

A guideline: If you must play games with object lifetimes and you know that doing so is okay, and you're certain that the operated-upon objects' copy constructors can never throw, and you're very sure that the unusually "imposed" value semantics will be all right in your application for those specific objects, then (and only then) you might legitimately decide to use such an approach in those specific situations only. But even then, don't make such an operation a general template that could be accidentally instantiated for any type, and be very sure to document the living day lights out of it so that the poor unsuspecting programmer next door knows what to expect, because this technique falls squarely into the "Unusual Beasties" section of the C++ coding catalog.

Previous Next

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 t same name do overload, so function templates are allowed to overload too. This is pretty natural. Wha 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 specializat
// 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)
```

Guideline

Remember that function templates can't be partially specialized; they overload instead. Writing what looks like a function template partial specialization is really writing a distinct primary function template.

Finally, let's focus on function templates only and consider the overloading rules to see which ones get called in different situations. The rules are pretty simple, at least at a high level, and can be expressed a classic two-class system:

- Nontemplate functions are first-class citizens. A plain old nontemplate function that matches the parameter types as well as any function template will be selected over an otherwise-just-as-good function template.
- If there are no first-class citizens to choose from that are at least as good, then primary function templates as the second-class citizens get consulted next. Which primary function template gets selected depends on which matches best and is the "most specialized" according to the following of fairly arcane rules. (Important note: This use of "specialized" oddly enough has nothing to do with template specializations; it's just an unfortunate colloquialism.)
    - If it's clear that there's one "most specialized" primary function template, that one gets used. that primary template happens to be specialized for the types being used, the specialization will get used, otherwise the primary template instantiated with the correct types will be used
    - Else if there's a tie for the "most specialized" primary function template, the call is ambiguo because the compiler can't decide which is a better match. The programmer will have to do something to qualify the call and say which one is wanted.
    - Else if there's no primary function template that can be made to match, the call is bad and the programmer will have to fix the code.

Putting these rules together, here's a sample of what we get:

```
// Example 7-1, continued: Overload resolution
```

```
//
bool b;
int i;
double d;

f(b);         // calls (b) with T = bool
f(i, 42, d); // calls (c) with T = int
f(&i);        // calls (d) with T = int
f(i);         // calls (e)
f(d);         // calls (f)
```

So far I've deliberately chosen simpler cases, because here's where we step off into the deep end of th pool.

Why Not Specialize: The Dimov/Abrahams Example

Consider the following code:

```
// Example 7-2(a): Explicit specialization
//
template<class T>   // (a) a primary template
void f(T);

template<class T>   // (b) a primary template, overloads (a)function templates
void f(T*);         // can't be partially specialized; they overload instead

template<>          // (c) explicit specialization of (b)
void f<int>(int*);

// …

int *p;
f(p);               // calls (c)
```

The result for the last line in Example 7-2(a) is just what you'd expect. The question of the day, howev is why you expected it. If you expected it for the wrong reason, you will be very surprised by what comes next. After all, "So what," someone might say, "I wrote a specialization for a pointer to `int`, so obviously that's what should be called"and that's exactly the wrong reason.

Consider now Question 2's code, put in this form by Peter Dimov and Dave Abrahams:

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

```
// Example 7-2(b): The Dimov/Abrahams Example
//
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?
```

The answer is… the third `f`. Here's the code again, this time annotated similarly to Example 7-2(a) to compare and contrast the two examples:

```
template<class T> // (a) same old primary template as before
void f(T);

template<>        // (c) explicit specialization, this time of (a)
void f<int*>(int*);

template<class T> // (b) a second primary template, overloads (a)
void f(T*);

// …

int *p;
f(p);             // calls (b)! overload resolution ignores specializations
                  // and operates on the base function templates only
```

If this surprises you, you're not alone; it has surprised a lot of experts in its time. The key to understanding this is simple, and here it is: Specializations don't overload.

Only the primary templates overload (along with nontemplate functions, of course). Consider again the salient part from the summary I gave earlier of the overload resolution rules, this time with specific words highlighted:

- …
- If there are no first-class citizens to choose from that are at least as good, then primary function templates as the second-class citizens get consulted next. Which primary function template gets selected depends on which matches best and is the "most specialized" […] according to a set of fairly arcane rules:
    - If it's clear that there's one "most specialized" primary function template, that one gets used. that primary template happens to be specialized for the types being used, the specialization will get used, otherwise the primary template instantiated with the correct types will be used
    - … etc.

Overload resolution selects only a primary template (or a nontemplate function, if one is available). O
after it's been decided which primary template is going to be selected and that choice is locked in will
the compiler look around to see if there happens to be a suitable specialization of that template availab
and if so that specialization will get used.

Guideline

Remember that function template specializations don't participate in overload resolution. A specialization will be used only when its primary template is chosen, and the choice of primary template isn't affected by whether it happens to have specializations or not.

Important Morals

If you're like me, the first time you see this you'll ask the question: "Hmm. But it seems to me that I specifically wrote a specialization for the case when the parameter is an `int*`, and it is an `int*` that is an exact match, so shouldn't my specialization always get used?" That, alas, is a mistake: If you want t be sure it will always be used in the case of an exact match, that's what a plain old function is for—so just make it a function instead of a specialization.

The rationale for why specializations don't participate in overloading is simple, once explained, becau the surprise factor is exactly the reverse: The standards committee felt it would be surprising that, just because you happened to write a specialization for a particular template, it would in any way change which template gets used. Under that rationale, and because we already have a way of making sure our version gets used if that's what we want (we just make it a function, not a specialization), we can understand more clearly why specializations don't affect which template gets selected.

Guidelines

Moral #1: If you want to customize a primary function template and want that customization to participate in overload resolution (or to always be used in the case of exact match), don't make it a specialization—make it a plain old function.

Corollary: If you do provide overloads of a function template, avoid also providing specializations.

But what if you're the one who's writing, not just using, a function template? Can you do better and avo this (and other) problem(s) up front, for yourself and for your users? Indeed you can:

```
// Example 7-2(c): Illustrating Moral #2
```

```
//
template<class T>
struct FImpl;

template<class T>
void f(T t) { FImpl<T>::f(t); }      // clients, don't touch this!

template<class T>
struct FImpl {
 static void f(T t);                // clients, go ahead and specialize this
};
```

Guideline

Moral #2: If you're writing a primary function template that is likely to need specialization, prefer to write it as a single function template that should never be specialized or overloaded, and then implement the function template entirely as a simple handoff to a class template containing a static function with the same signature. Everyone can specialize that—both fully and partially, and without affecting the results of overload resolution.

Summary

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

It's a lot less intuitive to specialize function templates. For one thing, you can't partially specialize them—you overload them instead. For another thing, function template specializations don't overload. This means that any specializations you write will not affect which template gets used, which runs counter to what most people would intuitively expect. After all, if you had written a nontemplate functi with the identical signature instead of a function template specialization, the nontemplate function wou always be selected because it's always considered to be a better match than a template.

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

If you're using someone else's plain old function template (one that's not implemented in terms of a cla template), and you want to write your own special-case version that should participate in overloading don't make it a specialization; just make it an overloaded function with the same signature.

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.

Why It's Legal But Dark

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

When declaring friends, there are four options (enumerated in the [C++03] §14.5.3). They boil down to this:

When you declare a friend without saying the keyword `template` anywhere:

1. If the name of the friend looks like the name of a template specialization with explicit template arguments (e.g., `Name<SomeType>`)

   Then the friend is the indicated specialization of that template.

2. Else if the name of the friend is qualified with a class or namespace name (e.g., `Some::Name`) AND that class or namespace contains a matching non-template function

   Then the friend is that function.

3. Else if the name of the friend is qualified with a class or namespace name (e.g., `Some::Name`) AND that class or namespace contains a matching function template (deducing appropriate template parameters)

   Then the friend is that function template specialization.

4. Else the name must be unqualified and declare (or redeclare) an ordinary (nontemplate) function.

Clearly #2 and #4 match only nontemplates, so to declare the template specialization as a friend we have two choices: Write something that puts us into bucket #1, or write something that puts us into bucket #3. In our example, the options are

```
// The original code, legal because it falls into bucket #3
friend void boost::checked_delete(Test* x);
```

or

```
// Adding "<Test>", still legal because it falls into bucket #1
friend void boost::checked_delete<Test>(Test* x);
```

The first is shorthand for the second… but only if the name is qualified (here by `boost::`) and there's no matching nontemplate function in the same indicated scope. Even though both are legal, the first makes use of a dark corner of the friend declaration rules that is sufficiently surprising to people—and to most current compilers!—that I will propose no fewer than three reasons to avoid using it, even though it's technically legal.

Issue 1: It Doesn't Always Work

As already noted, the Bucket #3 syntax is a shorthand for explicitly naming the template arguments in angle brackets, but the shorthand works only if the name is qualified and the indicated class or namespace does not also contain a matching nontemplate function.

In particular, if the namespace has (or later gets) a matching nontemplate function, that would get chosen instead, because the presence of a nontemplate function means bucket #2 preempts #3. Kind of subtle and surprising, isn't it? Kind of easy to mistake, isn't it? Let's avoid such subtleties.

Issue 2: It's Surprising to People

Bucket #3 is edgy and fragile and surprising to programmers who look at the code and try to figure out what it does. For example, consider this very slight variant—all that I've changed is to remove the qualification `boost::`.

```
// Variant: Make the name unqualified, and it means something very different
//
class Test {
 ~Test() { }
 friend void checked_delete(Test* x);
};
```

If you omit `boost::` (i.e., if the call is unqualified), you fall into a completely different bucket, namely #4, which cannot match a function template at all, ever, not even with pretty please. I'll bet you dollars to donuts that just about everyone on our beautiful planet will agree with me that it's Pretty Surprising that just omitting a namespace name changes the meaning of the friend declaration so drastically. Let's avoid such edgy constructs.

Issue 3: It's Surprising to Compilers

Bucket #3 is edgy and fragile and surprising to compilers, and that can make it unusable in practice even if we disregard the other shortcomings mentioned earlier.

Let's try the two options, bucket #1 and bucket #3, on a wide range of current compilers and see what they think. Will the compilers understand the standard as well as we do (having read this Item so far)? Will at least all the strongest compilers do what we expect? No and no, respectively.

Let's try bucket #3 first:

```
// Example 8-1 again
//
namespace boost {
 template<typename T> void checked_delete(T* x) {

  // … other stuff …

  delete x;
 }
}

class Test {
 ~Test() { }
 friend void boost::checked_delete(Test* x);            // the original code
};

int main() {
 boost::checked_delete(new Test);
}
```

Try this on your own compiler, and then compare with our results. If you've ever watched the game show "Family Feud" on television, you can now imagine Richard Dawson's voice saying: "Survey saaaaays…" (see Table 8-1).

Table 8-1. The results of compiling Example 8-1 on various compilers

Compiler	Result	Error message
Borland 5.5	OK	
Comeau 4.3.0.1	OK	
Digital Mars 8.38	Error	`Symbol Undefined ? checked_delete@@YAXPAV-Test@@@Z (void cdecl checked_delete(Test *))`
EDG 3.0.1	OK	
Intel 6.0.1	OK	
gcc 2.95.3	Error	`` `boost::checked_delete(Test *)' should have beendeclared inside `boost' ``
gcc 3.4	Error	`` `void boost::checked_delete(Test*)' should have been declared inside `boost' ``
Metrowerks 8.2	Error	`friend void boost::checked_delete(Test* x); name has not been declared in`

		namespace/class
MS VC++ 6.0	Error	nonexistent function 'boost::checked_delete' specified as friend
MS VC++ 7.0 (2002)	OK	
MS VC++ 7.1 (2003)	Error	'boost::checked_delete' : not a function
MS VC++ 8.0 (2005) beta	OK	

In this case, the survey says that this syntax is not well recognized on actual compilers; one implementation changed its mind several times across versions. By the way, it shouldn't surprise us that Comeau, EDG, and Intel all agree, because they're all based on the EDG C++ language implementation; of the five distinct C++ language implementations tested here, three don't accept this version (Digial Mars, gcc, Metrowerks), two do (Borland, EDG), and one varies across versions (Microsoft).

Let's try writing it the other standards-conforming way, for bucket #1:

```
// Example 8-2: The other way to declare friendship
//
namespace boost {
 template<typename T> void checked_delete(T* x) {
  // … other stuff …
  delete x;
 }
}



class Test {
 ~Test() { }
 friend void boost::checked_delete<>(Test* x);       // the alternative
};

int main() {
 boost::checked_delete(new Test);
}
```

Or, equivalently, we could have spelled out:

```
friend void boost::checked_delete<Test>(Test* x);    // equivalent
```

Either way, when we twist our compilers' tails, our survey says that this is noticeably better supported (see Table 8-2.).

Table 8-2. The results of compiling Example 8-2 on various compilers

Compiler	Result	Error message
Borland 5.5	OK	
Comeau 4.3.0.1	OK	
Digital Mars 8.38	OK	
EDG 3.0.1	OK	
Intel 6.0.1	OK	
gcc 2.95.3	Error	`` `boost::checked_delete(Test *)' should have been declared inside `boost' ``
gcc 3.1.1	Error	`` `void boost::checked_delete(Test*)' should have been declared inside `boost' ``
gcc 3.4	Error	`` `void boost::checked_delete(Test*)' should have been declared inside `boost' ``
Metrowerks 8.2	OK	
MS VC++ 6.0	Error	`nonexistent function 'boost::checked_delete' specified as friend`
MS VC++ 7.0 (2002)	OK	
MS VC++ 7.1 (2003)	OK	
MS VC++ 8.0 (2005) beta	OK	

Bucket #1 sure feels safer—Example 8-2 works on every current compiler except gcc, and every older compiler except MS VC++ 6.0.

Aside: It's the Namespace That's Confusing Them

Note that if the function template we're trying to befriend weren't in a different namespace, we could use bucket #1 correctly today on nearly all these compilers:

```
// Example 8-3: If only checked_delete weren't in a namespace…
//
template<typename T> void checked_delete(T* x) { // no longer in boost::
 // … other stuff …
 delete x;
}

class Test {
 friend void checked_delete<Test>(Test* x);      // no longer need "boost:"
};

int main() {
 checked_delete(new Test);
}
```

Survey says (see Table 8-3).

Table 8-3. The results of compiling Example 8-3 on various compilers

Compiler	Result	Error message
Borland 5.5	OK	
Comeau 4.3.0.1	OK	
Digital Mars 8.38	OK	
EDG 3.0.1	OK	
Intel 6.0.1	OK	
gcc 2.95.3	OK	
gcc 3.4	OK	
Metrowerks 8.2	OK	
MS VC++ 6.0	Error	syntax error (just can't handle it)
MS VC++ 7.0 (2002)	Error	friend declaration incorrectly interpreted as declaring a brand-new (and undefined) ordinary nontemplate function, even though we used template syntax
MS VC++ 7.1 (2003)	OK	
MS VC++ 8.0 (2005) beta	OK	

So the problem on most compilers that can't handle Example 8-1 is specifically declaring friendship

for a function template specialization in another namespace. (Whew.Say that three times fast.)

Two Non-Workarounds

When this question arose on Usenet, some responses suggested writing a `using`-declaration (or equivalently a `using`-directive) and making the friend declaration unqualified:

```
namespace boost {
 template<typename T> void checked_delete(T* x) {
  // … other stuff …
  delete x;
 }
}

using boost::checked_delete;
// or "using namespace boost;"

class Test {
~Test() { }

friend void checked_delete(Test* x);    // not the template specialization!

};
```

This friend declaration falls into bucket #4: "4. Else the name must be unqualified and declare (or redeclare) an ordinary (nontemplate) function." This is actually declaring a brand-new, ordinary nontemplate function at the enclosing namespace scope called `::checked_delete(Test *)`.

If you try this code, many of these compilers will reject it saying that `checked_delete` hasn't been defined, and all of them will reject it if you actually try to make use of the friendship and put a private member access call into the `boost::checked_delete` template.

Finally, one expert suggested changing it slightly—using the `using` but also using the template syntax `<>`:

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

  // … other stuff …

  delete x;
 }
}

using boost::checked_delete;
// or "using namespace boost;"

class Test {
 ~Test() { }
```

```
friend void checked_delete<>(Test* x);   // legal?

};
```

This code is probably not legal C++: The Standard is not clear that this is legal, there's an open issue in the standards committee to decide whether or not this ought to be legal, there is sentiment that it should not be legal, and in the real world virtually all current compilers that I tried reject it.

Why do people feel that it should not be legal? For consistency, because `using` exists to make it easier to use names to call functions and to use type names in variable or parameter declarations. Declarations are different: Just as you must declare a template specialization in the template's original namespace (you can't do it in another namespace "through a `using`"), so you should be able to declare a template specialization as a friend only by naming the template's original namespace (not "through a `using`").

Summary

To befriend a function template specialization, you can choose one of two syntaxes:

```
// From Example 8-1
friend void boost::checked_delete (Test* x);

// From Example 8-2: add <> or <Test>
friend void boost::checked_delete<>(Test* x);            // or "<Test>"
```

This Item has demonstrated a pretty high portability price to pay in practice for not just writing `<>` or `<Test>` as in Example 8-2.

Guidelines

Say what you mean. Be explicit. If you're talking about a template and there's any question about what you mean, include a (possibly empty) template argument list.

Avoid the dark corners of the language, including constructs that might be arguably legal but that are liable to confuse programmers, or even compilers.

When you befriend a function template specialization, always explicitly add at least the `<>` template syntax. For example:

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

```
}

class Test {
 friend void boost::checked_delete (Test* x);              // bad
 friend void boost::checked_delete <> (Test* x);           // good
};
```

If your compiler doesn't yet allow either of these legal alternatives for the friend declaration, however, you'll have to make the necessary function(s) `public`[12]  but add a comment saying why, and make a note to change it back to `private` as soon as you upgrade your compiler.

[12] There are other workarounds, but they're all much more cumbersome. For example, you could create a proxy class inside namespace `boost` and befriend that.

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 te 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 mode
//
// --- 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 worl
  // uses some other helper classes and functions the functions aren't necessari
  // declared "inline", but the body's code is all here in the same file just the
 }
}
```

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 `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 other types not already mentioned in their respective declarations, then all their respective callers will need to see those types' full definitions too.

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 {
 export template<typename T>
 void g(T&);              // MYOB
}
// --- file g.cpp, ??shipped to user?? ---
namespace MyLib {
 template<typename T>
 void g(T&) {
  // avant-garde, truly stellar implementation, the product of many years of worl
  // uses some other helper classes and functions and is now "separately" compil
 }
}
```

Highly surprisingly to many people, this does not solve both problems in the case of `g`. It might have ameliorated one of them, depending. Let's consider the issues in turn.

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 `#include`d 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.

Previous Next

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`.

If `export` doesn't appear to deliver the advantages that many people expect, why does it exist? The reason is quite simple: In the mid-1990s, a majority of the committee believed that shipping a standard that did not have separate compilation for templates, as C already did for functions, would be incomplete and embarrassing. In short, `export` was retained in the then-draft standard on principle.

Principle is very often a good thing. It should never be disparaged, especially by armchair quarterbacks like us, looking back with the benefit of many years' worth of hindsight. (That "like ourselves" part includes me. Although now, years later, I chair the ISO committee, I didn't start personally attending committee meetings till the following year.)

Remember that, in 1995 and 1996, templates themselves were still pretty new:

- The first presentation of the initial C++ template design was made by Bjarne Stroustrup in October 1988 [Stroustrup88].

- In 1990, Margaret Ellis and Bjarne Stroustrup published The Annotated C++ Reference Manual (the ARM) [Ellis90]. The same year, the ISO/ANSI C++ standards committee got going and selected the ARM as its "starting point" base document. The ARM was the first C++ reference to include a description of templates, and they weren't templates as we know them today; the entire specification and description of these simple templates was only ten pages long.

  At that time, the focus was entirely on enabling parameterized types and functions, the given examples being a `List` container that could hold different types of objects and a `sort` that could sort different types of sequences. Even in these early days, however, templates were conceived with the desire for a separate compilation model in mind. Cfront (Stroustrup's C++ compiler) had support for a form of "separate" template compilation for these simple templates, although its approach was not scalable; see the note in the previous Item.

- During 1990–1996, C++ compiler vendors flourished and took different routes with their template implementations, and at the same time the standards committee greatly enhanced (and complexified) templates. A complete description of standard C++ templates alone now occupies some 133 pages of a 552-page tome that describes the feature and how to use it effectively—the lucidly written and recommended work C++ Templates [Vandevoorde03].

- In the early and mid-1990s, the committee was principally trying to make templates more robust and practical to support the intended basic uses. Few suspected the enormously flexible and

slightly monstrous wonder they had created—it was known that templates were a Turing-complete metalanguage allowing programs of arbitrary complexity to be written that could execute entirely at compile time, but all the modern template metaprogramming and advanced library design that's in vogue today was largely unanticipated by the people who gave us the very templates that make it possible in the first place, and the techniques were largely unknown during 1990—1996. Remember, it wasn't until late 1994 that Stepanov made his first presentation of the STL to the committee, which adopted it in 1995 as a groundbreaking achievement—and by today's standards the STL was "just" a container and algorithm library. Groundbreaking to be sure in 1995, and a powerful differentiator of C++ from other languages still today, but it was nonetheless just the first testing of the template waters by today's standards.

This is why I say that, "in 1995—1996, templates themselves were still pretty new." Modern templates in their (mostly) final form existed, but even the people who invented them didn't fully realize what they were capable of. The global C++ community was much smaller than it is today, few compilers supported more than ARM templates, and most compilers' template support of any kind was poor or essentially useless. Around that time, only one commercial compiler could cope with the initial STL, for example.

So it was that the community in general and the standards committee in particular still had a comparatively short record of real-world experience with even the simpler ARM templates that existed. The climate in 1996 was no longer quite embryonic, but it was young and still growing and forming.

And it was in this formative climate, with that limited experience, that the standards committee was forced to decide whether to keep `export`ed templates in the then-draft standard.

1996

In 1996, even with the little information that was available, enough was known that `export` made a lot of experts nervous. In particular, it made all the compiler vendors nervous. Even supporters of `export` viewed it as a necessary compromise while disliking `export` as a source of complexity; some would have preferred general separate compilation with no special keyword.

In 1996 there was a coordinated push within the committee to remove the notion of "separate" template compilation. (It was finally as a concession to this objection that the `export` keyword was soon thereafter invented to help out compilers by providing a means to at least tag which templates were supposed to be separately compiled.) In particular, it was argued, the separate template compilation model had never actually been implemented, and the committee had no idea whether it would work as intended. Several C++ vendors had implemented various forms of template source organization models, but `export` followed none of them; `export` was a completely new and experimental beast with no implementation experience behind it. Infact, there were papers presented at that time—papers that in retrospect could be called insightful bordering on prescient—that detailed some of the major potential shortcomings of the export model as described in the draft standard.

In particular, all the compiler implementers unanimously opposed including separate template

compilation in the standard on the grounds that it was too early to know if they were doing the right thing. They had serious unanswered concerns about the existing formulations (with or without an `export` keyword), and they didn't feel they had enough experience yet to come up with a fully baked alternative (not to mention insufficient time; the standard was being stabilized and would be set in stone the following year, 1997). For those reasons, the compiler vendors unanimously didn't want to rush a separation model into the first standard [C++98]. Rather, they wanted to take time to design it right and do it in the next standard. They favored the idea of separate template compilation in principle, but felt that `export` wasn't fully baked and they still didn't know enough to do it right.

They lost, narrowly, and `export` stayed in the standard.[18] It would be armchair-quarterbacking at best to be unduly critical about this outcome, however. As I summarized earlier, a (slim) majority of the committee believed that shipping a standard that did not have some form of "separate" compilation for templates, as C already did for functions, would be incomplete and embarrassing. Several compilers had already been experimenting with forms of "separate" template compilation and it seemed to be a good idea in principle. In short, `export` was retained in the then-draft standard on principle. And it's a good principle, not to be disparaged.

> [18] Things were quite turbulent and support seesawed back and forth, balanced on a fulcrum. At the March 1996 meeting, the straw vote was 2-to-1 against separate template compilation. At the July 1996 meeting where the `export` keyword was introduced, the vote was 2-to-1 in favor of export.

To emphasize, note that the world's compiler vendors opposed `export` in particular, and did not oppose the principle of separate template compilation. They just felt they needed more time to be confident that the standard would get it right. Although some of the world-class experts who in 1996 voted in favor of retaining `export` now see it as a mistake, the intent and motivation was good, and there is still hope that export will deliver some benefits—if not all the big ones that were initially hoped for—as we gain experience with the first shipping compiler to implement `export` (Comeau 4.3.01, released in 2002).

Our Export Experience to Date

The world's only implementers of `export`, EDG, report that in their experience `export` is by far the most difficult C++ feature to implement, as much work as any three other major C++ language features they've done (such as namespaces or member templates). The `export` feature alone took more than three person-years to code and test, not counting design work; by comparison, implementing the entire Java language took the same three people only two person-years.

Why is `export` so difficult to implement, and so complex? Here are two major reasons:

1. Export relies on Koenig lookup. Most compilers still get Koenig lookup wrong even within a single translation unit (informally, this means a source file). Export requires performing Koenig lookup across translation units. (For more about Koenig lookup, see [Sutter00, Item 27].)

2. Conceptually, export requires a compiler to deal simultaneously with many symbol tables.

Instantiating an exported template can trigger cascaded instantiations in other translation units. Each must be able to refer to entities that existed (or "sort of existed") when the template definition was parsed. In C++, dealing with one symbol table is complicated enough. With `export`, at least conceptually you need to simultaneously deal with an arbitrary number of symbol tables.

Ch-ch-ch-changes: Export's Shadow Falls on Existing Language Features

3. In what ways does `export` change the meaning of other C++ language features? Briefly explain the interactions.

`export` has a few surprising effects on existing language features. Many of these real effects of `export` are not mentioned or addressed in the standard. In particular, `export` "exports" more than its template:

- Some functions and objects in unnamed namespaces must now be accessible and callable across translation units if they are used in exported templates. Similarly, some file-`static` functions and objects must now have external linkage, or at least behave as though they did, if they are used in exported templates. This is counter to the intent of unnamed namespaces and namespace-scope `static`, which was to make those names strictly internal to their original translation unit. (File-`static` functions and objects are deprecated, and you should use the unnamed namespace instead, but they're still part of standard C++.)

- Overload resolution must also be able to resolve names from an arbitrary number of different translation units—including, amusingly, overloading names from an arbitrary number of unnamed namespaces. A major benefit of putting internal functions into the unnamed namespace (and the deprecated file `static`) was to "privatize" those functions so you could give them simple names without worrying about name conflicts and overloading effects across source files. Now, because part of the protection is being removed and they can and do participate in overload resolution with each other via exported templates, it's (alas) a good idea to obfuscate their names again if you use such functions or objects in an exported template, even if the function is in an unnamed namespace or file `static`, so as to avoid silent changes of meaning.

- There are new ambiguities and potential One Definition Rule (ODR) violations. For example, a class might have multiple befriending entities in different translation units, and declarations of that class from those different translation units might all be participating in an instantiation. If so, which set of access rules should be applied? These issues might seem minor and many of the errors might be innocuous, but on some popular platforms, ODR violations are increasingly important (see [Sutter02c] for one example).

Export Can Be Difficult to Use Correctly

4. How does `export` affect the programmer?

`export` will probably be somewhat more difficult to use correctly than normal templates. Here are

three examples to illustrate why this is so.

Example 1: It is easier than before for programmers to write programs that have hard-to-predict meaning. Like an inclusion-model template, an exported template commonly has different paths by which it could be instantiated, and each path commonly has a different context. For those who might say, "But we already have that problem with functions defined in header files (e.g., `inline`)," note that this template problem is already greater than that because there are more opportunities for names to change meaning, in particular because templates use a wider set of names than closed functions do. Templates use dependent names, names that are dependent on (and therefore vary with) the template arguments, and so for each instantiation of the template with the very same template arguments the template's user must be careful to provide exactly the same context (e.g., overloaded functions that operate on that template argument type) to prevent the instantiation from inadvertently having a different meaning in different files, which would be a classic ODR violation. Why is this expected to be somewhat worse under the `export` model than for inclusion-model templates? The big thing about `export` is the fact that, in addition to the usual complexities, there are names from multiple translation units available to name lookup, which is not true in any other context in standard C++.

Example 2: It is harder for the compiler to generate high-quality diagnostics to aid programmers. Template error messages are already notoriously hard to understand because of long and verbose names, but besides that, what's less obvious to programmers is that it's already harder for compiler writers to give good error messages for templates because templates can generate multiple and cascading instantiations. With `export`, there is now the additional dimension of multiple translation units—a message such as "error on line X, caused by the instantiation of this function, caused by the instantiation of this function, caused by the instantiation of this function, …" must now add which translation unit it was in when it happened, and each line in the traceback could be from a different translation unit. Detecting ODR violations for exported templates is a challenging problem in itself, but detecting what was really meant so as to provide "did you mean" guidance is even harder. Many of us would be happy just to have our compiler emit readable error messages for plain old templates.

Example 3: Export puts new constraints on the build environment. The build environment does not consist of just `.cpp` and `.h` files any more, and many of today's tools don't understand how to handle apparently-circular dependencies when the linker can go back and change `.obj` (or `.o`) files. As noted in the previous Item, if you change an exported template file, you need to recompile that, but you also need to recompile the instantiations; that is, `export` really does not separate dependencies, it just hides them.

As the world's top template guru, John Spicer of EDG, notes: "`export` is intricate in nature and it takes a lot of work to understand the consequences. It's hard to make up simple usage guidelines that will keep users out of trouble." [Emphasis mine.]

Potential Benefits of Export

5. What real and potential benefits does `export` have?

Now that an implementation of `export` is finally available, for the first time ever, the time is ripe for

the early adopters in the C++ community to start kicking the tires and see how it runs in the field. Here are two actual and potential values of `export` that some early adopters hope to achieve:

1. Build speed (still). It is still unknown what, if any, impact `export` will have on build speed in common real-world template-using code. If the feature becomes more widely adopted and used, exploration in this area will let us discover how common the beneficial cases are and how easy or difficult those cases are to construct. In particular, it is hoped that translation units that use exported templates will be less sensitive to (i.e., less costly to rebuild when there are) changes in the template's definition.

   Caveats to #1: For reasons why being able to break dependencies might not be the case and why dependencies still exist, see the previous Item. Also, note that in the EDG implementation of templates, this potential advantage is available equally to both inclusion and `export` source organization models—which would mean that for this implementation at least (the only available `export` implementation today) `export` could have no benefit in this respect over inclusion-model templates.

2. Macro leakage. This is a real advantage of `export`. Macros leak across traditional inclusion-model header files. Because the inclusion-model source code is entirely available in each translation unit, outside macros pulled in from elsewhere earlier in that translation unit can affect the template's definition. With `export`, macros don't leak across translation units, and this will help the template author maintain better control over his template definitions (which are off in a separate file) and prevent outside macros from as easily interfering with his template definitions' internals.

This is a real advantage of `export`, but it is not unique to `export`. Note that within the C++ standards committee's early work on the next standard (C++0x), the Evolution Working Group is already pursuing better and more general solutions to the macro problem in all contexts, such as Stroustrup's work on potential new `#scope` and `#endscope` preprocessor extensions. If such a solution is adopted, it would eliminate entirely this advantage of `export`, because the preprocessor scope control solution would deliver all the macro-protection benefits of `export` and many more, in a better and more general way.

In summary, it remains to be seen in the coming years how much benefit `export` gives over normal include-all-the-code-in-the-header templates, but I'd like to strongly encourage the people who run those tests to also report the results of organizing their code to take full advantage of the EDG implementation's non-`export` template optimization capabilities and see whether any advantages to `export` actually remain.

Morals

So should you use `export`, and if so, how can you use it safely? Well, for the time being, only a fraction of C++ programmers will be using an `export`-capable compiler that they can experiment with. For most C++ programmers, then, the question of whether to use `export` is moot: They can't, not anytime soon, so they won't.

What if you're using one of those up-and-coming newfangled `export`-capable compilers? Ah, now we can finally come up with an initial guideline:

Guideline

For portable code, don't use `export`.

This borders on being a truism: `export` certainly can't be used for portable code, because given today's meager compiler support, any code that uses `export` is not portable in practice today and will not be portable for some time to come.

What if you don't need portable code, have `export`, and are tempted to use it? Then caveat emptor: Be aware that `export` is still experimental, that it does not necessarily deliver the benefits people expect, and that it adds some new operational wrinkles to existing C++ language features. Be aware that exported templates can also be trickier to write for the reasons mentioned in these two Items and summarized again below.

My best advice as of this writing would be that, even if you just use one compiler and it has `export` today, in general you should try to avoid `export` for now in production code because it is still an experimental design. Let someone else be the guinea pig as we spend the next year or two trying it out and learning about what `export` will really give us.

Guideline

(For now) Avoid `export`.

But, if you do decide to be one of the early-adopter experimenters, here are somethings we already know you can do to make life safer and less stressful:

Guidelines

If you do choose to use `export` selectively for some templates, then:

Don't expect that `export` means you don't have to ship source code (or its equivalent) anyway. You still do, and this will not change.

Don't expect that `export` means your builds will be earth-shatteringly faster. Initial experience is inconclusive, but your builds could well be slower.

Do check that your tools and environment can handle the new build requirements and dependencies (e.g., make sure all your tools understand that the linker can change its input `.obj`/`.o` files, if that's the technique your `export` implementation uses).

If your exported template uses any functions or objects that are in an unnamed namespace or file `static`:

- Understand that those functions/objects will behave as though they were `extern`, and that the functions are liable to participate in overload resolution with an arbitrary number of functions in other unnamed namespaces from an arbitrary number of source files.

- Always obfuscate (uglify) the names of those functions so as to prevent unintended semantic changes. (This is a pity because the unnamed namespace and file `static` are supposed to protect you from this so you don't have to obfuscate the names, but if you use `export` you can too easily silently lose this protection and should obfuscate them again.)

Do understand that this is not a complete list and that you will probably encounter some other issues beyond the ones we already know about for today's normal template uses. As Spicer put it: "It's hard to make up simple guidelines that will keep users out of trouble." Do understand that `export` is still somewhat experimental and that as a community we haven't yet had a chance to learn how to use `export`, so we don't have a complete set of good safety and usage guidelines yet. This will likely change in the future.

It's too early too tell whether the "avoid `export`" guideline will turn into permanent advice. Time and experimentation will tell. As vendors slowly begin to adopt and support `export` in the coming years and the community gets a chance to finally try it out, we'll know much more about how and when to use itor not.

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. Where and when should I handle an exception? This is the only consideration that is in part about writing `try` and `catch` in the right places, and even this can be automated most of the time. First, consider the questions we need to answer:

   - What code could `catch`? That is, what code has enough context and knowledge to handle the error being reported by the exception (possibly by translating the exception into another form)? In particular, note that the catching code also needs to have enough knowledge to perform any necessary cleanup, such as of dynamic resources.

   - What code should `catch`? That is, of the code that could catch the exception, which is best suited to do so?

Once we've answered those questions, note that using the "resource acquisition is initialization" idiom can eliminate many try-blocks by automating the cleanup work. If you wrap dynamically allocated resources in owner-manager objects, typically the destructor can perform automatic cleanup at the right time without any `try` or `catch` at all. This is clearly desirable, not to mention that it's also usually easier to code now and to read later.

Guideline

Prefer handling exception cleanup automatically by using de-structors instead of `try/catch`.

