CS106L

Standard C++ Programming Laboratory

Course Reader
Fall 2010

Keith Schwarz
Stanford University

Acknowledgements

This course reader represents the culmination of two years' work on CS106L and its course handouts. Neither the class nor this reader would have been possible without Julie Zelenski's support and generosity during CS106L's infancy. I strongly encourage you to take one of Julie's classes – you will not be disappointed.

I'd also like to extend thanks to all of the CS106L students I've had the pleasure of teaching over the years. It is truly a joy to watch students light up when they see exactly what C++ can do. The long hours that went into this course reader would not have been possible without the knowledge that students are genuinely interested in the material.

Additionally, I would like to thank the brave souls who were generous enough to proofread draft versions of this course reader. Yin Huang and Steven Wu offered particularly apt advice on style and grammar. Ilya Sherman gave wonderful suggestions on typesetting and layout and caught many errors that slipped under my radar. Kyle Knutson helped double-check the correctness of the code in the extended examples. David Goldblatt helped me stay on top of recent developments in C++0x and pointed out how to make many of the STL practice problems truer to the spirit of the library. Sam Schreiber provided excellent advice about the overall structure of the reader and was the inspiration for the "Critiquing Class Design" chapter. Leonid Shamis astutely suggested that I expand the section on development environments. Brittney Fraser's amazing feedback made many of the examples easier to understand and prevented several major errors from making it into this reader.

This is the third edition of this course reader. There are certainly layout problems, typoz, grammatically errors, and speling misstakes that have made it into this version. If you have any comments, corrections, or suggestions, please send me an email at htiek@cs.stanford.edu.

Table of Contents

Part Zero

Introduction

Suppose we want to write a function that computes the average of a list of numbers. One implementation is given here:

```
double GetAverage(double arr[], int numElems) {
    double total = 0.0;
    for(int h = 0; h < numElems; ++h)
        total += arr[h] / numElems;

    return total;
}
```

An alternative implementation is as follows:

```
template <typename ForwardIterator>
double GetAverage(ForwardIterator begin, ForwardIterator end) {
    return accumulate(begin, end, 0.0) / distance(begin, end);
}
```

Don't panic if you don't understand any of this code – you're not expected to at this point – but even without an understanding of how either of these functions work it's clear that they are implemented differently. Although both of these functions are valid C++ and accurately compute the average, experienced C++ programmers will likely prefer the second version to the first because it is safer, more concise, and more versatile. To understand why you would prefer the second version of this function requires a solid understanding of the C++ programming language. Not only must you have a firm grasp of how all the language features involved in each solution work, but you must also understand the benefits and weaknesses of each of the approaches and ultimately which is a more versatile solution.

The purpose of this course is to get you up to speed on C++'s language features and libraries to the point where you are capable of not only writing C++ code, but also critiquing your design decisions and arguing why the cocktail of language features you chose is appropriate for your specific application. This is an ambitious goal, but if you take the time to read through this reader and work out some of the practice problems you should be in excellent C++ shape.

Who this Course is For

This course is designed to augment CS106B/X by providing a working knowledge of C++ and its applications. C++ is an industrial-strength tool that can be harnessed to solve a wide array of problems, and by the time you've completed CS106B/X and CS106L you should be equipped with the skill set necessary to identify solutions to complex problems, then to precisely and efficiently implement those solutions in C++.
This course reader assumes a knowledge of C++ at the level at which it would be covered in the first two weeks of CS106B/X. In particular, I assume that you are familiar with the following:

0. How to print to the console (i.e. `cout` and `endl`)
1. Primitive variable types (`int`, `double`, etc.)
2. The `string` type.
3. `enum`s and `struct`s.
4. Functions and function prototypes.
5. Pass-by-value and pass-by-reference.
6. Control structures (`if`, `for`, `while`, `do`, `switch`).
7. CS106B/X-specific libraries (`genlib.h`, `simpio.h`, the ADTs, etc.)

If you are unfamiliar with any of these terms, I recommend reading the first chapter of *Programming Abstractions in C++* by Eric Roberts and Julie Zelenski, which has an excellent treatment of the material. These concepts are fundamental to C++ but aren't that particular to the language – you'll find similar constructs in C, Java, Python, and other languages – and so I won't discuss them at great length. In addition to the language prerequisites, you should have at least one quarter of programming experience under your belt (CS106A should be more than enough). We'll be writing a lot of code, and the more programming savvy you bring to this course, the more you'll take out of it.

How this Reader is Organized

The course reader is logically divided into six sections:

0. **Introduction**: This section motivates and introduces the material and covers information necessary to be a working C++ programmer. In particular, it focuses on the history of C++, how to set up a C++ project for compilation, and how to move away from the `genlib.h` training wheels we've provided you in CS106B/X.

1. **A Better C**: C++ supports *imperative programming*, a style of programming in which programs are sequences of commands executed in order. In this sense, C++ can be viewed as an extension to the C programming language which makes day-to-day imperative programming more intuitive and easier to use. This section of the course reader introduces some of C++'s most common libraries, including the standard template library, and shows how to use these libraries to build imperative programs. In addition, it explores new primitives in the C++ language that originally appeared in the C programming language, namely pointers, C strings, and the preprocessor.

2. **Data Abstraction**. What most distinguishes C++ from its sibling C is the idea of *data abstraction*, that the means by which a program executes can be separated from the ways in which programmers talk about that program. This section of the course reader explores the concept of abstraction, how to model it concretely in C++ using the `class` keyword, and an assortment of language features which can be used to refine abstractions more precisely.

3. **Object-Oriented Programming**. Object-oriented programming is an entirely different way of thinking about program design and can dramatically simplify complex software systems. The key concepts behind object-orientation are simple, but to truly appreciate the power of object-oriented programming you will need to see it in action time and time again. This section of the course reader explores major concepts in object-oriented programming and how to realize it in C++ with inheritance and polymorphism.

4. **Generic Programming**. Generic programming is a style of programming which aims to build software that can tackle an array of problems far beyond what it was initially envisioned to perform. While a full treatment of generic programming is far beyond the scope of an introductory C++ programming class, many of the ideas from generic programming are accessible and can fundamentally change the ways in which you think about programming in C++. This section

explores the main ideas behind generic programming and covers several advanced C++ programming techniques not typically found in an introductory text.

5. **More to Explore**. C++ is an enormous language and there simply isn't enough time to cover all of its facets in a single course. To help guide further exploration into C++ programming, this course reader ends with a treatment of the future of C++ and a list of references for further reading.

Notice that this course reader focuses on C++'s standard libraries before embarking on a detailed tour of its language features. This may seem backwards – after all, how can you understand libraries written in a language you have not yet studied? – but from experience I believe this is the best way to learn C++. A comprehensive understanding of the streams library and STL requires a rich understanding of templates, inheritance, functors, and operator overloading, but even without knowledge of these techniques it's still possible to write nontrivial C++ programs that use these libraries. For example, after a quick tour of the streams library and basic STL containers, we'll see how to write an implementation of the game Snake with an AI-controlled player. Later, once we've explored the proper language features, we'll revisit the standard libraries and see how they're put together.

To give you a feel for how C++ looks in practice, this course reader contains several extended examples that demonstrate how to harness the concepts of the previous chapters to solve a particular problem. I *strongly* suggest that you take the time to read over these examples and play around with the code. The extended examples showcase how to use the techniques developed in previous chapters, and by seeing how the different pieces of C++ work together you will be a much more capable coder. In addition, I've tried to conclude each chapter with a few practice problems. Take a stab at them – you'll get a much more nuanced view of the language if you do. Solutions to some of my favorite problems are given in Appendix One. Exercises with solutions are marked with a diamond (♦).

C++ is a large language and it is impossible to cover all of its features in a single course. To help guide further exploration into C++ techniques, most chapters contain a "More to Explore" section listing important topics and techniques that may prove useful in your future C++ career.

Supplemental Reading

This course reader is by no means a complete C++ reference and there are many libraries and language features that we simply do not have time to cover. However, the portions of C++ we do cover are among the most-commonly used and you should be able to pick up the remaining pieces on a need-to-know basis. If you are interested in a more complete reference text, Bjarne Stroustrup's *The C++ Programming Language, Third Edition* is an excellent choice. Be aware that *TC++PL* is *not* a tutorial – it's a reference – and so you will probably want to read the relevant sections from this course reader before diving into it. If you're interested in a hybrid reference/tutorial, I would recommend *C++ Primer, Fourth Edition* by Lippman, Lajoie, and Moo. As for online resources, the C++ FAQ Lite at www.parashift.com/c++-faq-lite/ has a great discussion of C++'s core language features. cplusplus.com has perhaps the best coverage of the C++ standard library on the Internet, though its discussion of the language as a whole is fairly limited.

Onward and Forward!

Chapter 0: What is C++?

C++ is a general purpose programming language with a bias towards systems programming that

- *is a better C.*
- *supports data abstraction.*
- *supports object-oriented programming.*
- *supports generic programming*

– Bjarne Stroustrup, inventor of C++ [Str09.2]

Every programming language has its own distinct flavor influenced by its history and design. Before seriously studying a programming language, it's important to learn *why* the language exists and what its objectives are. This chapter covers a quick history of C++, along with some of its design principles.

An Abbreviated History of C++[*]

The story of C++ begins with Bjarne Stroustrup, a Danish computer scientist working toward his PhD at Cambridge University. Stroustrup's research focus was *distributed systems*, software systems split across several computers that communicated over a network to solve a problem. At one point during his research, Stroustrup came up with a particularly clever idea for a distributed system. Because designing distributed systems is an enormously complicated endeavor, Stroustrup decided to test out his idea by writing a simulation program, which is a significantly simpler task. Stroustrup chose to write this simulation program in a language called Simula, one of the earliest object-oriented programming languages. As Stroustrup recalled, initially, Simula seemed like the perfect tool for the job:

> It was a pleasure to write that simulator. The features of Simula were almost ideal for the purpose, and I was particularly impressed by the way the concepts of the language helped me think about the problems in my application. The class concept allowed me to map my application concepts into the language constructs in a direct way that made my code more readable than I had seen in any other language...
>
> I had used Simula before... but was very pleasantly surprised by the way the mechanisms of the Simula language became increasingly helpful as the size of the program increased. [Str94]

In Simula, it was possible to model a *physical* computer using a computer *object* and a *physical* network using a network *object*, and the way that physical computers sent packets over physical networks corresponded to the way computer objects sent and received messages from network objects. But while Simula made it easier for Stroustrup to develop the simulator, the resulting program was so slow that it failed to produce any meaningful results. This was not the fault of Stroustrup's implementation, but of the language Simula itself. Simula was bloated and language features Stroustrup didn't use in his program were crippling the simulator's efficiency. For example, Stroustrup found that eighty percent of his program time was being spent on garbage collection despite the fact that the simulation didn't create any garbage. [Str94] In other words, while Simula had *decreased* the time required to build the simulator, it dramatically *increased* the time required for the simulator to execute.

Stroustrup realized that his Simula-based simulator was going nowhere. To continue his research, Stroustrup scrapped his Simula implementation and rewrote the program in a language he knew ran quickly and efficiently: BCPL. BCPL has since gone the way of the dodo, but at the time was a widely used,

* This section is based on information from *The Design and Evolution of C++* by Bjarne Stroustrup.

low-level systems programming language. Stroustrup later recalled that writing the simulator in BCPL was "horrible." [Str94] As a low-level language, BCPL lacked objects and to represent computers and networks Stroustrup had to manually lay out and manipulate the proper bits and bytes. However, BCPL programs were far more efficient than their Simula counterparts, and Stroustrup's updated simulator worked marvelously.

Stroustrup's experiences with the distributed systems simulator impressed upon him the need for a more suitable tool for constructing large software systems. Stroustrup sought a hybridization of the best features of Simula and BCPL – a language with both high-level constructs and low-level runtime efficiency. After receiving his PhD, Stroustrup accepted a position at Bell Laboratories and began to create such a language. Settling on C as a base language, Stroustrup incorporated high-level constructs in the style of Simula while still maintaining C's underlying efficiency. After several revisions, *C with Classes*, as his language was known, accumulated other high-level features and was officially renamed C++. C++ was an overnight success and spread rapidly into the programming community; for many years the number of C++ programmers was doubling every seven months. By 2007, there were over three million C++ programmers worldwide, and despite competition from other languages like Java and Python the number of C++ programmers is still increasing. [Str09] What began as Stroustrup's project at Bell Laboratories became an ISO-standardized programming language found in a variety of applications.