3. In all other places, is my code going to be safe if an exception comes roaring through out of any given function call? This consideration is about using good resource management to avoid leaks, maintaining class and program invariants, and other kinds of program correctness. Put another way, it's about keeping the program from blowing up just because an exception happens to pass from its throw site through code that shouldn't have to particularly care about it before arriving at an appropriate handler. For most programmers I've encountered, it turns out that this is typically by far the most time-consuming and difficult-to-learn aspect of exception safety.

Notice that only one of these three considerations has anything to do with writing `try` and `catch`. And even that one can often be avoided with the judicious use of de-structors to automate cleanup.

4. When should `try` and `catch` be used? When should they not be used? Express the answer as a good coding standard guideline.

Here's one suggestion. In brief:

1. Determine an overall error reporting and handling policy for your application or subsystem, and stick to it. In particular, the policy should cover the following basic aspects (and generally includes much more):

   - Error reporting. Define what kinds of errors are to be reported and how; prefer using exceptions as opposed to other error reporting methods. Generally it's good to choose the most readable and maintainable method for each case by default; for example, exceptions are most useful for constructors and operators that cannot emit return values or where the throw site and the handler are widely separated.

   - Error propagation. Among other things, define the boundaries that exceptions shall not cross; typically these are module or API boundaries.

   - Error handling. Among other things, mandate that owning objects and destructors be used to manage cleanup instead of `try`/`catch`, wherever possible.

2. Write `tHRow` in the places that detect an error and cannot deal with it themselves. (Clearly, code that can resolve an error immediately doesn't need to report it!)

For every operation, document what exceptions the operation might throw, and why, as part of the documentation for every function and module. You don't need to actually write an exception specification on each function (and you shouldn't; see Item 13), but you do need to document clearly and rigorously what the caller can expect, because error semantics are part of the function's or module's interface.

3. Write `TRy` and `catch` in the places that have sufficient knowledge to handle the error, to translate it, or to enforce boundaries defined in the error policy. In particular, I've found that there are three main reasons to write `try` and `catch`:

   - To handle an error. This is the simple case: An error happened, we know what to do about it, and we do it. Life goes on (sans the original exception, which has been safely put to rest). Again, do this in a destructor if possible; if not, go ahead and use `TRy`/`catch`.

   - To translate an exception. This means catching one exception that reports a lower-level problem and throwing another that is couched in the context of the translating code's own higher-level semantics. Alternatively, the original exception can be translated to another representation, such as an error code.

     For example, consider a communications session utility class that works across many host types and transport protocols: An attempt to open a session to another host can fail for any number of low-level reasons that the session class can detect (for example, a failure to detect the network or authentication/permission rejection from the remote host). The `Open` function can handle these conditions itself, and there's no use reporting them to the caller, who after all has no idea what a `Foo` packet is or what to do if it `Bar`ifies; the session class

handles its internal low-level errors directly, keeps itself in a consistent state, and reports its own higher-level error or exception to inform its caller that the session could not be opened.

```
void Session::Open(/*…*/) {
 try {
  // entire operation
 }
 catch(const ip_error& err) {
  // - do something about an IP error
  // - clean up
  throw Session::OpenFailed();
 }
 catch(const KerberosAuthentFail& err) {
  // - do something about an authentication error
  // - clean up
  throw Session::OpenFailed();
 }

 // … etc. …
}
```

- To `catch(...)` on subsystem boundaries or other run-time firewalls. This usually also involves translating the error, usually to an error code or other nonexceptional representation. For example, when your stack unwinds up to a C API, you have only two choices: Return an error code right away for the current API function, or set an error state that the caller can query later via a complementary `GetLastError` API function.

Guidelines

Determine an overall error reporting and handling policy for your application or subsystem, and stick to it. Include a policy for error reporting, error propagation, and error handling.

Write `throw` in the places that detect an error and cannot deal with it themselves.

Write `try` and `catch` in the places that have sufficient knowledge to handle the error, to translate it, or to enforce boundaries defined in the error policy (e.g., to `catch(...)` on subsystem boundaries or other run-time firewalls).

Summary

A wise man once said:

> Lead, follow, or get the blazes out of the way!

In exception safety analysis, we might say instead:

> `tHRow`, `catch`, or get the blazes out of the way!

In practice, the last get-out-of-the-way case accounts for the bulk of exception safety analysis and testing. That's the major reason why exception-safe coding is not fundamentally about writing `TRy` and `catch` in the right places. Rather, it's fundamentally about getting out of the bullet's way in the right places.

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 objects such as `Lock` classes and `shared_ptr`s (see [Boost, Sutter02a]) is just a good idea in general. It should come as no surprise that among their many benefits we should also find exception safety. How many times have you seen a function (here we're talking about someone else's function, of course, not something you wrote) where one of the code branches that leads to an early `return` fails to do some cleanup because cleanup wasn't being managed automatically using RAII?

- Use "do all the work off to the side, then commit using nonthrowing operations only" to avoid changing internal state until you're sure the whole operation will succeed. Such transactional programming is clearer, cleaner, and safer even with error codes. How many times have you seen a function (and naturally here again we're talking about someone else's function, of course, not something you wrote) where one of the code branches that leads to an early return fails to preserve the object's state, because some fiddling with internal state had already happened before a later operation failed?

- Prefer "one class (or function), one responsibility." Functions that have multiple effects, such as the `Stack::Pop` and `EvaluateSalaryAndReturnName` functions described in Items 10 and 18 of Exceptional C++ [Sutter00], are difficult to make strongly exception-safe. Many exception safety problems can be made much simpler, or eliminated without conscious thought, simply by following the "one function, one responsibility" guideline. And that guideline long predates our knowledge that it happens to also apply to exception safety; it's just a good idea in and of itself.

Doing these things is just plain good for you.

Having said that, then, which guarantee should we use when? In brief, here's the guideline followed by the C++ standard library, and one that you can profitably apply to your own code:

Guideline

A function should always support the strictest guarantee that it can support without penalizing callers who don't need it.

So if your function can support the nofail guarantee without penalizing callers who don't need that

guarantee, it should do so. Note also that a handful of key functions simply must be nofail operations:

Guideline

Never allow a destructor, deallocation, or swap function to emit an exception, because otherwise it's often impossible to reliably and safely perform cleanup.

Otherwise, if your function can support the strong guarantee without penalizing some users, it should do so. Note that `vector::insert` is an example of a function that does not support the strong guarantee in general because doing so would force us to make a full copy of the `vector`'s contents every time we insert an element, and not all programs care so much about the strong guarantee that they're willing to incur that much overhead. (Those programs that do can wrap `vector::insert` with the strong guarantee themselves, trivially: Take a copy of the `vector`, perform the insert on the copy, and once it's successful, perform a `swap` with the original `vector`, and you're done.)

Otherwise, your function should support the basic guarantee.

For more information about these concepts, such as what a nonthrowing `swap` is all about or why destructors should never emit exceptions, see also Exceptional C++ [Sutter00] and More Exceptional C++ [Sutter02].

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

The idea behind exception specifications is easy to understand: In a C++ program, unless otherwise specified, any function might conceivably emit any type of exception. Consider a function named `Func` (because the name `f` is so dreadfully over-used):

2. For each of the following functions, describe what exceptions the function could throw.

```
// Example 13-2(a)
//
int Func();                          // can throw anything
```

By default, in C++, `Func` could indeed throw anything, just as the comment added hereto says. Now, often we know just what kinds of things a function might throw, and then it's certainly reasonable to want to supply the compiler and the human programmer with some information limiting what exceptions could come tearing out of a function. For example:

```
// Example 13-2(b)
//
int Gunc() throw();                  // will throw nothing

int Hunc() throw(A,B);               // can only throw A or B
```

In these cases, the function's exception specification exists to say something about what the functions `Gunc` and `Hunc` could emit. The comments document colloquially what the specifications say. We'll return to that "colloquially" part in a moment, because as it turns out, these two comments are deceptively close to being correct.

One might naturally think that making a statement about what the functions might throw would be a good thing, that more information is better. One would not necessarily be right, because the devil is in the details: Although the motivation is noble, the way exception specifications are, well, specified in C++ isn't always useful and can often be downright detrimental.

Issue the First: A "Shadow Type System"

3. Is an exception specification part of the function's type? Explain.

John Spicer, of Edison Design Group fame and an author of large swathes of the template chapter of the C++ standard, has been known to call C++'s exception specifications a "shadow type system." One of C++'s strongest features is its strong type system, and that's well and good. Why would we call exception specifications a shadow type system instead of just part of the type system?

The reason is simple, and twofold:

- Exception specifications don't participate in a function's type.
- Except when they do.

Consider first an example of when exception specifications don't participate in a function's type. Reflect on the following code:

```
// Example 13-3(a): You can't write an exception specification in a typedef.
//
void f() throw(A,B);

typedef void (*PF)() throw(A,B);     // syntax error

PF pf = f;                           // can't get here because of the error
```

The exception specification on the `typedef` is illegal. C++ doesn't let you write that, so the exception specification is not allowed to participate in the type of a function… at least, not in the context of a `typedef`, it's not. But in other cases, exception specifications do indeed participate in the function's type, such as if you wrote the same function declaration without the `typedef`:

```
// Example 13-3(b): But you can if you omit the typedef!
//
void f() throw(A,B);

void (*pf)() throw(A,B);             // ok

pf = f;                              // ok
```

Incidentally, you can do this kind of assignment of a pointer to a function as long as the target's exception specification is no more restrictive than the source's:

```
// Example 13-3(c): Also kosher, low-carb, and fat-free.
//
void f() throw(A,B);

void (*pf)() throw(A,B,C);            // ok
```

```
pf = f;                                 // ok, pf's type is less restrictive
```

Exception specifications also participate in a virtual function's type when you try to override it:

```
// Example 13-3(d): Exception specifications matter for virtual functions.
//
class C {
 virtual void f() throw(A,B);           // same exception specification
};
class D : C {
 void f();                              // error, now the ES matters
};
```

So the first issue with exception specifications as they exist in today's C++ is that they're really a shadow type system that plays by different rules than the rest of the type system.

Issue the Second: (Mis)understandings

The second issue has to do with knowing what you're getting. As many notable persons, including the authors of the Boost exception specification rationale [BoostES], have put it, programmers tend to use exception specifications as though they be-haved the way the programmer would like, instead of the way they actually do be-have.

Hence the question:

4. What are exception specifications, and what do they do? Be precise.

Here's what many people think exception specifications do:

- Guarantee that functions will throw only listed exceptions (possibly none).
- Enable compiler optimizations based on the knowledge that only listed exceptions (possibly none) will be thrown.

These expectations are, again, deceptively close to being correct. Consider again the code in Example 13-2(b):

```
// Example 13-2(b) reprise, and two potential white lies:
//
int Gunc() throw();             // will throw nothing     ←?

int Hunc() throw(A,B);          // can only throw A or B  ←?
```

Are the comments correct? Not quite. `Gunc` might indeed throw something, and `Hunc` might well throw something other than `A` or `B`! The compiler just guarantees to beat them senseless if they do… oh, and to beat your program senseless too, most of the time.

Because `Gunc` or `Hunc` could indeed throw something they promised not to, not only can't the compiler assume it won't happen, but the compiler is also responsible for being the policeman with the billy club who checks to make sure such a bad thing doesn't happen undetected. If it does happen, then the compiler must invoke the `unexpected` function. Most of the time, that will terminate your program. Why? Because there are only two ways out of `unexpected`, neither of which is a normal return. You can pick your poison:

- Throw instead an exception that the exception specification does allow. If so, the exception propagation continues as it would normally have. But remember that the `unexpected` handler is global—there is only one for the whole program. A global handler is highly unlikely to be smart enough to Do the Right Thing for any given particular case, and the result is to go to `terminate`, go directly to `terminate`, do not pass `catch`, do not collect $200.

- Throw instead (or rethrow) an exception that the exception specification (still) doesn't allow. If the original function allowed a `bad_exception` type in its exception specification, okay, then it's a `bad_exception` that will now get propagated. But if not, then go to `terminate`, go directly to `terminate`...

Because violated exception specifications end up terminating your program the vast majority of the time, I think it's legitimate to call that "beating your program senseless."

Earlier, we saw two bullets stating what many people think that exception specifications do. Here is an edited statement that more accurately portrays what they actually do do [sic]:[19]

[19] Yes, this is a sic joke.

- Guarantee Enforce at run-time that functions will throw only listed exceptions (possibly none).

- Enable or prevent compiler optimizations based on the knowledge that only listed exceptions (possibly none) will be thrown having to check whether listed exceptions are indeed being thrown.

To see what a compiler has to do, consider the following code, which provides a body for one of our sample functions, `Hunc`:

```
// Example 13-4(a)
//
int Hunc() throw(A,B) {
 return Junc();
}
```

Functionally, the compiler must generate code like the following, and it's typically just as costly at run-time as if you'd hand-written it yourself (though less typing because the compiler generates it for you):

```
// Example 13-4(b): A compiler's massaged version of Example 13-4(a)
//
int Hunc()
try {
 return Junc();
}
catch(A) {
 throw;
}
catch(B) {
 throw;
}
catch(…) {
 std::unexpected();  // won't return! but might throw an A or a B if you're lucky
}
```

Here we can see more clearly why, rather than letting the compiler make optimizations by assuming only certain exceptions will be thrown, it's exactly the reverse: The compiler has to do more work to enforce at run-time that only those exceptions are indeed thrown.

The Scoop on Exception Specifications

Most people are surprised to discover that exception specifications can cause performance penalties. One reason this is true has now been amply demonstrated: The exception specification incurs the overhead for the implicitly generated `TRy/catch` blocks, although this might be minor on efficient compilers.

There are at least two other ways that exception specifications can commonly cost you in run-time performance:

- Some compilers will automatically refuse to inline a function having an exception specification, just as they can apply other heuristics such as refusing to inline functions that have more than a certain number of nested statements or that contain any kind of loop construct.

- Some compilers don't optimize exception-related knowledge well at all and will add the compiler-generated `TRy/catch` blocks even when the function body provably can't throw. (I mean it; that's not a typo.)

Moving beyond run-time performance, exception specifications can cost you development time because they increase coupling. For example, removing a type from the base class virtual function's exception specification is a quick and easy way to break lots of derived classes in one swell foop (if you're looking for a way). Try it on a Friday afternoon check-in and start a pool to guess the number of angry emails that will be waiting for you in your inbox on Monday morning.

Hence our natural next question would be:

5. When is it worth it to write an exception specification on a function? Why would you choose to write one, or why not?

Here's what seems to be the best advice we as a community have learned as of this writing:

Guidelines

Moral #1: Never write an exception specification.

Moral #2: Except possibly an empty one, but if I were you, I'd avoid even that.

Boost's experience is that a throws-nothing specification on a non-inline function is the only place where an exception specification "may have some benefit with some compilers." That's a rather underwhelming statement in its own right but a useful consideration if you have to write portable code that will be used on more than one compiler platform.

It's actually even a bit worse than that in practice, because it turns out that popular implementations vary in how they actually handle exception specifications. At least one popular C++ compiler (Microsoft's, up to the current version as of this writing, 7.1 (2003)) parses exception specifications but does not actually enforce them, reducing the exception specifications to glorified comments. But wait, there's more: At the same time, there are legal optimizations a compiler can perform outside a function, and which the Microsoft 7.x compiler does perform, that rely on the exception specification enforcement's being done inside each function; the idea is that if the function did try to throw something it shouldn't, then the internal handler would stop the program and control would never return to the caller, so because control did return to the caller the code generated for the call site can assume nothing was thrown and do such things as eliminate external `try/catch` blocks.

On such a compiler that fails to enforce the exception specification but still relies on its being enforced, the meaning of `throw()` changes from the standard "check me on this, stop me if I inadvertently throw" to a "trust me on this, assume I'll never throw and optimize away." So beware: If you do choose to use even an empty exception specification, read your compiler's documentation and check to see what it will really do with it. You might just be surprised. Be aware, drive with care.

Summary

In brief, don't bother with exception specifications. Even experts don't bother.

Slightly less briefly, the major issues are:

- Exception specifications can cause surprising performance hits, for example if the compiler turns off inlining for functions with exception specifications.

- A run-time `unexpected` error is not always what you want to have happen for the kinds of mistakes that exception specifications are meant to catch.

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

While presenting this material as part of a broader talk at a conference not long ago, I asked how many of the about 100 people in the room each time had used exception specifications. About half put up their hands. Then a wag at the back said (quite correctly) that I should also ask how many of those people later took the exception specifications back out again, so I asked; about the same number of hands went up. This is telling. The world-class library designers at Boost went through the same experience, and that's why their coding policy on writing exception specifications pretty much boils down to "don't do that" [BoostES].

True, many well-intentioned people wanted exception specifications in the language, and that's why we have them. This reminds me of a cute poem that I first encountered about 15 years ago as it circulated in midwinter holiday emails. Set to the cadence of "'Twas the Night Before Christmas," these days it's variously titled "'Twas the Night Before Implementation" or "'Twas the Night Before Crisis." It tells of a master programmer who slaves away late at night in the holiday season to meet user deadlines and performs multiple miracles to pull out a functioning system that perfectly implements the requirements… only to experience a final metaphorical kick in the teeth as the last four lines of the ditty report:

> The system was finished, the tests were concluded,
>
> The users' last changes were even included.
>
> And the users exclaimed, with a snarl and a taunt,
>
> "It's just what we asked for, but not what we want!"

The thought resonates as we finish considering our current experience with exception specifications. The feature seemed like a good idea at the time… and it is just what some people had asked for.

Be careful what you wish for. You might get your wish.

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 { };

class X : public D1, public D2 {
 M1 m1_;
 M2 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

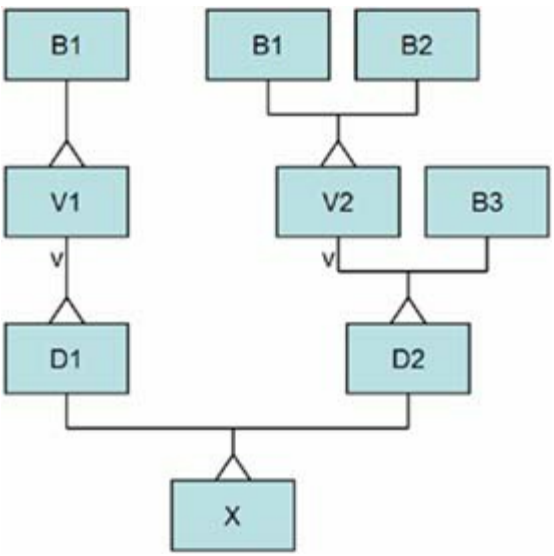
doesn't affect initialization order, so I'm showing all inheritance as public.

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 { };
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 { };

class X : public D1, public D2 {
 M1 m1_;
 M2 m2_;
};
```

The inheritance hierarchy is structured as shown in Figure 14-1.

Figure 14-1. Inheritance hierarchy for Example 14-2



The initialization order for an x object in Example 14-2 is as follows, where each constructor call shown represents the execution of the body of that constructor:

- First, construct the virtual bases:

construct `V1`: `B1::B1()` `V1::V1()`

construct `V2`: `B1::B1()` `B2::B2()` `V2::V2()`

- Next, construct the nonvirtual bases:

  construct `D1`: `D1::D1()`

  construct `D2`: `B3::B3()` `D2::D2()`

- Next, construct the members: `M1::M1()` `M2::M2()`
- Finally, construct `X` itself: `X::X()`

Summary: A(nother) Word About Inheritance

Of course, although the main point of this Item was to understand the order in which objects are constructed (and, in reverse order, destroyed), it doesn't hurt to repeat a tangentially related guideline:

Guideline

Avoid overusing inheritance.

Except for friendship, inheritance is the strongest relationship that can be expressed in C++ and should be used only when it's necessary. For more details, see also Item 20 in Exceptional C++ [Sutter00] on "Uses and Abuses of Inheritance" and Item 19 in More Exceptional C++ [Sutter02] on "Exception-Safe Class Design, Part 2: Inheritance."

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

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

1. Is this a hole in C++'s access control mechanism, and therefore a hole in C++'s encapsulation? Discuss.

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:
 friend ::Hijack(X&);
};

void Hijack(X& x) {
 x.private_ = 2;                        // evil laughter here
}
```

This man is a Forger. Mark him well, for he cannot be trusted.

What the Forger is doing is illegal, of course. It's illegal because it violates the One Definition Rule, which says that if a type (here `X`) is defined more than once, the definitions must be identical. The object being used might be called an `X` and might look like an `X`, but it's not the same kind of X all the other code in the program is using.

Still, this hack will work on most compilers because usually the underlying object data layout will still be the same, and if so, the Forger might be able to live his lie for a time before Tom Hanks finally manages to catch up with him.

Criminal #2: The Pickpocket

The Pickpocket's hack of choice is to silently change the meaning of the class definition. For example:

```
// Example 15-2: Evil macro magic
//
#define private public                // illegal
#include "x.h"

void Hijack(X& x) {
 x.private_ = 2;                       // evil laughter here
}
```

This man is a Pickpocket. Mark him well, for his fingers are light.

What the Pickpocket is doing is illegal, of course. The code in Example 15-2 is non-portable for two reasons:

- It is illegal to `#define` a reserved word.
- It violates the One Definition Rule in the same way that the Forger's tactic did. Still, if the object's data layout is unchanged, the hack might seem to work for a while.

Criminal #3: The Cheat

The Cheat's modus operandi is to substitute one item when you're expecting another. For example:

```
// Example 15-3: Nasty attempt to simulate the object layout.
//
class BaitAndSwitch {                // hopefully has the same data layout as X
public:                              // so we can pass him off as one
 int notSoPrivate;
};

void f(X& x) {
 (reinterpret_cast<BaitAndSwitch&>(x)).notSoPrivate = 2;
}                                  // evil laughter here
```

This man is a Cheat. Mark him well, for he's the kind of bait-and-switch artist who runs newspaper ads just to get you into his store and then claims not to have the advertised item and tries to fob off something of lesser value and higher price.

What the Cheat is doing is illegal, of course. The code in Example 15-3 is illegal for two reasons:

- The object layouts of `X` and `BaitAndSwitch` are not guaranteed to be the same, although in practice they probably always will be.
- The results of the `reinterpret_cast` are undefined, although most compilers will let you try to use the resulting reference in the way the hacker intended. After all, uttering `reinterpret_cast` tells the compiler to trust you, shut its eyes, and not watch the wickedness you're about to perpetrate.

But we were also asked to look for a fully standards-conforming and portable technique. Alas, although many criminals and hackers are smelly and unwashed and nonconforming, some do conform and have an air of respectability. And here he comes, wearing all his spiffed-up finest:

Person #4: The Language Lawyer

Many of us wisely fear dishonest, toothy-smiled lawyers more than (other) criminals.

Consider the following code:

```
// Example 15-4: The legal weasel
//
namespace {
 struct Y {};
}
template<>
void X::f(const Y&) {
 private_ = 2;                       // evil laughter here
}
```

```
void Test() {
 X x;
 cout << x.Value() << endl;          // prints 1
 x.f(Y());
 cout << x.Value() << endl;          // prints 2
}
```

This man is a Language Lawyer who knows the loopholes. He will never be caught, for he is careful to obey the letter of the law while pillaging its spirit. Mark and avoid such ungentlemen.

I wish I could say, "What the Language Lawyer is doing is illegal, of course." Unfortunately, I can't, because it's not illegal. How come? Well, Example 15-4 exploits the fact that `x` has a member template. The code is entirely conforming and is guaranteed by the standard to work as expected. The reason is twofold:

- It's legal to specialize a member template on any type.

The only room for error would be if you tried to specialize it on the same type twice in different ways, which would be a One Definition Rule violation, but we get around that because:

- The code uses a type that's guaranteed to be unique, because it's in the hacker's own unnamed namespace. Therefore it is guaranteed to be legal and won't tromp on anyone else's specialization.

Don't Subvert

There remains only one question:

3. Is this a hole in C++'s access control mechanism, and therefore a hole in C++'s encapsulation? Discuss.

This demonstrates an interesting interaction between two C++ features: the access control model and the template model. It turns out that member templates appear to implicitly "break encapsulation" in the sense that they effectively provide a portable way to bypass the class access control mechanism.

This isn't actually a problem. The issue here is of protecting against Murphy vs. protecting against Machiavelli… that is, protecting against accidental misuse (which the language does very well) as opposed to protecting against deliberate abuse (which is effectively impossible). In the end, if a programmer wants badly enough to subvert the system, he'll find a way, as demonstrated in Examples 15-1 to 15-3.

The real answer to the issue is: Don't do that! Admittedly, there are times when it's tempting to have a quick way to bypass the access control mechanism temporarily, such as to produce better diagnostic output during debugging… but it's just not a habit you want to get into for production code, and it

should appear on the list of one-warning offenses in your development shop.

Guideline

Never subvert the language. For example, never attempt to break encapsulation by copying a class definition and adding a `friend` declaration or by providing a local instantiation of a template member function.

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 class's private virtual function. For example:

```
// Example 16-1, continued: Derived classes can
// override, but not access, private virtual members
//
class Derived : public No {
 virtual void Satisfaction() {          // ok, overrides
  No::Satisfaction();                   // error
 }
};
```

There's just no way for outside (nonmember, nonfriend) code to incant the name of the function. To the question "Just how private is `private`?," we now have the first bit of an answer:

- A `private` member's name is accessible only to other members and friends.

If that were the whole story, this would be a short (and rather pointless) Item. But, of course, accessibility is not the whole story.

The Other Story: Visibility

The keyword `private` does indeed control a member's accessibility. But there is another concept that is related but often confused with accessibility, and that is visibility. Let's return now to the code in the question: Will it compile and run correctly?

The short answer is: No. In the form shown, the program is not legal and will not compile correctly. There are two reasons why. The first one is a fairly obvious error:

```
// Example 16-2 (question code repeated with annotation)
//
// 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);
}
```

Every C++ programmer knows that, even though the version of `Twice` that takes a `complex` object isn't accessible to the code in `main`, it's still visible and constitutes a source dependency. In particular, even though the code in `main` can't possibly ever care about `complex`—it can't even so much as use the name of `Twice(complex<float>)` (it can't call it or even take its address), and the use of `complex` can't possibly affect `Calc`'s size or layout in any way—there still at minimum must be at least a forward declaration of `complex` for this code to hope to compile. (If `Twice(complex<float>)` were also defined inline, then a full definition of `complex` would be required too, even though it still couldn't possibly matter to this code.)

To the question "Just how private is `private`?," we now have another bit of the answer:

- A `private` member is visible to all code that sees the class's definition. This means that its parameter types must be declared even if they can never be needed in this translation unit…

Everyone knows we can fix this easily enough by adding `#include <complex>`, so let's do that. This leaves us with the second, and probably less obvious, problem:

```
// Example 16-3: A partly fixed version of Example 16-2
//
#include <complex>

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);       // error, Twice is inaccessible
}
```

This result surprises a fair number of C++ developers. Some programmers expect that because the only accessible overload of `Twice` takes a `double`, and `21` can be converted to a `double`, that function should be called. That's not, in fact, what happens, for a simple reason: Overload resolution

happens before accessibility checking.

When the compiler has to resolve the call to `Twice`, it does three main things, in order:

1. Name lookup. Before doing anything else, the compiler searches for a scope that has at least one entity named `Twice` and makes a list of candidates. In this case, name lookup first looks in the scope of `Calc` to see if there is at least one member named `Twice`; if there weren't, base classes and enclosing namespaces would be considered in turn, one at a time, until a scope having at least one candidate was found. In this case, though, the very first scope the compiler looks in already has an entity named `Twice`—in fact, it has three of them, and that trio becomes the set of candidates. (For more information about name lookup in C++, with discussion about how it affects the way you should package your classes and their interfaces, see also Items 31 to 34 in Exceptional C++ [Sutter00].)

2. Overload resolution. Next, the compiler performs overload resolution to pick the unique best match out of the list of candidates. In this case, the argument is `21`, which is an `int`, and the available overloads take a `double`, an `int`, and a `complex<float>`. Clearly the `int` parameter is the best match for the `int` argument (it's an exact match and no conversions are required), so `Twice(int)` is selected.

3. Accessibility checking. Finally, the compiler performs accessibility checking to determine whether the selected function can be called. In this case… thud boom splatter.

It doesn't matter that the only accessible function, `Twice(double)`, could in fact be a match; it can never be called, because there is a better match, and being a better match always matters more than being an accessible match.

Interestingly, being even an ambiguous match matters more than being an accessible match. Consider this slight change to Example 16-3:

```
// Example 16-4(a): Introducing ambiguity
//
#include <complex>

class Calc {
public:
 double                Twice(double d);
private:
 unsigned              Twice(unsigned i);
 std::complex<float>   Twice(std::complex<float> c);
};

int main() {
 Calc c;
 return c.Twice(21);              // error, Twice is ambiguous
}
```

In this case, we never get past the second step: Overload resolution fails to find a unique best match

out of the candidate list, because the actual parameter type `int` could be converted to either unsigned or double and those two conversions are considered equally good according to the language rules. Because the two functions are equally good matches, the compiler can't choose between them and the call is ambiguous. The compiler never even gets to the accessibility check.

Even more interestingly, perhaps, even an impossible match matters more than a more accessible match. Consider this rearrangement of Example 16-3:

```
// Example 16-4(b): Introducing plain old name hiding
//
#include <string>

int Twice(int i);                       // now a global function

class Calc {
private:
 std::string Twice(std::string s);
public:
 int Test() {
  return Twice(21);                     // error, Twice(string) is unviable
 }
};

int main() {
 return Calc().Test();
}
```

Again, we never get past the second step: Overload resolution fails to find any viable match out of the candidate list (which now is only `Calc::Twice(string)`), because the actual parameter type `int` can't be converted to `string`. The compiler again never even gets to the accessibility check. Remember, as soon as a scope is found that contains at least one entity with the given name, the search ends—even if that candidate turns out to be uncallable and/or inaccessible. Other potential matches in enclosing scopes will never be considered.

To the question "Just how private is `private`?," we now have yet another bit of the answer:

- A `private` member is visible to all code that sees the class's definition. This means that … it participates in name lookup and overload resolution and so can make calls invalid or ambiguous even though it itself could never be called.

Bjarne Stroustrup writes about this effect in The Design and Evolution of C++ [Stroustrup94, page 55]:

> "Making `public`/`private` control visibility, rather than access, would have a change from public to private quietly change the meaning of the program from one legal interpretation (access [in our example, `Calc::Twice(int)`]) to another (access [in our example, `Calc::Twice(double)`]). I no longer consider this argument conclusive (if I ever did) but the decision made has proven useful in that it allows programmers to add and remove

`public` and `private` specifications during debugging without quietly changing the meaning of programs. I do wonder if this aspect of the C++ definition is the result of a genuine design decision."

Back to the First Story: Granting Access

As the first part of our answer to the "how private is `private`?" question, we said that a private member is accessible only to (its name can be used only by) other members and friends. Note that I deliberately avoided saying anything like 'it can be called only by other members or friends,' because that's actually not true. What accessibility establishes is the code's right to use the name. Let me emphasize that point from the earlier quote from the C++ standard:

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.

If code that has the right to directly use the name (in this case, a member or friend) uses the name to form a function pointer and then passes that pointer out to other code, the receiving code can use that pointer whether or not the receiving code has the right to use the member's name—it no longer needs the name, because it's got this pointer, see. Example 16-5 illustrates this technique at work, where a member function that has access to the name of `Twice(int)` uses that access to leak a pointer to that member:

```
// Example 16-5: Granting access
//
class Calc;
typedef int (Calc::*PMember)(int);

class Calc {
public:
 PMember CoughItUp() { return &Calc::Twice; }

private:
 int Twice(int i);
};
int main() {
 Calc c;
 PMember p = c.CoughItUp();     // yields access to Twice(int)
 return (c.*p)(21);             // ok
}
```

See also [Newkirk97], which describes a useful application of deliberately leaking private implementation details.

To the question "Just how private is `private`?," we now have one more bit of the answer:

- Code that has access to a member can grant that access to any other code by leaking a (name-free) pointer to that member.

Finally, there's another way that some classes can and do give the world a perfectly portable and standards-conforming way to access their private members: member templates. In Item 15, we covered several ways to get access to private parts of a class. Most are illegal or operate outside the laws of the standard, but one is perfectly conforming:

```
// Example 16-6: Adapted from Example 15-4
//

// In a header file
//
class X {
public:
 template<class T>
 void f(const T& t) {
  // …
 }

// …

private:
 int private_;
};

// In user code
//
namespace {
 struct Y {};
}

template<>
void X::f(const Y&) {
 private_ = 2;                    // evil laughter here
}
```

What's going on here? In short, any member template can be specialized for any type. Specialize it on a type that you know nobody else will ever specialize it on (say, a type in your own unnamed namespace), and poof! you've just written a member, and members have access to all parts of the class.

To the question "Just how private is `private`?," we now have one final (at least, final for this Item) bit of the answer:

- A `private` member's name is accessible only to other members (including explicit instantiations of member templates, whether anticipated or not) and friends.

Summary

So, how private is `private`? Here's what we've found:

A `private` member's name is accessible only to other members (including explicit instantiations of member templates, whether anticipated or not) and friends. But code that has access to a member can grant that access to any other code, by leaking a (name-free) pointer to that member.

A `private` member is visible to all code that sees the class's definition. This means that its parameter types must be declared even if they can never be needed in this translation unit, and it participates in name lookup and overload resolution and so can make calls invalid or ambiguous even though it itself could never be called.

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 class by hiding those internals behind a surrounding and encasing interface visible to the outside worl

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 separ and distinct from the internals. Because a capsule surrounds exactly one cohesive group of subobjects, 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 wit 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 can

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.

- Protect, shell

A good class interface protects the internals against unauthorized access and manipulation. In particula a primary job of the interface is to ensure that all access to and manipulation of internal structures is guaranteed to preserve class invariants.

The principal methods for killing bacteria (and humans) involve fashioning devices to break outer and inner capsules. On the micro level, these include chemicals, enzymes, or organisms (and possibly eventual nanomachines) capable of making appropriate holes. On the macro level, knives and guns are perennial favorites.

Encapsulation's Place in OO

> How important is it to object-oriented design and programming?

Encapsulation is the prime concept in object-oriented programming. Period.

Other OO techniques—such as data hiding, inheritance, and polymorphism—are important principally because they support special cases of encapsulation. For example:

- Encapsulation nearly always implies data hiding.
- Run-time polymorphism, using virtual functions, more completely separates the interface (provid by a base class) from the implementation (provided by the derived class, which need not even ex at the time that the code that will eventually use it is written).
- Compile-time polymorphism, using templates, completely divorces interface from implementatio because any class having the required operations can be used interchangeably without requiring a inheritance or other relationship.

Encapsulation is not always data hiding, but data hiding is always a form of encapsulation. Encapsulat is not always polymorphism, but polymorphism is always a form of encapsulation.

Object-orientation is often defined as:

> the bundling together of data and the functions that operate on that data

That definition is true to a point—it excludes nonmember functions that are also logically part of a cla such as `operator<<` in C++—and it stresses high cohesion. It does not, however, adequately emphasi the other essential element of object-orientation, namely:

the simultaneous separation of data from calling code through an interface of functions that operate on that data

This complementary aspect stresses low coupling and that the purpose of the assembled functions is to form a protective interface.

In short, object-orientation is all about separating interfaces from implementation in a way that promot high cohesion and low coupling, both of which have been known to be sound software engineering goa since long before objects were invented. These concepts address dependency management, which is o of the key concepts in modern software engineering, especially for large systems. (See [Martin95], an various 1996 articles by Martin, especially those with "Principle" in the title, at [ObjectMentor].)

Public, Protected, or Private Data?

2. Under what circumstances, if any, should nonstatic class data members be made `public`, `protected`, and `private`? Express your answer as a coding guideline.

Normally we first look at the rule and then at the exception. This time, let's do things the other way around and consider the exception first.

The only exception to the general rule that follows is when all class members (both functions and data) are public, as with a C-style `struct`. In this case the "class" isn't really a full-fledged class with its interface, behavior, and invariants—it's not even a half-fledged class; it's just a bundle-o-data. The "class" is merely a convenient bundling of objects, and that's fine, especially for backward compatibil with C programs that manipulate C-style `struct`s.

Other than that special case, however, data members should always be private.

Public data is a breach of encapsulation because it permits calling code to manipulate the object's internals directly. This implies a high level of trust! After all, in real life, most other people don't get t manipulate my internals directly (e.g., by operating directly on my stomach), because they might then easily and unintentionally do the wrong thing; at best, they only get to manipulate my internals indirectl by going through my public interface with my knowledge and consent (e.g., by handing me a bottle labeled "Drink Me," which I will then decide to drink or shampoo my hair with or wash my car with, according to my own feelings and judgment). Of course, some people really are qualified to manipulat my internals directly (e.g., a surgeon), but even then: a) it's rare; b) I get to elect whether or not to have the surgery; and c) I get to choose which surgeon I will declare has my requisite high level of trust.

Similarly, most calling code shouldn't ever manipulate a class's internals directly (e.g., by viewing or changing member data) because they might quite easily and unintentionally do the wrong thing; at best, they only get to manipulate the class's internals indirectly by going through the class's public interface with the class's knowledge and consent (e.g., by handing a `Bottle("Drink Me")` object to a public member function, which will then decide what, if anything, to do with the object according to the class author's own feelings and judgment). Of course, some non-member code might really be qualified to manipulate a class's internals directly (usually such code should be a member function, but for exampl

`operator<<` cannot be a member), but even then: a) it's rare; b) the class gets to elect whether or not t declare `friend`s at all; and c) the class gets to choose which such outside code will be declared a `friend` with that declaration's attendant high level of trust.

In short, public data is evil (except only for C-style `struct`s).

Likewise, in short, protected data is evil—this time with no exceptions.

"Now just wait a minute," someone might say. "I'm with you on the public data thing, but why say that a protected data is evil?" Because the same argument applies equally to protected data, which is part of interface too: the protected interface, which is still an interface to outside code, just a smaller set of it, namely the code down in the derived classes. Why is there no exception? Because protected data is never just a bundle-o-data; if it were, it could be used as such only by derived classes, and since when do you use additional instances of one of your base classes as a convenient bundle-o-data? That would be bizarre.

For more on the history of why protected data was originally permitted and why even the person who campaigned for it now agrees it was a bad idea, see

Stroustrup's The Design and Evolution of C++ [Stroustrup94].

Guideline

Always make all data members private. The only exception is the case of a C-style `struct` that isn't intended to encapsulate anything and where all members are public.

A General Transformation

Now let's prove the "member data should always be private" guideline by assuming the opposite (that public/protected member data can be appropriate) and showing that in every such case the data should not in fact be public/protected at all.

```
// Example 17-2(a): Nonprivate data (evil)
//
class X {
 // …
public:
 T1 t1_;
protected:
 T2 t2_;
};
```

First, we note that this can always be transformed, without loss of either generality or efficiency, to:

```
// Example 17-2(b): Encapsulated data (good)
//
class X {
 // …
public:
 T1& UseT1() { return t1_; }
protected:
 T2& UseT2() { return t2_; }
private:
 T1 t1_;
 T2 t2_;
};
```

Therefore, even if there's a reason to allow direct access to `t1_` or `t2_`, there exists a simple transformation that causes the access to be performed through a(n initially inline) member function. Examples 17-2(a) and 17-2(b) are equivalent. But is there any benefit to using the method in Example 17-2(a)?

To prove that Example 17-2(a) should never be used, all that remains is to show that:

1. Example 17-2(a) has no advantages not present in Example 17-2(b);
2. Example 17-2(b) has concrete advantages; and
3. Example 17-2(b) costs nothing.

Taking them in reverse order:

Point 3 is trivial to show. The inline function, which returns by reference and hence incurs no copying cost, will probably be optimized away entirely by the compiler.

Point 2 is easy: Just look at the source dependencies. In Example 17-2(a), all calling code that uses `t1` and/or `t2_` mentions them explicitly by name; in Example 17-2(b), all calling code that uses `t1_` or `t2` mentions only the names of the functions `UseT1` and `UseT2`. Example 17-2(a) is rigid, because any cha to `t1_` or `t2_` (e.g., removing them and replacing them with something else or just tacking on some instrumentation) requires all calling code to be changed to suit. In Example 17-2(b), however, instrumentation can be added, and `t1_` and/or `t2_` can even be removed entirely, without any change t calling code, because the member function completes the class's interface and "surrounds," "seals," an "protects" the internals.

Finally, Point 1 is demonstrated by observing that anything that calling code could do with `t1_` or `t2_` directly in Example 17-2(a), it can still do by using the member accessor in Example 17-2(b). The cal might have to write an extra pair of brackets, but that's it.

Let's consider a concrete example: Say you want to add some instrumentation, perhaps something as simple as counting the number of accesses to `t1_` or `t2_`. If it's a data member, as in Example 17-2(a), here's what you have to do:

1. You create accessor functions that do what you want, and make the data private. (In other word you do Example 17-2(b) anyway, only later as a retrofit.)

2. All your users get to experience the joy of finding and changing every use of `t1_` and `t2_` in the code to the functional equivalent. This is just bound to cause rejoicing among a user community w pressing deadlines who already have other real work to do. Your users might thank you profusely and buy you gifts as a reward; don't open them if they're ticking.

3. All your users recompile.

4. The compile will break if they missed any instances; fix them by repeating steps 2 and 3 until done.

If you already have simple accessor member functions, as in Example 17-2(b), here's what you have to do:

1. You make the change inside the existing accessor functions.
2. All your users relink (if the functions are in a separate `.cpp` and not inline), or, at worst, recompi (if the functions are in the header).

The worst part is that, in real life, if you started with Example 17-2(a), you might never even be allow later to make the change to get to Example 17-2(b). The more users there are that depend on an interfa the more difficult it is to ever change the interface. This brings us to a guideline that's tantamount to a Law of Second Chances:

Guideline

The most important thing to get right is the interface. Everything else can be fixed later.

Get the interface wrong, and you might never be allowed to fix it.

Once an interface is in widespread use, so many people may depend on it that it becomes infeasible to change it. True, interfaces can always be extended (added to instead of changed) without impacting anyone, but just adding member functions doesn't help fix existing parts that you later decide were a ba idea—at most it lets you add alternative ways of doing things that will confuse your users, who will legitimately ask: "But there are two (or three, or N) ways of doing it… why? Which one do I use?"

In short, a bad interface can be difficult or impossible to fix after the fact. Do your best to get the interface right the first time, and make it surround, seal, and protect its internals.

A Case in Point

3. The `std::pair` class template uses public data members because it is not an encapsulated class only a simple way of grouping data.

Note that this is the exceptional valid use of public data. Even so, `std::pair` would have been no wo off with accessors instead of public 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:

```
// Example 17-3(a): Mixing public and private data?
//
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?

This `Couple` class was proposed as a counterexample to the usual coding guidelines. It attempts to sho a class that is "mostly a `struct`" but has some private housekeeping data. The housekeeping data (here simple attribute) has an invariant. The claim is that updates to the attribute flag are totally independent updates to the `Couple`'s values.

Let's start with the last "totally independent" statement: I don't buy it. The updates might be independer but the attribute is clearly not independent of the values, else it wouldn't be grouped together cohesive with them. Of course the `deleted_` attribute isn't independent of the accompanying objects—it applies them!

Note how, instead of mixing public and private data, we can model the solution by using accessors eve if the accessors' initial implementation was to give up references:

```
// Example 17-3(b): Proper encapsulation, initially with inline accessors. Later
```

```
// in life, these might grow into nontrivial functions if needed; if not, then no
//
template<class T, class U>
class Couple {
  Couple()           : deleted_(false) { }
  T& First()         { return first_; }
  U& Second()        { return second_; }
  void MarkDeleted() { deleted_ = true; }
  bool IsDeleted()   { return deleted_; }

private:
 T first_;
 U second_;
 bool deleted_;
};
```

"Huh?" someone might quickly say. "Why bother writing do-nothing accessor functions?" Answer: As described in the earlier discussion about Example 17-2(b). If today calling code can change some aspe of this object (in this example, the tagalong `deleted_` attribute), tomorrow you might well want to add new features even if they do no more than add debugging information or add checks. Example 17-3(b) lets you do that, and that flexibility doesn't cost you anything in terms of efficiency because of the inlin functions.

For example, say that one month in the future you decide that you want to check all attempted accesses an object marked deleted:

- In Example 17-3(a), you can't, period—not without changing the design and requiring all code tha uses `first_` and `second_` to be rewritten.
- In Example 17-3(b), you simply put the check into the `First` and `Second` members. The change is transparent to all past and present users of `Couple`. At most a recompile is needed; no code chang are needed.

It turns out that Example 17-3(b) has other practical side benefits in the real world. For example, as N Mein points out: "You can put a breakpoint (or whatever) in the accessor to find out just where and wh the value is being modified. This can be pretty helpful in tracking down a bug." It sure can be, and ofte is.

Summary

Except for the case of a C-style `struct` (all members public), all data members should always be private. Doing otherwise violates all the principles of encapsulation noted at the start of this Item and creates dependencies on the data names, which makes it harder to later encapsulate them correctly. Th is never a good reason to write public or protected data members; they can always be trivially wrappe in (at first) inline accessor functions at no cost, so it's always right to do the right thing the first time. (True, there are examples of protected data in the standard library. These examples are not exemplary.

Get your interfaces right first. Internals are easy enough to fix later, but if you get the interface wrong y
might never be allowed to fix it.

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 `struct`s, 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 appears again in a specialization of a class template, here's what the standard library has:

- 6 public virtual functions, all of which are `std::exception::what` and its overrides; and
- 142 nonpublic virtual functions.

Recently I came across another major example. While writing this book, I've also been working at Microsoft on the C++ aspects of the .NET platform and the .NET Frameworks (FX), which is being evolved into WinFX, the object-oriented successor to the Win32 API and the programming model for Longhorn, Microsoft's next-generation Windows operating system due out around the middle of this decade. WinFX, even in its current in-progress internal form, is already a huge API—as of this writing, WinFX already contains over 14,000 classes and nearly 100,000 member functions (which includes all the currently shipping .NET Frameworks and a lot more stuff that busy people have been busy building). Yes, that's big. No, you wouldn't be faulted for wondering whether it's too big, but there it is.

So here's why I mention .NET Frameworks and its evolution into WinFX: For a be-hemoth class library this big, one would hope there would be good class design guidelines and that they would be followed reliably. I'm happy to report that, at this point, one would be correct about both of those things. Here is one WinFX design guideline that might sound awfully familiar, and although I agree with it I had no influence on it… it was completely independently arrived at, and for similar but overlapping reasons:

> It is recommended that you provide customization through protected (family) methods. The public interface of a base class should provide a rich set of functionality for the consumer of that class. However, customizers of that class often want to implement the fewest methods possible to provide that rich set of functionality to the consumer. To meet this goal, provide a set of nonvirtual or final public methods that call through to a single protected (family) method with the 'Core' suffix that provides implementations for such a method. Such pattern is also known as 'Template Method'.
>
> —from the .NET Framework (WinFX) Design Guidelines, internal draft, January 2004

Why is this pattern, which keeps cropping up in practice, such a good idea? Let's investigate.

Traditionally, many programmers were used to writing base classes using public virtual functions. For example, we might write:

```
// Example 18-1: A traditional base class.
//
class Widget {
public:
 // Each of these functions might optionally be pure virtual, and if so
 // might or might not have an implementation in Widget; see Item 23
 // in More Exceptional C++ [Sutter02].
 //
 virtual intProcess(Gadget&);
 virtual bool IsDone();

 // …
};
```

These public virtual functions, like all public virtual functions, simultaneously specify both the interface and the customizable behavior. The problem is in that "simultaneously" part, because every public virtual function is forced to serve two audiences with distinct needs and different purposes:

- One audience is the outside callers of the class, who depend on the class's public interface to use the class.
- The other audience is derived classes, who depend also on the "customization interface" of virtual functions, through which they extend and customize their base class.

A public virtual function is forced to do two jobs: It specifies interface because it's public and therefore directly part of the interface `Widget` presents to the rest of the world; and it specifies implementation detail, namely the internally customizable behavior, because it's virtual and therefore provides a hook for derived classes to replace the base implementation of that function (if any). That a public virtual function inherently has two significantly different jobs, and two competing audiences, is a sign that it's not separating concerns well and that we should consider a different approach.

What if we want to separate the specification of interface from the specification of the implementation's customizable behavior? Then we end up with something that should remind us strongly of the Template Method pattern [Gamma95] because it's very similar. But this one has a narrower purpose, and so deserves a more focused name: The Nonvirtual Interface (NVI) pattern. Here it is in action:

```
// Example 18-2: A more modern base class, using Nonvirtual Interface (NVI)
// to separate interface from internals.
//
class Widget {
public:
 // Stable, nonvirtual interface.
 //
 intProcess(Gadget&); // uses DoProcess...()
 bool IsDone(); // uses DoIsDone()
```

```
 // ...
private:
 // Customization is an implementation detail that might or might not directly
 // correspond to the interface. Each of these functions might optionally be
 // pure virtual, and if so might or might not have an implementation in
 // Widget; see Item 23 in More Exceptional C++ [Sutter02].
 //
 virtual intDoProcessPhase1(Gadget&);
 virtual intDoProcessPhase2(Gadget&);
 virtual bool DoIsDone();

 // ...
};
```

Prefer to use the NVI to make the interface stable and nonvirtual while delegating customizable work to nonpublic virtual functions that are responsible for implementing the customizable behavior. After all, virtual functions are designed to let derived classes customize behavior; it's better to not let publicly derived classes also customize the inherited interface, which is supposed to be consistent.

The NVI approach has several benefits and no significant drawbacks.

First, note that the base class is now in complete control of its interface and policy and can enforce interface preconditions and postconditions, insert instrumentation, and do any similar work all in a single convenient reusable place—the nonvirtual interface function. This promotes good class design because it lets the base class enforce the substitutability compliance of derived classes in accord with the Liskov Substitution Principle [Liskov88], to whatever extent enforcement makes sense. If efficiency is an issue, the base class can elect to check certain kinds of pre- and post-conditions only in a debug mode—for example, via a non-debug "release" build that completely removes the checking code from the executable image, or via a configurable debug mode that suppresses selected checking code at run-time.

Second, when we've better separated interface and implementation, we're free to make each take the form it naturally wants to take instead of trying to find a compromise that forces them to look the same. For example, notice that in Example 18-2 we've incidentally decided that it makes more sense for our users to see a single `Process` function while allowing more flexible customization in two parts, `DoProcessPhase1` and `DoProcessPhase2`. And it was easy. We couldn't have done this with the public virtual version without making the separation also visible in the interface, thereby adding complexity for the user who would then have to know to call two functions in the right way. (For more discussion of a related example, see also Item 19 in Exceptional C++ [Sutter00].)

Third, the base class is now less fragile in the face of change. We are free to change our minds later and add pre- and postcondition checking, or separate processing into more steps, or refactor, or implement a fuller interface/implementation separation using the Pimpl idiom (see Exceptional C++ [Sutter00]), or make other modifications to `Widget`'s customizability, without affecting the code that uses `Widget`. For example, it's much more difficult to start with a public virtual function and later try to wrap it for pre- and postcondition checking after the fact, than it is to provide a dumb passthrough

nonvirtual wrapper up front (even if no checking or other extra work is immediately needed) and insert the checking later. (For more discussion of how a class such as `Widget` is less fragile and more amenable to future revision and refactoring, see [Hyslop00].)

"But but but," some have objected, "let's say that all the public nonvirtual function does initially is pass through to the private virtual one. It's just one stupid little line. Isn't that pretty useless, and indeed haven't we lost something? Haven't we lost some efficiency (the extra function call) and added some complexity (the extra function)?" No, and no. First, a word about efficiency: No, none is lost in practice because if the public function is a one-line passthrough declared `inline`, all compilers I know of will optimize it away entirely, leaving no overhead.[22] Second, a word about complexity: The only complexity is the extra time it takes to write the one-line wrapper function, which is trivial. Period. That's it. C'est tout. The interfaces are unaffected: The class still has exactly the same number of public functions for a public user to learn, and it has exactly the same number of virtual functions for a derived class programmer to learn. Neither the interface presented to the outside world nor the inheritance interface presented to derived classes has become any more complex in itself for either audience. The two interfaces are just explicitly separated, is all, and that is a Good Thing.

> [22] Indeed, some compilers will always make such a function `inline` and eliminate it, whether you personally really wanted it to or not, but that's another story; see Item 25.

Well, that justifies nonvirtual interfaces and tells us that virtual functions benefit from being nonpublic, but we haven't answered whether virtual functions should be private or protected. So let's answer that:

Guideline

Prefer to make virtual functions private.

That's easy. This lets the derived classes override the function to customize the behavior as needed, without further exposing the virtual functions directly by making them callable by derived classes (as would be possible if the functions were just protected). The point is that virtual functions exist to allow customization; unless they also need to be invoked directly from within derived classes' code, there's no need to ever make them anything but private. But sometimes we do need to invoke the base versions of virtual functions (see [Hyslop00] for an example), and in that case only it makes sense to make those virtual functions protected, thus:

Guideline

Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.

The bottom line is that NVI as applied to virtual functions nicely helps us separate interface from implementation. It's possible to make the separation even more complete, of course, by completely divorcing interface from implementation using patterns such as Bridge [Gamma95], idioms such as Pimpl (principally for managing compile-time dependencies and exception safety guarantees) [Sutter00, Sutter02] or the more general handle/body or envelope/letter [Coplien92], or other approaches. Unless you need a more complete interface/implementation separation, though, NVI will often be sufficient for your needs. On the flip side, I am arguing that this use of NVI is also a good idea to adopt by default and view as a necessary minimum separation in practice in new code. After all, it costs nothing (beyond writing an extra line of code) and buys quite a bit of pain reduction down the road.

For more examples of using the NVI pattern to privatize virtual behavior, see [Hyslop00].

Speaking of [Hyslop00], did you notice that the code there presented a public virtual destructor? This brings us to the second topic of this Item:

Virtual Question #2: What About Base Class Destructors?

Now we're ready to tackle the second classic question, that old destructor chestnut: "Should base class destructors be virtual?"

As already noted, 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.

The slightly less usual and somewhat more correct answer is: "Huh? Of course base class destructors should be virtual if you're going to delete polymorphically (i.e., delete via a pointer to base)!" This answer is technically right but doesn't go far enough.

I've recently come to conclude that the fully correct answer is this:

Guideline

A base class destructor should be either public and virtual, or protected and nonvirtual.

Let's see why this is so.

First, clearly any operation that will be performed through the base class interface and should behave virtually, should be virtual. That's true even with NVI, because although the public function is nonvirtual, the work is delegated to a nonpublic virtual function and we get the virtual behavior that we need.

If deletion, therefore, can be performed polymorphically through the base class interface, then it must behave virtually and must be virtual. Indeed, the language requires it—if you delete polymorphically without a virtual destructor, you summon the dreaded specter of "undefined behavior," a specter I personally would rather not meet in even a moderately well-lit alley, thank you very much. Hence:

```
// Example 18-3: Obvious need for virtual destructor.
//
class Base { /*...*/ };
class Derived : public Base { /*...*/ };

Base* b = new Derived;
delete b;                               // Base::~Base had better be virtual!
```

Note that the destructor is the one case where the NVI pattern cannot be applied to a virtual function. Why not? Because once execution reaches the body of a base class destructor, any derived object parts have already been destroyed and no longer exist. If the `Base` destructor body were to call a virtual function, the virtual dispatch would reach no further down the inheritance hierarchy than `Base` itself. In a destructor (or constructor) body, further-derived classes just don't exist any more (or yet).

But base classes need not always allow polymorphic deletion. For example, in the standard library itself [C++03], consider class templates such as `std::unary_function` and `std::binary_function`. Those two class templates look like this:

```
template <class Arg, class Result>
struct unary_function {
 typedef Arg argument_type;
 typedef Result result_type;
};

template <class Arg1, class Arg2, class Result>
struct binary_function {
 typedef Arg1 first_argument_type;
 typedef Arg2 second_argument_type;
 typedef Result result_type;
};
```

Both of these templates are specifically intended to be instantiated as base classes (in order to inject those standardized `typedef` names into derived classes) and yet do not provide virtual destructors because they are not intended to be used for polymorphic deletion. That is, code like the following is not merely unsanctioned but downright illegal, and it's reasonable for you to assume such code will never exist:

```
// Example 18-4: Problematic code that you can assume will never exist.
//
void f(std::unary_function* f) {
 delete f;                              // error, illegal
}
```

Note that the standard tut-tuts and declares Example 18-4 to fall squarely into the Undefined Behavior Pit if you pass it a pointer to an object derived from `std::unary_function`, but the standard doesn't actually require a compiler to prevent you or anyone else from writing that code (more's the pity). It would be easy and nice—and it wouldn't break any standards-conforming C++ programs that exist today—to give `std::unary_function` (and other classes like it) an empty but protected destructor, in which case a compiler would actually be required to diagnose the error and toss it back in the offender's face. Maybe we'll see such a change in a future revision to the standard, maybe we won't, but it would be nice to make compilers reject such code instead of just making tut-tut noises in standardish legalese. (Yes, making the destructor nonpublic means that you wouldn't be able to instantiate `unary_function` directly, but that doesn't matter because it is useful only as a base class.)

Finally, what if a base class is concrete (can be instantiated on its own) but also wants to support polymorphic destruction? Doesn't it need a public destructor then, because otherwise you can't easily create objects of that type? That's possible, but only if you've already violated another guideline, to wit: Don't derive from concrete classes. Or, as Scott Meyers puts it in Item 29 of [Meyers96], "Make non-leaf classes abstract." (Admittedly, it can happen in practice—in code written by someone else, of course, not by you—and in this one case you might have to have a public virtual destructor just to accommodate what's already a poor design. Better to refactor and fix the design, though, if you can.)

In brief, then, you're left with one of two situations. Either: a) you want to allow polymorphic deletion through a base pointer, in which case the destructor must be virtual and public; or b) you don't, in which case the destructor should be nonvirtual and protected, the latter to prevent the unwanted usage.

Summary

In summary, prefer to make base class virtual functions private (or protected if you really must). This separates the concerns of interface and implementation, which stabilizes interfaces and makes implementation decisions easier to change and refactor later. For normal base class functions:

- Guideline #1: Prefer to make interfaces nonvirtual, using the Nonvirtual Interface pattern (NVI).
- Guideline #2: Prefer to make virtual functions private.
- Guideline #3: Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.

For the special case of the destructor only:

- Guideline #4: A base class destructor should be either public and virtual, or protected and nonvirtual.

True, the standard library itself does not always follow all these design criteria. In part, that's a reflection of how we as a community have learned over the years.

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 */);
 Count& operator=(const Count&);              // does the usual
 virtual ~Count();                            // does the usual
};
```

Unfortunately, programmers are human too, and they sometimes forget that they should write two of the functions explicitly.

```
class BadDerived : private virtual Count {
 int i_;

 // default constructor: should call special ctor, but does it?
 // copy constructor: should call special ctor, but does it?
 // copy assignment: ok?
 // destructor: ok?
};
```

In the context of this example, is there a way for the author of `Count` to force derived classes to be coded correctly—that is, to cause an error at compile time (preferable) or run time (at minimum) if the derived class is not coded correctly?

More generally: Is there any way that the author of a base class can force authors of derived classes to explicitly write each of these four basic operations? If so, how? If not, why not?

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 `struct`s 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 declared function can throw any of the types mentioned in those exception specifications:

```
// public:
inline C::C() throw (
 // anything that a C base or member default constructor might throw;
 // i.e., the union of all types mentioned in C's bases' or members' default
 // constructors' exception specifications
                  );
```

It turns out that there's a potential trap lurking here. Consider: What if one of the implicitly generated functions overrides an inherited virtual function? This can't happen for constructors (because constructors are never virtual), but it can happen for the copy assignment operator (if you try hard and make a base version that matches the implicitly generated derived version's signature), and it can happen pretty easily for the destructor:

```
// Example 19-1(b): Danger, Will Robinson!
//
class Derived;

class Base {
public:
 // Somewhat contrived, and almost certainly deplorable, but technically it is
 // possible to use this trick to declare a Base assignment operator that takes a
 // Derived argument; be sure to read [Meyers96] Item 33 before even thinking
 // about trying anything like this:
 //
 virtual Base& /* or Derived& */
 operator=(const Derived&) throw(B1);

 virtual ~Base() throw(B2);
};

class Member {
public:
 Member& operator=(const Member&) throw(M1);
 ~Member() throw(M2);
```

```
};

class Derived : public Base {
 Member m_;

 // implicitly declares four functions:
 // Derived::Derived();                                          // ok
 // Derived::Derived(const Derived&);                            // ok
 // Derived& Derived::operator=(const Derived&) throw(B1, M1); // error
 // Derived::~Derived() throw(B2, M2);                           // error
};
```

What's the problem? The two functions are ill-formed because whenever you over-ride any inherited virtual function, your derived function's exception specification must be at least as restrictive as the version in the base class. That only makes sense, after all: If it weren't that way, it would mean that code that calls a function through a pointer to the base class could get an exception that the base class version promised not to emit. For instance, if the context of Example 19-1(b) were allowed, consider the code:

```
Base* p = new Derived;

// Ouch this could throw B2 or M2, even though
// Base::~Base promised to throw at most B2:
delete p;
```

This is Yet Another Good Reason why every destructor should have either an exception specification of `throw()` or none at all. Besides, destructors should never throw anyway and should always be written as though they had an exception specification of `throw()` even if that specification isn't written explicitly. (See Item 12 of Exceptional C++ [Sutter00], which includes a section titled "Destructors That Throw and Why They're Evil.")

Guidelines

Never allow a destructor to emit an exception. Always write destructors as though they had a throw-nothing exception specification.

Never write an exception specification. Except possibly an empty one, but if I were you I'd avoid even that (see Item 13).

This is also Yet Another Good Reason to be careful about virtual assignment operators. See [Meyers96] Item 29 and [Dewhurst03] Item 76 for more about the hazards of virtual assignment and how to avoid them.

Guideline

Avoid making assignment operators virtual.

Now let's consider the four implicitly generated functions one at a time:

Implicit Default Constructor

a. default constructor

A default constructor is implicitly declared if you don't declare any constructor of your own. An implicitly declared default constructor is `public` and `inline`.

An implicitly declared default constructor is only implicitly defined when you actually try to call it, has the same effect as if you'd written an empty default constructor yourself, and can throw anything that a base or member default constructor could throw. It is illegal if that empty default constructor would also have been illegal had you written it yourself (for example, it would be illegal if some base or member doesn't have a default constructor).

Implicit Copy Constructor

b. copy constructor

A copy constructor is implicitly declared if you don't declare one yourself. An implicitly declared default constructor is `public` and `inline` and will take its parameter by reference to `const` if possible (it's possible if and only if every base and member has a copy constructor that takes its parameter by reference to `const` or `const volatile` too), and by reference to non-`const` otherwise.

That's right, just like most C++ programmers, the standard itself pretty much ignores the `volatile` keyword a lot of the time. Although the compiler will take pains to tack `const` onto the parameter of an implicitly declared copy constructor (and copy assignment operator) whenever possible, frankly—to use the immortal words of Clark Gable in Gone with the Wind—it doesn't give a hoot about tacking on `volatile`. Oh well, that's life.

An implicitly declared copy constructor is only implicitly defined when you actually try to call it to copy an object of the given type, performs a memberwise copy of its base and member subobjects, and can throw anything that a base or member copy constructor could throw. It is illegal if any base or member has an inaccessible or ambiguous copy constructor.

Implicit Copy Assignment Operator

c. copy assignment operator

A copy assignment operator is implicitly declared if you don't declare one yourself. An implicitly declared default constructor is `public` and `inline`, returns a reference to non-`const` that refers to the assigned-to object, and will take its parameter by reference to `const` if possible (it's possible if and only if every base and member has a copy assignment operator that takes its parameter by reference to `const` too), and by reference to non-`const` otherwise. Once again, just as with the copy constructor, `volatile` can go hang.

An implicitly declared copy assignment operator is implicitly defined only when you actually try to call it to assign an object of the given type, performs a member-wise assignment of its base and member subobjects (including possibly multiple assignments of virtual base subobjects), and can throw anything that a base or member copy constructor could throw. It is illegal if any base or member is `const`, is a reference, or has an inaccessible or ambiguous copy assignment operator.

Implicit Destructor

d. destructor

A destructor is implicitly declared if you don't declare any destructor of your own. An implicitly declared destructor is `public` and `inline`.

An implicitly declared destructor is implicitly defined only when you actually try to call it, has the same effect as if you'd written an empty destructor yourself, and can throw anything that a base or member destructor could throw. It is illegal if any base or member has an inaccessible destructor, or if any base destructor is `virtual` and not all base and member destructors have identical exception specifications (see Exception Specifications of Implicitly Declared Functions, page 141).

An auto_ptr Member

2. What functions are implicitly declared and implicitly defined for the following class `X`? With what signatures?

```
// Example 19-2
//
class X {
 auto_ptr<int> i_;
};
```

Aside: Note that this is an example to illustrate the rules governing implicitly-declared and defined functions. It isn't intended to be an example of good style. In general you should avoid `auto_ptr` members and try to avoid `auto_ptr` altogether. Instead, prefer `shared_ptr`, now being added to the C++ standard library.

In class `X`, the following functions are implicitly declared as public members. Each is implicitly defined, with the indicated effects, when you write code that tries to use it.

```
inline X::X() throw()                : i_() { }
inline X::X(X& other) throw()        : i_(other.i_) { }
inline X& X::operator=(X&) throw() { i_ = other.i_; return *this; }
inline X::~X() throw()               { }
```

The copy constructor and copy assignment operators take references to non-`const` because they can't—that's what `auto_ptr`'s versions do. Similarly, all of these functions have throw-nothing specifications because they can't—no related `auto_ptr` operation throws, and indeed no `auto_ptr` operation at all can throw.

Note that the copy constructor and copy assignment operator transfer ownership. That might not be what the author of `X` necessarily wants, so `X` often should provide its own versions of these functions. (For more details about this and related topics, see also Items 30 and 31 of More Exceptional C++ [Sutter02].)

Renegade Children and Other Family Problems

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:

```
// Example 19-3
//
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 */);
 Count& operator=(const Count&);              // does the usual
 virtual ~Count();                            // does the usual
```

Here we have a class that wants its derived child classes to play nice and call `Count`'s special constructor, perhaps so that `Count` can keep track of the number of objects of derived types created in the system. That's a good reason to require virtual inheritance, to avoid double-counting if some derived class happens to inherit multiply in such a way that `Count` is a base class more than once.[23]

[23] This example is adapted from one by Marco Dalla Gasperina in his unpublished article "Counting Objects and Virtual Inheritance." His code didn't include the design errors I talk about next. That article was about something else, not about enforcing rules for derived classes, but his example made me think, "hmm, how could I find a way to prevent authors of derived classes from for-getting to use the special constructor when they use that `Counter` base class?"

Interestingly, did you notice that `Count` might have a design error already? It has an implicitly

generated copy constructor, which probably isn't what is wanted to keep track of a correct count. To disable that, simply declare it private without a definition:

```
private:
 // undefined, no copy construction
 Count(const Count&);
};
```

So `Count` wants its derived child classes to behave. But kids don't always play nice, do they? Indeed, we don't have to look far to find an example of a badly behaved problem child:

> Unfortunately, programmers are human too, and they sometimes forget that they should write two of the functions explicitly.
>
> ```
> class BadDerived : private virtual Count {
>  int i_;
>
>  // default constructor: should call special ctor, but does it?
> ```

In short, no, the default constructor not only doesn't call the special constructor, but there's an even more fundamental concern: Is there even a `BadDerived` default constructor at all? The answer, which probably isn't reassuring, is: Sort of. There is an implicitly-declared default constructor (okay), but if you ever try to call it, the program becomes ill-formed (oops).

Let's see why this is so. First, `BadDerived` doesn't define any of its own constructors, so a default constructor will be implicitly declared. That's cool. But the minute you try to use that constructor (i.e., the minute you try to create a `BadDerived` object, which you might think is kind of an important thing to be able to do, and you'd be right), that default constructor gets implicitly defined—or at least it should be, but because that implicit definition is supposed to call a base default constructor that doesn't exist, the program is ill-formed. Bottom line, any program that tries to create a `BadDerived` object is not a conforming C++ program, and for that reason `BadDerived` is properly viewed as delinquent.

So is there a default constructor? Sort of. It's declared, but you can't call it, which makes it considerably less useful. When kids go renegade like this, it's just not a happy family.

```
// copy constructor: should call special ctor, but does it?
```

For similar reasons, the implicitly generated copy constructor will be declared but, when defined, won't call the special `Count` constructor. With the `Count` class as originally shown, this copy constructor will simply call `Count`'s implicitly generated copy constructor.

If we decide to suppress `Count`'s implicitly generated copy constructor, as indicated earlier, then this

`BadDerived` would have a copy constructor implicitly declared, but because it can't be implicitly defined (because `Count`'s wouldn't be accessible), any attempt to use it would make the program not valid C++.

Fortunately, now the news starts getting a little better:

```
// copy assignment: ok?
 // destructor: ok?
};
```

Yes, the implicitly generated copy assignment operator and destructor will both do the right thing, namely invoke (and, in the destructor's case, override) the base class versions. So the good news is that at least something worked right.

Still, all is not happy in this class family. Every household must have some minimum of order, after all. Can we not find a way to give the parents better tools to keep the peace?

Enforcing Rules for Derived Classes

> In the context of this example, is there a way for the author of `Count` to force derived classes to be coded correctly—that is, to cause an error at compile time (preferable) or run time (at minimum) if the derived class is not coded correctly?

The idea isn't to suppress the implicit declaration (we can't) but to make the implicit definition not well-formed so that the compiler should emit an understandable error about it.

> More generally: Is there any way that the author of a base class can force authors of derived classes to explicitly write each of these four basic operations? If so, how? If not, why not?

Well, as we went through the review of what happens to implicitly declare and define the four basic operations, we kept coming across the words "inaccessible" and "ambiguous." It turns out that adding ambiguous overloads, even with different access specifiers, doesn't help much. It's hard to do much better than simply make the base class functions selectively inaccessible by declaring them private (whether they're actually defined is optional)—and this approach works for all the functions but one.

```
// Example 19-4: Try to force derived classes to not use their implicitly
// generated functions, by making Base functions inaccessible.
//
class Base {
public:
 virtual ~Base();

private:
 Base(const Base&);                 // undefined
 Base& operator=(const Base&);      // undefined
};
```

This `Base` has no default constructor (because a user-defined constructor has been declared, if not defined), and it has a hidden copy constructor and copy assignment operator. There's no way we could hide the destructor, which must always be accessible to derived classes after all, and which should normally be public and virtual as in Example 19-4, or else protected and nonvirtual (see Item 18).

The idea is that, even if we do want to support a form of a given operation (for example, copy assignment), if we can't do it with the usual function, then we make the usual function inaccessible and provide a named, or otherwise distinguishable, function to do the work the way we want it done.

Where does this get us? Let's see:

```
class Derived : private Base {
int i_;

 // default constructor:  declared, but definition is ill-formed
 //                       (there is no Base default constructor)

 // copy constructor:     declared, but definition ill-formed
 //                       (Base's copy constructor is inaccessible)

 // copy assignment:      declared, but definition ill-formed
 //                       (Base's copy assignment is inaccessible)

 // destructor:           well-formed, will compile
};
```

Not bad… we got three compile-time errors out of a possible four, and it turns out that's pretty much the best we can do, or indeed need to do.