C++ as a Language

When confronted with a new idea or concept, it's often enlightening to do a quick Wikipedia search to see what others have to say on the subject. If you look up C++ this way, one of the first sentences you'll read (at least, at the time of this writing) will tell you that C++ is a general-purpose, compiled, statically-typed, multiparadigm, mid-level programming language. If you are just learning C++, this description may seem utterly mystifying. However, this sentence very aptly captures much of the spirit of C++, and so before continuing our descent into the realm of C++ let's take a few minutes to go over exactly what this definition entails.

C++ is a General-Purpose Programming Language

Programming languages can be broadly categorized into two classes – domain-specific programming languages and general-purpose programming languages. A language is *domain-specific* if it is designed to solve a certain class of problems in a particular field. For example, the MATLAB programming language is a domain-specific language designed for numerical and mathematical computing, and so has concise and elegant support for matrix and vector operations. Domain-specific languages tend to be extremely easy to use, particularly because these languages let programmers express common operations concisely and elegantly because the language has been designed with them in mind. As an example, in MATLAB it is possible to solve a linear system of equations using the simple syntax `x = A\b`. The equivalent C++ or Java code would be significantly more complex. However, because domain-specific languages are optimized on a particular class of problems, it can be difficult if not impossible to adapt those languages into other problem domains. This has to do with the fact that domain-specific languages are custom-tailored to the problems they solve, and consequently lack the vocabulary or syntactic richness to express structures beyond their narrow scope. This is best illustrated by analogy – an extraordinary mathematician with years of training would probably have great difficulty holding a technical discussion on winemaking with the world's expert oenologist simply because the vocabularies of mathematics and winemaking are entirely different. It might be possible to explain viticulture to the mathematician using terms from differential topology or matrix theory, but this would clearly be a misguided effort.

Contrasting with domain-specific languages are *general-purpose* languages which, as their name suggests, are designed to tackle all categories of problems, not just one particular class. This means that general-purpose languages are more readily adapted to different scenarios and situations, but may have a harder time describing some of the fundamental concepts of those domains than a language crafted specifically

for that purpose. For example, an American learning German as a second language may be fluent enough in that language to converse with strangers and to handle day-to-day life, but might have quite an experience trying to hold a technical conversation with industry specialists. This is not to say, of course, that the American would not be able to comprehend the ideas that the specialist was putting forth, but rather that any discussion the two would have would require the specialist to define her terms as the conversation unfolded, rather than taking their definitions for granted at the start.

C++ is a general-purpose programming language, which means that it is robust enough to adapt to handle all sorts of problems without providing special tools that simplify tasks in any one area. This is a trade-off, of course. Because C++ is general-purpose, it will not magically provide you a means for solving a particular problem; you will have to think through a design for your programs in order for them to work correctly. But because C++ is general-purpose, you will be hard-pressed to find a challenge for which C++ is a poor choice for the solution. Moreover, because C++ is a general-purpose language, once you have learned the structures and techniques of C++, you can apply your knowledge to any problem domain without having to learn new syntax or structures designed for that domain.

C++ is a Compiled Language

The programs that actually execute on a computer are written in machine language, an extremely low-level and hardware-specific language that encodes individual instructions for the computer's CPU. Machine languages are indecipherable even to most working programmers because these languages are designed to be read by computer hardware rather than humans. Consequently, programmers write programs in programming languages, which are designed to be read by humans. In order to execute a program written in a programming language, that program must somehow be converted from its source code representation into equivalent machine code for execution. How this transformation is performed is not set in stone, and in general there are two major approaches to converting source code to machine code. The first of these is to *interpret* the program. In *interpreted languages*,a special program called the *interpreter* takes in the program's source code and translates the program as it is being executed. Whenever the program needs to execute a new piece of code, the interpreter reads in the next bit of the source code, converts it into equivalent machine code, then executes the result. This means that if the same interpreted program is run several times, the interpreter will translate the program anew every time. The other option is to *compile* the program. In a *compiled language*, before running the program, the programmer executes a special program called the *compiler* on the source code which translates the entire program into machine code. This means that no matter how many times the resulting program is run, the compiler is only invoked once. In general, interpreted languages tend to run more slowly than compiled languages because the interpreter must translate the program as it is being executed, whereas the translation work has already been done in the case of compiled languages. Because C++ places a premium on efficiency, C++ is a compiled language. While C++ interpreters do exist, they are almost exclusively for research purposes and rarely (if at all) used in professional settings.

What does all of this mean for you as a C++ programmer? That is, why does it matter whether C++ is compiled or interpreted? A great deal, it turns out; this will be elaborated upon in the next segment on static type checking. However, one way that you will notice immediately is that you will have to compile your programs every time you make a change to the source code that you want to test out. When working on very large software projects (on the order of millions to hundreds of millions of lines of code), it is not uncommon for a recompilation to take hours to complete, meaning that it is difficult to test out lots of minor changes to a C++ program. After all, if every change takes three minutes to test, then the number of possible changes you can make to a program in hopes of eliminating a bug or extending functionality can be greatly limited. On the other hand, though, because C++ is compiled, once you have your resulting program it will tend to run much, *much* faster than programs written in other languages. Moreover, you don't need to distribute an interpreter for your program in addition to the source – because C++ programs compile down directly to the machine code, you can just ship an executable file to whoever wants to run your program and they should be able to run it without any hassle.

C++ is a Statically-Typed Language

One of the single most important aspects of C++ is that it is a statically-typed language. If you want to manipulate data in a C++ program, you must specify in advance what the *type* of that data is (for example, whether it's an integer, a real number, English text, a jet engine, etc.). Moreover, this type is set in stone and cannot change elsewhere in the source code. This means that if you say that an object is a coffee mug, you cannot treat it as a stapler someplace else.

At first this might seem silly – *of course* you shouldn't be able to convert a coffee mug into a stapler or a ball of twine into a jet engine; those are entirely different entities! You are completely correct about this. Any program that tries to treat a coffee mug as though it is a stapler is bound to run into trouble because a coffee mug *isn't* a stapler. The reason that static typing is important is that these sorts of errors are caught at *compile-time* instead of at *runtime*. This means that if you write a program that tries to make this sort of mistake, the program won't compile and you won't even have an executable containing a mistake to run. If you write a C++ program that tries to treat a coffee mug like a stapler, the compiler will give you an error and you will need to fix the problem before you can test out the program. This is an extremely powerful feature of compiled languages and will dramatically reduce the number of runtime errors that your programs encounter. As you will see later in this book, this also enables you to have the compiler verify that complex relationships hold in your code and can conclude that if the program compiles, your code does not contain certain classes of mistakes.

C++ is a Multi-Paradigm Language

C++ began as a hybrid of high- and low-level languages but has since evolved into a distinctive language with its own idioms and constructs. Many programmers treat C++ as little more than an object-oriented C, but this view obscures much of the magic of C++. C++ is a *multiparadigm* programming language, meaning that it supports several different programming styles. C++ supports *imperative* programming in the style of C, meaning that you can treat C++ as an upgraded C. C++ supports *object-oriented* programming, so you can construct elaborate class hierarchies that hide complexity behind simple interfaces. C++ supports *generic* programming, allowing you to write code reusable in a large number of contexts. Finally, C++ supports a limited form of *higher-order* programming, allowing you to write functions that construct and manipulate other functions at runtime.

C++ being a multiparadigm language is both a blessing and a curse. It is a blessing in that C++ will let you write code in the style that you feel is most appropriate for a given problem, rather than rigidly locking you into a particular framework. It is also a blessing in that you can mix and match styles to create programs that are precisely suited for the task at hand. It is a curse, however, in that multiparadigm languages are necessarily more complex than single-paradigm languages and consequently C++ is more difficult to pick up than other languages. Moreover, the interplay among all of these paradigms is complex, and you will need to learn the subtle but important interactions that occur at the interface between these paradigms.

This book is organized so that it covers a mixture of all of the aforementioned paradigms one after another, and ideally you will be comfortable working in each by the time you've finished reading.

C++ is a Mid-Level Language

Computer programs ultimately must execute on computers. Although computers are capable of executing programs which perform complex abstract reasoning, the computers themselves understand only the small set of commands necessary to manipulate bits and bytes and to perform simple arithmetic. Low-level languages are languages like C and assembly language that provide minimal structure over the actual machine and expose many details about the inner workings of the computer. To contrast, high-level languages are languages that abstract away from the particulars of the machine and let you write

programs independently of the computer's idiosyncrasies. As mentioned earlier, low-level languages make it hard to represent complex program structure, while high-level languages often are too abstract to operate efficiently on a computer.

C++ is a rare language in that it combines the low-level efficiency and machine access of C with high-level constructs like those found in Java. This means that it is possible to write C++ programs with the strengths of both approaches. It is not uncommon to find C++ programs that model complex systems using object-oriented techniques (high level) while taking advantage of specific hardware to accelerate that simulation (low-level). One way to think about the power afforded by C++ is to recognize that C++ is a language that provides a set of abstractions that let you intuitively design large software systems, but which lets you break those abstractions when the need to optimize becomes important. We will see some ways to accomplish this later in this book.

Design Philosophy

C++ is a comparatively old language; its first release was in 1985. Since then numerous other programming languages have sprung up – Java, Python, C#, and Javascript, to name a few. How exactly has C++ survived so long when others have failed? C++ may be useful and versatile, but so were BCPL and Simula, neither of which are in widespread use today.

One of the main reasons that C++ is still in use (and evolving) today has been its core guiding principles. Stroustrup has maintained an active interest in C++ since its inception and has steadfastly adhered to a particular design philosophy. Here is a sampling of the design points, as articulated in Stroustrup's *The Design and Evolution of C++*.

- **C++'s evolution must be driven by real problems**. When existing programming styles prove insufficient for modern challenges, C++ adapts. For example, the introduction of exception handling provided a much-needed system for error recovery, and abstract classes allowed programmers to define interfaces more naturally.

- **Don't try to force people**. C++ supports multiple programming styles. You can write code similar to that found in pure C, design class hierarchies as you would in Java, or develop software somewhere in between the two. C++ respects and trusts you as a programmer, allowing you to write the style of code you find most suitable to the task at hand rather than rigidly locking you into a single pattern.

- **Always provide a transition path**. C++ is designed such that the programming principles and techniques developed at any point in its history are still applicable. With few exceptions, C++ code written ten or twenty years ago should still compile and run on modern C++ compilers. Moreover, C++ is designed to be mostly backwards-compatible with C, meaning that veteran C coders can quickly get up to speed with C++.

The Goal of C++

There is one quote from Stroustrup ([Str94]) I believe best sums up C++:

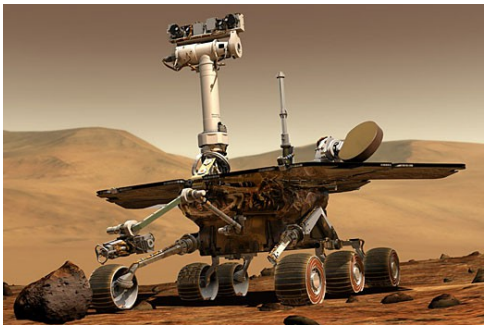
*C++ makes programming **more enjoyable** for **serious programmers**.*

What exactly does this mean? Let's begin with what constitutes a *serious programmer*. Rigidly defining "serious programmer" is difficult, so instead I'll list some of the programs and projects written in C++ and leave it as an exercise to the reader to infer a proper definition. For example, you'll find C++ in:

Mozilla Firefox. The core infrastructure underlying all Mozilla projects is written predominantly in C++. While much of the code for Firefox is written in Javascript and XUL, these languages are executed by interpreters written in C++.

The WebKit layout engine used by Safari and Google Chrome is also written in C++. Although it's closed-source, I suspect that Internet Explorer is also written in C++. If you're browsing the web, you're seeing C++ in action.

Java HotSpot. The widespread success of Java is in part due to *HotSpot*, Sun's implementation of the Java Virtual Machine. HotSpot supports just-in-time compilation and optimization and is a beautifully engineered piece of software. It's also written in C++. The next time that someone engages you in a debate about the relative merits of C++ and Java, you can mention that if not for a well-architected C++ program Java would not be a competitive language.