This simple solution can't handle the destructor case, but that's okay because destructors are less amenable to special-purpose replacement anyway; base versions must always be called, no two ways about it, and, after all, there can be only one destructor. The difficult part is usually getting any unusual constructors to be correctly called so that the base class is correctly initialized; the base class can then normally save the information it needs to do the right thing in the destructor.

There, that wasn't bad. Simple solutions are usually best. In this case there were some more complex alternatives; let's consider them briefly to reassure ourselves that none of them could do better for the destructor case, or any other case for that matter:

Alternative #1: Make base class functions ambiguous. This isn't any better, it still doesn't break the implicitly generated destructor, and it's more work.

Alternative #2: Provide base versions that blow up. For example, we could make them throw a `std::logic_error` exception. This also doesn't break the implicitly generated destructor (without

breaking all possible destructors), and it turns a compile-time error into a run-time error, which is not as good.

Guideline

Prefer compile-time errors to run-time errors.

Alternative #3: Provide base versions that are pure virtual. This is useless: It doesn't apply to constructors (either default or copy); it won't help with copy assignment, because the derived versions have different signatures; and it won't help with the destructor because the implicitly generated version will satisfy the requirement to define a destructor.

Alternative #4: Use a virtual base class without a default constructor. This forces each most-derived class to explicitly call the virtual base's constructor. This approach has potential for the two constructors and has the additional advantage of working even for classes that aren't immediately derived from `Base`, so this is really the only alternative that could be used in conjunction with the Example 19-4 solution. Derived classes' implicitly generated copy assignment operators and destructors will still be valid, though.

Summary

To prevent derived classes from getting an implicitly generated default constructor, copy constructor, or copy assignment operator, the simplest and best choice is to make the base class have nonpublic (or absent) versions of those functions.

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 overhead, but more about that in a moment.

The obvious next question is: Who selects the memory management strategy?

Plotting Strategy

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?

Several possible layers of memory manager are involved, each of which might override the previous (lower-level) one:

- The operating system kernel provides the most basic memory allocation services. This underlying allocation strategy, and its characteristics, can vary from one operating systems platform to another, and this level is the most likely to be affected by hardware considerations.

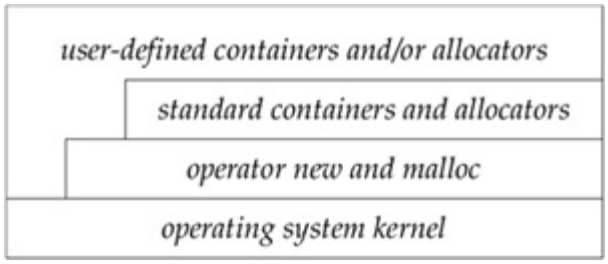
- The compiler's default run-time library builds its allocation services, such as C++'s `operator new` and C's `malloc`, upon the native allocation services. The compiler's services might just be a thin wrapper around the native ones and inherit their characteristics, or the compiler's services might override the native strategies by buying larger chunks from the native services and then parceling those out according to their own methods.

- The standard containers and allocators in turn use the compiler's services and possibly further override them to implement their own strategies and optimizations.

- Finally, user-defined containers and/or user-defined allocators can further reuse any of the lower-level services (for example, they might want to directly access native services if portability doesn't matter) and do pretty much whatever they please.

These levels are summarized in Figure 20-1.

Figure 20-1. Major levels at which memory management is done. Each level can typically be implemented in terms of lower level(s) adjacent to it as shown.



Thus memory allocators come in various flavors and can or will vary from operating system to operating system, from compiler to compiler on the same operating system, from container to container—and even from object to object, say in the case of a `vector<int>` object that uses the strategy implemented by `allocator<int>`, and a `vector<int, MyAllocator>` object that could express-mail memory blocks from Taiwan unless it's a weekday night and the Mets are playing or implement whatever other strategy you like.

Guideline

Know who does what: Understand the actual (and system-dependent) allocation strategies and responsibilities of your platform and standard library.

Summary

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, first through its allocation and deallocation primitives, then through its standard containers and allocators, and then to all the newfangled containers and allocators you can write yourself.

But when you ask for memory, what do you really know about what you get—and what it will cost? How much memory do the standard containers use—in theory, and in reality? Stay tuned, as we ask and answer precisely these questions in the next Item.

Previous Next

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

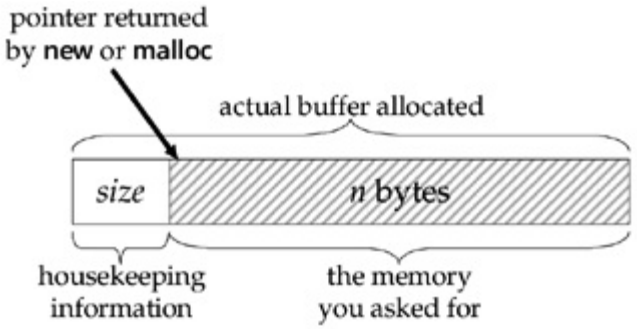
"I'll Take 'Operator New' for 200 Bytes, Alex"

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



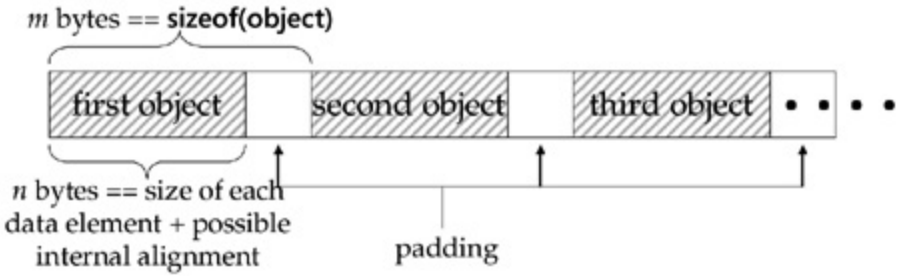
Of course, fixed-size allocation schemes (i.e., ones that return blocks of a given known size) don't need to store such overhead because they always know how big the block will be.

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)



For example:

```
// Example 21-1: Assume sizeof(long) == 4 and longs require 4-byte alignment
//
struct X1 {
  char c1;    // at offset 0, 1 byte
              // bytes 1-3: 3 padding bytes
  long l;     // bytes 4-7: 4 bytes, aligned on 4-byte boundary
  char c2;    // byte 8: 1 byte
              // bytes 9-11: 3 padding bytes (see narrative)
}; // sizeof(X1) == 12
```

In Figure 21-2's terms, n == 1 + 3 + 4 + 1 == 9 and m == `sizeof(X1)` == 12 in this example.[24] Note that all the padding contributes to `sizeof(X1)`. The at-the-end padding bytes might seem odd but are needed so that when you build an array of `X1` objects one after the other in memory, the `long` data is always 4-byte aligned. This at-the-end padding is the padding that's the most noticeable and the most often surprising for folks examining object data layout for the first time. It can be particularly surprising in this rearranged `struct`:

[24] Only a perverse implementation would add more than the minimum padding.

```
// Example 21-2: A rearranged version of Example 21-1
//
struct X2 {
  long l;     // bytes 0-3
  char c1;    // byte 4
  char c2;    // byte 5
              // bytes 6-7: 2 padding bytes
}; // sizeof(X2) == 8
```

Now the data members really are all contiguous in memory (n == 6),[25] yet there's still extra space at the end that counts toward m == `sizeof(X2)` == 8 and that padding is most noticeable when you build an array of `X2` objects. Bytes 6 and 7 are the padding highlighted in Figure 21-2.

[25] The compiler isn't allowed to perform the rearrangement of Example 21-1 to Example 21-2 by it-self, though. The standard requires that all data that's in the same `public:`, `protected:`, or `private:` group must be laid out in that order by the compiler. If you intersperse your data with access specifiers, then the compiler is allowed to rearrange the access-specifier-delimited groups of data to improve the layout, which is one reason why some people like putting an access specifier in front of every data member.

Incidentally, this is why when writing the standard it's surprisingly tricky to wordsmith the requirement that "`vector`s must be contiguous" in the same sense as arrays—in Figure 21-2, the memory is considered contiguous even though there are "gaps" of dead space, so what is "contiguous," really? Essentially, the individual `sizeof(struct)` chunks of memory are contiguous, and that definition works because `sizeof(struct)` already includes padding overhead.

The C++ standard guarantees that all memory allocated via `operator new` or `malloc` will be suitably aligned for any possible kind of object you might want to store in it, which means that `operator new` and `malloc` have to respect the strictest possible type alignment requirement of the native platform.

Alternatively, as described earlier, a fixed-size allocation scheme might maintain memory arenas for blocks of certain sizes that are multiples of some basic size m, and a request for n bytes will get rounded up to the next multiple of m.

Memory and the Standard Containers: The Basic Story

Now we can address the ultimate question of this Item:

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

Each standard container uses a different underlying memory structure and therefore imposes different overhead per contained object:

- A `vector<T>` internally stores a contiguous C-style array of `T` objects and so has no extra per-element overhead at all (besides padding for alignment, of course; note that here "contiguous" has the same meaning as it does for C-style arrays, as shown in Figure 21-2).

- 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; the actual page size isn't specified by the standard and depends mainly on how big `T` objects are and on the size choices made by your

standard library implementer. This paging requires the `deque` to store one extra pointer of management information per page, which usually works out to a mere fraction of a bit per contained object; for example, on a system with 8-bit bytes and 4-byte `int`s and pointers, a `deque<int>` with a 4K page size incurs an overhead per `int` of 0.03125 bits—just 1/32 of a bit. There's no other per-element overhead because `deque<T>` doesn't store any extra pointers or other information for individual `T` objects. There is no requirement that a `deque`'s pages be C-style arrays, but that's the usual implementation.

- 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 that 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 at least two pointers' worth of overhead per element.

- A `set<T>` (and, for that matter, a `multiset<T>`, `map<Key,T>`, or `multi-map<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?" The reason three are needed is that 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 enough. (Besides trees, other internal implementations of `set` are possible; for example, an alternating skip list can be used, which still requires at least three pointers per element in the `set`. See [Marrie00] for an example.)

Table 21-1 summarizes this basic overhead for each container. Note that smaller per-element costs are sometimes possible if you're willing to pay a higher per-iterator cost. That is, some of the bookkeeping can be moved into the iterators, creating fatter iterators in exchange for slimmer containers. I am not aware of any commercial implementation that uses this technique as of this writing.

Table 21-1. Basic overhead per contained object for various containers

Container	Typical housekeeping overhead per contained object
`vector`	No overhead per `T`.
`deque`	Nearly no overhead per `T`, typically just a fraction of a bit.
`list`	Two pointers per `T`.
`set, multiset`	Three pointers per `T`.
`map, multimap`	Three pointers per `pair<const Key, T>`.

Memory and the Standard Containers: The Real World

Now we get to the interesting part: Don't be too quick to draw conclusions from Table 21-1. For example, judging from just the housekeeping data required for `list` and `set`, you might conclude that `list` requires less overhead per contained object than `set`—after all, `list` stores only two extra pointers, whereas `set` stores three. The interesting thing is that this might not be true once you take into consideration the run-time memory allocation policies.

To dig a little deeper, consider Table 21-2, which shows the node layouts typically used internally by `list`, `set`/`multiset`, and `map`/`multimap`.

Table 21-2. Dynamic memory blocks used per contained object for various containers

Container	Typical memory block used per contained object
`vector`	None; objects are not allocated individually.
`deque`	None; objects are allocated in pages, and nearly always each page will store many objects.

`list`	`struct LNode {`
	`  LNode* prev;`
	`  LNode* next;`
	`  T object;`
	`};`

`set`, `multiset`	`struct SNode {`
	`  SNode* prev;`
	`  SNode* next;`
	`  SNode* parent;`
	`  T object;`
	`}; // or equivalent`

`map`, `multimap`	`struct MNode {`
	`  MNode* prev;`
	`  MNode* next;`
	`  MNode* parent;`
	`  std::pair<const Key, T> data;`
	`}; // or equivalent`

Next, consider what happens in the real world under the following assumptions, which happen to be drawn from a popular platform:

- Pointers and `int`s are 4 bytes long. (Typical for 32-bit platforms.)
- `sizeof(string)` is 16. Note that this is just the size of the immediate `string` object and ignores any data buffers the `string` may itself allocate; the number and size of `string`'s internal

buffers will vary from implementation to implementation, but doesn't affect the comparative results shown. (This `sizeof(string)` is the actual value of one popular implementation.)

- The default memory allocation strategy is to use fixed-size allocation where the block sizes are multiples of 16 bytes. (Typical for Microsoft Visual C++'s native allocator.)

Table 21-3 contains a sample analysis with these numbers. You can try this at home; just plug in the appropriate numbers for your platform to see how this kind of analysis applies to your own current environment. To see how to write a program that figures out what the actual block overhead is for allocations of specific sizes on your platform, see Appendix 3 of Jon Bentley's updated classic [Bentley00].

Table 21-3. Same actual overhead per contained object (implementation-dependent assumptions: `sizeof(string)` == 16, 4-byte pointers and `int`s, and 16-byte fixed-size allocation blocks)

Container	Basic node data size	Actual size of allocation block for node, including alignment and block allocation overhead
`list<char>`	9 bytes	16 bytes
`set<char>`, `multiset<char>`	13 bytes	16 bytes
`list<int>`	12 bytes	16 bytes
`set<int>`, `multiset<int>`	16 bytes	16 bytes
`list<string>`	24 bytes	32 bytes
`set<string>`, `multiset<string>`	28 bytes	32 bytes

Looking at Table 21-3, we immediately spy one interesting result: For many cases—that is, for about 75% of all possible sizes of the contained type `T`—`list` and `set`/`multiset` actually incur the same memory overhead in this particular environment. What's more, here's an even more interesting result: `list<char>` and `set<int>` have the same actual overhead in this particular environment, even though the latter stores both more object data and more housekeeping data in each node.

If memory footprint is an important consideration for your choice of data structure in specific situations, take a few minutes to do this kind of analysis and see what the difference really is in your own environment—sometimes it's less than you might think!

Summary

Each kind of container chooses a different space/performance tradeoff. You can do things efficiently with `vector` and `set` that you can't do with `list`, such as O(log N) searching;[26] you can do things efficiently with `vector` that you can't do with `list` or `set`, such as random element access; you can do things efficiently with `list`, less so with `set`, and more slowly still with `vector`, such as insertion in the middle; and so on. To get more flexibility often requires more storage overhead inside the container, but after you account for data alignment and memory allocation strategies, the difference in overhead might be significantly different than you'd think! For related discussion about data alignment and space optimizations, see also Item 26 in Exceptional C++ [Sutter00].

[26] If the `vector`'s contents are sorted.

Guideline

Know what you're getting: Understand the actual (and system-dependent) space costs of dynamic memory allocation and of the different containers.

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
				`std::bad_alloc`	

Plain	None	Yes	Yes (throws)		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.

Here is an example showing some ways to use these versions of `new`:

```
// Example 22-1(b): Using various indigenous and user-supplied overloads of new
//
new (FastMemory()) T;          // calls some user-supplied
                               // operator new(std::size_t, FastMemory&)
                               // (or similar, with argument type conversions),
                               // presumably to select a custom memory arena

new (42, 3.14159, "xyzzy") T; // calls some user-supplied
                               // operator new(std::size_t, int, double, const cha
                               // (or similar, with argument type conversions)

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

In each case shown in Examples 22-1(a) and 22-1(b), the parameters inside the brackets in the new-expression turn into additional parameters tacked onto the call to `operator new`. Of course, unlike the case in Example 22-1(a), the cases in Example 22-1(b) probably do allocate memory in one way or another rather than using some existing location.

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-supplie
                                // 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.

Previous Next

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()
```

But what if we wanted to use the global ones, at least in the first two cases, because the overload in the base class can't be a match anyway? The only reasonable way to re-enable the global ones is for `Derived` to provide the necessary passthrough functions itself—calling code must otherwise know to write globally qualified new-expressions to select a global `operator new`.

This leads to a few interesting conclusions, best expressed as a coding and design guideline. (Scott Meyers covers part of the first bullet in Item 5 of [Meyers97], but the other points are just as important.)

Guideline

If you provide any class-specific `new`, always also provide class-specific plain (no-extra-parameters) `new`.

The class-specific version should almost always preserve the global version's semantics, so declare it with an exception specification of `throw(std::bad_alloc)`, and prefer to implement it in terms of the global version unless you really need to put in some special handling:

```
// Preferred implementation of class-specific plain new.
//
void* C::operator new(std::size_t s) throw(std::bad_alloc) {
 return ::operator new(s);
}
```

Note that you might be calling a replaced global version, rather than the standard's default one, but that's normally a good thing: In most cases, a replaced global `operator new` exists for debugging or heap instrumentation reasons, and it's desirable to reflect such behavior in class-specific versions.

If you don't do this, you won't be able to use the class with any code that tries to dynamically allocate objects the usual way.

Guideline

If you provide any class-specific `new`, then always also provide class-specific in-place `new`.

This should always preserve the global version's semantics, so declare it to throw nothing, and implement it in terms of the global version:

```
// Preferred implementation of class-specific in-place new.
//
void* C::operator new(std::size_t s, void* p) throw() {
 return ::operator new(s, p);
}
```

If you don't do this, you will surprise (and break) any calling code that tries to use in-place `new` for your class. In particular, standard library container implementations commonly use in-place construction and expect such in-place construction to work the usual way; this is, after all, the only way to make an explicit constructor call in C++. Unless you write this, you probably won't be able to use even a `std::vector<C>`.

Guideline

If you provide any class-specific `new`, then 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`.

Either implement it using the global nothrow `new`:

```
// "Option A" implementation of class-specific nothrow new. Favors consistency
// with global nothrow new. Should have the same effect as Option B.
//
void* C::operator new(std::size_t s, const std::nothrow_t& n) throw() {
 return ::operator new(s, n);
}
```

Or, to ensure consistent semantics with the normal `new` (which the global nothrow `new` ought to do, but

what if someone replaced it in a way such that it doesn't?), implement it in terms of that:

```
// "Option B" implementation of class-specific nothrow new. Favors consistency
// with the corresponding class-specific plain new. Should have the same effect
// as Option A.
//
void* C::operator new(std::size_t s, const std::nothrow_t&) throw() {
 try {
  return C::operator new(s);
 }
 catch(...) {
  return 0;
 }
}
```

Note that these passthroughs can't be simulated with a `using` declaration, such as `using ::operator new;`. The only place such a `using` declaration would be helpful would be inside the definition for class `C`, but it's illegal there; within a class, you can only write `using` declarations that bring in names from base classes, not other names such as global names or names from other classes. Requiring that the calling code add the `using` declaration itself would not only be onerous, but wouldn't even help because we might not be able to modify it; some of the calling code would be inside what to us are read-only modules such as third-party libraries, or even inside the C++ standard library if we were trying to provide the standard containers with access to in-place `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`. Doi 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 real 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 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 th 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 mer available.

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

The question is whether this really buys us anything: Some people have had the mistaken idea that noth `new` enhances exception safety because it prevents some exceptions from occurring. So here's the $64,(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 an

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 w of reporting an error. Therefore, detecting and handling the failure is just a choice between checking fo 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 whic 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(); thr MyOwnException(); }` vs. something like `catch(bad_alloc) { HandleError(); throw MyOwnException(); }`). Neither way of `new` error reporting provides additional information or addit inherent safety, so neither way inherently makes programs somehow "safer" or "able to be more correc assuming of course that the handling is coded accurately.

But what's the difference to a calling program that doesn't check for errors? In that case, the only differ is that the eventual failure will be different in mode but not severity. Either an uncaught `bad_alloc` wi unceremoniously terminate the program (with or without unwinding the program stack all the way back `main`) or an unchecked null pointer will cause a memory violation and immediate halt when it's later dereferenced. Both failure modes are fairly catastrophic, but there's some advantage to the uncaught exception: It will make an attempt to destroy at least some objects and therefore release some resource and if some of those objects are things like a `TextEditor` object that automatically saves recovery information when prematurely destroyed, then not all the program state need be lost if the program is carefully written. (Caveat emptor: When memory really is exhausted, it's harder than it appears to writ code that will correctly unwind and back out, without trying to use a teensy bit more memory.) An abru halt due to use of a bad pointer, on the other hand, is far less likely to be healthy.

From this we derive Moral #1:

Guideline

Moral #1: Avoid nothrow `new`

Nothrow `new` does not inherently benefit program correctness or exception safety. For some failures—namely, failures that are ignored—it's worse than an exception, because at least an exception would get a chance to do some recovery via unwinding. As pointed out in the previous Item, if classes provide their own `new` but forget to provide nothrow `new` too, nothrow `new` will be hidden and won't e work. In most cases, nothrow `new` offers no benefit, and for all these reasons it should be avoided.

I can think of two corner cases where nothrow `new` can be beneficial. The first case is one that these d getting pretty hoary with age: When you're migrating a lot of legacy really-old-style C++ application c written before the mid-1990s that still assumes that (and checks whether) `new` returns null to report fai it can be easier to globally replace "`new`" with "`new(nothrow)`" in those files—but it's been a long tin now since unadorned `new` behaved the old way! The amount of such hoary old code that's still sitting a and hasn't yet been migrated to (or recompiled using) a modern compiler is dwindling fast.

The second case for using nothrow `new` is if `new` is being used a lot in a time-critical function or inner and the function is being compiled under a weaker compiler that generates inefficient exception-handli code overhead, and this produces a measurable run-time difference in this time-critical function betwe using normal `new` and using nothrow `new`. Note that when I say "measurably," I mean that we've actuall written a test harness that includes at least the entire piece of time-critical code (not just a toy example `new` by itself) and timed two versions, one with `new` and one with `new(nothrow)`. If after all that, we'v proved that it makes a difference to the performance of the time-critical code, we might consider `new(nothrow)`and should at the same time consider other ways to improve the allocation performar including the option of writing a custom `new` using a fixed-size allocator or other fast-memory arena (b before you do that, read Item 21).

This brings us to Moral #2:

Guideline

Moral #2: There's often little point in checking for `new` failure anyway.

This statement might horrify some people. "How can you suggest that we not check for `new` failure, or checking failures is not important?!" some might say, righteously indignant. "Checking failures is a cornerstone of robust programming!" That's very true in general, but alas it often isn't as meaningfu `new` for reasons that are unique to memory allocation, as opposed to other kinds of operations whose f should indeed be checked and handled. Hence, the third question:

When the Theoretical Rubber Meets the Real-World Road

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.

Here are some reasons why checking for `new` failure might not be as important as one might think, in particular situations:

1. Checking `new` failure is useless on systems that don't commit memory until the memory is used. O some operating systems, including but not limited to Linuxes,[29] memory allocation always succe Full stop.

[29] Lazy-commit is also a configurable feature on some other operating systems.

"Hey, wait a minute," you might rightly wonder. "How can allocation always succeed, even when the requested memory really isn't available?" The reason is that the allocation itself merely records a requ for the memory; under the covers, the (physical or virtual) memory is not actually committed to the requesting process, with real backing store, until the memory is actually used. Even when the memory used, often real (physical or virtual) memory is only actually committed as each page of the allocated

is touched, so it can be that only the parts of the block that have actually been touched get committed.

Note that, if `new` uses the operating system's facilities directly, then `new` will always succeed, but any l innocent code like `buf[100] = 'c';` can throw or fail or halt. From a standard C++ point of view, bo effects are nonconforming, because the C++ standard requires that if `new` can't commit enough memory must fail (this doesn't) and that code like `buf[100] = 'c'` shouldn't throw an exception or otherwise (this might).

Background: Why do some operating systems do this kind of lazy allocation? There's a noble and prag idea behind this scheme, namely that a given process that requests memory might not actually immedia need all of said memory—the process might never use all of it, or it might not use it right away, and in mean-time maybe the memory can be usefully "lent" to a second process which may need it only briefl Why immediately commit all the memory a process demands when it might not really need it right awa this scheme does have some potential advantages.

The main problem with this approach, besides that it makes C++ standards conformance difficult, is th makes program correctness in general difficult, because any access to successfully allocated dynamic memory might cause the program to halt. That's just not good. If allocation fails up front, the program k that there's not enough memory to complete an operation, and then the program has the choice of doing something about it, such as trying to allocate a smaller buffer size or giving up on only that particular operation or at least attempting to clean up some things by unwinding the stack. But if there's no way to know whether the allocation really worked, then any attempt to read or write the memory may cause a halt—and that halt can't be predicted, because it might happen on the first attempt to use part of the buf on the millionth attempt after lots of successful operations have used other parts of the buffer.

On the surface, it would appear that our only way to defend against this is to immediately write to (or from) the entire block of memory to force it to really exist. For example:

```
// Example 23-1: Manual initialization: Deliberately go and touch each byte.
//
char* p = new char[1000000000];
memset(p, 0, 1000000000);
```

If the type being allocated happens to be a non-POD[30] class type, the memory is in fact touched automatically for you:

> [30] POD stands for "plain old data." Informally, a POD means any type that's just a bundle of plain data, though possibly with user-defined member functions just for convenience. More formally, a POD is a class or union that has no user-defined constructor or copy assignment operator or destructor, and no (non-static) data member that is a reference, pointer to member, or non-POD.

```
// Example 23-2: Default initialization: If T is a non-POD, this code initializes
// the T objects immediately and will touch every (significant, non-padding) byte
//
T* p = new T[1000000000];
```

If `T` is a non-POD, each object is default-initialized, which means that all the significant bytes of each object are written to, so the memory has to be accessed.[31]

[31] This ignores the pathological case of a `T` whose constructor doesn't actually initialize the object's data.

You might think that's helpful. It's not. Sure, if we successfully complete the `memset` in Example 23-1 c `new` in Example 23-2, we do in fact know that the memory has actually been allocated and committed. accessing the memory fails, the twist is that the failure won't be what we might naively expect it to be: won't get a null return or a nice `bad_alloc` exception, but rather we'll get an access violation and a program halt, none of which the code can do anything about (unless it can use some platform-specific to trap the violation). That might be marginally better and safer than just allocating without writing and pressing on regardless, hoping that the memory really will be there when we need it and that all will b well, but not by much.

This brings us back to standards conformance, because it might be possible for the compiler-supplied `::operator new` and `::operator new[]` itself to do better with this approach than we as programm could do easily. In particular, the compiler implementer might be able to use knowledge of the operati system to intercept the access violation and therefore prevent a program halt. That is, it might be possi for a C++ implementer to do all this work—allocate, and then confirm the allocation by making sure w write to each byte, or at least to each page—and catch any failure in a platform-specific way and conv to a standard `bad_alloc` exception (or a null return, in the case of a nothrow `new`). I doubt that any implementer would go to this trouble, though, for two reasons: first, it means a performance hit, and probably a big one to incur for all cases; and second, `new` failure is pretty rare in real life anyway… w happens to lead us nicely to the next point:

2. In the real world, `new` failure is a rare beast, made nearly extinct by the thrashing beast. As a prac matter, many modern server-based programs rarely encounter memory exhaustion.

On a virtual memory system, most real-world server-based software performs work in various parts o memory while other processes are actively doing the same in their own parts of memory; this causes increasing paging as the amount of memory in use grows, and often the processes never reach `new` failu Rather, long before memory can be fully exhausted, the system performance will hit the thrash wall an grind ever more unusably slowly as pages of virtual memory are swapped in and out from disk, and th sysadmin will start killing processes.

Caveat lector: I use words like "most" because it is possible to create a program that allocates more a more memory but doesn't actively use much of it. That's possible but unusual, at least in my own exper This also of course does not apply to systems without virtual memory, such as many embedded and rea time systems; some of these are so failure-intolerant that they won't even use any kind of dynamic mem all, never mind virtual memory.

3. There's not always much you can do when you detect `new` failure. As Andy Koenig pointed out in

article "When Memory Runs Low" [Koenig96], the default behavior of halting the program on `ne` failure (usually with at least an attempt to un-wind the stack) is actually the best option in most situations, especially during testing.

Sometimes when `new` fails, there are a few things you can do, of course: If you want to record some diagnostic info, the `new` handler is a nice hook for doing logging. It is sometimes possible to apply the strategy of keeping a reserve "emergency memory" buffer; although anyone who does this should know they are doing and actually carefully test the failure case on their target platforms, because this doesn't necessarily work the way people think. Finally, if memory really is exhausted, you can't necessarily re being able to throw a nontrivial (e.g., non-built-in) exception; even `throw string("failed");` will usually attempt a dynamic allocation using `new`, depending on how highly optimized your implementati `string` happens to be.

So yes, sometimes there are useful things you can do to cope with specific kinds of `new` failure, but oft not worth it beyond letting stack unwinding and the `new` handler (including perhaps some logging) do t thing.

What Should You Check?

There are special cases for which checking for memory exhaustion and trying to recover from it do ma sense. Koenig mentions some in [Koenig96]. For example, you could choose to allocate (and if necess write to) all the memory you're ever going to use up front, at the beginning of your program, and then manage it yourself; if your program crashes at all, it will crash right away before actually doing work. approach requires extra effort and is only an option if you know the memory requirements in advance.

The main category of recoverable `new` failure error I've seen in production systems has to do with crea buffers whose size is externally supplied from some input. For example, consider a communications application where each transmitted packet is prepended with the packet length, and the first thing the receiver does with each packet is to read the length and then allocate a buffer big enough to store the r the packet. In just such situations, I've seen attempts to allocate monstrously large buffers, especially v data-stream corruption or programming errors cause the length bytes to get garbled. In this case, the application should be checking for this kind of corruption (better still, designing the protocol to prevei from happening in the first place, if possible) and aborting on invalid data and unreasonable buffer siz because the program can often continue doing something sensible, such as retrying the transmission wi smaller packet size or even just abandoning that particular operation and going on with other work. Af all, the program is not really "out of memory" when an attempt to allocate 2GB failed but there's still 1 of virtual memory left![32]

[32] Interestingly, allocating buffers whose size is externally specified is a classic security vulnerability. Attacks by malicious users or programs specifically trying to cause buffer problem is a classic, and still favorite, security exploit to bring down a system. Note that trying to crash the program by causing allocation to fail is a denial-of-service attack, not an attempt to actually gain access to the system; the related, but distinct, overrun-a-fixed-length-buffer attack is also a perennial favorite in the hacker community, and it's amazing just how many people still use `strcpy` and other unchecked calls and thereby leave themselves wide open to this sort of abuse.

Another special case where `new` failure recovery can make sense is when your program optimistically to allocate a huge working buffer, and on failure just keeps retrying a smaller one until it gets somethin fits. This assumes that the program as a whole is designed to adjust to the actual buffer size and does chunking as necessary.

Summary

Avoid using nothrow `new`, because it offers no significant advantages these days and usually has worse failure characteristics than plain throwing `new`.

Remember that there's often little point in checking for `new` failure anyway, for several reasons.

If you are rightly concerned about memory exhaustion, be sure that you're checking what you think you' checking, because:

- Checking `new` failure is typically useless on systems that don't commit memory until the memory i used.
- In virtual memory systems, `new` failure is encountered rarely or never because long before virtual memory can be exhausted, the system typically thrashes and a sysadmin begins to kill processes.
- Except for special cases, even when you detect `new` failure, there's not always much you can do i there really is no memory left.

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 correct`ness 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_cast`s inside member functions. Indeed, the code inside `f` itself might perform `const_cast`s 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."

How Writing const Really Can Optimize

3. Consider the following code:

```
// Example 24-3
//
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.

If the compiler knows that `z` truly is a `const` object, it could perform some useful optimizations even without global analysis. For example, if the body of `f` contains a call like `g(&z)`, the compiler can be sure that the non-mutable parts of `z` do not change during the call to `g`.

Other than that, however, writing `const` in Example 24-3 is not an optimization for most classes `Z` and in those cases where it is an optimization, it's not a compiler code generation optimization.

For compiler code generation, the question mostly comes down to whether the compiler could elide copy construction or could put `z` into read-only memory. That is, it would be nice if we knew that `z` was really untouched by what goes on inside `f`, because theoretically that would mean we might be able to just directly use the out-side object that the caller passed into this function instead of making a copy of it, or else we could make a copy but put that copy into read-only memory if that happens to be faster or otherwise more desirable.

In general, the compiler can't use the parameter's constness to elide the copy construction or assume bitwise constness. As already noted, too many things can go wrong. In particular, `Z` might have `mutable` members, or someone somewhere (in `f` itself, in some other function, or in some directly or

indirectly called `Z` member function) might perform `const_cast`s or other chicanery.

There is one case where the compiler might be able to generate better code. If:

- the definitions for `Z`'s copy constructor and for all of `Z`'s functions that are used directly or indirectly in the body of `f` are visible at this point; and
- those functions are all pretty simple and free of side effects; and
- the compiler has an aggressive optimizer

then the compiler can be sure of what exactly is going on every step of the way, and might choose to elide the copy under the "as if" rule, which says that a compiler is allowed to do something different from what the standard technically says it must as long as a conforming program can't tell the difference.

As an aside, it's worth mentioning one small red herring: Some people might say there's another case where the compiler could generate better code with `const`, namely, if the compiler performs global optimization. The thing is that that sentence is true even if you remove the "with `const`." Never mind that global optimization is still fairly rare and very expensive; the real issue here is that global optimization makes use of all knowledge about how an object is actually used and therefore will do the same thing whether the object is actually declared `const` or not—it makes decisions based on what you really do, not on what you promised to do, so it doesn't matter if the two happen to be the same thing. If you're getting true global optimization anyway, then promising constness definitely doesn't help the optimizer do its job any better in this respect.

Note that a few paragraphs ago I said that "writing `const` in Example 24-3 is not an optimization for most classes `Z`," and "for compiler code generation." There are, however, more optimizations possible than are dreamt of in your compiler's optimizer! And indeed `const` can enable some real optimizations:

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

In particular, there are also programmatic optimizations, where the author of `Z` can intelligently choose to do things a different way for `const` objects.

As a case in point, let's say that Example 24-3's `Z` is a handle/body class, such as a `String` class that uses reference counting to perform lazy copying:

```
// Example 24-4
//
void f(const String s) {

 // …

 s[4];       // or use iterators
```

```
 // …
}
```

Such a `String`, knowing that calling `operator[]` on a `const String` shouldn't be able to change the contents of the string, might decide to provide a `const` overload for `operator[]` that returns a `char` by value instead of a `char&`:

```
class String {
 // ...

public:
 const char operator[](size_t) const;
 char&      operator[](size_t);

 // ...
}
```

Similarly, `String` could also provide `const_iterator`s whose `operator*` returns a `char` by value instead of a `char&`.

If `String` does this work, and if you happen to use the smartened-up `operator[]` or iterators, and you declare the pass-by-value parameter as `const`, then—wonder of wonders!—the `String` can, with no further help from you, automagically and whole- somely optimize your code by avoiding a deep copy…

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

… but you get all this and more by simply writing pass-by-reference instead:

```
// Example 24-5: Just do it—better than Example 24-3
//
void f(const Z& z) {

 // …

}
```

and it works whether the object is handle/body or reference-counted or not, so just do that!

Guideline

Avoid passing objects by `const` value. Prefer passing them by reference to `const` instead, except only if they're cheap-to-copy objects such as `int`s.

Summary

It's a common belief that const correctness helps compilers generate tighter code. Yes, `const` is indeed a Good Thing, but the point of this Item is that `const` is mainly for humans, rather than for compilers and optimizers.

When it comes to writing safe code, `const` is a great tool that lets programmers write safer code with compiler checking and enforcement. Even when it comes to optimization, `const` is still principally useful as a tool that lets human class designers better implement handcrafted optimizations and less so as a tag for omniscient compilers to automatically generate better code.

Previous Next

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. 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?

Answer A: At Coding Time

At coding time, developers can incant the `inline` keyword in their programs. That's not really "performing" inlining in the sense of actually moving code around to eliminate a function call, but it is an attempt to choose the appropriate places for inlining to take place, so we'll consider that as the earliest opportunity to make decisions about inlining.[33]

[33] Arguably, another interpretation of "at coding time" is that some developers literally inline at coding time by physically moving blocks of source code around. That's even more of a stretch from the usual meaning of "inlining" and so I'll ignore it.

When you're tempted to write `inline` in your code, there are three important things to remember.

- By default, don't do it. Premature optimization is evil, and you shouldn't be tempted to write `inline` until after profiling demonstrates the need in specific cases. See Item 8 of More Exceptional C++ [Sutter02] or Google for "premature site:www.gotw.ca" for further harangues and dire warnings about premature optimization in general and premature inlining in particular.

- It only means "pretty please. " As described at length in Item 8 of More Exceptional C++ [Sutter02], the `inline` keyword is merely a hint to the compiler, a special hook in language to let you (try to) sweet-talk the compiler. (See below for the drawbacks of sweet talk.) And that's all it's good for: The `inline` keyword is not required to have any semantic effect whatsoever in a C++ program. It doesn't affect other parts of the standard language, in that writing `inline` on a function does not change how you use the function (for example, you can still take the function's address), and it is not possible for standardsconforming C++ code to programmatically detect whether a given function was declared as `inline` or not.

- It's often at the wrong level of granularity. We write `inline` on a function, but when inlining is performed, it actually is done to a function call. This distinction is important because the very same function can (and often should) be inlined at one call site but not at another. Writing `inline` does not give you any way to express that fact, because we can only write `inline` on a function itself, and therefore when we do so we are implicitly saying that we think we know that it is appropriate to inline this function at all possible call sites. That kind of prescience is rarely accurate. So although we often colloquially speak of "inlining a function," to be accurate it would be better to change our vocabulary to consistently talk about "inlining a function call."

Guideline

Avoid writing `inline` or other attempted optimizations until performance measurements show the need.

Answer B: At Compile Time

At compile time, compilers routinely perform precisely the kind of inlining described in Example 25-1.

What does the compiler do when we try to sweet-talk it by declaring certain functions to be `inline`? It depends. Not all compilers respond well to sweet talk, even when it's accompanied by chocolate and flowers. Your compiler (or other tools, as we will see) may ignore you, in three interesting ways:

- By refusing to inline calls to functions that you declared `inline`.

- By inlining calls to functions you didn't declare `inline`.

- By inlining some calls, but not others, to the same function (whether or not that function was declared `inline`).

Note again that Example 25-1 doesn't say `inline` anywhere. This was deliberate, because I wanted to illustrate that inlining can still happen. Indeed, don't be surprised if the compiler you're using today will in fact inline the function call in Example 25-1. Because you can't write a conforming program

that can tell the difference, this falls into the category of perfectly legitimate optimizations that a compiler can (and often should) perform on your behalf.

Modern compilers are usually better than programmers are at deciding which function calls to inline, including whether to perform inline expansion of the same function at some call sites but not at others. Why? The simplest reason is that the compiler has more context because it knows the "real" structure of the call site—the machine code actually generated for the call site after other optimizations, such as loop unrolling and dead branch elimination, have already been performed. For example, the compiler might be able to detect that inlining a function call in a certain inner loop would make the loop too large to fit into cache, which would slow down performance, and elect not to inline that call site while still inlining other calls to the same function.

Answer C: At Link Time

Now we start to get into the more interesting, and more modern, aspects of inlining.

Question: Can a function be inlined at link time? Answer: Yes. This gets to the heart of the for-extra-marks question posed at the outset: What kinds of functions are guaranteed to never be inlined? I posed this extra question because there's a common belief that certain kinds of functions can't be inlined. In particular, functions whose definitions are not put into header files but put into separate modules are commonly thought to be uninlineable.

So let's try to make inlining as hard as we can. Consider this slight variation to Ex- ample 25-1:

```
// Example 25-2: Make life hard for the optimizer: Put the function in a
// different module, and make the function's source definition unavailable.
//

//--- file main.cpp ---
//
double Square(double x);

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

//--- file square.obj (or .o) ---
//
// contains the compiled definition for:
// double Square(double x) { return x * x; }
```

The idea here is that the implementation of `Square` has been moved out of the `main.cpp` translation unit. In fact, more than that: `Square` is not even available in source form, only in object form. "Certainly this call to `Square` is guaranteed to never be inlined!" some might be tempted to exclaim. As far as the compiler goes, they would be correct. While compiling `main.cpp`, even an inordinately precocious compiler could not possibly peek at the definition of `Square`.

A precocious linker, however, could, and some popular commercial implementations do. Several compiler products, including Hewlett-Packard's implementation, have supported such cross-module inlining; one popular product that supports it today is Microsoft Visual C++ version 7.0 (aka ".NET") and higher, using the /LTCG switch, which stands for "link time code generation." One real advantage of doing such late inlining is, again, that the tool knows more of the actual context of each call site and can make smarter choices about where and when it's worth inlining a given call.

But wait, there's more: Did you notice that there's nothing in the description of Example 25-2 that says `Square` is necessarily written in C++? This illustrates a second advantage of post-compilation inlining: It is language-neutral. `Square` could just as easily have been written in Fortran or Pascal. Sweet. All the linker has to do is be aware of the parameter-passing convention and then remove the parameter-pushing and -popping code along with the program jump instruction.

But wait (again), there's more—much more. Fasten your seat belts, because it turns out that even now we're still just getting started.

Answer D: At Application Installation Time

Fast-forward now to the joyous day when we've compiled and linked our application, bundled it all happily into a tarball or setup, and proudly shipped the shrink- wrapped CD to our first customer. Surely now it's time to:

a. break open the morale fund;

b. have a ship party; and

c. declare that all opportunities for inlining are past.

Right?

Yes, yes, and no, respectively.

Particularly since the mid-1990s, increasing numbers of shipping applications are targeted for managed run-time environments. That is, instead of being compiled to machine code that's specific to a particular chip and to API calls that are specific to a particular operating system, the applications are compiled to a bytecode stream that will be interpreted or compiled by a run-time environment on the user's machine that abstracts away some or all CPU and OS facilities. Common examples include, but are not limited to, a Java Virtual Machine (JVM) and the .NET Common Language Run-time (CLR).[34] When targeting these environments, the compiler translates the original C++ source code into said bytecode stream (also known as the virtual machine's instruction language, or IL) that represents a program written as opcodes in the run-time environment's instruction set.

[34] And the CLR's cousins, such as Mono, DotGNU, and Rotor, that also implement the ISO Common Language Infrastructure (CLI) standard which specifies a subset of the CLR.

Aside: Some of these environments have an instruction set so rich that high-level object-oriented concepts, including classes and inheritance and virtual functions, have direct first-class support. A compiler that is targeting such a platform can (and many do) translate the source program into the instruction language on a straight class- for-class, function-for-function basis, possibly after applying some optimizations of its own, including performing some function-call inlining early at compile time. If the compiler does this, then a C++ function in the source code can be more or less directly represented by a function having the same signature, in the target instruction set. Of course, a compiler doesn't have to mirror things that closely, and even if it doesn't necessarily do it that way, the following inlining notes will still apply.

Back to the topic at hand: What does this have to do with inlining at application installation time? Well, even in managed environments, eventually the CPU on the user's machine has to be fed instructions in its native instruction set. Therefore, the managed environment is responsible for translating the instruction language into native machine code that the local CPU can understand. This is often done when the application is first installed, and at this point, just as in any other compilation-like process, further optimizations can be (and frequently are) applied. In particular, .NET's NGEN IL compiler performs some inlining at application installation time when some or all of the installed IL is translated into native instructions ready for execution.

Again, it's worth noting that some of these managed environments are language-neutral, so that the optimizations (including inlining) being performed at application install time can be applied across language boundaries. Don't be too surprised if your C# program makes a call to a small C++ function, and the call ends up being inlined.

So, when is it too late to perform inlining? Never say never, because even now the story is not quite over…

Answer E: At Run Time

All right, enough is enough: Surely by the time we hit run-time, we must be well be- yond opportunities for code optimizations. Right?

It might seem impossible that inlining can still be performed at run-time, but in fact there are several ways it can be done. In particular, I want to mention profile-directed optimization and guarded inlining. Like a JIT-compiled environment (see the previous and next sections), this requires some tool support to exist on the user's machine at run-time.

The idea behind profile-directed optimization is that when the application is actually run, instrumentation hooks inserted into the executing program can gather data about how the program is actually being used—in particular, what functions are be- ing called heavily and under what conditions (e.g., the size of the working set com- pared to the total cache memory when the function is called). The data gathered from these instrumented hooks can be used to modify the executable image so that selected function calls can be inlined to tune the application to its target environ- ment based on actual run-time performance measurements.

Guarded inlining is another example of how aggressive the run-time inline optimizations can be. In particular, [Arnold00] and [JikesRVM] document the Jikes Re- search Virtual Machine (RVM), née the Jalapeño dynamic optimizing compiler, for JVM targets. Among other things, this compiler is able to inline calls to virtual functions by assuming that the receiver of the virtual call will be of a given declared type (to avoid not only the cost of the function call but also the extra expense of virtual dispatch). Now, compilers can already routinely nonvirtualize (and therefore also optionally inline) certain virtual function calls today, if the type of the target is statically known. What's new here is that the Jikes/Jalapeño environment can speculatively nonvirtualize and inline calls to virtual functions even if the static type of the target is not known. Because the guess might not be right, however, the compiler inserts a guard that performs a run-time check that validates that the target object's type is what was expected; if it's not, execution falls back to performing a normal virtual function call.

Answer F: At Some Other Time

Finally, I'll add one "other time" example I can think of, which is similar to some of the others but distinct enough that I'll give it its own section.

Recall that in answer (d) we considered inlining that happens when installing applications on certain managed run-time environments, such as a JVM or .NET CLR environment. Of course, Astute Readers will already have noticed that earlier I only mentioned translation from bytecode to native machine instructions at application installation time, but there's another and more common time when that translation takes place, namely at JIT time, where JIT refers to "just-in-time" compilation.

The idea behind JITting is to compile functions "just in time," just before they're about to be used. This has the advantage of amortizing the cost of compiling the program down to native code, because instead of one big compilation step, you get lots of little compilations for individual parts of the code, just in the nick of time as they're about to be executed. It has the corresponding disadvantages of potentially making the first runs of a program somewhat slower and of reducing the quality of optimization because the JIT has to be fast and can't afford to spend a lot of time analyzing inlining and other optimization opportunities. A JITter can still perform optimizations like inlining, and many JITters do, but we can generally get better re- sults by doing the same work earlier, say at application installation time (see Answer D) when we're not so time-sensitive and the optimizer can afford to be less thrifty with machine cycles and spend a little more time on analysis.

Guideline

Inlining can happen anytime.

Summary

Like all optimizations, inlining is frequently better when performed by tools that are aware of the

generated code and/or the execution environment rather than by the programmer. The later inlining is performed, the more specific and targeted it can be.

We talk about inlining functions, but it's more correct to say that we perform inlin- ing on function calls. After all, the same function might be inlined in one place but not in others. And, because of the many opportunities that exist for inlining even well after initial compilation has finished, the same function can be inlined, not only in different places, but by different tools in each place.

There's more to inlining than the `inline` keyword alone.

Previous Next

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
R	Rook
B	Bishop
N	Knight
P	Pawn
rank	row on the chessboard, typically numbered 1 (White's home row) to 8 (Black's home row)
file	column on the chessboard, typically numbered a (left, from White's point of view) to h (right)

The questions:

> 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

One representation that would take this amount of space would be to assume that the program already knows the current board position (which it can deduce from the previous moves) and store the entire new board position, using two bytes per square. In this case, we are mimicking one of the standard online notations, which uses a `'W'` or `'B'` or `'.'` to designate the side that owns the piece in the given square, and a `'K'`, `'Q'`, `'R'`, `'B'`, `'N'`, `'P'`, or `'.'` to designate the type of piece in the given square.

Using this scheme and storing the board from rank 1 to rank 8 and file a to file h within each rank, one possible half-move representation might be:

```
WRWNWBWQWKWBWNWRWPWPWP..WPWPWPWP.....  WP............
     ....................BPBPBPBPBPBPBPBPBRBNBBBQBKBBBNBR
```

If this is the first move, it represents "1. d4" (or, "1. P-Q4"), my usual opening move.

b. 32 bytes per half-move

The representation in (a) seems a little wasteful, because it's in ASCII text that humans can read whereas we really only need a machine-readable format. After all, the software running in front of the chess database is going to take care of displaying positions to the user.

We can get down to 32 bytes per half-move by keeping the basic strategy of storing the entire new board position, but this time storing only 4 bits for each square: we need 3 bits to store the piece type (e.g., `0` for K, `1` for Q, `2` for R, `3` for B, `4` for N, `5` for P, or `6` for none, which requires 3 bits although it wastes two possible values), and 1 bit to store the color (which can be ignored if there is no piece on the square).

Using this scheme, and storing the board from rank 1 to rank 8, and file a to file h within each rank, one possible half-move representation might consume this many bytes:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

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

We can achieve this by storing the half-move as text in old-style chess notation.

Old-style "descriptive" chess notation identifies squares using variable-length tags like K3 and QN8 instead of using two-character tags like e3 and b8. To write down a half-move this way requires at least 4 characters (e.g., P-Q4) and possibly as many as 8 characters (e.g., RKN1-KB1, P-KB8(Q)). Note that no extra trailing null or other delimiter is needed, because the move format can be decoded unambiguously.

Using this scheme, one possible half-move representation might be:

P-KB8(Q)

c. minimum 2 bytes and maximum 4 bytes per half-moves

We can achieve this by storing the half-move as text in modern chess notation.

Modern "algebraic" chess notation is more compact, and any half-move can be written using at least 2 characters (e.g., d4) and at most 4 characters (e.g., Rgf1, gh=Q).

Again, no special move delimiter is needed because the format can be decoded unambiguously.[35]

[35] Incidentally, a major advantage of this representation outside the computing world is that it can be written down quickly on paper by a human, even under time pressure. The reduction from a maximum of 8 characters to a maximum of 4 characters, coupled with some improved conceptual simplicity, turns out to make a big difference to users—also known as players.

Using this scheme, one possible half-move representation might be:

gh=Q

e. 12 bits per half-move

We can get more compact still by taking a different approach: What if we were to store just the moving piece's origin and destination squares? To encode one square location requires 6 bits because there are 64 possibilities, so to encode two square locations to allow for both the origin and the destination to be recorded requires 12 bits. That suffices for usual moves; however, in the case of a pawn promotion, this scheme will need more than 12 bits.

That's already a lot better than the earlier attempts. Let's put the compression front on hold for just a moment, though, and begin the next Item by considering how we might create auxiliary data structures to store such bits of information that won't play nice and fall on even byte boundaries.

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 char`s, 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 going wrong, plus half a dozen sessions in the debugger stepping through code, to determine and fix all the problems.

Here's the result. I don't claim it's perfect, but it passed the unit tests I threw at it, including single- and multi-byte appends and boundary cases. (You always write unit test harnesses for your code, right? And make sure your code passes them all, before you check the code in?) Note that this version of the code operates on chunks of bytes at a time—for example, if we're using 8-bit bytes and have an offset of 3 bits, we'll copy the first 3 bits as a unit and copy the last 5 bits as a unit, for two operations per byte. For simplicity, I also require the user to provide buffers that are a byte bigger than might otherwise be necessary, just so that I can simplify my own code by allowing a little running off the end.

```
// Example 27-1: BitBuffer implemented in terms of
// vector<unsigned char>. Hard, finicky work. Ugh.
//
class BitBuffer {
public:
 BitBuffer() : buf_(0), size_(0) { }

 // Append num bits starting with the first bit of p.
 //
 void Append(unsigned char* p, size_t num) {
  int bits = numeric_limits<unsigned char>::digits;

  // first destination byte & bit offset
  int dst = size_ / bits;
  int off = size_ % bits;

  while(buf_.size() < (size_+num) / bits + 1)
   buf_.push_back(0);

  for(int i = 0; i < (num+bits-1)/bits; ++i) {
   unsigned char mask = FirstBits(num - bits*i);
  buf_[dst+i] |= (*(p+i) & mask) >> off;
  if(off > 0)
    buf_[dst+i+1] = (*(p+i) & mask) << (bits - off);
 }

 size_ += num;
}
```

```
// Query #bits in use (initially zero).
//
size_t Size() const {
  return size_;
}

// Get num bits starting with the start-th bit (0-based), and store the result
// starting with the first bit of dst. The buffer pointed at by dst should be at
// least one byte bigger than the minimum needed to hold num bits.
//
void Get(size_t start, size_t num, unsigned char* dst) const {
 int bits = numeric_limits<unsigned char>::digits;

 // first source byte & bit offset
 int src = start / bits;
 int off = start % bits;

 for(int i = 0; i < (num+bits-1)/bits; ++i) {
  *(dst+i) = buf_[src+i] << off;
  if(off > 0)
    *(dst+i) |= buf_[src+i+1] >> (bits - off);
 }
}

private:
 vector<unsigned char> buf_;
 size_t size_; // in bits

 // Build a mask where the first n bits are 1 and the rest are 0.
 //
 unsigned char FirstBits(size_t n) {
  int num = min(n, numeric_limits<unsigned char>::digits);
  unsigned char b = 0;
  while(num-- > 0)
   b = (b >> 1) | (1 << (numeric_limits<unsigned char>::digits-1));
  return b;
 }
};
```

This code is nontrivial. Take some time to read it and to convince yourself that it's doing the right thing. (If you think you've found a bug, first write a test harness that attempts to demonstrate the bug; once the bug has been confirmed, please do go ahead and send me both the bug report and the test harness that tickles the problem behavior.)

Attempt #2: Reusing a Standard Bit-Packed Container

The second idea is to note that the standard library already includes two containers that store bits: `bitset` and `vector<bool>`. Now, `bitset` is a bad choice simply because `bitset<N>` has fixed length `N` and we'll be encoding variable-length bitstreams. No dice. Here `vector<bool>`, for all its other faults, is a tempting choice and in this case turns out to be just what the doctor ordered. (Of

course, the standard doesn't actually require that `vector<bool>` implementations must use packed storage, it just encourages it; but most implementations do.)

The most important thing I can say about the following code is this:

> The Example 27-2 code was essentially correct on first writing.

Yes, really. Between my first compile and the final code, all I did was fix a few syntax typos, in particular to add a missing semicolon (these are, after all, things the compiler is supposed to find for you) and add parentheses in two places where I'd for-gotten that `%` has higher precedence than `+`. That's it.

```
// Example 27-2: BitBuffer implemented in terms of vector<bool>
//
class BitBuffer {
public:
 // Append num bits starting with the first bit of p.
 //
 void Append(unsigned char* p, size_t num) {
  int bits = numeric_limits<unsigned char>::digits;
  for(int i = 0; i < num; ++i) {
   buf_.push_back(*p & (1 << (bits-1 - i%bits)));
   if((i+1) % bits == 0)
     ++p;
   }
 }

  // Query #bits in use (initially zero).
  //
  size_t Size() const {
   return buf_.size();
 }

 // Get num bits starting with the start-th bit (0-based), and store the result
 // starting with the first bit of dst.
 //
 void Get(size_t start, size_t num, unsigned char* dst) const {
  int bits = numeric_limits<unsigned char>::digits;

  *dst = 0;
  for(int i = 0; i < num; ++i) {
   *dst |= unsigned char(buf_[start+i]) << (bits-1 - i%bits);
   if((i+1) % bits == 0)
     *++dst = 0;

  }
 }

private:
 vector<bool> buf_;
};
```

That writing this version was so much easier than writing Example 27-1 shouldn't be surprising. This version reuses existing bit-fiddling code instead of writing its own, it uses about 50% fewer lines of code—and it's disproportionately less buggy as a result. It's also cleaner: This time I didn't even have to ask the caller to supply a bit[36] of extra output space just to make my `Get` code simpler, as I did in the first version.

[36] Pun unintended.

I suspect that both solutions, especially the first, could probably be further improved—there might be bugs I didn't detect, and maybe the code could be simplified a bit[37] in ways I didn't see—but I think they're pretty close to optimal in terms of both correctness and style.

[37] Pun still unintended.

The Big Squeeze

Let's take one final look at the compressed representation of a chess game and see if there's anything more we can do to squeeze it down.

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?

Yes, but if you're going to represent them in code you need a bit-twiddling container like `BitBuffer`.

For example, here are three ways:

We can get down to 10 bits per half-move by encoding the destination square and the piece that moved there. Encoding a square requires 6 bits, as before. Encoding which piece moved there can be done by simply identifying the number of the piece, assigning an arbitrary ordering to the squares, say from rank 1 to rank 8 and file a to file h within each rank, and numbering the pieces in the order their current squares appear in that ordering. There can be only 16 pieces on the board, so the piece can be identified using 4 bits, for a total of 10 bits.

Could we do better still? Let's reason it through: We can encode all possible squares as destination, but usually only a minority of squares could actually be moved to with a legal move, so there must be some redundancy left in that part of our encoding. Similarly, we can represent all pieces moving to the given square, even though almost certainly not all pieces could move to that square—indeed, some pieces might not even have a legal move at all to any square—so somehow we're probably encoding more than we need to encode. For example, we could compress the second part further by storing from 0 to 4 bits to identify the piece that moved: There are 1 to 16 possibilities, and if there is only one piece that could move to the square then we don't need to encode any bits at all. On decoding, we know how many possible pieces could have moved to the square, so we know how many bits to pull from the input format for that half-move.

We can get down to an encoding that uses at least 0 and at most 8 bits per half-move as follows: First,

invent an ordering of legal moves; for example, we could order the pieces according to their squares as before, and for each piece order its possible moves as possible destination squares according to the same square ordering. Then, store the number of the actual move made using the minimum number of bits required; for example, the opening position has 20 legal moves, and to store them as a plain binary number requires ceiling($\log_2(20)$) = 5 bits.

The result is that we need a minimum of 0 bits to represent each half-move. Zero bits are needed if there is only one possibility, a forced move. But how many bits could be needed in the worst case? This corresponds directly to the question: How many moves could there be in a legal chess position? As far as I have been able to determine, the current known maximum is 218 in the following position:

In this worst case, 8 bits are needed to encode the move as a plain binary number. On average, probably 5 bits will be required to store a typical move; the opening position has 20 moves, and a typical endgame with the side to move having, say, K+R+2P on an open board can yield about 30 legal moves if the pawns are getting ready to promote, both of which cases require 5 bits to store, using this method.

Thinking about this briefly should convince us that this encoding ought to be pretty close to optimal because it is representing directly and exactly the answer to the question at hand: "Which legal move was made?" We are using the minimum number of bits to represent the possibilities for any given move as a plain binary number, with full knowledge of what has gone before.

Can we do better still? Yes, although now we'll start to see diminishing returns as the further gains become incremental. This is because further gains rely on having more knowledge and/or saving only partial bits. To illustrate how we could do a bit better still, consider that the opening position has 20 moves, which under the previous scheme we would store using ceiling($\log_2(20)$) = 5 bits. Really that choice of first move theoretically holds only $\log_2(20)$ = 4.3 bits of actual information, even assuming that all moves are equally likely, and on average we should require even fewer bits because the two most popular opening moves for White account for the majority of all chess games. In brief, if we can additionally gain knowledge about the relative probabilities of each move (for example, by building into the compression engine a deterministic chess-playing program that can guess which moves are better or more likely than others for any given position), then we could use variable-length encodings such as Huffman and arithmetic compression that use fewer bits to store the more likely moves. This trades off computing time, using domain-specific knowledge in return for better compression.

Summary

This Item shows how domain-specific knowledge can be applied to make a significant difference in the solution of a given problem.

In summary, even without any knowledge of which moves are more likely in any given position, a typical 40-move (80-half-move) game can be stored in about 50 bytes. That's pretty small, and it's possible only by applying knowledge of the problem domain to arrive at an optimally optimized

solution.

Guideline

Optimize based only on solid information: a) that you should optimize; and b) how you should optimize. There's no substitute for domain expertise.

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 `float`ing 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 statements we knew and loved in Example 28-1(a)? Or are they uses of the type `if`, which happens to helpfully have a suitable constructor, in which case the statements mean to create two unnamed temporary objects? After thinking about this question for a few seconds, I hope you'll quickly come to the conclusion that not even a human could know the answer for sure in this case, never mind the general case. And if a human can't know, what more could we reasonably expect from a compiler?

"But wait," the person still trying to force his way down the dirty little road might say, "We can still invent a rule for this and make it work! In lines 4 and 5, creating a temporary just to destroy it again isn't very useful, so we can just arbitrarily decide it's not what the programmer must have meant, and that therefore it must be a plain old conditional statement." I hope that you've recoiled in some combination of horror, disgust, shock, and dismay at the very suggestion of such a filthy hack, but let's pursue it long enough to note two killer objections that blast such a hack to smithereens: 1. One could just as easily say that writing "`if(true);`" is a no-op and couldn't possibly be what the programmer meant, so we should treat both statements as declaring a temporary object. 2. Whichever way you choose, you're in the situation where lines 4 and 5 have an utterly different meaning depending solely on whether there happens to be such a class `if` in scope or not, and that would be disgraceful.

Having ad-hoc hacked-up special-case foul-smelling rules like that ought to be a big flashing red light that's warning of a serious design problem. Indeed, it is.

There are, of course, other ways to create such ambiguities if keywords are not specially reserved. Example 28-1(c) illustrates another simple way:

```
// Example 28-1(c): Not legal C++, but what if it were?
//
class SomeFunctor {
public:
 int operator()(bool) { return 42; }
};

SomeFunctor if;   // Let's call the variable "if" (not legal, but what if it were
```

```
// Now let's go back to that code again…
int main() {
 if(true);          // 6: Hmm… what does this mean?
 if(42);            // 7: Hmm… and this?nnn
}
```

Here again, what would lines 6 and 7 mean? Are they the same old plain-jane conditional statements we knew and loved in Example 28-1(a)? Or are they uses of the variable `if`, which happens to helpfully understand `operator()` ☺hey, it even takes compatible parameters!☺in which case the statements mean `if.operator()(true);` and `if.operator()(42);`? I think it's clear that it's next to impossible for even a human like you or me to come up with a sane rule to decide what this ought to mean, and if a human can't know, he can't write a compiler that knows.

It's clear that C++ (like other languages) does indeed need to firmly nail down the meanings of some names. It needs to reserve the names for the language's own use, so such things are called reserved words.

Our Rather Reserved Cast: The Keywords

The C++ standard reserves 63 names as keywords. I've listed them in Table 28-1. Most or all of those names should be familiar to us. We use most of them daily.

Table 28-1. Standard C++ keywords

`asm`	`do`	`if`	`return`	`typedef`
`auto`	`double`	`inline`	`short`	`typeid`
`bool`	`dynamic_cast`	`int`	`signed`	`typename`
`BReak`	`else`	`long`	`sizeo`	`union`
`case`	`enum`	`mutable`	`static`	`unsigned`
`catch`	`explicit`	`namespace`	`static_cast`	`using`
`char`	`export`	`new`	`struct`	`virtual`
`class`	`extern`	`operator`	`switch`	`void`
`const`	`false`	`private`	`template`	`volatile`
`const_cast`	`float`	`protected`	`this`	`wchar_t`
`continue`	`for`	`public`	`throw`	`while`
`default`	`friend`	`register`	`TRue`	
`delete`	`goto`	`reinterpret_cast`	`try`	

On top of that, 11 of the operators and punctuators can be spelled out as words instead of in their usual form; for example, you can write `and` instead of `&&` in a conditional expression. The standard

reserves those names too, so that you can't use them for your own names; see Table 28-2.

Table 28-1. Standard C++ keywords

`and`	`and_eq`	`bitand`	`bitor`	`compl`	`not`
`not_eq`	`or`	`or_eq`	`xor`	`xor_eq`	

That's 74—count 'em—74 specific names your program is not allowed to use for its own purposes, such as for the names of types, functions, or variables (see [C++03, §2.11]).

Keywords: The Lesser Ones

Most of these keywords do something. That's good—otherwise, why have them?

Some keywords, however, don't do nearly as much as one might hope. In fact, several have no semantic impact on your program at all—really. I mean it. That's right, some keywords are semantically equivalent to whitespace, a glorified comment. In particular, I have three in mind: `auto`, `register`, and, in many respects, `inline` (see Item 25). (For those who may wonder whether the keyword `export` has any effect in theory and/or practice, we have Items on that too; see Item 9 and Item 10.)

auto

Consider first poor `auto`:

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

In short: Not at all. `auto` is an entirely redundant storage class specifier. It can only appear on the names of objects declared in a code block and designates that those objects are automatically destroyed when their function or block ends; but in all the cases where `auto` can appear, it's implied anyway if it's not written, and that's what makes it redundant. In short, `auto` is exactly as meaningful as whitespace.

Now, at this point, some astute readers of the C++ standard might pipe up and say in a high, shrill voice, "But that's not quite what the Standard says! Why, it even says, in a note, that `auto` is only almost always redundant!" And so it does:

> …the `auto` specifier is almost always redundant and not often used; one use of `auto` is to distinguish a declaration-statement from an expression-statement explicitly.
>
> —[C++03] §7.1.1

Yes, that's what the Standard says, but no, it's not correct. (I've submitted a defect report to correct the non-normative note.) Why not? The rule in C++ that specifically deals with such ambiguity is that

anything that can possibly be a declaration must be a declaration; adding `auto` never changes that. For example:

```
// Example 28-2: auto does not disambiguate.
//
int i;
int j;
int main() {
int(i);            // declares i; not a reference to ::i
auto int(j);       // still declares j; not a reference to ::j
 int f();            // a function declaration, not a default-constructed int var
 auto int f();       // still a function declaration, though this time one that
}                    // will get an error on strict compilers
```

For further discussion of the declaration ambiguity in C++, turn to Item 29 (see also Item 39 of [Meyers01]). In sum, `auto` cannot be used to disambiguate any such ambiguity.

Guideline

Never write `auto`. It's exactly as meaningful as whitespace.

Aside: In the future, `auto` might become meaningful. As the C++ committee works on the next version of the C++ standard, colloquially known as C++0x, it keeps casting an eye toward `auto` as a keyword to reuse for "auto" matic type deduction so that in time we might be able to replace declarations like this:

```
vector<SomeNamepace::SomeType>::const_iterator i = v.begin();
```

with just this:

```
auto i = v.begin(); // illegal today, but a possible future extension for C++
```

register

Enough about `auto`. What about `register`? Let's ask:

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

In short: Not at all, on most modern compilers.

To see why, consider what the C++ standard has to say immediately following the quoted note about `auto`… it begins:

> A `register` specifier has the same semantics as an `auto` specifier…

Uh, oh. According to what we've just discovered, that would mean "no semantics." Not an auspicious start. Forging ahead, the text continues:

> …together with a hint to the implementation that the object so declared will be heavily used. [Note: the hint can be ignored and in most implementations it will be ignored if the address of the object is taken.]end note]
>
> [C++03] §7.1.1

The idea behind `register` is that if some variables are going to be heavily used, then it makes sense to put them in physical CPU registers whenever possible, which lets them be operated on much faster than if they need to be fetched from (relatively) slow cache memory or, worse still, from main memory.

That's fine as far as it goes, but it doesn't go as far as the programmer.

You should almost never want to write `register`. These days, the idea of having the programmer pepper the code with register allocation hints is a wild goose chase more than ever it has been, because it's virtually impossible for even the best programmer (that's you) to come up with the best allocation of registers to make his code run fastest. Even if the programmer knows the exact chip his code will run on (which is rare) and knows it as well as his compiler's code generation development team (which is unlikely in the extreme), the programmer can never assign objects to registers as well as a good compiler can do it because the programmer has no idea what other transformations for example, inlining, loop unrolling, dead branch elimination, variable folding have already been performed by the time the code generator sees the code and can start to decide what parts of what's left will benefit most from register use. Not only can't you do as good a job as your compiler, but you shouldn't want to this sort of thing is just what automated tools are for, not to mention much better at.

Guideline

Never write `register` (unless you know you're on a compiler, and in code, where it will actually matter). On most compilers, it's exactly as meaningful as whitespace.

Summary

We've seen why the C++ language treats keywords as reserved words, and we've seen two keywords `auto` and `register` that make no semantic difference whatso-ever to a C++ program.

Don't use them; they're just whitespace anyway, and there are faster ways to type whitespace.

Never write `auto`. It's exactly as meaningful as whitespace.

Never write `register` (unless you know you're on a compiler, and in code, where it will actually matter). On most compilers, it's exactly as meaningful as whitespace.

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 `string`s 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 a 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 t

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;

// append elements using push_back
copy(coll1.begin(), coll1.end(), back_inserter(coll2));
```

The (minor) difference is that `coll2`'s default constructor is called first and then the elements are pushed into the collection as a separate step, using `push_back`. The original code simply did it all usi the constructor that takes an iterator pair, which probably (though not necessarily) does exactly the sa thing under the covers.

You might wonder why I've belabored this syntax. The reason will become clear as we take a look at the last part of the code, which is completely benign—and "benign" is usually a good thing, but unfortunately in this case it's much more benign than some might think or actually want:

```
// Example 29-2(c): Declaring yet another deque?
//
deque<string> coll3(istream_iterator<string>(cin), istream_iterator<string>());
```

This code looks at first blush like it's trying to do the same thing as Example 29-1(a), namely create a `deque` of `string`s populated from the standard input, except that it's trying to do it using the syntax of Example 29-2(a), namely using the iterator range constructor.

This has one potential problem and one actual problem. The potential problem is that `cin` is exhausted so there's no input left to read as was probably intended, which might be a logical problem.

The big problem, though, is that the code doesn't actually do anything at all. Why not? Because it does actually declare a `deque<string>` object named `coll3`. What it actually declares is (take a deep brea here):

a function named `coll3`
 that returns a `deque<string>` by value
 and takes two parameters:
  an `istream_iterator<string>` with a formal parameter name of `cin`,
  and a function with no formal parameter name

that returns an `istream_iterator<string>`
and takes no parameters.

(Say that three times fast.)

What's going on here? Basically, we're running across one of the painful rules that C++ inherited from C, to maintain C compatibility: If a piece of code can be interpreted as a declaration, it will be. In the words of the C++ standard:

> There is an ambiguity in the grammar involving expression-statements and declarations: An expression-statement with a function-style explicit type conversion (_expr.type.conv_) as its leftmost subexpression can be indistinguishable from a declaration where the first declarator starts with a `(`. In those cases the statement is a declaration. [C++03] §6.8

Without going into the gory details, the reason why this is the way that it is comes down to helping compilers deal with C's horrid declaration syntax, which can be ambiguous so to make things manageable the compiler resolves such ambiguities by universally assuming that "if in doubt, it must b a function declaration." 'Nuff said.