NASA / JPL. The rovers currently exploring the surface of Mars have their autonomous driving systems written in C++. *C++ is on Mars!*

C++ makes programming ***more enjoyable*** *for serious programmers*. Not only does C++ power all of the above applications, it powers them *in style*. You can program with high-level constructs yet enjoy the runtime efficiency of a low-level language like C. You can choose the programming style that's right for you and work in a language that trusts and respects your expertise. You can write code once that you will reuse time and time again. This is what C++ is all about, and the purpose of this book is to get you up to speed on the mechanics, style, and just plain excitement of C++.

With that said, let's dive into C++. Our journey begins!

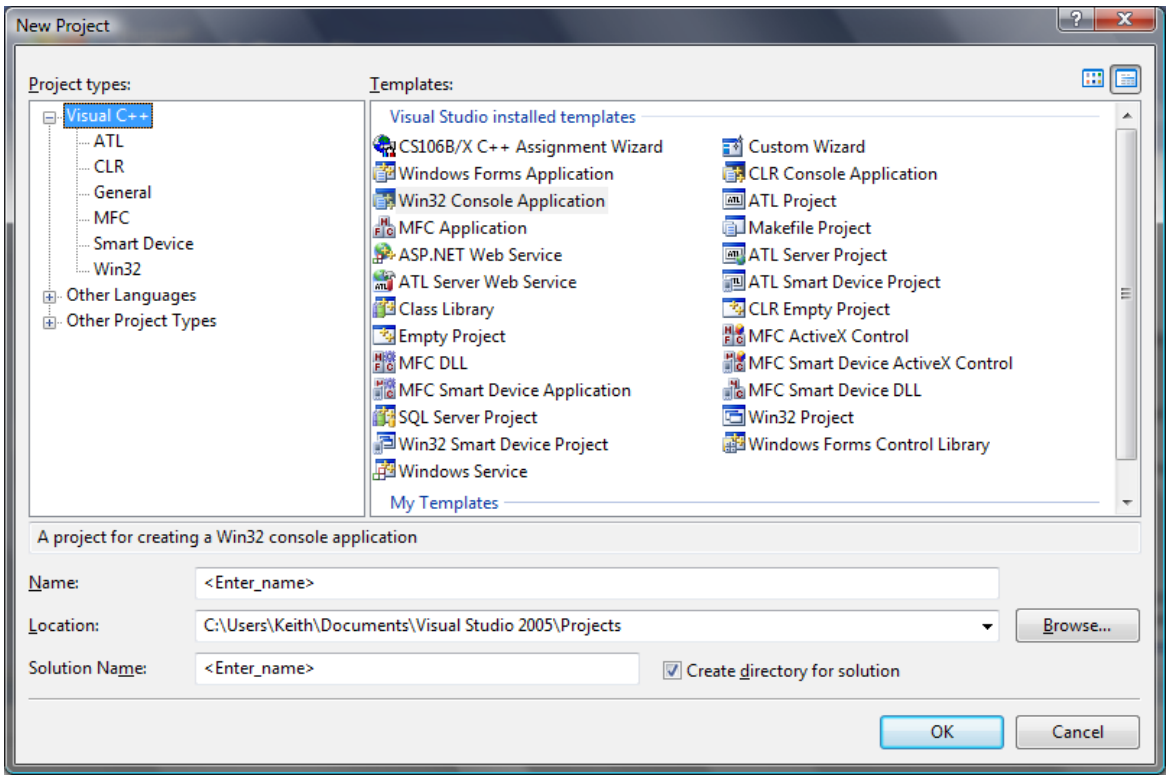
Chapter 1: Getting Started

Every journey begins with a single step, and in ours it's getting to the point where you can compile, link, run, and debug C++ programs. This depends on what operating system you have, so in this section we'll see how to get a C++ project up and running under Windows, Mac OS X, and Linux.

Compiling C++ Programs under Windows

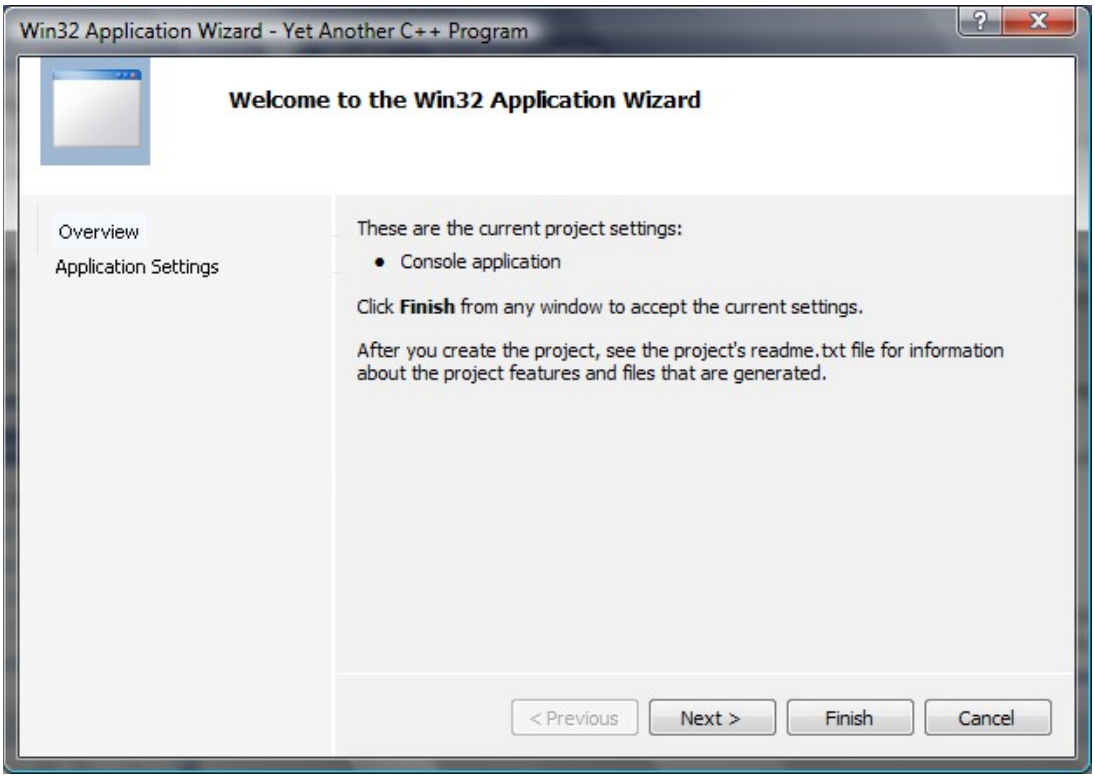
This section assumes that you are using Microsoft Visual Studio 2005 (VS2005). If you are a current CS106B/X student, you can follow the directions on the course website to obtain a copy. Otherwise, be prepared to shell out some cash to get your own copy, though it is definitely a worthwhile investment.* Alternatively, you can download Visual C++ 2008 Express Edition, a free version of Microsoft's development environment sporting a fully-functional C++ compiler. The express edition of Visual C++ lacks support for advanced Windows development, but is otherwise a perfectly fine C++ compiler. You can get Visual C++ 2008 Express Edition from http://www.microsoft.com/express/vc/. With only a few minor changes, the directions for using VS2005 should also apply to Visual C++ 2008 Express Edition, so this section will only cover VS2005.

VS2005 organizes C++ code into "projects," collections of source and header files that will be built into a program. The first step in creating a C++ program is to get an empty C++ project up and running, then to populate it with the necessary files. To begin, open VS2005 and from the **File** menu choose **New > Project...**. You should see a window that looks like this:



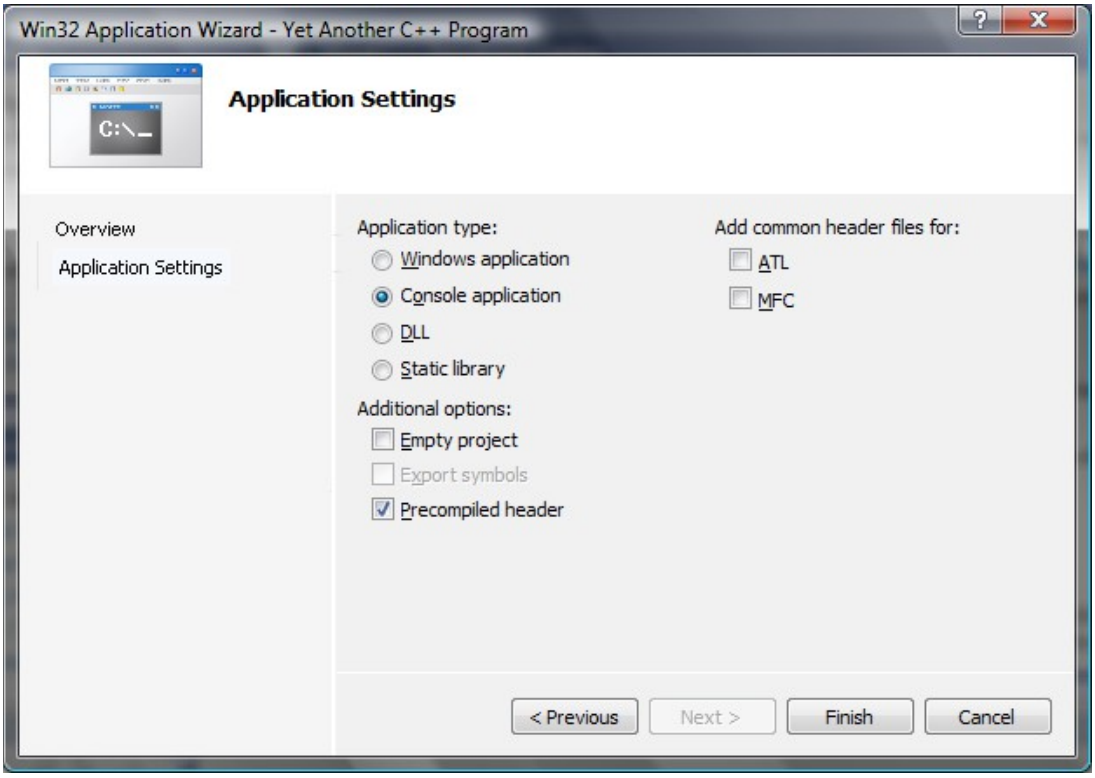
* I first began programming in C++ in 2001 using Microsoft Visual C++ 6.0, which cost roughly eighty dollars. I recently (2008) switched to Visual Studio 2005. This means that the compiler cost just over ten dollars a year. Considering the sheer number of hours I have spent programming, this was probably the best investment I have made.

As you can see, VS2005 has template support for all sorts of different projects, most of which are for Microsoft-specific applications such as dynamic-link libraries (DLLs) or ActiveX controls. We're not particularly interested in most of these choices – we just want a simple C++ program! To create one, find and choose **Win32 Console Application**. Give your project an appropriate name, then click **OK**. You should now see a window that looks like this, which will ask you to configure project settings:



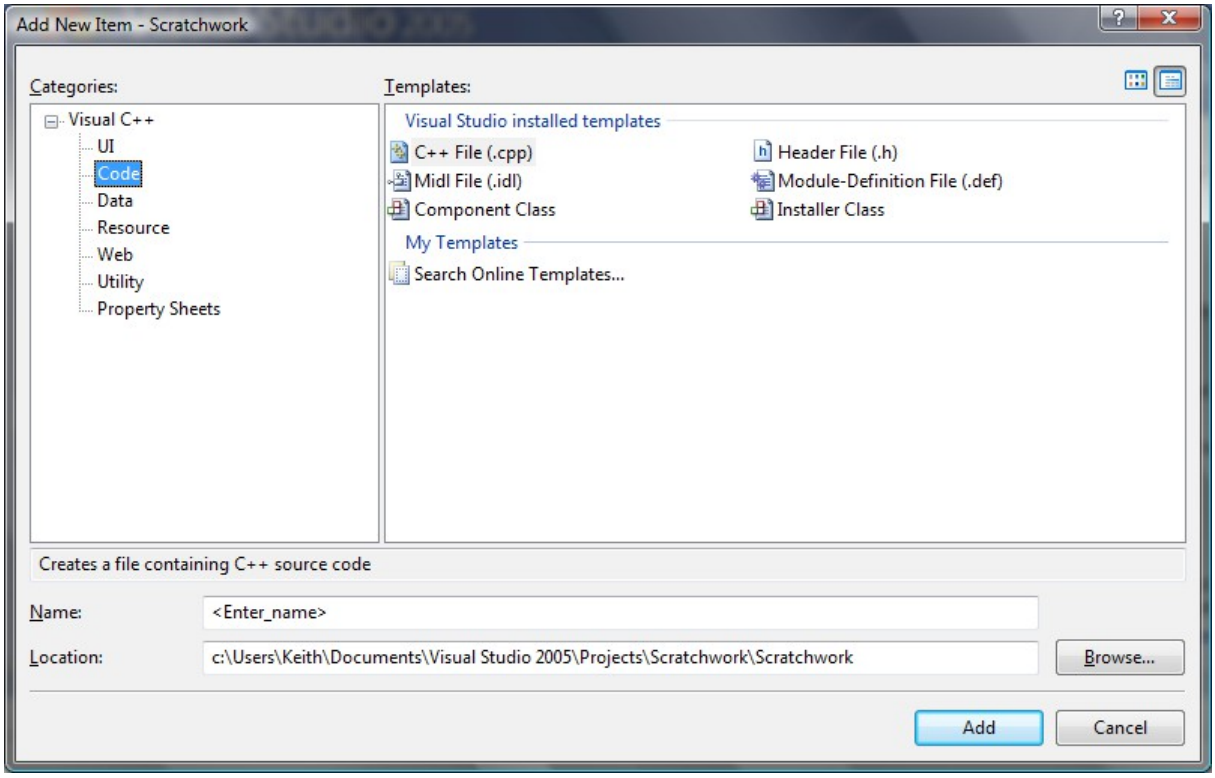
Note that the window title will have the name of the project you entered in the previous step in its title; “Yet Another C++ Program” is a placeholder.