If you haven't already, take a quick look at Item 42 of Exceptional C++ [Sutter00], which contains a similar but simpler example. Let's dissect the declaration step by step to see what's going on:

```
// Example 29-2(d): Identical to Example 29-2(c), removing
// redundant parentheses and adding a typedef
//
typedef istream_iterator<string> (Func)();

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

Does that look more like a function declaration? Maybe so, maybe not, so let's take another step and remove the formal parameter name `cin`, which is ignored anyway, and change the name `coll3` to something that we usually expect to see as a function name:

```
// Example 29-2(e): Still identical to Example 29-2(c),
// other than slight name changes
//
typedef istream_iterator<string> (Func)();

deque<string> f(istream_iterator<string>, Func);
```

Now it's pretty clear: This "could be" a function declaration, so according to the C and C++ syntax rules, it is one. What makes it confusing is that it looks a lot like constructor syntax; what makes it downright obscure is that the formal parameter name, `cin`, happens to resemble the name of a variable that is indeed in scope and is even defined by the standard because that's what it was in fact intended to be but, misleading as it is, that doesn't matter, for the formal parameter name and `std::cin` have

nothing in common other than the accident that they happen to be spelled the same way.

People still run across this problem from time to time in real-world coding, and that's the reason why this problem deserves its own Item. Because the code is (probably surprisingly) just a function declaration, it doesn't actually do anything—no code gets generated by the compiler, no actions are performed, no `deque` constructors are called, no objects are created.

Proper Population Control

It wouldn't be fair to throw up an example like this, however, without also showing how you can fix it. This brings us to the final question:

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

All we need is something that makes it impossible for the compiler to treat the code as a function declaration. There are two easy ways to do it. Here's the seductive way:

```
// Example 29-3(a): Disambiguate, say by adding parens (okay solution, score 7/1(
//
deque<string> coll3((istream_iterator<string>(cin)), istream_iterator<string>()),
```

Here, just adding the extra parentheses around the parameters makes it clear to the compiler that what we intend to be constructor parameter names can't be parameter declarations. This is because although `istream_iterator<string>(cin)` can be a variable (or parameter declaration, as already noted), `(istream_iterator<string>(cin))` can't—the code in Example 29-3(a) can't be a function declaration for the same reason that `void f((int i))` can't be, namely because of the extra parentheses, which are illegal around a whole parameter declaration.

There are other ways to try to disambiguate this by forcing the statement out of the declaration syntax, but I won't present them for a simple reason: They only work if both you and your compiler understand this corner case of the standard very well.

Guideline

Avoid the dark corners of the language, including constructs that might be arguably legal but that are liable to confuse programmers, or even compilers.

This declaration-vs.-constructor syntax ambiguity is by its nature such a thorny edge case that the best thing to do is just avoid the ambiguity altogether, and not rely on methods that essentially amount to coaxing and wheedling a surly three-year-old compiler into treating it as a declaration. Put another wa if you were talking to someone, would you purposely say something ambiguous and then change it

slightly by adding, "Well, honey, what I really meant was…"? Hardly.

It's far better to avoid the ambiguity in the first place. I prefer and recommend the following alternativ because it's much easier to get right, it's utterly understandable to even the weakest compilers, and it makes the code clearer to read to boot:

```
// Example 29-3(b): Use named variables (recommended solution, score 10/10)
//
istream_iterator<string> first(cin), last;

deque<string> coll3(first, last);
```

Actually, in both Example 29-3(a) and Example 29-3(b), making the suggested change to just one of th parameters would have been sufficient, but for consistency I've treated both parameters the same way.

Guideline

Prefer using named variables as constructor parameters. This avoids possible declaration ambiguities. It also often makes the purpose of your code clearer and thus is easier to maintain.

Summary

Avoid the language's dusty corners; programmers only know there are enough of them.

Be clear and explicit, and say what you mean: By all means use named variables as constructor parameters to avoid subtle language weirdness and make your code clearer and more maintainable.

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 `float`s might be somewhat faster, as fast, or somewhat slower than `double`s) 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.

A Word About Narrowing Conversions

Some people might wonder, "Well, besides universal heat death, isn't there another problem? The constant `1e8` has type `double`. So if we just changed `double` to `float`, the program wouldn't compile because of the narrowing conversion, right?" Well, let's quote standardese again, this time from a different section:

> An rvalue of floating point type can be converted to an rvalue of another floating point type. If the source value can be exactly represented in the destination type, the result of the conversion is that exact representation. If the source value is between two adjacent destination values, the result of the conversion is an implementation-defined choice of either of those values. Otherwise, the behavior is undefined.
>
> [C++03] §4.8/1

This means that a `double` constant can be implicitly (i.e., silently) converted to a `float` constant, even if doing so loses precision (i.e., data, information, knowledge, state, meaning). This was allowed to remain for C compatibility and usability reasons, but it's worth keeping in mind when you do floating-point work.

Summary

This Item barely scratches the surface of floating-point issues. Floating-point math is hard, deep, and almost entirely nonobvious. I can say with some confidence that there are three kinds of people in the world: people who know they don't understand floating-point math (and are correct), people who think they understand floating-point math (and are wrong), and those few true experts who wonder if they will ever fully understand floating-point math (and are wise).

Guideline

Remember that floating-point math is weird and deeply strange. Be alert when using floating point, and avoid relying on floating point conversions. Almost everything that people think they know about arithmetic is subtly or grossly incorrect when it concerns floating-point math.

A quality compiler will warn you if you try to do something that's undefined behavior, namely, put a `double` quantity into a `float` that's less than the minimum or greater than the maximum value that a `float` is able to represent. A really good compiler will provide an optional warning if you try to do something that might be defined but could lose information, namely, put a `double` quantity into a `float` that is between the minimum and maximum values representable by a `float` but which can't be represented exactly as a `float`.

Be alert to possible odd floating-point behavior, try hard to avoid relying on floating-point conversions, and turn on all the diagnostics that your compiler gives you, and you can wade more or less safely in the strange and exciting but decidedly murky waters of floating-point arithmetic.

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.

To get around the compile error, the programmer comments out the `...Ex` functions, and now the program compiles and he starts testing the sleeping functions. Unfortunately, the `Animal::Sleep` member function doesn't seem to always work correctly; when he tries to call the member function `Animal::Sleep` directly, all is well. But when he tries to call it through the `Sleep` free function wrapper, which does nothing but call the member function version, sometimes nothing happens… not all the time, only in some cases. Finally, when the programmer goes into the debugger or the linker-generated symbol map in an attempt to diagnose the problem, he can't seem to even find the code for `Animal::Sleep` at all.

Is the compiler on the fritz? Should the programmer send the compiler vendor an angry flame e-mail and submit an irate letter to the New York Times? Is it a delayed Year 2000 problem? Or is it just due to a rogue virus caught from the Internet?

What's going on?

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 other accounts doesn't seem to exist. It's usually at this point that you quickly figure out what's really going on and mutter a sotto voce grumbling at stupid macro tricks.

But wait, it gets better:

1(b). C++ already has features to deal with names. That macros provide a different feature to deal with similar work causes what might be best termed an unhealthy interaction.

You might think that it's not such a big deal to change a function's name. All right, fine; often it's not. But say you change a function's name to be the same as another function that also exists… what does C++ do if it finds two functions with the same name? It overloads them. That's not quite so fine when you don't realize it's happening silently.

This, alas, seems to be the case with `Sleep`. The whole reason the library vendor decided to "helpfully" provide a `Sleep` macro to automatically rename things to `SleepEx` is that both such functions in fact do already exist in the vendor's library. Consider that the functions might have different signatures; then when we write our own `Sleep` function, we might well be aware of the overloading on the library supplied `Sleep` and take care to avoid ambiguities or other errors that might cause us problems. We might even rely on the overloading because we want to provide library-`Sleep`-like behavior intentionally. If, instead, our function is being silently overloaded with some other function, the overload isn't just going to behave differently than we expect, but if our original overloading was intentional it's not going to happen at all, at least not in the way we thought.

In the context of our question, such a `Sleep`-renaming macro can partly explain why different functions could end up being called in different circumstances; which function gets called can depend on how the overload resolution happened to work out for the specific types used at different call sites. Sometimes it's ours, and sometimes it's the library's. It just depends, perhaps in nonobvious ways.

If this sordid tale ended with these lamentable effects on nonmember functions, that would be bad enough. Unfortunately, like shrapnel from a grenade, there's dangerous stuff flying in several other directions too.

2. Macros don't care about type.

The original intent for the `Sleep` macro I described was to change a global nonmember function's name. Unfortunately, the macro will change the name `Sleep` wherever it finds it; if we happen to have a global variable named `Sleep`, that member's name will silently change too. This is altogether a Bad Thing.

3. Macros don't care about scope.

Worse still, a macro designed to change a global nonmember function name will happily change any matching function (or other) names that happen to be members of your classes or nicely encapsulated within your own namespaces. In this case, we wrote a class with both `Sleep` and `SleepEx` functions; many of the described problems could be accounted for at least in part by a `Sleep`-renaming macro that makes our own functions invisibly overload with each other. Indeed, as with the invisible overloading already mentioned under point #1, this can explain why sometimes an unexpected member function can be called, depending on how the overload resolution happened to work out for the specific types used at different call sites.

If you think this is yet another Bad Thing, you're right. It's like having some ungloved, uncertified doctor (the injudicious library header writer) with dirty hands (unsterilized macros) slice open your torso (class or namespace) and reach into your body cavity to rearrange things (members and other code)… while sleepwalking (not even realizing they're doing it).

Summary

In short, macros don't care about much of anything.

Guidelines

Avoid macros.

Do not ever ever ever even consider starting to think about writing a macro that is a common word or abbreviation.

Your default response to macros should be something like "Macros! Ew, yuck!" unless there's a compelling reason to use them in special cases where they are not a hack. Macros are not type-safe, they're not scope-safe… they're just not safe, period. If you must write macros, avoid putting them in header files and try to give them long and personalized names that are highly unlikely to ever tromp upon things that they aren't intentionally meant to beat into unwilling submission and grind under their heels into the dust.

Guideline

Prefer to use namespaces to encapsulate names.

In short, practice good encapsulation. Not only does good encapsulation make for better designs, but it can also defend you against unexpected threats you might not even see coming. Macros are the antithesis of encapsulation because their actual range of effect cannot be controlled, not even by the macro writer. Classes and namespaces are among C++'s useful tools to help manage and minimize interdependencies between parts of your program that should be unrelated, and judicious use of these and other C++ facilities to promote encapsulation will not just make for superior designs but will at the same time offer a measure of protection against the shrapnel that ill-considered code from fellow programmers, however wellintentioned, might occasionally send your way.

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?

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

The short answer is: Zero. This code is perfectly legal and standards-conforming (whether the author might have wanted it to be or not).

Let's consider in turn each of the expressions that might be questionable and see why they're really okay:

- `0[p]` is legal and has exactly the same meaning as `p[0]`. In C (and C++), an expression of the form `x[y]`, where one of `x` and `y` is a pointer type and the other is an integer value, always means `*(x+y)`. In this case, `0[p]` and `p[0]` have the same meaning because they mean `*(0+p)` and `*(p+0)`, respectively, which comes out to the same thing. For more details, see [C99] §6.5.2.1.

- `and` and `not` are valid keywords that are alternative spellings of `&&` and `!`, respectively.

- `:>` is legal. It is a digraph for the `]` character, not a smiley (smileys are unsupported in the C++ language outside of comment blocks, which is rather a shame). This turns the final part of the expression into `p[1]>p[2]`.

- The "extra" semicolon at the end of the member function declaration is allowed and is completely benign. The C++ class definition syntax allows an empty member declaration (a bald semicolon) to appear anywhere, as often as you like. For example, the following is a perfectly legal definition of a class with no members:

```
class X { ;;;;;;;;;; };
```

Of these, most people seem to find the `:>` digraph the most surprising. Of course, it could well be that the colon ":" was a typo and the author really meant something else, such as perhaps `p[1]>>p[2]`, but even if it was a typo it's still (unfortunately, in that case) perfectly legal code.

Previous Next

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 additionally provide a + operator. For example:

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

Now we can (if the afternoon is particularly slow) write code like:

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

which works out to:

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

This trick is fairly simple. Creating longer-than-usual sequences of other characters turns out to be a little more challenging, but still possible.

Abuses of Operators

The code in Examples 33-1(a) and 33-1(b) doesn't especially abuse the ++ and + operators' usual semantics. What we're going to do next, however, goes far beyond anything you'd ever want to see in production code; this is for fun only.

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?

For this question, let's create and use the following helper class:

```
// Example 33-2
//
class A {
public:
 void operator&&(int) { }
 void operator<<(int) { }
 void operator||(int) { }
};

typedef void (A::*F)(int);
```

Now let's consider the challenges:

a. `&`

Answer: Five.

Well, `&&` is easy and `&&&` not too much harder, so let's go right to the next level: Can we create a series of four `&`'s, namely `&&&&`? Well, if we did, they would be interpreted as `&& &&`, but expressions like `a && && b` are syntactically illegal; we can't have two binary `&&` operators immediately after each other.

The trick is to see that we can use the second `&&` as an operator, and make the first `&&` come out as the end of something that's not an operator. With that in mind, it doesn't take too long to see that the first `&&` could be the end of a name, specifically the name of a function, so all we need is an `operator&&()` that can accept a pointer to some other `operator&&()` as its first parameter:

```
void operator&&(F, A) { }
```

This lets us write:

```
&A::operator&&&&a;      // && &&
```

which means:

```
operator&&(&A::operator&&, a);
```

That's the longest even-numbered run of `&`'s we can make, because `&&&&&&` has to be illegal. Why?

Because it would mean `&& && &&`, and even after making the first `&&` part of a name again, we can't make the final `&&` the beginning of another name, so we're left with two binary `&&` operators in a row, which is illegal.

But can we squeeze in one more `&` by going to an odd number of `&`'s? Yes, indeed. Specifically, `&&&&&` means `&& && &`; we already have a solution for the first part, and with not too much more thought it's easy to tack on a unary `&`:

```
&A::operator&&&&&a; // && && &
```

which uses the built-in `operator&&()` that can take pointers:

```
operator&&(&A::operator&&, &a);
```

Now let's try (b) and (c):

b. `<`

c. `|`

Answer for both: Four.

Having seen the solution to 2(a), this one should be easy. To make a series of four, we just use the same trick as before, defining a:

```
void operator<<(F, A) { }
void operator||(F, A) { }
```

which lets us write:

```
&A::operator<<<<a; // << <<
&A::operator||||a;       // || ||
```

which means:

```
operator<<(&A::operator<<, a);
operator||(&A::operator||, a);
```

These are the longest even-numbered runs we can make, because `<<<<<<` and `||||||` have to be illegal, just as we've already noted that `&&&&&&` has to be illegal. But this time we can't manage an

extra < or | to make a series of five, because there is no unary < or | operator.

Bonus Challenge Question

Here's a bonus question: How many question mark (?) characters can appear in a row in a valid C++ program?

Think about the question before reading on. It's a lot harder than it looks.

• • • • •

Do you have an answer?

You might think that the answer must be "one," because the only legitimate token in the C++ language that includes a ? is the ternary ? : operator. It's true that that's the only legitimate language feature that includes a ?, but "there are more things in the translator and preprocessor, Horatio, than are dreamt of in your language syntax rules…" In particular, there's more to C++ than just the language, or even just the preprocessor.

For ?, the correct answer is three. For example:

```
1???-0:0;
```

This question is harder than the others in part because it's the only one that doesn't follow the maximal munch rule. The three question marks, ???, are not interpreted as the tokens ?? and ?. Why not? Because ??- happens to be a trigr… quit groaning there in the back… trigraph, and trigraphs are replaced very early during source code processing—before tokenization begins, even before preprocessor instructions are handled. If you haven't heard about trigraphs before, don't worry; that just means that you don't use an exotic foreign-language keyboard or you haven't yet read Item 32. A trigraph is an alternate way to spell certain unusual source code characters (specifically #, \, ^, [,], |, {, }, and ~) provided for the benefit of program mers whose keyboards don't happen to have a key for that character.

In this case, long before any tokenization can occur, the trigraph ??- is replaced with ~, the one's-complement operator. Therefore the statement becomes:

```
1?~0:0;
```

which is tokenized as:

```
1 ? ~ 0 : 0 ;
```

and means:

```
1 ? (~0) : 0 ;
```

Summary

Trigraphs are a feature inherited from C, they are rare in practice, and they were principally useful politically during standardization. (Don't ask.) Just to give you an idea of how rare they are, note that as of this writing several compilers I know of do not have trigraph support turned on by default, and one of those documents it as "enables the undesirable and rarely used ANSI trigraph feature." This is an accurate and well-founded comment.

Previous Next

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++)
  (*vit).set(it,i);

 std::sort(v.begin(), v.end());

 int *pi = pidxtbl;
 vit = v.begin();
 for(; vit<v.end(); pi++, vit++)
  *pi = (*vit).i;
}

main()
{
 int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };

 cout << "##################" << endl;
 std::vector<int> vecai(ai, ai+10);
 int aidxtbl[10];
 sort_idxtbl(vecai.begin(), vecai.end(), aidxtbl);

 for (int i=0; i<10; i++)
 cout << "i="<< i
    << ", aidxtbl[i]="<< aidxtbl[i]
    << ", ai[aidxtbl[i]]="<< ai[aidxtbl[i]]
    << endl;
 cout << "##################" << endl;
}
```

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 in the first place, so writing clean code makes your own life easier even in the very short term. (For a in point, see the discussion surrounding Example 27-2 in Item 27.) Further, when you return to the code month or a year later—as you surely will if the code is any good and is actually being used—it's much to pick it up again and understand what's going on. Most programmers find keeping full details of code 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 benefi 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 understan the internal workings of the code itself, as well as its side effects and interactions with other sub-syste is altogether too easy to introduce new errors when changing code one does not fully understand. Code is clear and understandable is easier to grok, and therefore, fixes to such code become less fragile, les risky, less likely to have un-intended side effects.

Most important, however, your end users benefit from clear and understandable code for all these reas Such code is likely to have had fewer initial bugs in the first place, and it's likely to have been maintai 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 in 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 data (for example, a `vector<Employee>`) and secondary containers of iterators into the main containe support variant access methods (for example, a `set<vector<Employee>::iterator, Funct>` where `Funct` is a functor that compares `Employee` objects indirectly, yielding a different ordering than the or which the objects are physically stored in the `vector`).

Having said that, style matters too. The original author has kindly allowed me to use his code as a case point, and I'm not trying to pick on him here; I'm just adopting the technique, pioneered long ago by suc luminaries as P. J. Plauger, of expounding coding style guidelines via the dissection and critique of published code. I've critiqued other published material before and have had other people critique my c and I'm positive that further dissections will no doubt follow.

Having said all that, let's see what we might be able to improve in this particular piece of code.

Correcting Mechanical Errors

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

The first area for constructive criticism is mechanical errors in the code, which on most platforms wo compile as shown.

```
#include <algorith>
```

1. Spell standard headers correctly. Here the header `<algorithm>` is misspelled as `<algorith>`. M first guess was that this is probably an artifact of an 8-character file system used to test the origin code, but even my old version of VC++ on an old version of Windows (based on the 8.3 filename system) rejected this code. Anyway, it's not standard, and even on hobbled file systems the compi itself is required to support any standard long header names, even if it silently maps it onto a sho filename (or onto no file at all).

Next, consider:

```
main()
```

2. Define `main` correctly. This unadorned signature for `main` has never been standard C++ [C++98] although it is a conforming extension as long as the compiler warns about it. It used to be valid in 1999 C, which had an implicit `int` rule, but it's nonstandard in both C++ (which never had impli

`int`) and C99 [C99] (which as far as I can tell didn't merely deprecate implicit `int`, but actually removed it outright). In the C++ standard, see:

- §3.6.1/2: portable code must define `main` as either `int main()` or `int main(int,char*[]`
- §7/7 footnote 78, and §7.1.5/2 footnote 80: implicit `int` banned
- Annex C (Compatibility), comment on 7.1.5/4: explicitly notes that bare `main()` is invalid C and must be written `int main()`

Guideline

Don't rely on implicit `int`; it's not standard-conforming portable C++. In particular "`void main()`" or just "`main()`" has never been standard C++, although many compilers still support them as conforming extensions.

```
cout << "##################" << endl;
```

3. Always `#include` the headers for the types whose definitions you need. The program uses `cout` `endl` but fails to `#include <iostream>`. Why did this probably work on the original developer' system? Because C++ standard headers can `#include` each other, but unlike C, C++ does not spe which standard headers `#include` which other standard headers.

In this case, the program does `#include <vector>` and `<algorithm>`, and on the original system it probably just so happened that one of those headers also happened to indirectly `#include <iostrea` too. That might work on the library implementation used to develop the original code, and it happens t work on mine too, but it's not portable and not good style.

4. Follow the guidelines in Item 36 in More Exceptional C++ [Sutter02] about using namespaces. Speaking of `cout` and `endl`, the program must also qualify them with `std::` or write `using std::cout; using std::endl;`. Unfortunately it's still common for authors to forget namespac scope qualifiers—I hasten to point out that this author did correctly scope `vector` and `sort`, whi good.

Improving Style

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

Beyond the mechanical errors, there were several things I personally would have done differently in th code example. First, a couple of comments about the helper struct:

```
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() {}
};
```

1. Be const correct. In particular, `sort_idxtbl_pair::operator<` doesn't modify `*this`, so it oug be declared as a `const` member function.

> **Guideline**
>
> Practice const correctness.

2. Remove redundant code. The program explicitly writes class `sort_idxtbl_pair`'s default const even though it's no different from the implicitly generated version. There doesn't seem to be much to this. Also, as long as `sort_idxbl_pair` is a `struct` with public data, having a distinct `set` operation adds a little syntactic sugar but because it's called in only one place the minor extra complexity doesn't gain much.

> **Guideline**
>
> Avoid code duplication and redundancy.

Next, we come to the core function, `sort_idxtbl`:

```
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++)
  (*vit).set(it,i);

 std::sort(v.begin(), v.end());

 int *pi = pidxtbl;
 vit = v.begin();
 for(; vit<v.end(); pi++, vit++)
  *pi = (*vit).i;
}
```

3. Choose meaningful and appropriate names. In this case, `sort_idxtbl` is a misleading name beca the function doesn't sort an index table… it creates one! On the other hand, the code gets good ma for using the template parameter name `RAIter` to indicate a random-access iterator; that's what's required in this version of the code, so naming the parameter to indicate this is a good reminder.

> **Guideline**
>
> Choose clear and meaningful names.

4. Be consistent. In `sort_idxtbl`, sometimes variables are initialized (or set) in `for` loop initializa statements, and sometimes they aren't. This just makes things harder to read, at least for me. Your mileage may vary on this one.

5. Remove gratuitous complexity. This function adores gratuitous local variables! It contains three examples. First, the variable `iDst` is initialized to `last-first` and then used only once; why not write `last-first` where it's used and get rid of clutter? Second, the `vector` iterator `vit` is creat where a subscript was already available and could have been used just as well, and the code wo have been clearer. Third, the local variable `it` gets initialized to the value of a function paramete after which the function parameter is never used; my personal preference in that case is just to use function parameter (even if you change its value—that's okay!) instead of introducing another nam

6. Reuse Part 1: Reuse more of the standard library. Now, the original program gets good marks for reusing `std::sort`—that's good. But why handcraft that final loop to perform a copy when `std:` does the same thing? Why reinvent a special-purpose `sort_idxtbl_pair` class when the only thi has that `std::pair` doesn't is a comparison function? Besides being easier, reuse also makes our code more readable. Humble thyself and reuse!

> **Guideline**
>
> Know about and (re)use the standard library's facilities wherever appropriate instead of hand-rolling your own.

7. Reuse Part 2: Kill two birds with one stone by making the implementation itself more re-usable. (
original implementation, nothing is directly reusable other than the function itself. The helper
`sort_idxtbl_pair` class is hardwired for its purpose and is not independently reusable.

8. Reuse Part 3: Improve the signature. The original signature

```
template <class RAIter>
void sort_idxtbl(RAIter first, RAIter last, int* pidxtbl)
```

takes a bald `int*` pointer to the output area, which I generally try to avoid in favor of managed storage
a `vector`). But at the end of the day the user ought to be able to call `sort_idxtbl` and put the output i
plain array or a `vector` or anything else. Well, the requirement "be able to put the output into any cont
simply cries out for an iterator, doesn't it? (See also Items 5 and 6.)

```
template< class RAIn, class Out >
void sort_idxtbl(RAIn first, RAIn last, Out result)
```

Guideline

Widen the reusability of your generic components by not hardcoding types that don't need to be hardcoded.

9. Reuse Part 4, or Prefer comparing iterators using `!=`: When comparing iterators, always use `!=` (
works for all kinds of iterators) instead of < (which works only for random-access iterators), unl
course you really need to use < and only intend to support random-access iterators. The original
program uses < to compare the iterators it's given to work on, which is fine for random-access
iterators, which was the program's initial intent: to create indexes into `vector`s and arrays, both
which support random-access iteration. But there's no reason we might not want to do exactly the
thing for other kinds of containers, like `list`s and `set`s, that don't support random-access iteratio
and the only reason the original code won't work for such containers is that it uses < instead of `!=`
compare iterators.

As Scott Meyers puts it eloquently in Item 32 of [Meyers96], "Program in the future tense." He elabora

> Good software adapts well to change. It accommodates new features, it ports to new platforms, i
> adjusts to new demands, it handles new inputs. Software this flexible, this robust, and this
> reliable does not come about by accident. It is designed and implemented by programmers who
> conform to the constraints of today while keeping in mind the probable needs of tomorrow. This

kind of software—software that accepts change gracefully—is written by people who program in the future tense.

Guideline

Prefer to compare iterators using `!=`, not `<`.

10. Prefer preincrement unless you really need the old value. Here, for the iterators, writing preincre (`++i`) should habitually be preferred over writing postincrement (`i++`); see [Sutter00, Item 39]. T that might not make a material difference in the original code because `vector<T>::iterator` mi be a cheap-to-copy `T*` (although it might not be, particularly on checked STL implementations), b we implement point 13 we may no longer be limited to just `vector<T>::iterator`s, and most o iterators are of class type—perhaps often still not too expensive to copy, but why introduce this possible inefficiency needlessly?

Guideline

Prefer to write preincrement rather than postincrement, unless you really need to use the previous value.

That covers most of the important issues. There are other things we could critique but that I didn't cons important enough to call attention to here; for example, production code should have comments that document each class's and function's purpose and semantics, but that doesn't apply to code accompany magazine articles where the explanation is typically written in better English and in greater detail than comments have any right to expect.

I'm deliberately not critiquing the mainline for style (as opposed to the mechanical errors already note would cause the mainline to fail to compile), because, after all, this particular mainline is only meant t a demonstration harness to help readers of the magazine article see how the index table apparatus is m to work, and it's the index table apparatus that's the intended focus.

Summary

Let's preserve the original code's basic interface choice instead of straying far afield and proposing alternate design choices.[40] Limiting our critique just to correcting the code for mechanical errors and basic style, then, consider the three alternative improved versions below. Each has its own benefits, drawbacks, and style preferences as explained in the accompanying comments. What all three versions in common is that they are clearer, more understandable, and more portable code—and that ought to co for something, in your company and in mine.

[40] The original author also reports separate feedback from another reader demonstrating another elegant, but substantially different, approach: He creates a containerlike object that wraps the original container, including its iterators, and allows iteration using the alternative ordering.

```
// An improved version of the code originally published in [Hicks00].
//
#include <vector>
#include <map>
#include <algorithm>

// Solution 1 does some basic cleanup but still preserves the general structure
// of the original's approach. We're down to 17 lines (even if you count "public
// and "private:" as lines), where the original had 23.
//
namespace Solution1 {
 template<class Iter>
 class sort_idxtbl_pair {
 public:
  void set(const Iter& it, int i) { it_ = it; i_ = i; }

  bool operator<(const sort_idxtbl_pair& other) const
   { return *it_ < *other.it_; }

  operator int() const { return i_; }
 private:
   Iter it_;
   int i_;
 };

 // This function is where most of the clarity savings came from; it has 5 lines,
 // where the original had 13. After each code line, I'll show the corresponding
 // original code for comparison. Prefer to write code that is clear and concise,
 // not unnecessarily complex or obscure!
 //
 template<class IterIn, class IterOut>
 void sort_idxtbl(IterIn first, IterIn last, IterOut out) {
  std::vector<sort_idxtbl_pair<IterIn> > v(last-first);
   // int iDst = last-first;
   // typedef std::vector< sort_idxtbl_pair<RAIter> > V;
   // V v(iDst);

  for(int i=0; i < last-first; ++i)
   v[i].set(first+i, i);
   // int i=0;
   // RAIter it = first;
   // V::iterator vit = v.begin();
   // for (i=0; it<last; it++, vit++, i++)
   // (*vit).set(it,i);

  std::sort(v.begin(), v.end());
   // std::sort(v.begin(), v.end());

  std::copy(v.begin(), v.end(), out);
   // int *pi = pidxtbl;
   // vit = v.begin();
   // for (; vit<v.end(); pi++, vit++)
```

```
   // *pi = (*vit).i;
 }
}

// Solution 2 uses a pair instead of reinventing a pair-like helper class. Now we
// down to 13 lines, from the original 23. Of the 14 lines, 9 are purpose-specif
// and 5 are directly reusable in other contexts.
//
namespace Solution2 {
 template<class T, class U>
 struct ComparePair1stDeref {
  bool operator()(const std::pair<T,U>& a, const std::pair<T,U>& b) const
   { return *a.first < *b.first; }
 };
 template<class IterIn, class IterOut>
 void sort_idxtbl(IterIn first, IterIn last, IterOut out) {
  std::vector< std::pair<IterIn,int> > s(last-first);
  for(int i=0; i < s.size(); ++i)
   s[i] = std::make_pair(first+i, i);

  std::sort(s.begin(), s.end(), ComparePair1stDeref<IterIn,int>());

  for(int i=0; i < s.size(); ++i, ++out)
  *out = s[i].second;
 }
}
  // Solution 3 just shows a couple of alternative details—it uses a map to avoi
// separate sorting step, and it uses std::transform() instead of a handcrafted l
// Here we still have 15 lines, but more are reusable. This version uses more spa
// overhead and probably more time overhead too, so I prefer Solution 2, but this
// is an example of finding alternative approaches to a problem.
//
namespace Solution3 {
 template<class T>
 struct CompareDeref {
  bool operator()(const T& a, const T& b) const
   { return *a < *b; }
 };
 template<class T, class U>
 struct Pair2nd {
  const U& operator()(const std::pair<T,U>& a) const { return a.second; }
 };

 template<class IterIn, class IterOut>
 void sort_idxtbl(IterIn first, IterIn last, IterOut out) {
  std::multimap<IterIn, int, CompareDeref<IterIn> > v;
  for(int i=0; first != last; ++i, ++first)
   v.insert(std::make_pair(first, i));

  std::transform(v.begin(), v.end(), out, Pair2nd<IterIn const,int>());
 }
}

// I left the test harness essentially unchanged, except to demonstrate putting
// the output in an output iterator (instead of necessarily an int*) and using th
// source array directly as a container.
//
#include <iostream>
```

```
int main() {
  int ai[10] = { 15,12,13,14,18,11,10,17,16,19 };

  std::cout << "#################" << std::endl;
  std::vector<int> aidxtbl(10);

  // use another namespace name to test a different solution
  Solution3::sort_idxtbl(ai, ai+10, aidxtbl.begin());

  for(int i=0; i<10; ++i)
  std::cout << "i="<< i
            << ", aidxtbl[i]="<< aidxtbl[i]
            << ", ai[aidxtbl[i]]="<< ai[aidxtbl[i]]
            << std::endl;
  std::cout << "#################" << std::endl;
}
```


Previous Next

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 incl
the kitchen sink. What it does mean is that generic code ought to make a reasonable and balanced effor
avoid at least three things:

1. Avoid undue type restrictions. For example, are you writing a generic container? Then it's perfec
reasonable to require that the contained type have, say, a copy constructor and a nonthrowing des
But what about a default constructor or an assignment operator? Many useful types that users mig
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 comple
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 us
wants to do Z, and Z isn't so much different from Y? Sometimes you'll want to make your facility
enough to support Z; sometimes you won't. Part of good generic design is choosing the ways and
by which your facility can be customized or extended. That this is important in generic design sho
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 gene
code. For examples of policy-based design, see any of several chapters in [Alexandrescu01]; the `Smar`
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 exam
under consideration below, and it deserves some dedicated consideration in its own right, hence
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 judgm
needed when deciding where to draw the line between failing to be sufficiently generic (the "I'm sure
would want to use it with anything but `char`" syndrome) on the one hand and overengineering (the "wh
someday someone wants to use this toaster-oven LED display routine to control the booster cutoff on a
interplanetary spacecraft?" misguided fantasy) on the other.

Dissecting Generic Callbacks

4. The following code presents an interesting and genuinely useful idiom for wrapping callback fun

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
 void execute() { (object.*F)(); }   // launch callback function

private:
 T& object;
};
```

Now, really, how many ways are there to go wrong in a simple class with just two one-liner member functions? Well, as it turns out, its extreme simplicity is part of the problem. This class template doesn to be heavyweight, not at all, but it could stand to be a little less lightweight.

Improving Style

Critique this code and identify:

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

How many did you spot? Here's what I came up with:

5. The constructor should be `explicit`. The author probably didn't mean to provide an implicit con from `T` to `callback<T>`. Well-behaved classes avoid creating the potential for such problems for users. So what we really want is more like this:

```
explicit callback(T& t) : object(t) {} // assign actual object to T
```

While we're already looking at this particular line, there's another stylistic issue that's not about the de se but about the description:

Guideline

Prefer making constructors `explicit` unless you really intend to enable type conversions.

(Nit) The comment is wrong. The word "assign" in the comment is incorrect and so somewhat mislead

More correctly, in the constructor, we're "binding" a `T` object to the reference and by extension to the c
object. Also, after many rereadings I'm still not sure what the "to `T`" part means. So better still would b
actual object."

```
explicit callback(T& t) : object(t) {} // bind actual object
```

But then, all that comment is saying is what the code already says, which is faintly ridiculous and a ste
example of a useless comment, so best of all would be:

```
explicit callback(T& t) : object(t) {}
```

1. The `execute` function should be `const`. The `execute` function isn't doing anything to the `callba`
   object's state, after all! This is a "back to basics" issue: Const correctness might be an oldie, but
   goodie. The value of const correctness has been known in C and C++ since at least the early 198
   that value didn't just evaporate when we clicked over to the new millennium and started writing l
   templates.

```
void execute() const { (object.*F)(); }       // launch callback function
```

Guideline

Be const correct.

While we're already beating on the poor `execute` function, there's an arguably more serious idiomatic
problem:

2. (Idiom) And the `execute` function should be spelled `operator()`. In C++, it's idiomatic to use th
   function-call operator for executing a function-style operation. Indeed, then the comment, already
   somewhat redundant, becomes completely so and can be removed without harm because now our
   already idiomatically commenting itself. To wit:

   ```
   void operator()() const { (object.*F)(); } // launch callback function
   ```

"But," you might be wondering, "if we provide the function-call operator, isn't this some kind of functi
object?" That's an excellent point, which leads us to observe that, as a function object, maybe callback
instances ought to be adaptable too.

Guideline

Provide `operator()` for idiomatic function objects rather than providing a named `execute` function.

Pitfall: (Idiom) Should this callback be derived from `std::unary_function`? See Item 36 in [Meyer a more detailed discussion about adaptability and why it's a Good Thing in general. Alas, here, there a excellent reasons why callback should not be derived from `std::unary_function`, at least not yet:

- It's not a unary function. It takes no parameter, and unary functions take a parameter. (No, `void` do count.)
- Deriving from `std::unary_function` isn't going to be extensible anyway. Later on, we're going that callback perhaps ought to work with other kinds of function signatures too, and depending on number of parameters involved, there might well be no standard base class to derive from. For ex if we supported callback functions with three parameters, we have no `std::ternary_function` derive from.

Deriving from `std::unary_function` or `std::binary_function` is a convenient way to give callba handful of important `typedef`s that binders and similar facilities often rely upon, but it matters only if going to use the function objects with those facilities. Because of the nature of these callbacks and how intended to be used, it's unlikely that this will be needed. (If in the future it turns out that they ought to b usable this way for the common one-and two-parameter cases, then the one-and two-parameter versio mention later can be derived from `std::unary_function` and `std::binary_function`, respectively

Correcting Mechanical Errors and Limitations

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

1. Consider making the callback function a normal parameter, not a template parameter. Non-type te parameters are rare in part because there's rarely much benefit in so strictly fixing a type at comp That is, we could instead have:

```
template < class T >
class callback {
public:
 typedef void (T::*Func)();

 callback(T& t, Func func) : object(t), f(func) {}       // bind  actual obje
 void operator()() const { (object.*f)(); }              // launch callback fu

private:
 T& object;
 Func f;
};
```

Now the function to be used can vary at run-time, and it would be simple to add a member function tha allowed the user to change the function that an existing call-back object was bound to, something not p in previous versions of the code.

Guideline

It's usually a good idea to prefer making non-type parameters into normal function parameters, unless they really need to be template parameters.

2. Enable containerization. If a program wants to keep one callback object for later use, it's likely to to keep more of them. What if it wants to put the callback objects into a container, such as a `vect list`? Currently that's not possible, because call-back objects aren't assignable—they don't suppo `operator=`. Why not? Because they contain a reference, and once that reference is bound during construction, it can never be rebound to something else.

Pointers, however, have no such compunction and are quite happy to point at what-ever you'd ask them this case it's perfectly safe for callback instead to store a pointer, not a reference, to the object it's to b on and then to use the default compiler-generated copy constructor and copy assignment operator:

```
template < class T >
class callback {
public:
 typedef void (T::*Func)();

 callback(T& t, Func func) : object(&t), f(func) {}      // bind actual object
 void operator()() const { (object->*f)(); }             // launch callback funct

private:
 T* object;
 Func f;
};
```

Now it's possible to have, for example, a `list< callback< Widget, &Widget::Some-Func > >`.

Guideline

Prefer to make your objects compatible with containers. In particular, to be put into a standard container, an object must be assignable.

"But wait," you might wonder at this point, "if I could have that kind of a `list`, why couldn't I have a l arbitrary kinds of callbacks of various types, so that I can remember them all and go execute them all v want to?" Indeed, you can, if you add a base class:

3. Enable polymorphism: Provide a common base class for callback types. If we want to let users h `list<callbackbase*>` (or, better, a `list< shared_ptr<callbackbase> >`) we can do it by providing just such a base class, which by default happens to do nothing in its `operator()`:

```
class callbackbase {
public:
 virtual void operator()() const { };
 virtual ~callbackbase() = 0;
};

callbackbase::~callbackbase() { }

template < class T >
class callback : public callbackbase {
public:
 typedef void (T::*Func)();

 callback(T& t, Func func) : object(&t), f(func) {}       // bind actual obje
 void operator()() const { (object->*f)(); }              // launch callback

private:
 T* object;
 Func f;
};
```

Now anyone who wants to can keep a `list<callbackbase*>` and polymorphically invoke `operator` its elements. Of course, a `list< shared_ptr<callback> >` would be even better; see [Sutter02b].

Note that adding a base class is a tradeoff, but only a small one: We've added the overhead of a secon indirection, namely a virtual function call, when the callback is triggered through the base interface. B overhead actually manifests only when you use the base interface. Code that doesn't need the base inte doesn't pay for it.

Guideline

Consider enabling polymorphism so that different instantiations of your class template can be used interchangeably, if that makes sense for your class template. If it does, do it by providing a common base class shared by all instantiations of the class template.

4. (Idiom, Tradeoff) There could be a helper `make_callback` function to aid in type deduction. Aft while, users may get tired of explicitly specifying template parameters for temporary objects:

```
list< callback< Widget > > l;
l.push_back(callback<Widget>(w, &Widget::SomeFunc));
```

Why write `Widget` twice? Doesn't the compiler know? Well, no, it doesn't, but we can help it to know contexts where only a temporary object like this is needed. Instead, we could provide a helper so that need only type:

```
list< callback< Widget > > l;
l.push_back(make_callback(w, &Widget::SomeFunc));
```

This `make_callback` works just like the standard `std::make_pair`. The missing `make_callback` hel should be a function template, because that's the only kind of template for which a compiler can deduc Here's what the helper looks like:

```
template<typename T >
callback<T> make_callback(T& t, void (T::*f) ()) {
 return callback<T>(t, f);
}
```

5. (Tradeoff) Add support for other callback signatures. I've left the biggest job for last. As the Barc have put it, "There are more function signatures in heaven and earth, Horatio, than are dreamt of i `void (T::*F) ()`!"

Guideline

Avoid limiting your template; avoid hardcoding for specific types or for less general types.

If enforcing that signature for callback functions is sufficient, then by all means stop right there. There' sense in complicating a design if we don't need to—for complicate it we will, if we want to allow for function signatures!

I won't write out all the code, because it's significantly tedious. (If you really want to see code this rep or you're having trouble with insomnia, see books and articles like [Alexandrescu01] for similar exam What I will do is briefly sketch the main things you'd have to support and how you'd have to support th

First, what about const member functions? The easiest way to deal with this one is to provide a paralle callback that uses the `const` signature type, and in that version, remember to take and hold the `T` by ref or pointer to `const`.

Second, what about non-`void` return types? The simplest way to allow the return type to vary is by ad

another template parameter.

Third, what about callback functions that take parameters? Again, add template parameters, remember parallel function parameters to `operator()`, and stir well.

Remember to add a new template to handle each potential number of callback arguments.

Alas, the code explodes, and you have to do things like set artificial limits on the number of function parameters that callback supports. Perhaps in a future C++0x language we'll have features like templat "varargs" that will help to deal with this, but not today.

Summary

Putting it all together, and making some purely stylistic adjustments like using `typename` consistently a naming conventions and whitespace conventions that I happen to like better, here's what we get:

```
class CallbackBase {
public:
 virtual void operator()() const { };
 virtual ~CallbackBase() = 0;
};

CallbackBase::~CallbackBase() { }

template<typename T>
class Callback : public CallbackBase {
public:
 typedef void (T::*F)();

 Callback(T& t, F f) : t_(&t), f_(f) { }
 void operator()() const { (t_->*f_)(); }

private:
 T* t_;
 F f_;
};

template<typename T>
Callback<T> make_callback(T& t, void (T::*f) ()) {
 return Callback<T>(t, f);
}
```

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();
  inline LIST& getlist();
  inline STRING& getstring();

protected:
  union {
   int i;
   unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
  } U;

  void cleanup();
 };

 inline int& MYUNION::getint()
 {
  if(currtype==_INT) {
    return U.i;
  } else {
    cleanup();
    currtype=_INT;
    return U.i;
  } // else
 }

 inline LIST& MYUNION::getlist()
 {
  if(currtype==_LIST) {
   return *(reinterpret_cast<LIST*>(U.buff));
  } else {
    cleanup();
    LIST* ptype = new(U.buff) LIST();
    currtype=_LIST;
    return *ptype;
  } // else
 }

 inline STRING& MYUNION::getstring()
 {
  if(currtype==_STRING) {
   return *(reinterpret_cast<STRING*>(U.buff));
   } else {
     cleanup();
     STRING* ptype = new(U.buff) STRING();
     currtype=_STRING;
     return *ptype;
  } // else
 }
 void MYUNION::cleanup()
```

```
{
 switch(currtype) {
  case _LIST: {
  LIST& ptype = getlist();
  ptype.~LIST();
  break;
 } // case
 case _STRING: {
  STRING& ptype = getstring();
  ptype.~STRING();
  break;
  } // case
  default: break;
 } // switch
 currtype=NONE;
}
```

4. Show a better way to achieve a generalized variant type, and comment on any tradeoffs you encounter.

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 m
   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 t
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? Expla

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 a 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 on
- 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 close-to-the-metal programming, which has been compatibly preserved in C++; that's why C++ also ha On the other hand, the C language does not have any tradition of language support for an object model class types with constructors and destructors and user-defined copying, which C++ definitely does; tha C++ also has to define what, if any, uses of such newfangled types make sense with the "oldfangled" u 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 co if the following code were allowed:

```
// Example 36-2: Not standard C++ code, but what if it were allowed?
//
void f() {
 union IllegalImmoralAndFattening {
  std::string s;
  std::auto_ptr<int> p;
 };

 IllegalImmoralAndFattening iiaf;

 iiaf.s = "Hello, world";   // has s's constructor run?
 iiaf.p = new int(4);       // has p's constructor run?
}    // will s get destroyed? should it be? will p get destroyed? should it be?
```

As the comments indicate, serious problems would exist if this were allowed. To avoid further compli language by trying to craft rules that at best might only partly patch up a few of the problems, the probl operations were simply banished.

But don't think that unions are only a holdover from earlier times. Unions are perhaps most useful for s by allowing data to overlap, and this is still desirable in C++ and in today's modern world. For examp the most advanced C++ standard library implementations in the world now use just this technique for implementing the "small `string` optimization," a great optimization alternative that reuses the storage `string` object itself. Here's the idea: For large strings, space inside the `string` object stores the usua the dynamically allocated buffer and housekeeping information like the size of the buffer; for small stri however, the same space is instead reused to store the string contents directly and completely avoid ar memory allocation. For more about the small `string` optimization (and other string optimizations and pessimizations in considerable depth), see Items 13 through 16 in More Exceptional C++ [Sutter02]; s discussion of current commercial `std::string` implementations in [Meyers01].

Dissecting Construction Unions

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

On the plus side, the cited article addresses a real problem, and clearly much effort has been put into c with a good solution. Unfortunately, from well-intentioned beginnings, more than one programmer has astray.

The problems with the design and the code fall into three major categories: legality, safety, and morali

> 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.

The first overall comment that needs to be made is that the fundamental idea behind this code is not leg standard C++. The original article summarizes the key idea:

> The idea is that instead of declaring object members, you instead declare a raw buffer [non-dynamically, as a char array member inside the object pretending to act like a union] and instantia needed objects on the fly [by in-place construction].
>
> [Manley02]

The idea is common, but unfortunately it is also unsound.

Allocating a buffer of one type and then using casts to poke objects of another type in and out, is noncc and nonportable because buffers that are not dynamically allocated (i.e., that are not allocated via `mal` are not guaranteed to be correctly aligned for any other type than the one with which they were origina declared. Even if this technique happens to accidentally work for some types on someone's current cor there's no guarantee it will continue to work for other types or for the same types in the next version of compiler. For more details and some directly related discussion, see Item 30 in Exceptional C++ [Sutt notably the sidebar titled "Reckless Fixes and Optimizations, and Why They're Evil." See also the alig discussion in [Alexandrescu02].

For C++0x, the standards committee is considering adding alignment aids to the language specifically techniques that rely on alignment like this, but that's all still in the future. For now, to make this work r reliably even some of the time, you'd have to do one of the following:

- Rely on the `max_align` hack (see [Manley02] which footnotes the `max_align` hack, or do a Goog for `max_align`).)
- Rely on nonstandard extensions like Gnu's `__alignof__` to make this work reliably on a particul that supports such an extension. (Even though Gnu provides an `ALIGNOF` macro intended to work reliably on other compilers, it too is admitted hackery that relies on the compiler's laying out obje certain ways and making guesses based on `offsetof` inquiries, which might often be a good gues guaranteed by the standard.)

You could work around this by dynamically allocating the array using `malloc` or `new`, which would gu the `char` buffer is suitably aligned for an object of any type, but that would still be a bad idea (it's still

safe) and it would defeat the potential efficiency gains that the original article was aiming for as part c original motivation. An alternative and correct solution would be to use `boost::any` (see below), wh similar allocation/indirection overhead but is at least both safe and correct; more about that later on.

Attempts to work against the language, or to make the language work the way we want it to work instea way it actually does work, are often questionable and should be a big red flag. In the Exceptional C++ sidebar cited earlier, while in an ornery mood, I also accused a similar technique of "just plain wrong followed by some pretty strong language. There can still be cases where it could be reasonable to use that are known to be nonportable but okay in a particular environment (in this case, perhaps using the `n` hack), but even then I would argue that that fact should be noted explicitly and, further, that it still has n general piece of code recommended for wide use.

Into the Code

Let's now consider the code:

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

Always include necessary headers. Because `new` is going to be used below, we need to also `#include` (Note: The `<iostream>` header is okay; later in the original code, not shown here, was a test harness t output using iostreams.)

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

typedef list<int> LIST;
typedef string STRING;

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

The first classic mechanical error here is that `MYUNION` is unsafe to copy because the programmer forg provide a suitable copy constructor and copy assignment operator.

`MYUNION` is choosing to play games that require special work to be done in the constructor and destruc functions are provided explicitly as shown; that's fine as far as it goes. But it doesn't go far enough, be same games require special work in the copy constructor and copy assignment operator, which are not explicitly. That's bad because the default compiler-generated copying operations do the wrong thing; n copy the contents bitwise as a `char` array, which is likely to have most unsatisfactory results, in most c leading straight to memory corruption. Consider the following code:

```
// Example 36-3: MYUNION is unsafe for copying
```

```
//
{
 MYUNION u1, u2;
 u1.getstring() = "Hello, world";
 u2 = u1;                              // copies the bits of u1 to u2
} // oops, double delete of the string (assuming the bitwise copy even made sense
```

Guideline

Observe the Law of the Big Three [Cline99]: If a class needs a custom copy constructor, copy assignment operator, or destructor, it probably needs all three.

Passing on from the classic mechanical error, we next encounter a duo of classic stylistic errors:

```
enum uniontype {NONE,_INT,_LIST,_STRING};
uniontype currtype;

inline int& getint();
inline LIST& getlist();
inline STRING& getstring();
```

There are two stylistic errors here. First, this `struct` is not reusable because it is hard-coded for spec Indeed, the original article recommended hand-coding such a struct every time it was needed. Second, its limited intended usefulness, it is not very extensible or maintainable. We'll return to this frailty agai once we've covered more of the context.

Guideline

Avoid hard-wiring information that needlessly makes code more brittle and limits flexibility.

There are also two mechanical problems. The first is that `currtype` is public for no good reason; this good encapsulation and means any user can freely mess with the type flag, even by accident. The seco mechanical problem concerns the names used in the enumeration; I'll cover that in its own section, "Ur Names," later on.

```
protected:
```

Here we encounter another mechanical error: The internals ought to be private, not protected. The only make them protected would be to make the internals available to derived classes, but there had better derived classes because `MYUNION` is unsafe to derive from for several reasons—not least because of th and abstruse games it plays with its internals and because it lacks a virtual destructor.

Guideline

Always make all data members private. The only exception is the case of a C-style `struct` which isn't intended to encapsulate anything and where all members are public.

```
 union {
  int i;
  unsigned char buff[max(sizeof(LIST),sizeof(STRING))];
 } U;

 void cleanup();
};
```

That's it for the main class definition. Moving on, consider the three parallel accessor functions:

```
inline int& MYUNION::getint()
{
 if(currtype==_INT) {
   return U.i;
 } else {
   cleanup();
   currtype=_INT;
   return U.i;
 } // else
}

inline LIST& MYUNION::getlist()
{
 if(currtype==_LIST) {
   return *(reinterpret_cast<LIST*>(U.buff));
 } else {
   cleanup();
   LIST* ptype = new(U.buff) LIST();
   currtype=_LIST;
   return *ptype;
 } // else
}

inline STRING& MYUNION::getstring()
{
 if(currtype==_STRING) {
   return *(reinterpret_cast<STRING*>(U.buff));
 } else {
   cleanup();
```

```
   STRING* ptype = new(U.buff) STRING();
   currtype=_STRING;
   return *ptype;
 } // else
}
```

A minor nit: The `// else` comments add nothing. It's unfortunate that the only comments in the code ar
ones.

Guideline

Write (only) useful comments. Never write comments that repeat the code; instead, write comments that explain the code and the reasons why you wrote it that way.

More seriously, there are three major problems here. The first is that the functions are not written symm
and whereas the first use of a `list` or a `string` yields a default-constructed object, the first use of `int`
uninitialized object. If that is intended, in order to mirror the ordinary semantics of uninitialized `int` v
then that should be documented; because it is not, the `int` ought to be initialized. For example, if the ca
accesses `getint` and tries to make a copy of the (uninitialized) value, the result is undefined behavior
platforms support copying arbitrary invalid `int` values, and some will reject the instruction at run-time

The second major problem is that this code hinders const-correct use. If the code is really going to be
way, then at least it would be useful to also provide `const` overloads for each of these functions; each
naturally return the same thing as its non-`const` counterpart, but by a reference to `const`.

Guideline

Practice const-correctness.

The third major problem is that this approach is fragile and brittle in the face of change. It relies on typ
switching, and it's easy to accidentally fail to keep all the functions in sync when you add or remove n

Stop reading here and consider: What do you have to do in the published code if you want to add a new
Make as complete a list as you can.

• • • • • •

Are you back? All right, here's the list I came up with. To add a new type, you have to remember to:

- Add a new `enum` value;
- Add a new accessor member;
- Update the `cleanup` function to safely destroy the new type; and
- Add that type to the `max` calculation to ensure `buff` is sufficiently large to hold the new type too.

If you missed one or more of those, well, that just illustrates how difficult this code really is to mainta extend.

Pressing onward, we come to the final function:

```
void MYUNION::cleanup()
{
 switch(currtype) {
  case _LIST: {
   LIST& ptype = getlist();
   ptype.~LIST();
   break;
  } // case
  case _STRING: {
   STRING& ptype = getstring();
   ptype.~STRING();
   break;
  } // case
  default: break;
 } // switch
 currtype=NONE;
}
```

Let's reprise that small commenting nit: The `// case` and `// switch` comments add nothing; it's unfor the only comments in the code are useless ones. It is better to have no comments at all than to have cor are just distractions.

But there's a larger issue here: Rather than having simply `default: break;`, it would be good to mak exhaustive list (including the `int` type) and signal a logic error if the type is unknown—perhaps via ar a `tHRow std::logic_error(...);`.

Again, type switching is purely evil. A Google search for switch C++ Dewhurst will yield all sorts of references on this topic, including [Dewhurst02]. See those references for more details if you need mc convince colleagues to avoid the type-switching beast.

Guideline

Avoid type switching; prefer type safety.

Underhanded Names

There's one mechanical problem I haven't yet covered. This problem first rears its ugly, unshaven, and unshampooed head in the following line:

```
enum uniontype {NONE,_INT,_LIST,_STRING};
```

Never, ever, ever create names that begin with an underscore or contain a double underscore; they're r your compiler and standard library vendor's exclusive use so that they have names they can use withou on your code. Tromp on their names, and their names might just tromp back on you![41]

> [41] The more specific rule is that any name with a double underscore anywhere in it `like__this` that starts with an underscore and a capital letter `_LikeThis` is reserved. You can remember that you like, but it's a bit easier to just avoid both leading underscores and double underscores entire

Don't stop! Keep reading! You might have read that advice before. You might even have read it from m might even be tired of it, and yawning, and ready to ignore the rest of this section. If so, this one's for y this advice is not at all theoretical, and it bites and bites hard in this code.

The enum definition line happens to compile on most of the compilers I tried: Borland 5.5, Comeau 4.1 2.95.3 / 3.1.1 / 3.4, Intel 7.0, and Microsoft Visual C++ 6.0 through 8.0 (2005) beta. But under two of them—Metrowerks CodeWarrior 8.2 and EDG 3.0.1 used with the Dinkumware 4.0 standard library—breaks horribly.

Under Metrowerks CodeWarrior 8, this one line breaks noisily with the first of 52 errors (that's not a t 225 lines of error messages (again, that's not a typo) begin with the following diagnostics, which point one of the commas:

```
### mwcc Compiler:
# File: 36.cpp
# --------------
# 17:         enum uniontype {NONE,_INT,_LIST,_STRING};
# Error: ^
# identifier expected
### mwcc Compiler:
# 18:         uniontype currtype;
# Error:      ^^^^^^^^^
# declaration syntax error
```

followed by 52 further error messages and 215 more lines. What's pretty obvious from the second and is that we should ignore them for now because they're just cascades from the first error—because `unio` never successfully defined, the rest of the code which uses `uniontype` extensively will of course brea

But what's up with the definition of `uniontype`? The indicated comma sure looks like it's in a reasona
doesn't it? There's an identifier happily sitting in front of it, isn't there? All becomes clear when we as
Metrowerks compiler to spit out the preprocessed output… omitting many many lines, here's what the
finally sees:

```
enum uniontype {NONE,_INT,, };
```

Aha! That's not valid C++, and the compiler rightly complains about the third comma because there's r
in front of it.

But what happened to `_LIST` and `_STRING`? You guessed it—tromped on and eaten by the ravenously h
Preprocessor Beast. It just so happens that Metrowerks' implementation has macros that happily strip a
names `_LIST` and `_STRING`, which is perfectly legal and legitimate because it (the implementation) is a
own those `_Names` (as well as `other__names`).

So Metrowerks' implementation happens to eat both `_LIST` and `_STRING`. That solves that part of the n
what about EDG's/Dinkumware's implementations? Judge for yourself:

```
"1.cpp", line 17: error: trailing comma is nonstandard
    enum uniontype {NONE,_INT,_LIST,_STRING};
                                    ^

"1.cpp", line 58: error: expected an expression
    if(currtype==_STRING) {
                 ^

"1.cpp", line 63: error: expected an expression
      currtype=_STRING;
                  ^

"1.cpp", line 76: error: expected an expression
      case _STRING: {
                 ^

4 errors detected in the compilation of "36.cpp".
```

This time, even without generating and inspecting a preprocessed version of the file, we can see what'
The compiler is behaving as though the word `_STRING` just wasn't there. That's because it was—you gu
it—tromped on, not to mention thoroughly chewed up and spat out, by the still-peckish Preprocessor B

I hope that this will convince you that when some of us boring writers natter on about not using `_Names
like__these`, the problem is far from theoretical, far more than mere academic tedium. It's a practica
indeed, because the naming restriction directly affects your relationship with your compiler and standa
writer. Trespass on their turf, and you might get lucky and remain unscathed; on the other hand, you mig

The C++ landscape is wide open and clear and lets you write all sorts of wonderful and flexible code

in pretty much whatever direction your development heart desires, including that it lets you choose pre
whatever names you like outside of namespace `std`. But when it comes to names, C++ also has one bi
grove, surrounded by gleaming barbed wire and signs that say things like "`Employees__Only`Must I
`_Badge` To Enter Here" and "Violators Might be Tromped and Eaten." This is a stellar example of the
one gets for disregarding the `_Warnings`.

Guideline

Never use "underhanded names"ones that begin with an underscore or that contain a double underscore. They are reserved for your compiler and standard library implementation.

Toward a Better Way: `boost::any`

4. Show a better way to achieve a generalized variant type, and comment on any tradeoffs you encou

The original article says:

> [Y]ou might want to implement a scripting language with a single variable type that can either be integer, a string, or a list.
>
> [Manley02]

This is true, and there's no disagreement so far. But the article then continues:

> A union is the perfect candidate for implementing such a composite type.
>
> [Manley02]

Rather, the article has served to show in some considerable detail just why a union is not suitable at al

But if not a union, then what? One very good candidate for implementing such a variant type is [Boost]
facility, along with its `many` and `any_cast`.[42] Interestingly, the complete implementation for the fully
(covering any number/combination of types and even some platform-specific `#ifdef`s) is about the sa
of code as the sample `MYUNION` solution hardwired for just the special case of the three types `int`, `lis`
and `string`and `any` is fully general, extensible, and type-safe to boot, and part of a healthy low-cho
diet.

> [42] For further discussion of this facility, see [Hyslop01].

There is still a tradeoff, however, and it is this: dynamic allocation. The `boost::any` facility does not
achieve the potential efficiency gain of avoiding a dynamic memory allocation, which was part of the
in the original article.

Note too that the `boost::any` dynamic allocation overhead is more than if the original article's code w modified to use (and reuse) a single dynamically allocated buffer that's acquired once for the lifetime because `boost::any` also performs a dynamic allocation every time the contained type is changed.

Here's how the article's demo harness would look if it instead used `boost::any`. The old code that us original article's version of `MYUNION` is shown in comments for comparison:

```
any u;                                  // instead of: MYUNION u;
```

Instead of a handwritten struct, which has to be written again for each use, just use `any` directly. Note t plain class, not a template.

```
// access union as integer
u = 12345;                              // instead of: u.getint() = 12345;
```

The assignment shows `any`'s more natural syntax.

```
cout << "int="<< any_cast<int>(u) << endl;          // or just int(u)
                        // instead of: cout << "int="<< u.getint() << endl;
```

I like `any`'s cast form better because it's more general (including that it is a nonmember) and more natu style; you could also use the less verbose `int(u)` without an `any_cast` if you know the type already. hand, `MYUNION`'s `get`[type] is more fragile, harder to write and maintain, and so forth.

```
// access union as std::list
u = list<int>();
list<int>& l = *any_cast<list<int> >(&u);    // instead of: LIST& list = u.getlis
l.push_back(5);                             // same: list.push_back(5);
l.push_back(10);                            // same: list.push_back(10);
l.push_back(15);                            // same: list.push_back(15);
```

I think `any_cast` could be improved to make it easier to get references, but this isn't too bad. (Aside: discourage using `list` as a variable name when it's also the name of a template in scope; too much roo expression ambiguity.)

So far we've achieved some typability and readability savings. The remaining differences are more mi

```
list<int>::iterator it = l.begin();       // instead of: LIST::iterator it = list

while(it != l.end()) {
 cout << "list item="<< *(it) << endl;
 it++;
} // while
```

Pretty much unchanged. I've let the original comment stand because it's not germane to the side-by-side comparison with `any`.

```
// access union as std::string
u = string("Hello world!");          // instead of: STRING& str = u.getstring();
                                     //             str = "Hello world!";
```

Again, about a wash; I'd say the `any` version is slightly simpler than the original, but only slightly.

```
cout << "string='"<< any_cast<string>(u) << "'"<< endl; //   or just "string(u)"
                    // instead of: cout << "string='"<< str.c_str() << "'"<< endl;
```

As before.

Alexandrescu's Discriminated Unions

Is it possible to fully achieve both of the original goals—safety and avoiding dynamic memory—in a c standard C++ implementation? That sounds like a problem that someone like Andrei Alexandrescu wo sink his teeth into, especially if it could somehow involve complicated templates. As evidenced in [Alexandrescu02], where he describes his discriminated unions (aka `Variant`) approach, it turns out t

- it is (something he would love to tackle), and
- it can (involve weird templates, and just one quote from [Alexandrescu02] says it all: "Did you k unions can be templates?"), so
- he does.

In short, by performing heroic efforts to push the boundaries of the language as far as possible, Alexan `Variant` comes very close to being a truly portable solution. It falls only slightly short and is probabl enough in practice even though it too goes beyond the pale of what the Standard guarantees. Its main pr that, even ignoring alignment-related issues, the `Variant` code is so complex and advanced that it actu on very few compilers—in my testing, I only managed to get it to work with one.

A key part of Alexandrescu's `Variant` approach is an attempt to generalize the `max_align` idea to mal reusable library facility that can itself still be written in conforming standard C++. The reason for wan specifically to deal with the alignment problems in the code we've been analyzing so that a non-dynam buffer can continue to be used in relative safety. Alexandrescu makes heroic efforts to use template metaprogramming to calculate a safe alignment. Will it work portably? His discussion of this question

> Even with the best `Align`, the implementation above is still not 100-percent portable for all types theory, someone could implement a compiler that respects the Standard but still does not work pr

with discriminated unions. This is because the Standard does not guarantee that all user-defined t
ultimately have the alignment of some POD type. Such a compiler, however, would be more of a
of a wicked language lawyer's imagination, rather than a realistic language implementation.

[…] Computing alignment portably is hard, but feasible. It never is 100-percent portable.

[Alexandrescu02]

There are other key features in Alexandrescu's approach, notably a `union` template that takes a `typeli`
template of the types to be contained, visitation support for extensibility, and an implementation techni
will "fake a vtable" for efficiency to avoid an extra indirection when accessing a contained type. Thes
more heavyweight than `boost::any` but are portable in theory. That "portable in theory" part is impor
with Andrei's great work in Modern C++ Design [Alexandrescu01], the implementation is so heavy or
that the code itself contains comments like, "Guaranteed to issue an internal compiler error on: [variou
compilers, Metrowerks, Microsoft, Gnu gcc]," and the mainline test harness contains a commented-ou
helpfully labeled "The construct below didn't work on any compiler."

That is `Variant`'s major weakness: Most real-world compilers don't even come close to being able to
implementation, and the code should be viewed as important but still experimental. I attempted to buil
Alexandrescu's `Variant` code using a variety of compilers: Borland 5.5; Comeau 4.3.0.1; EDG 3.0.1;
3.1.1, and 3.2; Intel 7.0; Metrowerks 8.2; and Microsoft VC++ 6.0, 7.0 (2002), and 7.1 (2003). As so
will know, some of the products in that list are very strong and standards-conforming compilers. None
compilers could successfully compile Alexandrescu's template-heavy source as it was provided.

I tried to massage the code by hand to get it through any of the compilers but was successful only with
VC++ 7.1 (2003). Most of the compilers didn't stand a chance, because they did not have nearly strong
template support to deal with Alexandrescu's code. (Some emitted a truly prodigious quantity of warni
errors—Intel 7.0's response to compiling `main.cpp` was to spew back an impressive 430K worth—re
half a megabyte—of diagnostic messages.)

I had to make three changes to get the code to compile without errors (although still with some narrow
conversion warnings at the highest warning level) under Microsoft VC++ 7.1 (2003):

- Added a missing `typename` in class `AlignedPOD`.
- Added a missing `this->` to make a name dependent in `ConverterTo<>::-Unit<>::DoVisit()`
- Added a final newline character at the end of several headers, as required by the C++ standard (s
conforming compilers aren't strict about this and allow the absence of a final newline as a confor
extension; VC++ is stricter and requires the newline).[43]

[43] Thanks to colleague Jeff Peil for pointing out this requirement in [C++03] § 2.1/1, whic
states: "If a source file that is not empty does not end in a new-line character, or ends in a ne
character immediately preceded by a backslash character, the behavior is undefined."

As the author of [Manley02] commented further about tradeoffs in Alexandrescu's design:

It doesn't use dynamic memory, and it avoids alignment issues and type switching. Unfortunately I have access to a compiler that can compile the code, so I can't evaluate its performance vs. `myun` and `any`. Alexandrescu's approach requires 9 supporting header files totaling ~80KB, which intro its own set of maintenance problems.

�K. Manley, private communication

Those points are all valid.

I won't try to summarize Andrei's three articles further here, but I encourage readers who are interested problem to look them up. They're available online as indicated in the bibliography.

Guideline

If you want to represent variant types, for now prefer to use `boost::any` (or something equally simple).

Once the compiler you are using catches up (in template support) and the Standard catches up (in true a support) and `Variant` libraries catch up (in mature implementations), it will be time to consider using like library tools as type-safe replacements for unions.

Summary

Even if the design and implementation of `MYUNION` are lacking, the motivating problem is both real and considering. I'd like to thank Mr. Manley for taking the time to write his article and raise awareness of for variant type support and Kevlin Henney and Andrei Alexandrescu for contributing their own soluti area. It is a hard enough problem that Manley's and Alexandrescu's approaches are not strictly portabl standards-conforming C++, although Alexandrescu's `Variant` makes heroic efforts to get there�Alexa design is very close to portable in theory, although the implementation is still far from portable in prac because very few compilers can handle the advanced template code it uses.

For now, an approach like `boost::any` is the preferred way to go. If in certain places your measurem that you really need the efficiency or extra features provided by something like Alexandrescu's `Variar` have time on your hands and some template know-how, you might experiment with writing your own s version of the full-blown `Variant` by applying only the ideas in [Alexandrescu02] that are applicable situation.

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 monc
The word "monolith" comes from the words "mono" (one) and "lith" (stone), whose vivid image of a s
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, t
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 memb
functions instead of nonmember functions, even when nonmember nonfriend functions would be possib
at least as good. This has at least two drawbacks:

- (Major) It isolates potentially independent functionality inside a class. The operation in question
otherwise be nice to use with other types, but because it's hardwired into a particular class, that v
possible, whereas if it were exposed as a nonmember function template, it could be more widely

- (Minor) It can discourage extending the class with new functionality. "But wait!" someone might s
doesn't matter whether the class's existing interface leans toward member or nonmember function
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 (
via member functions and therefore presents that as the class's natural idiom, the extended nonme
functions cannot follow the original natural idiom and will always remain visibly second-class jc
come-latelies. That the class presents its functions as members is a semantic statement to its users
will be accustomed to the member syntax that extenders of the class cannot use. (Do we really wa
out of our way to make our users commonly have to look up which functions are members and wh
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.

Counting constructors, there are no fewer than 103 member functions. Really. If that's not monolithic, i
to imagine what would be.