At this point, you **do not** want to click **Finish**. Instead, hit **Next >** and you'll be presented with the following screen:



Keep all of the default settings listed here, but make sure that you check the box marked **Empty Project**. Otherwise VS2005 will give you a project with all sorts of Microsoft-specific features built into it. Once you've checked that box, click **Finish** and you'll have a fully functional (albeit empty) C++ project.

Now, it's time to create and add some source files to this project so that you can enter C++ code. To do this, go to **Project > Add New Item...** (or press CTRL+SHIFT+A). You'll be presented with the following dialog box:



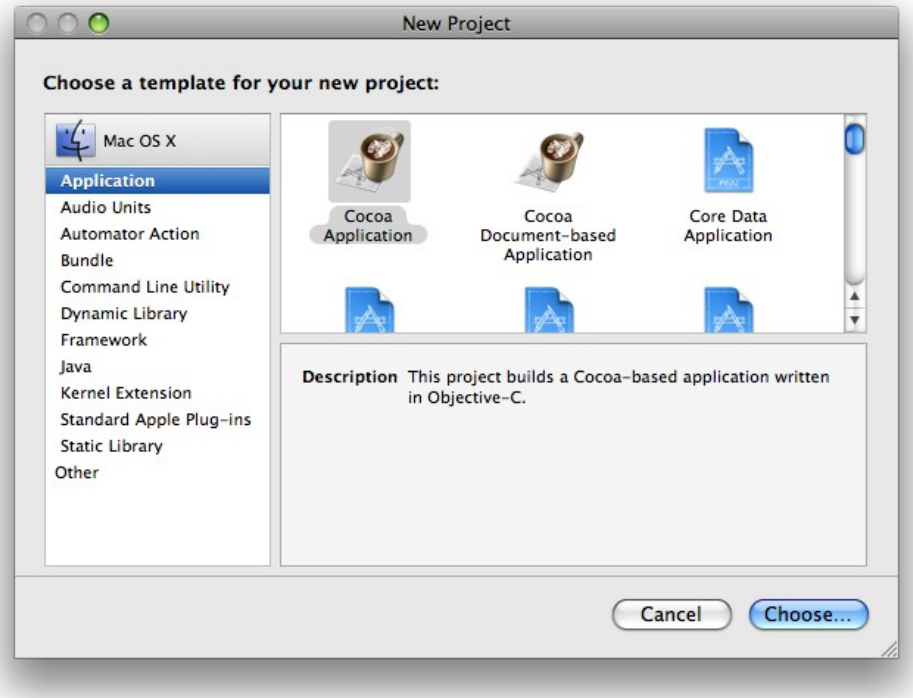
Choose **C++ File (.cpp)** and enter a name for it inside the Name field. VS2005 automatically appends .cpp to the end of the filename, so don't worry about manually entering the extension. Once you're ready, click **Add** and you should have your source file ready to go. Any C++ code you enter in here will be considered by the compiler and built into your final application.

Once you've written the source code, you can compile and run your programs by pressing **F5**, choosing **Debug> Start Debugging**, or clicking the green "play" icon. By default VS2005 will close the console window after your program finishes running, and if you want the window to persist after the program finishes executing you can run the program without debugging by pressing **CTRL+F5** or choosing **Debug > Start Without Debugging**. You should be all set to go!

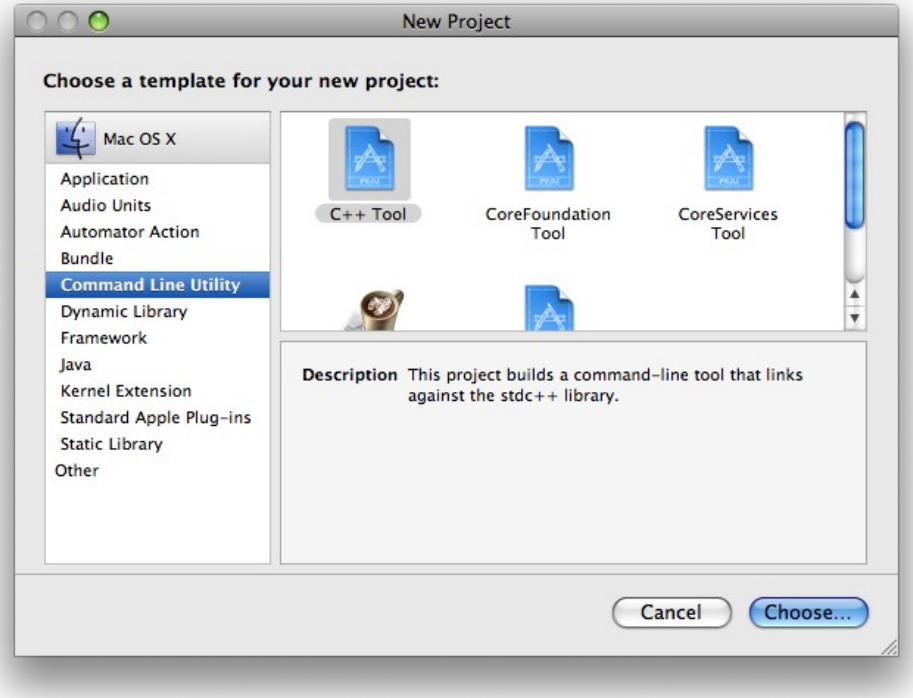
Compiling C++ Programs in Mac OS X

If you're developing C++ programs on Mac OS X, your best option is to use Apple's Xcode development environment. You can download Xcode free of charge from the Apple Developer Connection website at http://developer.apple.com/.

Once you've downloaded and installed Xcode, it's reasonably straightforward to create a new C++ project. Open Xcode. The first time that you run the program you'll get a nice welcome screen, which you're free to peruse but which you can safely dismiss. To create a C++ project, choose **File > New Project...**. You'll be presented with a screen that looks like this:

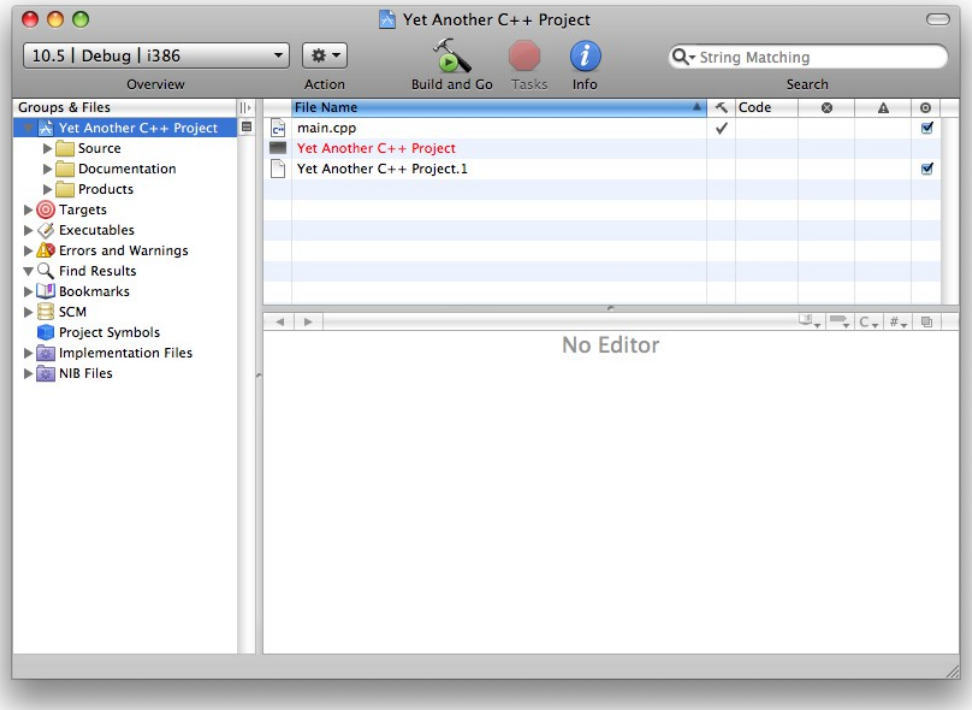


There are a lot of options here, most of which are Apple-specific or use languages other than C++ (such as Java or Objective-C). In the panel on the left side of the screen, choose **Command Line Utility** and you will see the following options:

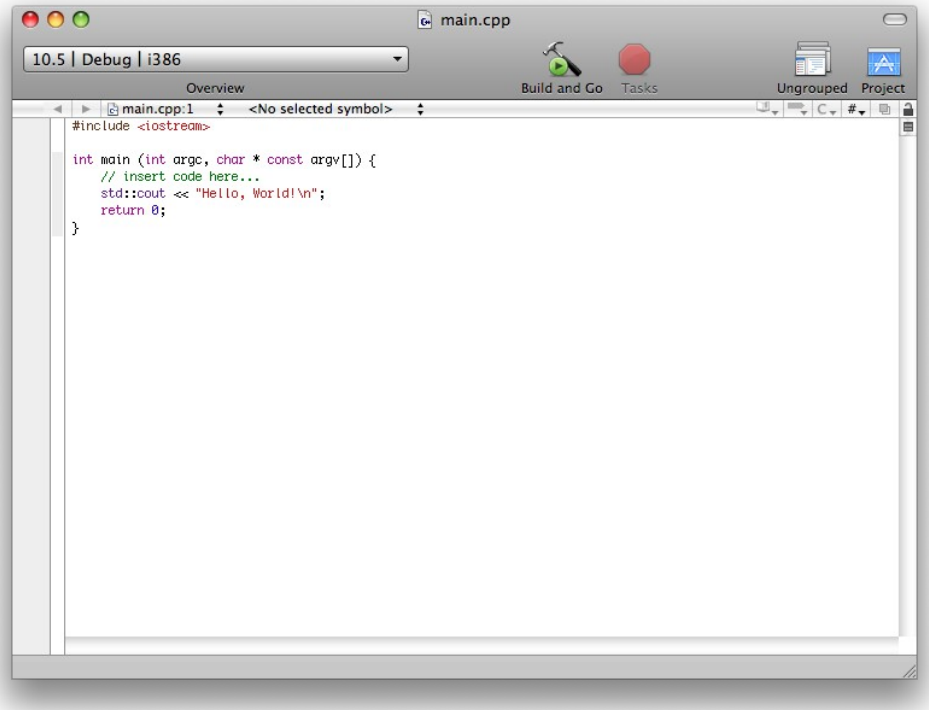


Select **C++ Tool** and click the **Choose...** button. You'll be prompted for a project name and directory; feel free to choose whatever name and location you'd like. In this example I've used the name “Yet Another C+

+ Project," though I suggest you pick a more descriptive name. Once you've made your selection, you'll see the project window, which looks like this:

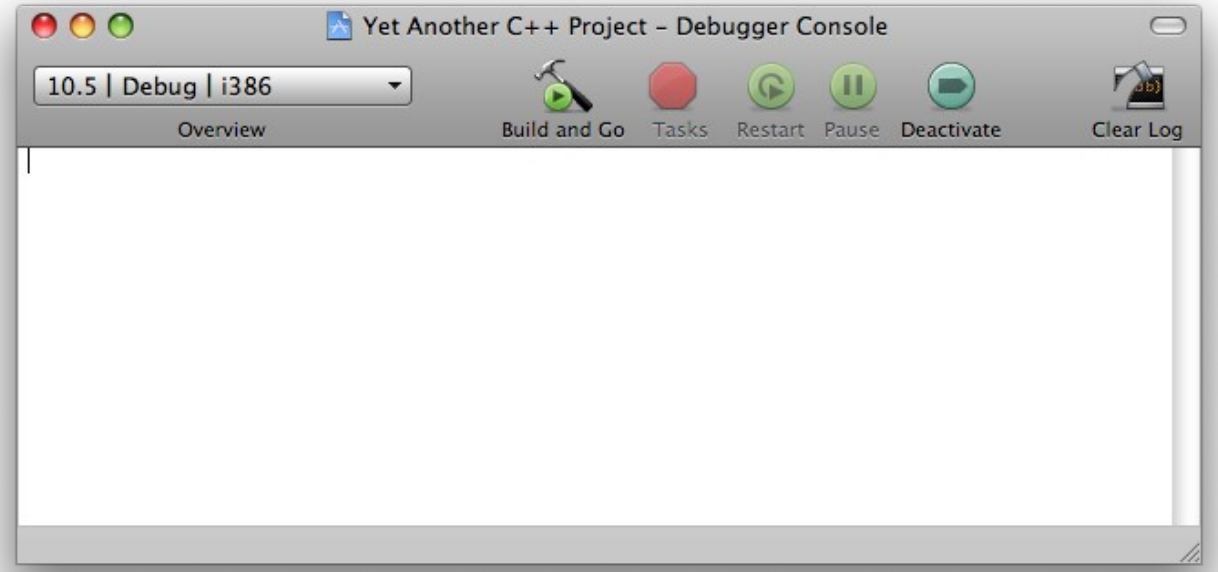


Notice that your project comes prepackaged with a file called main.cpp. This is a C++ source file that will be compiled and linked into the final program. By default, it contains a skeleton implementation of the Hello, World! program, as shown here:



Feel free to delete any of the code you see here and rewrite it as you see fit.

Because the program we've just created is a command-line utility, you will need to pull up the console window to see the output from your program. You can do this by choosing **Run > Console** or by pressing ⌘R. Initially the console will be empty, as shown here:



Once you've run your program, the output will be displayed here in the console. You can run the program by clicking the **Build and Go** button (the hammer next to a green circle containing an arrow). That's it! You now have a working C++ project.

If you're interested in compiling programs from the Mac OS X terminal, you might find the following section on Linux development useful.

Compiling C++ Programs under Linux

For those of you using a Linux-based operating system, you're in luck – Linux is extremely developer-friendly and all of the tools you'll need are at your disposal from the command-line.

Unlike the Windows or Mac environments, when compiling code in Linux you won't need to set up a development environment using Visual Studio or Xcode. Instead, you'll just set up a directory where you'll put and edit your C++ files, then will directly invoke the GNU C++ Compiler (`g++`) from the command-line.

If you're using Linux I'll assume that you're already familiar with simple commands like `mkdir` and `chdir` and that you know how to edit and save a text document. When writing C++ source code, you'll probably want to save header files with the .h extension and C++ files with the .cc, .cpp, .C, or .c++ extension. The .cc extension seems to be in vogue these days, though .cpp is also quite popular.

To compile your source code, you can execute `g++` from the command line by typing `g++` and then a list of the files you want to compile. For example, to compile `myfile.cc` and `myotherfile.cc`, you'd type

```
g++ myfile.cc myotherfile.cc
```

By default, this produces a file named `a.out`, which you can execute by entering `./a.out`. If you want to change the name of the program to something else, you can use `g++`'s `-o` switch, which produces an output file of a different name. For example, to create an executable called myprogram from the file myfile.cc, you could write

```
g++ myfile.cc -o myprogram
```

`g++` has a whole host of other switches (such as `-c` to compile but not link a file), so be sure to consult the man pages for more info.

It can get tedious writing out the commands to compile every single file in a project to form a finished executable, so most Linux developers use *makefiles*, scripts which allow you to compile an entire project by typing the make command. A full tour of makefiles is far beyond the scope of an introductory C++ text, but fortunately there are many good online tutorials on how to construct a makefile. The full manual for make is available online at http://www.gnu.org/software/make/manual/make.html.

Other Development Tools

If you are interested in using other development environments than the ones listed above, you're in luck. There are dozens of IDEs available that work on a wide range of platforms. Here's a small sampling:

- **NetBeans**: The NetBeans IDE supports C++ programming and is highly customizable. It also is completely cross-platform compatible, so you can use it on Windows, Mac OS X, and Linux.

- **MinGW**: MinGW is a port of common GNU tools to Microsoft Windows, so you can use tools like `g++` without running Linux. Many large software projects use MinGW as part of their build environment, so you might want to explore what it offers you.

- **Eclipse**: This popular Java IDE can be configured to run as a C++ compiler with a bit of additional effort. If you're using Windows you might need to install some additional software to get this IDE working, but otherwise it should be reasonably straightforward to configure.

- **Sun Studio**: If you're a Linux user and command-line hacking isn't your cup of tea, you might want to consider installing Sun Studio, Sun Microsystem's C++ development environment, which has a wonderful GUI and solid debugging support.

- **Qt Creator**: This Linux-based IDE is designed to build C++ programs using the open-source Qt libraries, but is also an excellent general-purpose C++ IDE. It is a major step above what the terminal and your favorite text editor have to offer, and I highly recommend that you check this program out if you're a Linux junkie.

Chapter 2: C++ Without `genlib.h`

When you arrived at your first CS106B/X lecture, you probably learned to write a simple "Hello, World" program like the one shown below:

```
#include "genlib.h"
#include <iostream>

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

Whether or not you have previous experience with C++, you probably realized that the first line means that the source code references an external file called `genlib.h`. For the purposes of CS106B/X, this is entirely acceptable (in fact, it's required!), but once you migrate from the educational setting to professional code you will run into trouble because `genlib.h` is *not* a standard header file; it's included in the CS106B/X libraries to simplify certain language features so you can focus on writing code, rather than appeasing the compiler.

In CS106L, none of our programs will use `genlib.h`, `simpio.h`, or any of the other CS106B/X library files. Don't worry, though, because none of the functions exported by these files are "magical." In fact, in the next few chapters you will learn how to rewrite or supersede the functions and classes exported by the CS106B/X libraries.* If you have the time, I encourage you to actually open up the genlib.h file and peek around at its contents.

To write "Hello, World" without `genlib.h`, you'll need to add another line to your program. The "pure" C++ version of "Hello, World" thus looks something like this:

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

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

We've replaced the header file `genlib.h` with the cryptic statement "`using namespace std;`" Before explaining exactly what this statement does, we need to take a quick diversion to lessons learned from development history. Suppose you're working at a company that produces two types of software: graphics design programs and online gunfighter duels (admittedly, this combination is pretty unlikely, but humor me for a while). Each project has its own source code files complete with a set of helper functions and classes. Here are some sample header files from each project, with most of the commenting removed:

---

* The exceptions are the graphics and sound libraries. C++ does not have natural language support for multimedia, and although many such libraries exist, we won't cover them in this text.

GraphicsUtility.h:

```
/* File: graphicsutility.h
 * Graphics utility functions.
 */

/* ClearScene: Clears the current scene. */
void ClearScene();

/* AddLine: Adds a line to the current scene. */
void AddLine(int x0, int y0, int x1, int y1);

/* Draw: Draws the current scene. */
void Draw();
```

GunfighterUtility.h:

```
/* File: gunfighterutility.h
 * Gunfighter utility functions.
 */

/* MarchTenPaces: Marches ten paces, animating each step. */
void MarchTenPaces(PlayerObject &toMove);

/* FaceFoe: Turns to face the opponent. */
void FaceFoe();

/* Draw: Unholsters and aims the pistol. */
void Draw();
```

Suppose the gunfighter team is implementing `MarchTenPaces` and needs to animate the gunfighters walking away from one another. Realizing that the graphics team has already implemented an entire library geared toward this, the gunfighter programmers import graphicsutility.h into their project, write code using the graphics functions, and try to compile. However, when they try to test their code, the linker reports errors to the effect of "`error: function 'void Draw()' already defined`."

The problem is that the graphics and gunfighter modules each contain functions named `Draw()` with the same signature and the compiler can't distinguish between them. It's impractical for either team to rename their `Draw` function, both because the other programming teams expect them to provide functions named `Draw` and because their code is already filled with calls to `Draw`. Fortunately, there's an elegant resolution to this problem. Enter the C++ `namespace` keyword. A *namespace* adds another layer of naming onto your functions and variables. For example, if all of the gunfighter code was in the namespace "`Gunfighter`," the function `Draw` would have the full name `Gunfighter::Draw`. Similarly, if the graphics programmers put their code inside namespace "`Graphics`," they would reference the function `Draw` as `Graphics::Draw`. If this is the case, there is no longer any ambiguity between the two functions, and the gunfighter development team can compile their code.

But there's still one problem – other programming teams expect to find functions named `ClearScene` and `FaceFoe`, not `Graphics::ClearScene` and `Gunfighter::FaceFoe`. Fortunately, C++ allows what's known as a *`using` declaration* that lets you ignore fully qualified names from a namespace and instead use the shorter names.

Back to the Hello, World example, reprinted here:

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

int main() {
    cout << "Hello, World!" << endl;
    return 0;
}
```

The statement "`using namespace std;`" following the `#include` directive tells the compiler that all of the functions and classes in the namespace std can be used without their fully-qualified names. This "`std`" namespace is the *C++ standard namespace* that includes all the library functions and classes of the standard library. For example, cout is truly named std::cout, and without the using declaration importing the std namespace, Hello, World would look something like this:

```
#include <iostream>

int main() {
  std::cout << "Hello, World!" << std::endl;
  return 0;
}
```

While some programmers prefer to use the fully-qualified names when using standard library components, repeatedly writing `std::` can be a hassle. To eliminate this problem, in `genlib.h`, we included the `using` declaration for you. But now that we've taken the training wheels off and `genlib.h` is no more, you'll have to remember to include it yourself!

There's one more important part of `genlib.h`, the `string` type. Unlike other programming languages, C++ lacks a primitive string type.* Sure, there's the class `string`, but unlike `int` or `double` it's not a built-in type and must be included with a `#include` directive. Specifically, you'll need to write `#include <string>` at the top of any program that wants to use C++-style strings. And don't forget the `using` declaration, or you'll need to write `std::string` every time you want to use C++ strings!

* Technically speaking there are primitive strings in C++, but they aren't objects. See the chapter on C strings for more information.

Part One

A Better C

Chapter 3: Streams

It's time to begin our serious foray into the magical world of C++ programming. In this first chapter, we'll explore C++'s *streams library*, a collection of functions that allow you to read and write formatted data from a variety of sources. The streams library allows your program to print text to the user and read back responses. It also lets you load persistent data from external files and to save custom information on-disk. As you continue your exploration of C++, you will use the contents of this chapter time and time again, whether for simple error-reporting or more complex data management.

Streams: An Overview

In the physical world, all interesting devices have some way of interacting with their environment. Take a common alcohol thermometer, for example. The thermometer has a liquid-filled bulb that is warmed up by the environment and a graduated meter which allows the user to read off the temperature near the bulb. Or consider a car, which has an accelerator, brake, gearbox, and steering wheel to control the direction and speed of the vehicle and a dashboard which reports the current state of the automobile. C++'s streams library is the primary means by which a C++ program can interact with its environment, namely the user and the file system.

The basic unit of communication between a program and its environment is a *stream*. A stream is a channel between a *source* and a *destination* which allows the source to push formatted data to the destination. The type of the source and the sink varies from stream to stream. In some streams the source is the program itself and the destination is a file on disk, and the stream can be used to write persistent data to the user's hard drive. In others, the source is the keyboard and the destination is the program, and the stream can be used to read user input from the physical world into the computer.

The use of the term "stream" in the context of the streams library is similar to the use of "stream" in the context of "streaming video." When a data provider (for example, YouTube) streams video over the Internet, the video is not sent all at once. Instead, the program receiving the video continuously queries the server for more and more information, and the video is sent in fixed-size chunks and reassembled by the video player. When using the streams library to read or write data, you do not need to read or write all of the data at once. It's perfectly legal (and quite common) to read the data one piece at a time. For example, if you want to read data from a file, instead of loading all of the file contents at once, you can read the file line-by-line, or character-by-character, or using some hybrid approach. This gives you great flexibility, since you can read different pieces of the file in different ways to get the data in a format appropriate to your application.

To give you a better sense of how streams work in practice, let's consider an actual stream, `cout`. `cout` (for **c**haracter **out**put) is a stream connected to the *console*, a text window that displays plain text data. Any information pushed across `cout` displays in the console, and so you can think of `cout` as a way of displaying data to the user. For example, here's a simple program which displays a message to the user and then quits:

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

int main() {
    cout << "I'm sorry Dave, I'm afraid I can't do that." << endl;
    return 0;
}
```

There's a lot of code here, so let's take a few minutes to dissect it. The first line of the program, `#include <iostream>`, instructs the C++ compiler to import the `cout` stream into the program. The line `using namespace std` is covered in the previous chapter and makes the `cout` stream available. Inside of `main`, we have the following line of code:

```
cout << "I'm sorry Dave, I'm afraid I can't do that." << endl;
```

The special `<<` operator is called the *stream insertion operator* and is a C++ operator that is used to push data into a stream object. Here, we push the text string `I'm sorry Dave, I'm afraid I can't do that` into the `cout` stream. This causes the this text to display on-screen. Afterwards, we push the special object `endl` into the stream. `endl` stands for "**end l**ine" and prints a newline character to the `cout` stream. This means that the next time we push text into `cout`, the text will display on the next line, rather than directly after the text string we just printed. We'll discuss `endl` in more detail later in this chapter.

Streams are very versatile and you can write data of multiple types to stream objects. In fact, you can push data of any primitive type into a stream. For example, here's a program showing off the sorts of data that can move across a stream:

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

int main() {
    cout << "Streams can take in text." << endl;
    cout << 137 << endl;     // Streams can take in integers.
    cout << 2.71828 << endl; // Streams can take in real numbers.
    cout << "Here is text followed by a number: " << 31415 << endl;
    return 0;
}
```

Running this program will produce the following output:

```
Streams can take in text.
137
2.71828
Here is text followed by a number: 31415
```

In the first line of this program, we sent a text string to the console. In the second and third, we sent an integer and a natural number, respectively. The last line is perhaps the most interesting. In it, we push both a string *and* an integer to the console by chaining together the stream insertion operator. The designers of the streams library were fairly clever, and so it's perfectly legal to chain together as many stream insertions as you'd like.

To give you a better feel for why each of the stream operations in the above program end by pushing `endl` into the stream, let's consider what would happen if this weren't the case. Here's a revised version of the above program will all instances of `endl` removed:

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

int main() {
    cout << "Streams can take in text.";
    cout << 137;
    cout << 2.71828;
    cout << "Here is text followed by a number: " << 31415;
    return 0;
}
```

This produces the following output:

```
Streams can take in text.1372.71828Here is text followed by a number: 31415
```

Notice that all of this text runs together. C++ will not "automatically" insert newlines into any text you write, and when outputting data to the console you will need to manually insert line breaks. As a general rule, most of the time that you use `cout` to push data to the console, you will need to append `endl` to ensure the output doesn't all run together.

All of the stream examples we have seen so far have revolved around `cout` and pushing data from the program to the console. To build a truly interactive program, however, we'll need to get input from the user. In CS106B/X, we provide the `simpio.h` header file, which exports the input functions `GetLine`, `GetInteger`, `GetReal`, and `GetLong`. Though useful, these functions are not part of the C++ standard library and will not be available outside of CS106B/X. Don't worry, though, because by the end of this chapter we'll see how to implement them using only standard C++.

The streams library exports another stream object called `cin` (**c**haracter **in**put) which lets you read values directly from the user. To read a value from `cin`, you use the *stream extraction operator* `>>`. Syntactically, the stream extraction operator mirrors the stream insertion operator. For example, here's a code snippet to prompt the user for an integer.

```
cout << "Please enter an integer: ";

int myInteger;
cin >> myInteger; // Value stored in myInteger
```

When the program encounters the highlighted line, it will pause and wait for the user to type in a number and hit enter. Provided that the user actually enters an integer, its value will be stored inside the `myInteger` variable. What happens if the user *doesn't* enter an integer is a bit more complicated, and we'll return to this later in the chapter.

You can also read multiple values from `cin` by chaining together the stream extraction operator in the same way that you can write multiple values to `cout` by chaining the stream insertion operator:

```
int myInteger;
string myString;
cin >> myInteger >> myString; // Read an integer and string from cin
```

This will pause until the user enters an integer, hits enter, then enters a string, then hits enter once more. These values will be stored in `myInteger` and `myString`, respectively.

Note that when using `cin`, you should not read into `endl` the way that you write `endl` when using `cout`. Hence the following code is illegal:

```
int myInteger;
cin >> myInteger >> endl; // Error: Cannot read into endl.
```

Intuitively, this makes sense because `endl` means "print a newline." Reading a value into `endl` is therefore a nonsensical operation.

In practice, it is not a good idea to read values directly from `cin`. Unlike `GetInteger` and the like, `cin` does not perform any safety checking of user input and if the user does not enter valid data, `cin` will begin behaving unusually. Later in this chapter, we will see how the `GetInteger` function is implemented and you will be able to use the function in your own programs. In the meantime, though, feel free to use `cin`, but make sure that you always type in input correctly!

Reading and Writing Files

So far, we have seen two examples of streams – `cout`, which sends data to the console, and `cin`, which reads data from the keyboard. In this next section we'll see two new kinds of streams – `ifstream`s and `ofstream`s – which can be used to read or write files on disk. This will allow your program to save data indefinitely, or to read in configuration data from an external source.

C++ provides a header file called `<fstream>` (**f**ile **stream**) that exports the `ifstream` and `ofstream` types, streams that perform file I/O. The naming convention is unfortunate – `ifstream` stands for **i**nput **f**ile **stream** (not "something that might be a stream") and `ofstream` for **o**utput **f**ile **stream**. There is also a generic `fstream` class which can do both input and output, but we will not cover it in this chapter. Unlike `cin` and `cout`, which are concrete stream objects, `ifstream` and `ofstream` are *types*. To read or write from a file, you will create an object of type `ifstream` or `ofstream`, much in the same way that you would create an object of type `string` to store text data or a variable of type `double` to hold a real number. Once you have created the file stream object, you can read or write to it using the stream insertion and extraction operators just as you would `cin` or `cout`.

To create an `ifstream` that reads from a file, you can use this syntax:

```
ifstream myStream("myFile.txt");
```

This creates a new stream object named `myStream` which reads from the file `myFile.txt`, provided of course that the file exists. We can then read data from `myStream` just as we would from `cin`, as shown here:

```
ifstream myStream("myFile.txt");
int myInteger;
myStream >> myInteger; // Read an integer from myFile.txt
```

Notice that we wrote `myStream >> myInteger` rather than `ifstream >> myInteger`. When reading data from a file stream, you must read from the stream variable rather than the `ifstream` type. If you read from `ifstream` instead of your stream variable, the program will not compile and will give you a fairly cryptic error message.

You can also open a file by using the `ifstream`'s `open` member function, as shown here:

```
ifstream myStream;            // Note: did not specify the file
myStream.open("myFile.txt"); // Now reading from myFile.txt
```

When opening a file using an `ifstream`, there is a chance that the specified file can't be opened. The filename might not specify an actual file, you might not have permission to read the file, or perhaps the file

is locked. If you try reading data from an `ifstream` that is not associated with an open file, the read will fail and you will not get back meaningful data. After trying to open a file, you should check if the stream is valid by using the `.is_open()` member function. For example, here's code to open a file and report an error to the user if a problem occurred:

```
ifstream input("myfile.txt");
if(!input.is_open())
    cerr << "Couldn't open the file myfile.txt" << endl;
```

Notice that we report the error to the `cerr` stream. `cerr`, like `cout`, is an output stream, but unlike `cout`, `cerr` is designed for error reporting and is sometimes handled differently by the operating system.

The output counterpart to `ifstream` is `ofstream`. As with `ifstream`, you specify which file to write to either by using the `.open()` member function or by specifying the file when you create the `ofstream`, as shown below:

```
ofstream myStream("myFile.txt"); // Write to myFile.txt
```

A word of warning: if you try writing to a nonexistent file with an `ofstream`, the `ofstream` will create the file for you. However, if you open a file that already exists, the `ofstream` will overwrite all of the contents of the file. Be careful not to write to important files without first backing them up!

The streams library is one of the older libraries in C++ and the `open` functions on the `ifstream` and `ofstream` classes predate the `string` type. If you have the name of a file stored in a C++ `string`, you will need to convert the `string` into a C-style string (covered in the second half of this book) before passing it as a parameter to `open`. This can be done using the `.c_str()` member function of the `string` class, as shown here:

```
ifstream input(myString.c_str()); // Open the filename stored in myString
```

When a file stream object goes out of scope, C++ will automatically close the file for you so that other processes can read and write the file. If you want to close the file prematurely, you can use the `.close()` member function. After calling `close`, reading or writing to or from the file stream will fail.

As mentioned above in the section on `cin`, when reading from or writing to files you will need to do extensive error checking to ensure that the operations succeed. Again, we'll see how to do this later.

Stream Manipulators

Consider the following code that prints data to `cout`:

```
cout << "This is a string!" << endl;
```

What exactly is `endl`? It's an example of a *stream manipulator*, an object that can be inserted into a stream to change some sort of stream property. `endl` is one of the most common stream manipulators, though others exist as well. To motivate some of the more complex manipulators, let's suppose that we have a file called `table-data.txt` containing four lines of text, where each line consists of an integer value and a real number. For example:

File: `table-data.txt`

```
137     2.71828
42      3.14159
7897987 1.608
1337    .01101010001
```

We want to write a program which reads in this data and prints it out in a table, as shown here:

```
--------------------+----------------------+---------------------
                  1 |                  137 |              2.71828
                  2 |                   42 |              3.14159
                  3 |              7897987 |                1.608
                  4 |                 1337 |              0.01101
```

Here, the first column is the one-indexed line number, the second the integer values from the file, and the third the real-numbered values from the file.

Let's begin by defining a few constants to control what the output should look like. Since there are four lines in the file, we can write

```
const int NUM_LINES = 4;
```

And since there are three columns,

```
const int NUM_COLUMNS = 3;
```

Next, we'll pick an arbitrary width for each column. We'll choose twenty characters, though in principle we could pick any value as long as the data fit:

```
const int COLUMN_WIDTH = 20;
```

Now, we need to read in the table data and print out the formatted table. We'll decompose this problem into two smaller steps, resulting in the following source code:

```
#include <iostream>
#include <fstream>
using namespace std;

const int NUM_LINES = 4;
const int NUM_COLUMNS = 3;
const int COLUMN_WIDTH = 20;

int main() {
    PrintTableHeader();
    PrintTableBody();
    return 0;
}
```

`PrintTableHeader` is responsible for printing out the top part of the table (the row of dashes and pluses) and `PrintTableBody` will load the contents of the file and print them to the console.

Despite the fact that `PrintTableHeader` precedes `PrintTableBody` in this program, we'll begin by implementing `PrintTableBody` as it illustrates exactly how much firepower we can get from the stream manipulators. We know that we need to open the file `table-data.txt` and that we'll need to read four lines of data from it, so we can begin writing this function as follows:

```
void PrintTableBody() {
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
     * program.
     */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        /* ... process data ... */
    }
}
```

You may have noticed that at the end of this `for` loop I've written `++k` instead of `k++`. There's a slight difference between the two syntaxes, but in this context they are interchangeable. When we talk about operator overloading in a later chapter we'll talk about why it's generally considered better practice to use the prefix increment operator instead of the postfix.

Now, we need to read data from the file and print it as a table. We can start by actually reading the values from the file, as shown here:

```
void PrintTableBody() {
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
     * program.
     */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;
    }
}
```

Next, we need to print out the table row. This is where things get tricky. If you'll recall, the table is supposed to be printed as three columns, each a fixed width, that contain the relevant data. How can we ensure that when we print the values to `cout` that we put in the appropriate amount of whitespace? Manually writing space characters would be difficult, so instead we'll use a stream manipulator called `setw` (**set w**idth) to force `cout` to pad its output with the right number of spaces. `setw` is defined in the `<iomanip>` header file and can be used as follows:

```
cout << setw(10) << 137 << endl;
```

This tells `cout` that the next item it prints out should be padded with spaces so that it takes up at least ten characters. Similarly,