Imagine yourself in an underground cavern, at the edge of a subterranean lake. There is a submerged cl
that leads to the next air-filled cavern. You prepare to dive into the black water and swim through the t
tunnel…

Take a deep breath, and repeat after ISO/IEC 14882:1998(E):[44]

[44] A thick tome also known as the ISO C++ standard [C++03].

```
namespace std {
 template<class charT, class traits = char_traits<charT>,
     class Allocator = allocator<charT> >
 class basic_string {

  // ... some typedefs ...

  explicit basic_string(const Allocator& a = Allocator());
  basic_string(const basic_string& str, size_type pos = 0,
               size_type n = npos, const Allocator& a = Allocator());
  basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
  basic_string(const charT* s, const Allocator& a = Allocator());
  basic_string(size_type n, charT c, const Allocator& a = Allocator());
  template<class InputIterator>
   basic_string(InputIterator begin, InputIterator end,
               const Allocator& a = Allocator());
  ~basic_string();
  basic_string& operator=(const basic_string& str);
  basic_string& operator=(const charT* s);
  basic_string& operator=(charT c);

  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;

  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;

  size_type size() const;
  size_type length() const;
```

```
size_type max_size() const;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const;
void reserve(size_type res_arg = 0);
void clear();
bool empty() const;

const_reference operator[](size_type pos) const;
reference operator[](size_type pos);
const_reference at(size_type n) const;
reference at(size_type n);

basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos, size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<class InputIterator>
 basic_string& append(InputIterator first, InputIterator last);
void push_back(const charT);

basic_string& assign(const basic_string&);
basic_string& assign(const basic_string& str, size_type pos, size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<class InputIterator>
 basic_string& assign(InputIterator first, InputIterator last);

basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(iterator p, charT c);
void insert(iterator p, size_type n, charT c);
template<class InputIterator>
 void insert(iterator p, InputIterator first, InputIterator last);
```

You break surface at a small air pocket and gasp. Don't give up—we're halfway there! Another deep b and:

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);

basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s, size_type n2
```

```
  basic_string& replace(size_type pos, size_type n1, const charT* s);
  basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);

  basic_string& replace(iterator i1, iterator i2, const basic_string& str);
  basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);
  basic_string& replace(iterator i1, iterator i2, const charT* s);
  basic_string& replace(iterator i1, iterator i2, size_type n, charT c);
  template<class InputIterator>
   basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterat

  size_type copy(charT* s, size_type n, size_type pos = 0) const;
  void swap(basic_string<charT,traits,Allocator>&);

  const charT* c_str() const;         // explicit
  const charT* data() const;
  allocator_type get_allocator() const;

  size_type find (const basic_string& str, size_type pos = 0) const;
  size_type find (const charT* s, size_type pos, size_type n) const;
  size_type find (const charT* s, size_type pos = 0) const;
  size_type find (charT c, size_type pos = 0) const;
  size_type rfind(const basic_string& str, size_type pos =  npos) const;
  size_type rfind(const charT* s, size_type pos, size_type n) const;
  size_type rfind(const charT* s, size_type pos = npos) const;
  size_type rfind(charT c, size_type pos = npos) const;

  size_type find_first_of(const basic_string& str, size_type pos = 0) const;
  size_type find_first_of(const charT* s, size_type pos, size_type n) const;
  size_type find_first_of(const charT* s, size_type pos = 0) const;
  size_type find_first_of(charT c, size_type pos = 0) const;
  size_type find_last_of (const basic_string& str, size_type pos = npos) const;
  size_type find_last_of (const charT* s, size_type pos, size_type n) const;
  size_type find_last_of (const charT* s, size_type pos = npos) const;
  size_type find_last_of (charT c, size_type pos = npos) const;

  size_type find_first_not_of(const basic_string& str, size_type pos = 0) const;
  size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
  size_type find_first_not_of(const charT* s, size_type pos = 0) const;
  size_type find_first_not_of(charT c, size_type pos = 0) const;
  size_type find_last_not_of (const basic_string& str, size_type pos = npos) cons
  size_type find_last_not_of (const charT* s, size_type pos, size_type n) const;
  size_type find_last_not_of (const charT* s, size_type pos = npos) const;
  size_type find_last_not_of (charT c, size_type pos = npos) const;

  basic_string substr(size_type pos = 0, size_type n = npos) const;
  int compare(const basic_string& str) const;
  int compare(size_type pos1, size_type n1, const basic_string& str) const;
  int compare(size_type pos1, size_type n1, const basic_string& str,
              size_type pos2, size_type n2) const;
  int compare(const charT* s) const;
  int compare(size_type pos1, size_type n1,
              const charT* s, size_type n2 = npos) const;
 };
}
```

Whew! 103 member functions or member function templates! The rocky tunnel expands and we breal through a new lake's surface just in time. Somewhat water-logged but feeling like better persons for th experience, we look around inside the new cavern to see if the exercise has let us discover anything interesting and useful.

Previous Next

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 library is not premature in the same way as advance optimization of an application program—you can do actual performance measurements of the final system for an application program, whereas you often have little idea just how and where your library will be used! In the latter case, it's best to err on the side of making anything that could be a common operation efficient by default, where doing so doesn't involve tradeoffs with the interface or needless complications. Also, note that the general-purpose library's implementer often has concrete information from past experience (e.g., past user reports, or the implementer's own experience in the problem domain) about what operations are being used by users in time-sensitive ways, so his optimizations are not done blindly and are less likely to be premature.

So don't get hung up about premature optimization—we're not falling into that trap here. Rather, we are primarily investigating just how many of `basic_string`'s members could "equally well" be implemented as nonmember nonfriends. Even if you are expecting many to be implementable as nonmembers, the actual results might well surprise you.

Operations That Must Be Members

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

As a first pass, let's sift out those operations that just have to be members. There are some obvious ones at the beginning of the list.

- constructors (6)
- destructor
- assignment operators (3)
- `[]` operators (2)

Clearly these functions must be members. It's impossible to write a constructor, destructor, assignment operator, or `[]` operator in any other way!

12 functions down, 91 to go…

Operations That Should Be Members

2. Which member functions of `std::string` should be members? Why?

Which operations need access to internal data we would otherwise have to grant via friendship? Clearly these have a reason to be members and normally ought to be. This list includes the following, some of which provide indirect access to (e.g., `begin`) or change (e.g., `reserve`) the internal state of the string:

- `begin` (2)
- `end` (2)
- `rbegin` (2)
- `rend` (2)
- `size`
- `max_size`
- `capacity`
- `reserve`
- `swap`
- `c_str`
- `data`
- `get_allocator`

These ought to be members not only because they're tightly bound to `basic_string` but also because they happen to form the public interface that nonmember nonfriend functions will need to use. Sure, you could implement these as nonmember friends, but why? (There actually is a reason you might prefer to write nonmember nonfriend versions too which just pass through to the members, namely uniformity of interface; this will be touched on again later.)

I'm going to add a few more to this list as fundamental string operations:

- `insert` (1 the three-parameter version)
- `erase` (1 the "iter, iter" version)

- `replace` (2 the "iter, iter, num, char" and templated versions)

We'll return to the question of `insert`, `erase`, and `replace` a little later. `For replace` in particular, it's important to be able to choose well and make the most flexible and fundamental version(s) into member(s).

Into the Fray: Possibly-Contentious Operations That Can Be Nonmember Nonfriends

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.

First in this section, allow me to perform a stunning impersonation of a lightning rod by pointing out that all of the following functions have something fundamental in common, to wit: Each one could obviously as easily and as efficiently be a nonmember nonfriend.

- `at` (2)
- `clear`
- `empty`
- `length`

Sure they can be nonmember nonfriends; that's obvious, no sweat. Of course, these functions also happen to have something else pretty fundamental in common: They're mentioned in the standard library's container and sequence requirements as member functions. Hence the lightning-rod effect…

"Yeah, wait!" I can already hear some standardiste-minded people saying, heading in my direction and resembling the beginnings of a lynch mob, "Not so fast! Don't you know that `basic_string` is designed to meet the C++ standard's container and sequence requirements, and those requirements require or suggest that some of those functions be members? So quit misleading the readers! Those functions are members they just are, so live with it!" Indeed, and true, but for the sake of this discussion I'm going to waive those container and sequence requirements with a dismissive wave of a hand and a quick escape across some back yards past small barking dogs and various clothes drying on the line.

Having left my pursuers far enough behind to resume reasoned discourse, here's the point: For once, the question I'm considering isn't what the container requirements say. It's which functions can, without loss of efficiency, be made nonmember nonfriends, and whether there's any additional benefit to be gained from doing so. If those benefits exist for something the container requirements say must be a member, well, why not point out that the container requirements could be improved while we're at it? And so we shall….

Take `empty` as a case in point. Can we implement it as a nonmember nonfriend? Sure… the standard itself requires the following behavior of `basic_string::empty`, in [C++03, §21.3.3/14]:

Returns: `size() == 0`.

Well, now, that's pretty easy to write as a nonmember without loss of efficiency:

```
template<class charT, class traits, class Allocator>
bool empty(const basic_string<charT, traits, Allocator>& s) {
 return s.size() == 0;
}
```

Of course, if we didn't already have `size` then implementing `empty` as a nonmember nonfriend would not be possible. Alternatively (and with in some cases more reliable performance) we could implement it as:

```
template<class charT, class traits, class Allocator>
bool empty(const basic_string<charT, traits, Allocator>& s) {
 return s.begin() != s.end ();
}
```

Or more generally still:

```
template<typename T>
bool empty(const T& t) {
 return t.begin() != t.end();
}
```

This final version doesn't rely on strings at all, although even then the template argument must provide `begin` and `end`. The class's public member functions do have to provide the necessary and sufficient functionality already. We'll see this point crop up several more times as we consider other functions. (For the rest of the sample code in this Item, I don't go all the way to writing the nonmember nonfriend function templates as fully general templates; I leave each one specific to `basic_string`. But they can all be thusly generalized, and we'll see as we go why there might in fact be good reasons to want to do so.)

Notice that, although we can make `size` a member and implement a nonmember `empty` in terms of it, we could not do the reverse. In several cases here, there's a group of related functions, and perhaps more than one could be nominated as a member and the others implemented in terms of that one as nonmembers. Which function should we nominate to be the member? My advice is to choose the most flexible one that doesn't force a loss of efficiency—that will be the enabling flexible foundation on which the others can be built. In this case, we choose `size` as the member because its result can always be cached (indeed, the standard encourages that it be cached because `size` "should" run in constant time), in which case an `empty` implemented only in terms of `size` is no less efficient than anything we could do with full direct access to the string's internal data structures.[45]

[45] Interestingly, note that the same performance analysis would not hold for `list`, because `list::size` runs in linear time, not constant time.

For another case in point, what about `at`? The same reasoning applies. For both the `const` and non-`const` versions, the standard requires the following:

> Throws: `out_of_range` if `pos >= size()`.
>
> Returns: `operator[](pos)`.

That's easy to provide as a nonmember, too. Each is just a two-line function template, albeit a bit syntactically tedious because of all those template parameters and nested type names—and remember all your `typename`s! Here's the code:

```
template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::const_reference
at(const basic_string<charT, traits, Allocator>& s,
   typename basic_string<charT, traits, Allocator>::size_type pos)
{
 if(pos >= s.size()) throw out_of_range("don't do that");
 return s[pos];
}

template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::reference
at(basic_string<charT, traits, Allocator>& s,
   typename basic_string<charT, traits, Allocator>::size_type pos)
{
 if(pos >= s.size())
   throw out_of_range("I said, don't do that");
 return s[pos];
}
```

What about `clear`? Easy, that's the same as `erase(begin(),end())`. No fuss, no muss. Exercise for the reader, and all that.

What about `length`? Easy, again—it's defined to give the same result as `size`. What's more, note that the other containers don't have `length`, and it's there in the `basic_string` interface as a sort of "string thing," but by making it a nonmember suddenly we can consistently say "`length`" about any container. Not too useful in this case because it's just a synonym for `size`, I grant you, but a noteworthy point in the principle it illustrates—making algorithms nonmembers immediately also makes them more widely useful and usable.

In summary, let's consider the benefits and drawbacks of providing functions like `at` and `empty` as members vs. nonmembers. First, just because we can write these members as nonmembers (and nonfriends) without loss of efficiency, why should we do so? What are the actual or potential benefits? Here are several:

1. Simplicity. Making them nonmembers lets us write and maintain less code. We can write `empty` just once and be done with it forever. Why write it many times as `basic_string::empty` and `vector::empty` and `list::empty` and so forth, including writing it over and over again for each new STL-compliant container that comes along in third-party or internal libraries and even in future versions of the C++ standard? Write it once, be done, and move on.

   Note that there are some limitations to this simplicity advantage. For one thing, some of the functions, such as `at`, won't be able to guarantee a single complexity guarantee, because the complexity will vary by the container that's being used; `at` for `map`s is logarithmic complexity, whereas for `vector`, it's constant time complexity. For another, it might be that not all present or future containers might supply exactly the interface the generalized function expects; as already shown, the sample nonmember version of `empty` that uses `size` won't work for a container that doesn't provide `size`,[46] and would need to be specialized or overloaded for containers that supply an interface that's sufficient but different.

   [46] Of course, such a container wouldn't conform to the standard STL container requirements, but there might be other legacy (or future) containers that are useful if not entirely STL-ish and that we might still want to support.

2. Consistency. It avoids gratuitous incompatibilities between the member algorithms of different containers, and between the member and nonmember versions of algorithms (some existing inconsistencies between members and nonmembers are pointed out in [Meyers01]). If customized behavior is needed, specialization or overloading of the nonmember function templates should be able to accommodate it.

3. Encapsulation. It improves encapsulation (as argued by Meyers strongly and at length in [Meyers00]).

   Having said that, what are the potential drawbacks? Here are two, although in my opinion they are outweighed by the advantages:

4. Namespace pollution. Because `empty` is such a common name, putting it at namespace scope risks namespace pollution—after all, will every function named `empty` want to have exactly these semantics? This argument is valid to a point, but weakened in two ways: First, by noting that encouraging consistent semantics for functions is a Good Thing and, second, by noticing that overload resolution has turned out to be a very good tool for disambiguation, and namespace pollution has never been as big a problem as some have made it out to be in the past. Really, by putting all the names in one place and sharing implementations, we're not polluting that one namespace as much as we're sanitizing the functions by gathering them up and helping to avoid the gratuitous and needless incompatibilities that Meyers mentions.

5. Consistency. You could argue that keeping things like `empty` as members follows the principle of least surprise—similar functions are members, and do we really want to remember to write `length(str)` but `str.size()`? This argument might seem convincing, but only until we notice that this wouldn't be inconsistent at all if people were in the habit of following Meyers' advice in

the first place, routinely writing functions as nonmembers whenever reasonably possible even when there also have to be member versions.

In particular, what if we added nonmember versions of several of the functions that are also members, such as `size`, where the nonmember nonfriends were simple passthroughs or synonyms for the member versions? This would unify the calling syntax and address this consistency concern. For example, if we provide a nonmember `size`, both `size(str)` and `str.size()` work, getting rid of the need to remember which are members in cases that would otherwise feel inconsistent, like `length(str)` vs. `str.size()`. Then people would get used to the nonmember syntax and benefit from its simplicity, consistency, and encapsulation advantages.

By the way, there are other valid technical and design reasons to prefer nonmember syntax. For one thing, it turns out that routinely and consistently providing nonmember functions makes writing templates significantly easier. If templates can rely on the functions' being nonmembers for all types instead of members for some types and nonmembers for others, they can avoid jumping through some traits-like hoops to figure out which is which and work correctly in both cases. (See Item 37 in More Exceptional C++ [Sutter02] for a detailed example.) For another, it would make (erstwhile) members and nonmembers overload—and if you're jumping back in horror and about to strenuously resist that as undesirable, well, experience has shown that it could often be a Good Thing, and some committee members feel that in general the overloading of members and nonmembers might ought to be added to C++0x. (Such a change might well never happen. I'm merely pointing out that even experts feel it's potentially a good idea).

So this objection about consistency really comes down to whether we ought to trade away real benefits in order to follow a tradition of member naming or to change the tradition of member naming in order to reap real benefits. Bottom line, I think this objection is weak, because writing the functions as nonmembers wherever reasonably possible yields a greater consistency, as noted, than the questionable inconsistency being claimed here.

With perhaps these more contentious choices out of the way, then, in the next Item we'll continue with other operations that can be nonmember nonfriends. Some, like those listed earlier, are mentioned in the container requirements as members; again, here we're considering not what the standard says we must do, but rather what, given a blank page, we might choose to do in designing these as members vs. nonmember nonfriends.

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)
{
  resize(s, n, charT());
}
```

That's one way to implement them as nonmember nonfriends that are just as efficient as members.

`assign` and `+=`/`append`/`push_back`

2. Analyze the following member functions of `std::string` and demonstrate whether or not they could or should instead be nonmember functions. Justify your answers.

    a. `assign` and `+=`/`append`/`push_back`

Of this group, let's first tackle `assign`: We have six—count 'em—six forms of `assign`. Fortunately, this case is simple: Most of them are already specified in terms of each other, and all can be implemented in terms of a constructor and `operator=` combination.

Next, `operator+=`, `append`, and `push_back`: What about all those pesky flavors of appending operations, namely the three forms of `operator+=`, the six—count 'em—six forms of `append`, and the lone `push_back`? Just the similarity ought to alert us to the fact that probably at most one needs to be a member. After all, they're doing about the same thing, even if the details differ slightly; for example, appending a character in one case, a string in another case, and an iterator range in still another case. Indeed, as it turns out, all of them can likewise be implemented as nonmember nonfriends without loss of efficiency:

- Clearly `operator+=` can be implemented in terms of `append`, because that's how it's specified in the C++ standard.
- Equally clearly, five of the six versions of `append` can be nonmember non-friends because they are specified in terms of the three-parameter version of `append`, and that in turn can be implemented in terms of `insert`, all of which quite closes the `append` family.
- Determining the status of `push_back` takes only slightly more work, because its operational semantics aren't specified in the `basic_string` clause but only in the container requirements clause, and there we find the specification that `a.push_back(x)` is just a synonym for `a.insert(a.end(),x)`.

"But wait!" someone might say, "the C++ standard says that assignment operators must be members, and `+=` is an assignment operator!" Yes and no. Without getting into the gory details, although `+=` is listed along with all the other `*=` operators as an assignment operator in the C++ language grammar, the only assignment operator that must be a member is, as already noted, `operator=` itself. Therefore,

the following sample implementations of the nonmember `operator+=` functions are perfectly valid C++:

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s,
           const basic_string<charT, traits, Allocator>& t) {
 return s.append(t);
}

template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s, const charT* p) {
 return s.append(p);
}

template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
operator+=(basic_string<charT, traits, Allocator>& s, charT c) {
 return s.append(1, c);
}
```

What's the linchpin holding all the members of this group together as valid non-member nonfriends? It's `insert`; hence `insert`'s status as a good choice to be the member that does the work and encapsulates the append-related access to the string's internals in one place instead of spreading the internal access all over the place in a dozen different functions.

`insert`

b. `insert`

Next, `insert`: For those of you who might think that six-count'em-six versions of `as-sign` and six-count'em-six versions of `append` might have been a little much, those were just the warmup. Now we consider the eight-count'em-eight versions of `insert`.

I've already nominated the three-parameter version of `insert` as a member, and now it's time to justify why. First, as noted before, `insert` is a more general operation than `append`, and having a member `insert` allowed all the `append` operations to be non-member nonfriends; if we didn't have at least one `insert` as a member, then at least one of the `append`s would have had to be a member, so I chose to nominate the more fundamental and flexible operation.

But we have eight-count'em-eight flavors of `insert`. Which one (or more) ought to be the member(s)? Five of the eight forms of `insert` are already specified in terms of the three-parameter form, and the others can also be implemented efficiently in terms of the three-parameter form, so we only need that one form to be a member. The others can all be nonmember nonfriends.

Interlude

For those of you who might think that the eight-count'em-eight versions of `insert` take the cake, well, that was a warmup too. In a moment we'll consider the ten-count'em-ten forms of `replace`. Before we attempt those, though, let's take a short break. In the next Item, we'll tackle an easier function first, because it turns out that `erase` is instructive in building up to dealing with `replace`...

Previous Next

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 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-3 flavors of `assign`, `append`, `insert`, and `replace`—and having dealt with 20 of the 30 already—you v relieved to know that there are only three forms of `erase` and that only two of them belong in this secti After what we just went through for the others, this is like knocking off for a well-deserved coffee bre

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

```
// erase(pos, length)
basic_string& erase(size_type pos = 0, size_type n = npos);
```

```
// erase(iter, …)
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

First, notice that the two families' return types are not consistent: The first version returns a reference t
string, whereas the other two return iterators pointing immediately after the erased character or range.
Second, notice that the two families' argument types are not consistent: The first version takes an offse
length, whereas the other two take an iterator or iterator range; fortunately, we can convert from iterato
offsets via `pos = iter - begin()` and from offsets to iterators via `iter= begin() + pos`.

(Aside: The standard does not require, but an implementer can choose, that `basic_string` objects sto
their data in a contiguous `charT` array buffer in memory. If they do, then the conversion from iterators t
positional offsets and vice versa clearly need incur no overhead. For that matter, I would argue that ev
segmented storage schemes could provide for very efficient conversion back and forth between offsets
iterators using only the container's and iterator's public interfaces. This is all aside from the main "mer
vs. nonmember nonfriend" theme and mentioned only for completeness.)

This allows the first two forms to be expressed in terms of the third. (Again, remember your `typename`
qualifications!)

```
template<class charT, class traits, class Allocator>
basic_string<charT, traits, Allocator>&
erase(basic_string<charT, traits, Allocator>& s,
      typename Allocator::size_type pos = 0,
      typename Allocator::size_type n =
      basic_string<charT, traits, Allocator>::npos)
{
 if(pos > s.size())
   throw out_of_range("yes, we have no bananas");
 typename basic_string<charT, traits, Allocator>::iterator
   first = s.begin()+pos,
   last = n == basic_string<charT, traits, Allocator>::npos
          ? s.end() : first + min(n, s.size() - pos);
 if(first != last)
   s.erase(first, last);
 return s;
}

template<class charT, class traits, class Allocator>
typename basic_string<charT, traits, Allocator>::iterator
erase(basic_string<charT, traits, Allocator>& s,
      typename basic_string<charT, traits, Allocator>::iterator position)
{
 return s.erase(position, position+1);
}
```

OK, coffee break's over…

Back to Work: Replacing `replace`

2. Analyze the remaining member functions of `std::string` and demonstrate whether or not they ca should instead be nonmember functions. Justify your answers.

    a. `replace`

Next, `replace`: Truth be told, the ten-count'em-ten `replace` members are less interesting than they are tedious and exhausting.

At least one of these `replace` functions must be a member (or friend), there being no other way to repl efficiently using the other already-mentioned member functions alone. In particular, note that you can't efficiently implement `replace` in terms of `erase` followed by `insert` or vice versa because both way would require more character shuffling and possibly buffer reallocation.

Note that we have two families of `replace` functions:

```
// replace(pos, length, …)
basic_string& replace(size_type pos1, size_type n1, const basic_string& str); //
basic_string& replace(size_type pos1, size_type n1, // #2
                      const basic_string& str, size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s, size_type n2),
basic_string& replace(size_type pos, size_type n1, const charT* s); // #4
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c); // #5

// replace(iter, iter, …)
basic_string& replace(iterator i1, iterator i2, const basic_string& str); // 6
basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n); //
basic_string& replace(iterator i1, iterator i2, const charT* s); // 8
basic_string& replace(iterator i1, iterator i2, size_type n, charT c); // 9
template<class InputIterator> // 10
 basic_string& replace(iterator i1, iterator i2, InputIterator j1, InputIterator
```

This time, the two families' return types are consistent; that's a small pleasure. But, as with `erase`, the families' argument types are not consistent: One family is based on an offset and length, whereas the ot based on an iterator range. As with `erase`, because we can convert between iterators and positions, w easily implement one family in terms of the other.

When considering which must be members, we want to choose the most flexible and fundamental versi as members and implement the rest in terms of those. Here are a few pitfalls one might encounter whil this analysis, and some tips for how one might avoid them. Consider first the first family:

- One function (#2)? One might notice that the standard specifies all of the first family in terms of t version. Unfortunately, some of the passthroughs would needlessly construct temporary `basic_st` objects, so we can't get by with #2 alone even for just the first family. The standard specifies the observable behavior, but the operational specification isn't necessarily the best way to actually implement a given function.

- Two functions (#3 and #5)? One might notice that all but #5 in the first family can be implemente efficiently in terms of #3, but then #5 would still need to be special-cased to avoid needlessly cr a temporary string object (or its equivalent).

Consider second, the second family:

- One function (#6)? One might notice that the standard specifies all of the second family in terms c Unfortunately, again some of the passthroughs would needlessly construct temporary `basic_stri` objects, so we can't get by with #6 alone even for just the second family.

- Three functions (#7, #9, #10)? One might notice that most of the functions in the second family ca implemented efficiently in terms of #7, except for #9 (for the same reason that made #5 the outcas first family, namely that there was no existing buffer with the correct contents to point to) and #1((which cannot assume that iterators are pointers or even, for that matter, `basic_string::iterat`

- Two functions (#9, #10)! One might then immediately notice that all but #9 in the second family c implemented efficiently in terms of #10, including #7. In fact, assuming string contiguity and position/iterator convertibility as we've already assumed, we could probably even handle all the members of the first family… aha! That's it.

So it appears that the best we can do is two member functions upon which we can then build everythin as nonmember nonfriends. The members are the "iter, iter, num, char" and templated versions. The nonmembers are everything else. (Exercise for the reader: For each of the other eight versions, write s implementations as efficient nonmember nonfriends.)

Note that #10 well illustrates the power of templates—this one function can be used to implement all b of the others without any loss of efficiency and to implement the remaining ones with what would prob be only a minor loss of efficiency (constructing a temporary `basic_string` containing `n` copies of a character).

Time for another quick coffee break…

Coffee Break #2: Copying `copy` and `substr`

b. `copy` and `substr`

Oh, `copy`, schmopy. Note that `copy` is a somewhat unusual beast and that its interface is inconsistent w `std::copy` algorithm. Note again the signature:

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

The function is `const`; it does not modify the string. Rather, what the string object has to do is copy pa itself (up to `n` characters, starting at position `pos`) and dump it into `s`, the target buffer (note, I delibera did not say "C-style string"), which is required to be big enough—if it's not, oh well, then the program

scribble onward into whatever memory happens to follow the string and get stuck somewhere in the Undefined Behavior swamp. And better still, `basic_string::copy` does not, repeat not, append a nu object to the target buffer, which is one reason it's not a C-style string. (The other reason is that `charT` doesn't need to be `char` ; this function will copy into a buffer of whatever kind of characters the string is made of). The lack of null-termination is also what makes `copy` a dangerous function.

Guideline

Never use functions that write to range-unchecked buffers (e.g., `strcpy`, `sprintf`) or could fail to null-terminate C-style strings (e.g.,`strncpy`, `basic_string::copy`). They are not only crashes waiting to happen but a clear and present security hazard—buffer overrun attacks continue to be a perennially popular pastime for hackers and malware writers. [For more on this topic, turn to Items 2 and 3.]

All the required work could be done pretty much as simply, and a lot more flexibly, just by using plain `std::copy`:

```
string s = "0123456789";

char* buf1 = new char[5];
s.copy(buf1, 0, 5);           // ok: buf will contain the chars '0', '1', '2', '3
copy(s.begin(), s.begin()+5, buf1);
                              // ok: buf will contain the chars '0', '1', '2', '3
int* buf2 = new int[5];
s.copy(buf2, 0, 5);           // error: first parameter is not char*
copy(s.begin(), s.begin()+5, buf2);
                             // ok: buf2 will contain the values corresponding to
                             // '0', '1', '2', '3', '4' (e.g., ASCII values)
```

Incidentally, this code also shows how `basic_string::copy` can trivially be written as a nonmembe nonfriend, most trivially in terms of the `copy` algorithm—another exercise for the reader, but do remen handle the `n == npos` special case correctly.

Well, that one didn't leave us breathing too hard, so while we're still relaxing, let's knock off another s one at the same time: `substr`. Recall its signature: [47]

[47] Astute readers might have noticed that this function chooses to take its parameters in the orde "position, length" whereas the `copy` we just considered takes the very same parameters in the order "length, position." Besides being aesthetically inconsistent, this can actually be dangerous. Trying to remember which function takes the parameters in which order makes for an easy trap fo users of `basic_string` to stumble into, and because both parameters also happen to be of the same type the poor users who get it wrong will find that their code continues to happily compile without any errors or warnings and continue to happily run… well, except only every so often

hiccupping and generating odd user support calls when odd strings get emitted after odd and wro substrings get taken. A jaundiced eye could view this as the moral equivalent of neglectfully leaving a land mine neatly labeled "Design By Committee™" lying around the countryside just waiting to blow up without warning beneath the unwary. But I digress.

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

Notice that `substr` can be easily implemented as a nonmember nonfriend because it's syntactic sugar f string constructor—after all, the standard itself specifies that it must simply return a fresh `basic_stri` object constructed using the equivalent of `basic_string<charT,traits,Allocator>(data()+pos min(n, size()-pos))`. That is, creating a new string that's a substring of an existing one can be done equally well with or without `substr`, using the more general `string` constructor, which already does and a lot more besides:

```
string s = "0123456789";

string s2 = s.substr(0, 5);          // s2 contains "01234"
string s3(s.data(), 5);              // s3 contains "01234"
string s4(s.begin(), s.begin()+5);   // s4 contains "01234"
```

All right, break's over. Back to work again… fortunately we have only two families left to consider: `compare` and the `*find*`s.

Almost There: Comparing `compare`s

c. `compare`

The penultimate family is `compare`. It has five members, all of which can be trivially shown to be implementable efficiently as nonmember nonfriends. How? Because in the standard they're specified i of `basic_string`'s `size` and `data`, which we already decided to make members, and `traits::compa` which does the real work. (For more information about `traits::compare`, see [Josuttis99].)

Wow, wait a minute. That was almost easy! Let's not question it but move right along….

The Home Stretch: Finding the `find`s

d. `find` family (`find`, `find_*`, and `rfind`)

Our relief doesn't last long, alas. Lest the sight of the eight-count'em-eight flavors of `insert` and the te count'em-ten versions of `replace` wasn't enough to bring you verily to the verge of tears, we end on a unsurpassed (in `basic_string`) distressing note, namely this: the 24-count'em-24 (yes, really, I am no making this up) variations on `find`-like algorithms.

There are six families of `find` functions, each with exactly four members:

- `find`. Forward search for the first occurrence of a string or character (`str`, `s`, or `c`) starting at po
- `rfind`. Backward search for the first occurrence of a string or character (`str`, `s`, or `c`) starting at `pos`
- `find_first_of`. Forward search for the first occurrence of any of one or more characters (`str`, `c`) starting at point `pos`
- `find_last_of`. Backward search for the first occurrence of any of one or more characters (`str`, `c`) starting at point `pos`
- `find_first_not_of`. Forward search for the first occurrence of any but one or more characters `s`, or `c`) starting at point `pos`
- `find_last_not_of`. Backward search for the first occurrence of any but one or more characters `s`, or `c`) starting at point `pos`

Each family has four members:

- "str, pos" where `str` contains the characters to search for (or not) and `pos` is the starting position string
- "ptr, pos, n" where `ptr` is a `charT*` pointing to a buffer of length `n` containing the characters to se for (or not) and `pos` is the starting position in the string
- "ptr, pos" where `ptr` is a `charT*` pointing to a null-terminated buffer containing the characters to for (or not) and `pos` is the starting position in the string
- "c, pos" where `c` is the character to search for (or not) and `pos` is the starting position in the strin

All of these can be written efficiently as nonmember nonfriends; the implementations are left as exerci the reader. Having said that, we're done!

But let's add one final note about string finding. In fact, you might have noticed that, in addition to the extensive bevy of `basic_string::*find*` algorithms, the C++ standard also provides a not-quite-as extensive-but-still-plentiful bevy of `std::*find*` algorithms. In particular:

- `std::find` can do the work of `basic_string::find`
- `std::find` using `reverse_iterator`s, or `std::find_end`, can do the work of `basic_string::rfind`
- `std::find_first_of`, or `std::find` with an appropriate predicate, can do the work of `basic_string::find_first_of`
- `std::find_first_of`, or `std::find` with an appropriate predicate, using `reverse_-iterator`

do the work of `basic_string::find_last_of`

- `std::find` with an appropriate predicate can do the work of `basic_string::find_first_not`
- `std::find` with an appropriate predicate and using `reverse_iterator`s can do the work of `basic_string::find_last_not_of`

What's more, the nonmember algorithms are more flexible, because they work on more than just strings Indeed, all the `basic_string::*find*` algorithms could be iplemented using the `std::find` and `std::find_end`, tossing in appropriate predicates and/or `reverse_iterator`s as necessary.

So what about just ditching the `basic_string::*find*` families altogether and just telling programm use the existing `std::find*` algorithms? Sure, but be careful: One caution here is that, even though the `basic_string::*find*` work can be emulated, doing it with the default implementations of `std::fi` would incur significant loss of performance in some cases, and there's the rub. The three forms each o and `rfind` that search for substrings (not just individual characters) can be made much more efficient t brute-force search that tries each position and compares the substrings starting at those positions. Ther well-known algorithms that construct finite state machines on the fly to run through a string and find a substring (or prove its absence) in linear time, and it might be desirable to take advantage of such tech

To take advantage of such optimizations, could we provide overloads (not specializations, see Item 7) `std::find*` that work on `basic_string::iterator`s? Yes, but only if `basic_string::iterator` i class type, not a plain `charT*`. The reason is that if it were a plain pointer type and we did specialize `std::find` for it, we'd be specializing `std::find` for all pointers of that type, which is clearly wrong because not all character pointers necessarily point into `std::string`s. Thus we would need `basic_string::iterator` to be a distinct type that we can detect and specialize on. Then those specializations could perform the optimizations and work at full efficiency for matching substrings.

Summary

Decomposition and encapsulation are Good Things. In particular, it's often best to separate the algorith from the container, which is what the STL does most of the time.

It's widely accepted that `basic_string` has way too many member functions. Of the 103 functions in `basic_string`, only 32 really need to be members, and 71 could be written as nonmember nonfriends without loss of efficiency. In fact, many of them needlessly duplicate functionality already available as algorithms or are themselves algorithms that would be useful more widely if only they were decoupled `basic_string` instead of being buried inside it.

Don't repeat `basic_string`'s mistakes in your designs—decouple your algorithms from your containe template specialization or overloading to get special- purpose behavior where warranted (as for subst searching), and above all follow these guidelines… your users will thank you, because you will be ac decreasing the surface area of the libraries that you ask them to learn.

Guidelines

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

Where possible, prefer writing functions as nonmember non-friends.

Previous Next

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).

[Hyslop00] H. Sutter and J. Hyslop. "Virtually Yours" (C/C++ Users Journal, 18(12), December 2000).

[Hyslop01] H. Sutter and J. Hyslop. "I'd Hold Anything For You" (C/C++ Users Journal, 19(12), December 2001).

[JikesRVM] Jikes RVM home page.

[Jones96] R. Jones and R. Lins. Garbage Collection (Wiley, 1996).

[Josuttis99] N. Josuttis. The C++ standard Library (Addison-Wesley, 1999).

[Kalev01] D. Kalev. "Designing a Generic Callback Dispatcher" (DevX, 2001).

[Koenig96] A. Koenig, "When Memory Runs Low" (C++ Report, 8(6), June 1996).

[Langer00] A. Langer and K. Kreft. Standard C++ IOStreams and Locales (Addison-Wesley, 2000).

[Lippman98] S. Lippman and J. Lajoie. C++ Primer, Third Edition (Addison-Wesley, 1998).

[Liskov88] B. Liskov. "Data Abstraction and Hierarchy" (SIGPLAN Notices, 23(5), May 1988).

[Manley02] K. Manley. "Using Constructed Types in Unions" (C/C++ Users Journal, 20(8), August 2002).

[Martin95] R. C. Martin. Designing Object-Oriented Applications Using the Booch Method (Prentice-Hall, 1995).

[Marrie00] L. Marrie. "Alternating Skip Lists" (Dr. Dobb's Journal, 25(8), August 2000).

[Meyers96] S. Meyers. More Effective C++ (Addison-Wesley, 1996).

[Meyers97] S. Meyers. Effective C++, Second Edition (Addison-Wesley, 1997).

[Meyers99] S. Meyers. Effective C++ CD: 85 Specific Ways to Improve Your Programs and Designs (Addison-Wesley, 1999).

[Meyers00] S. Meyers. "How Non-Member Functions Improve Encapsulation" (C/C++ Users Journal, 18(2), February 2000).

[Meyers01] S. Meyers. Effective STL (Addison-Wesley, 2001).

[Newkirk97] J. Newkirk. "Private Interface" (Object Mentor, 1997).

[ObjectMentor] Object Mentor Inc.

[Stroustrup88] B. Stroustrup. "Parameterized Types for C++" (Proc. USENIX Conference, Denver, October 1988).

[Stroustrup94] B. Stroustrup. The Design and Evolution of C++ (Addison-Wesley, 1994).

[Stroustrup99] B. Stroustrup. "Learning Standard C++ as a New Language" (C/C++ Users Journal, 17(5), May 1999).

[Stroustrup00] B. Stroustrup. The C++ Programming Language, Special Edition (Addison-Wesley, 2000).

[Sutter99] H. Sutter. "ACID Programming" (Guru of the Week #61, September 1999).

[Sutter00] H. Sutter. Exceptional C++ (Addison-Wesley, 2000).

[Sutter02] H. Sutter. More Exceptional C++ (Addison-Wesley, 2002).

[Sutter02a] H. Sutter. "The Group of Seven: Extensions Under Consideration for the C++ Standard Library" (C/C++ Users Journal Experts Forum, 20(4), April 2002).

[Sutter02b] H. Sutter. "Smart(er) Pointers" (C/C++ Users Journal, 20(8), August 2002).

[Sutter02c] H. Sutter. "Standard C++ Meets Managed C++" (C/C++ Users Journal, C++ .NET Solutions Supplement, 20(9), September 2002).

[Vandevoorde03] D. Vandevoorde and N. Josuttis. C++ Templates: The Complete Guide (Addison-Wesley, 2003).