```
cout << setw(20) << "Hello there!" << endl;
```

Would print out `Hello there!` with sufficient leading whitespace.

By default `setw` pads the next operation with spaces on the left side. You can customize this behavior with the `left` and `right` stream manipulators, as shown here:

```
cout << '[' << left  << setw(10) << "Hello!" << ']' << endl;  // [     Hello!]
cout << '[' << right << setw(10) << "Hello!" << ']' << endl;  // [Hello!     ]
```

Back to our example. We want to ensure that every table column is exactly `COLUMN_WIDTH` spaces across. Using `setw`, this is relatively straightforward and can be done as follows:

```
void PrintTableBody() {
    ifstream input("table-data.txt");
    /* No error-checking here, but you should be sure to do this in any real
     * program.
     */

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        cout << setw(COLUMN_WIDTH) << (k + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;
    }
}
```

This produces the following output when run on the input file described above:

```
                   1 |                 137 |             2.71828
                   2 |                  42 |             3.14159
                   3 |             7897987 |               1.608
                   4 |                1337 |             0.01101
```

The body of the table looks great, and now we just need to print the table header, which looks like this:

```
--------------------+----------------------+--------------------
```

If you'll notice, this is formed by printing twenty dashes, then the pattern `-+-`, another twenty dashes, the pattern `-+-`, and finally another twenty dashes. We could thus implement `PrintTableHeader` like this:

```
void PrintTableHeader() {
    /* Print the ---...---+- pattern for all but the last column. */
    for(int column = 0; column < NUM_COLUMNS - 1; ++column) {
        for(int k = 0; k < COLUMN_WIDTH; ++k)
            cout << '-';
        cout << "-+-";
    }

    /* Now print the ---...--- pattern for the last column. */
    for(int k = 0; k < COLUMN_WIDTH; ++k)
        cout << '-';

    /* Print a newline... there's nothing else on this line. */
    cout << endl;
}
```

As written there's nothing wrong with this code and the program will work just fine, but we can simplify the implementation by harnessing stream manipulators. Notice that at two points we need to print out `COLUMN_WIDTH` copies of the dash character. When printing out the table body, we were able to use the `setw` stream manipulator to print multiple copies of the space character; is there some way that we can use it here to print out multiple dashes? The answer is yes, thanks to `setfill`. The `setfill` manipulator

accepts a parameter indicating what character to use as a fill character for `setw`, then changes the stream such that all future calls to `setw` pad the stream with the specified character. For example:

```
cout << setfill('0') << setw(8) << 1000 << endl; // Prints 00001000
cout << setw(8) << 1000 << endl; // Prints 00001000 because of last setfill
```

Note that `setfill` does not replace all space characters with instances of some other character. It is only meaningful in conjunction with `setw`. For example:

```
cout << setfill('X') << "Some   Spaces" << endl; // Prints Some   Spaces
```

Using `setfill` and `setw`, we can print out `COLUMN_WIDTH` copies of the dash character as follows:

```
cout << setfill('-') << setw(COLUMN_WIDTH) << "" << setfill(' ');
```

This code is dense, so let's walk through it one step at a time. The first part, `setfill('-')`, tells `cout` to pad all output with dashes instead of spaces. Next, we use `setw` to tell `cout` that the next operation should take up at least `COLUMN_WIDTH` characters. The trick is the next step, printing the empty string. Since the empty string has length zero and the next operation will always print out at least `COLUMN_WIDTH` characters padded with dashes, this code prints out `COLUMN_WIDTH` dashes in a row. Finally, since `setfill` permanently sets the fill character, we use `setfill(' ')` to undo the changes we made to `cout`.

Using this code, we can rewrite `PrintTableHeader` as follows:

```
void PrintTableHeader() {
    /* Print the ---...---+- pattern for all but the last column. */
    for(int column = 0; column < NUM_COLUMNS - 1; ++column)
        cout << setfill('-') << setw(COLUMN_WIDTH) << "" << "-+-";

    /* Now print the ---...--- pattern for the last column and a newline. */
    cout << setw(COLUMN_WIDTH) << "" << setfill(' ') << endl;
}
```

Notice that we only call `setfill(' ')` once, at the end of this function, since there's no reason to clear it at each step. Also notice that we've reduced the length of this function dramatically by having the library take care of the heavy lifting for us. The code to print out a table header is now three lines long!

There are many stream manipulators available in C++. The following table lists some of the more commonly-used ones:

`boolalpha`	`cout << true << endl;` `// Output: 1`
	`cout << boolalpha << true << endl;` `// Output: true`
	
	Determines whether or not the stream should output boolean values as 1 and 0 or as "true" and "false." The opposite manipulator is `noboolalpha`, which reverses this behavior.

`setw(n)`	`cout << 10 << endl;` `// Output: 10`
	`cout << setw(5) << 10 << endl;` `// Output:  10`
	
	Sets the minimum width of the output for the next stream operation. If the data doesn't meet the minimum field requirement, it is padded with the default fill character until it is the proper size.

Common stream manipulators, contd.

`hex, dec, oct`	`cout << 10 << endl;  // Output: 10`
	`cout << dec << 10 << endl; // Output: 10`
	`cout << oct << 10 << endl; // Output: 12`
	`cout << hex << 10 << endl; // Output: a`
	`cin >> hex >> x;  // Reads a hexadecimal value.`
	
	Sets the radix on the stream to either octal (base 8), decimal (base 10), or hexadecimal (base 16). This can be used either to format output or change the base for input.

`ws`	`myStream >> ws >> value;` Skips any whitespace stored in the stream. By default the stream extraction operator skips over whitespace, but other functions like `getline` do not. `ws` can sometimes be useful in conjunction with these other functions.

When Streams Go Bad

Because stream operations often involve transforming data from one form into another, stream operations are not always guaranteed to succeed. For example, consider the following code snippet, which reads integer values from a file:

```
ifstream in("input.txt"); // Read from input.txt
for(int i = 0; i < NUM_INTS; ++i) {
    int value;
    in >> value;
    /* ... process value here ... */
}
```

If the file `input.txt` contains `NUM_INTS` consecutive integer values, then this code will work correctly. However, what happens if the file contains some other type of data, such as a string or a real number?

If you try to read stream data of one type into a variable of another type, rather than crashing the program or filling the variable with garbage data, the stream fails by entering an *error state* and the value of the variable will not change. Once the stream is in this error state, any subsequent read or write operations will automatically and silently fail, which can be a serious problem.

You can check if a stream is in an error state with the `.fail()` member function. Don't let the name mislead you – `fail` checks if a stream is in an error state, rather than putting the stream into that state. For example, here's code to read input from `cin` and check if an error occurred:

```
int myInteger;
cin >> myInteger;
if(cin.fail()) { /* ... error ... */ }
```

If a stream is in a fail state, you'll probably want to perform some special handling, possibly by reporting the error. Once you've fixed any problems, you need to tell the stream that everything is okay by using the `.clear()` member function to bring the stream out of its error state. Note that clear won't skip over the input that put the stream into an error state; you will need to extract this input manually.

Streams can also go into error states if a read operation fails because no data is available. This occurs most commonly when reading data from a file. Let's return to the table-printing example. In the `PrintTableData` function, we hardcoded the assumption that the file contains exactly four lines of data.

But what if we want to print out tables of arbitrary length? In that case, we'd need to continuously read through the file extracting and printing numbers until we exhaust its contents. We can tell when we've run out of data by checking the `.fail()` member function after performing a read. If `.fail()` returns true, something prevented us from extracting data (either because the file was malformed or because there was no more data) and we can stop looping.

Recall that the original code for reading data looks like this:

```
void PrintTableBody() {
    ifstream input("table-data.txt");

    /* Loop over the lines in the file reading data. */
    for(int k = 0; k < NUM_LINES; ++k) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        cout << setw(COLUMN_WIDTH) << (k + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;
    }
}
```

The updated version of this code, which reads all of the contents of the file, is shown here:

```
void PrintTableBody() {
    ifstream input("table-data.txt");

    /* Loop over the lines in the file reading data. */
    int rowNumber = 0;
    while(true) {
        int intValue;
        double doubleValue;
        input >> intValue >> doubleValue;

        if(input.fail()) break;

        cout << setw(COLUMN_WIDTH) << (rowNumber + 1) << " | ";
        cout << setw(COLUMN_WIDTH) << intValue << " | ";
        cout << setw(COLUMN_WIDTH) << doubleValue << endl;

        rowNumber++;
    }
}
```

Notice that we put the main logic into a `while(true)` loop that `break`s when `input.fail()` returns `true` instead of a `while(!input.fail())` loop. These two structures may at first appear similar, but are quite different from one another. In a `while(!input.fail())` loop, we only check to see if the stream encountered an error after reading and processing the data in the body of the loop. This means that the loop will execute once more than it should, because we don't notice that the stream malfunctioned until the top of the loop. On the other hand, in the above loop structure (`while(true)` plus `break`), we stop looping as soon as the stream realizes that something has gone awry. Confusing these two loop structures is a common error, so be sure that you understand why to use the "loop-and-a-half" idiom rather than a simple `while` loop.

A Useful Shorthand

In the above code, we used the loop-and-a-half idiom to determine whether we should continue reading and printing data out of the file or whether we should stop looping. The general pattern for this idiom is as follows:

```
while(true) {
    int intValue;
    double doubleValue;
    input >> intValue >> doubleValue;

    if(input.fail()) break;

    /* ... process values here ... */
}
```

This code is perfectly valid, but it's a bit clunky. The outermost loop is a `while(true)` loop, which means "loop forever," but in reality the idea we want to represent is "loop until there is no more available data." The designers of the streams library anticipated this use case and provided a remarkably simple shorthand to alleviate this complexity. The above code is entirely equivalent to

```
int intValue;
double doubleValue;

while(input >> intValue >> doubleValue) {
    /* ... process values here ... */
}
```

Notice that the condition of the `while` loop is now `input >> intValue >> doubleValue`. Recall that in C++, any nonzero value is interpreted as "`true`" and any zero value is interpreted as "`false`." The streams library is configured so that most stream operations, including stream insertion and extraction, yield a nonzero value if the operation succeeds and zero otherwise. This means that code such as the above, which uses the read operation as the looping condition, is perfectly valid. One particular advantage of this approach is that while the syntax is considerably more dense, the code is more intuitive. You can read this while loop as "while I can successfully read data into `intValue` and `doubleValue`, continue executing the loop." Compared to our original implementation, this is much cleaner.

This syntax shorthand is actually a special case of a more general technique. In any circumstance where a boolean value is expected, it is legal to place a stream object or a stream read/write operation. We will see this later in this chapter when we explore the getline function.

When Streams Do Too Much

Consider the following code snippet, which prompts a user for an age and hourly salary:

```
int age;
double hourlyWage;

cout << "Please enter your age: ";
cin >> age;

cout << "Please enter your hourly wage: ";
cin >> hourlyWage;
```

As mentioned above, if the user enters a string or otherwise non-integer value when prompted for their age, the stream will enter an error state. There is another edge case to consider. Suppose the input is `2.71828`. You would expect that, since this isn't an integer (it's a real number), the stream would go into an error state. However, this isn't what happens. The first call, `cin >> age`, will set `age` to `2`. The next call, `cin >> hourlyWage`, rather than prompting the user for a value, will find the `.71828` from the earlier input and fill in `hourlyWage` with that information. Despite the fact that the input was malformed for the first prompt, the stream was able to partially interpret it and no error was signaled.

As if this wasn't bad enough, suppose we have this program instead, which prompts a user for an administrator password and then asks whether the user wants to format her hard drive:

```
string password;
cout << "Enter administrator password: ";

cin >> password;
if(password == "password") { // Use a better password, by the way!
    cout << "Do you want to erase your hard drive (Y or N)? ";

    char yesOrNo;
    cin >> yesOrNo;

    if(yesOrNo == 'y')
        EraseHardDrive();
}
```

What happens if someone enters `password y`? The first call, `cin >> password`, will read only `password`. Once we reach the second `cin` read, it automatically fills in `yesOrNo` with the leftover `y`, and there goes our hard drive! Clearly this is not what we intended.

As you can see, reading directly from `cin` is unsafe and poses more problems than it solves. In CS106B/X we provide you with the `simpio.h` library primarily so you don't have to deal with these sorts of errors. In the next section, we'll explore an entirely different way of reading input that avoids the above problems.

An Alternative: `getline`

Up to this point, we have been reading data using the stream extraction operator, which, as you've seen, can be dangerous. However, there are other functions that read data from a stream. One of these functions is *`getline`*, which reads characters from a stream until a newline character is encountered, then stores the read characters (minus the newline) in a `string`. `getline` accepts two parameters, a stream to read from and a `string` to write to. For example, to read a line of text from the console, you could use this code:

```
string myStr;
getline(cin, myStr);
```

No matter how many words or tokens the user types on this line, because `getline` reads until it encounters a newline, all of the data will be absorbed and stored in `myStr`. Moreover, because any data the user types in can be expressed as a string, unless your input stream encounters a read error, `getline` will not put the stream into a fail state. No longer do you need to worry about strange I/O edge cases!

You may have noticed that the `getline` function acts similarly to the CS106B/X `GetLine` function. This is no coincidence, and in fact the `GetLine` function from `simpio.h` is implemented as follows:*

```
string GetLine() {
    string result;
    getline(cin, result);
    return result;
}
```

At this point, `getline` may seem like a silver-bullet solution to our input problems. However, `getline` has a small problem when mixed with the stream extraction operator. When the user presses return after entering text in response to a `cin` prompt, the newline character is stored in the `cin` internal buffer. Normally, whenever you try to extract data from a stream using the `>>` operator, the stream skips over newline and whitespace characters before reading meaningful data. This means that if you write code like this:

```
int first, second;
cin >> first;
cin >> second;
```

The newline stored in `cin` after the user enters a value for `first` is eaten by `cin` before `second` is read. However, if we replace the second call to `cin` with a call to `getline`, as shown here:

```
int dummyInt;
string dummyString;
cin >> dummyInt;
getline(cin, dummyString);
```

`getline` will return an empty string. Why? Unlike the stream extraction operator, `getline` does *not* skip over the whitespace still remaining in the `cin` stream. Consequently, as soon as `getline` is called, it will find the newline remaining from the previous `cin` statement, assume the user has pressed return, and return the empty string.

To fix this problem, your best option is to replace all normal stream extraction operations with calls to library functions like `GetInteger` and `GetLine` that accomplish the same thing. Fortunately, with the information in the next section, you'll be able to write `GetInteger` and almost any `Get____` function you'd ever need to use. When we cover templates and operator overloading in later chapters, you'll see how to build a generic read function that can parse any sort of data from the user.

Reading Files with `getline`

Our treatment of `getline` so far has only considered using `getline` to read data from `cin`, but `getline` is in fact much more general and can be used to read data from any stream object, including file streams. To give a better feel for how the `getline` function works in practice, let's go over a quick example of how to use `getline` to read data from files. In this example, we'll write a program that takes in a data file containing some useful information and display it in a nice, pretty format. In particular, we'll write a program that reads a data file called `world-capitals.txt` containing a list of all the world's countries and their capitals, then displays them to the user. We will assume that the `world-capitals.txt` file is formatted as follows:

---

* Technically, the implementation of `GetLine` from `simpio.h` is slightly different, as it checks to make sure that `cin` is not in an error state before reading.

File: `world-capitals.txt`

```
Abu Dhabi
United Arab Emirates
Abuja
Nigeria
Accra
Ghana
Addis Ababa
Ethiopia
...
```

In this file, every pair of lines represents a capital city and the country of which it is the capital. For example, the first two lines indicate that Abu Dhabi is the capital of the United Arab Emirates, the second two that Abuja is the capital of Nigeria, etc. Our goal is to write a program that prints this data in the following format:

```
Abu Dhabi is the capital of United Arab Emirates
Abuja is the capital of Nigeria
Accra is the capital of Ghana
...
```

How can we go about writing a program like this? Well, we can start by opening the file and printing an error if we can't find it:

```
int main() {
    ifstream capitals("world-capitals.txt")
    if (!capitals.is_open()) {
        cerr << "Cannot find the file world-capitals.txt" << endl;
        return -1;
    }

    /* ... */
}
```

Now, we need to process pairs of lines in the file. Using the concepts from this chapter, we have two general lines of attack to consider. First, we could use the stream extraction operator `>>` to read the data from the file. Second, we could use the `getline` function to read lines of text from the file. In this particular circumstance, it is not a particularly good idea to use the stream extraction operator. Remember that the extraction operator reads data from files one token at a time, rather than one line at a time. Not all world capitals are a single token long (for example, Abu Dhabi or Addis Ababa) nor are all countries one token long (for example, United Arab Emirates). If we were to try to read the file data using the stream extraction operator, we would have no way of knowing when we had read in the complete name of a capital city or country, and it would be all but impossible to print the data out in a meaningful format. However, `getline` does not have this problem, since `getline` blindly reads lines of text and has no notion of whitespace-delineated tokens. Thus for this particular program, we'll use the `getline` function to read file data.

As with most file reading operations, we will need to keep looping until we've exhausted all of the data in the file. This can usually be done with the loop-and-a-half idiom. In our case, one possible version of the code is as follows:

```
int main() {
    ifstream capitals("world-capitals.txt")
    if (!capitals.is_open()) {
        cerr << "Cannot find the file world-capitals.txt" << endl;
        return -1;
    }

    while (true) {
        string capital, country;
        getline(capitals, capital);
        getline(capitals, country);

        if (capitals.fail()) break;

        cout << capital << " is the capital of " << country << endl;
    }
}
```

The above code creates two strings, `capital` and `country`, and populates them with data from the file. It then checks whether the read succeeded, and, if so, prints out the formatted data string.

This code is perfectly correct, but it's clunky. The loop-and-a-half idiom is never pretty, and there has to be a better way to structure this code. Fortunately, there is a wonderful shorthand we can use to condense this code. Recall that when using the stream extraction operator `>>`, we could write code to the following effect to read data from a file and continue looping while the read operation succeeds:

```
while (myStream >> myValue) {
    /* ... process myValue here ... */
}
```

We can use a similar trick with `getline`. In particular, the `getline` function returns a nonzero value if data can be read from a file and a zero value otherwise. Consequently, we can rewrite the above code as follows:

```
int main()
{
    ifstream capitals("world-capitals.txt")
    if (!capitals.is_open()) {
        cerr << "Cannot find the file world-capitals.txt" << endl;
        return -1;
    }

    string capital, country;
    while (getline(capitals, capital) && getline(capitals, country))
        cout << capital << " is the capital of " << country << endl;
}
```

This code is considerably more concise than our original version and arguably easier to read. The condition of the `while` loop now reads "while we can read a line from the file into `capital` and a line from the file into `country`, keep executing the loop." If you ever find yourself reading a file line-by-line, feel free to adapt this trick into your own code – you'll save yourself a great deal of typing if you do.

A String Buffer: `stringstream`

Before we discuss writing `GetInteger`, we'll need to take a diversion to another type of C++ stream.

Often you will need to construct a string composed both of plain text and numeric or other data. For example, suppose you wanted to call this hypothetical function:

```
void MessageBoxAlert(string message);
```

and have it display a message box to the user informing her that the level number she wanted to warp to is out of bounds. At first thought, you might try something like

```
int levelNum = /* ... */;
MessageBoxAlert("Level " + levelNum + " is out of bounds."); // Error
```

For those of you with Java experience this might seem natural, but in C++ this isn't legal because you can't add numbers to strings (and when you can, it's almost certainly won't do what you expected; see the chapter on C strings).

One solution to this problem is to use another kind of stream object known as a *`stringstream`*, exported by the `<sstream>` header. Like console streams and file streams, `stringstream`s are stream objects and consequently all of the stream operations we've covered above work on `stringstream`s. However, instead of reading or writing data to an external source, `stringstream`s store data in temporary string buffers. In other words, you can view a `stringstream` as a way to create and read string data using stream operations.

For example, here is a code snippet to create a `stringstream` and put text data into it:

```
stringstream myStream;
myStream << "Hello!" << 137;
```

Once you've put data into a `stringstream`, you can retrieve the string you've created using the `.str()` member function. Continuing the above example, we can print out an error message as follows:

```
int levelNum = /* ... */;
stringstream messageText;

messageText << "Level " << levelNum << " is out of bounds.";
MessageBoxAlert(messageText.str());
```

`stringstream`s are an example of an `iostream`, a stream that can perform both input and output. You can both insert data into a `stringstream` to convert the data to a string and extract data from a `stringstream` to convert string data into a different format. For example:

```
stringstream myConverter;
int myInt;
string myString;
double myDouble;

myConverter << "137 Hello 2.71828";              // Insert string data
myConverter >> myInt >> myString >> myDouble;    // Extract mixed data
```

The standard rules governing stream extraction operators still apply to `stringstream`s, so if you try to read data from a `stringstream` in one format that doesn't match the character data, the stream will fail. We'll exploit this functionality in the next section.

Putting it all together: Writing `GetInteger`

Using the techniques we covered in the previous sections, we can implement a set of robust user input functions along the lines of those provided by `simpio.h`. In this section we'll explore how to write `GetInteger`, which prompts the user to enter an integer and returns only after the user enters valid input.

Recall from the above sections that reading an integer from `cin` can result in two types of problems. First, the user could enter something that is not an integer, causing `cin` to fail. Second, the user could enter too much input, such as `137 246` or `Hello 37`, in which case the operation succeeds but leaves extra data in `cin` that can garble future reads. We can immediately eliminate these sorts of problems by using the `getline` function to read input, since `getline` cannot put `cin` into a fail state and grabs all of the user's data, rather than just the first token.

The main problem with `getline` is that the input is returned as a `string`, rather than as formatted data. Fortunately, using a `stringstream`, we can convert this text data into another format of our choice. This suggests an implementation of `GetInteger`. We read data from the console using `getline` and funnel it into a `stringstream`. We then use standard stream manipulations to extract the integer from the `stringstream`, reporting an error and reprompting if unable to do so. We can start writing `GetInteger` as follows:

```
int GetInteger() {
    while(true) { // Read input until user enters valid data
        stringstream converter;
        converter << GetLine();

        /* Process data here.  On error: */
        cout << "Retry: "
    }
}
```

At this point, we've read in all of the data we need, and simply need to check that the data is in the proper format. As mentioned above, there are two sorts of problems we might run into – either the data isn't an integer, or the data contains leftover information that isn't part of the integer. We need to check for both cases. Checking for the first turns out to be pretty simple – because `stringstream`s are stream objects, we can see if the data isn't an integer by extracting an integer from our `stringstream` and checking if this puts the stream into a fail state. If so, we know the data is invalid and can alert the user to this effect.

The updated code for `GetInteger` is as follows:

```
int GetInteger() {
    while(true) { // Read input until user enters valid data
        stringstream converter;
        converter << GetLine();

        /* Try reading an int, continue if we succeeded. */
        int result;
        if(converter >> result) {
            /* ... check that there isn't any leftover data ... */
        } else
            cout << "Please enter an integer." << endl;

        cout << "Retry: "
    }
}
```

Finally, we need to check if there's any extra data left over. If so, we need to report to the user that something is wrong with the input, and can otherwise return the value we read. While there are several ways to check this, one simple method is to read in a single `char` from the `stringstream`. If it is possible to do so, then we know that there must have been something in the input stream that wasn't picked up when we extracted an integer and consequently that the input is bad. Otherwise, the stream must be out of data and will enter a fail state, signaling that the user's input was valid. The final code for `GetInteger`, which uses this trick, is shown here:

```
int GetInteger() {
    while(true) { // Read input until user enters valid data
        stringstream converter;
        converter << GetLine();

        /* Try reading an int, continue if we succeeded. */
        int result;
        if(converter >> result) {
            char remaining;
            if(converter >> remaining) // Something's left, input is invalid
                cout << "Unexpected character: " << remaining << endl;
            else
                return result;
        } else
            cout << "Please enter an integer." << endl;

        cout << "Retry: "
    }
}
```

More To Explore

C++ streams are extremely powerful and encompass a huge amount of functionality. While there are many more facets to explore, I highly recommend exploring some of these topics:

- **Random Access**: Most of the time, when performing I/O, you will access the data sequentially; that is, you will read in one piece of data, then the next, etc. However, in some cases you might know in advance that you want to look up only a certain piece of data in a file without considering all of the data before it. For example, a ZIP archive containing a directory structure most likely stores each compressed file at a different offset from the start of the file. Thus, if you wanted to write a program capable of extracting a single file from the archive, you'd almost certainly need the ability to jump to arbitrary locations in a file. C++ streams support this functionality with the `seekg`, `tellg`, `seekp`, and `tellp` functions (the first two for `istream`s, the latter for `ostream`s). Random access lets you quickly jump to single records in large data blocks and can be useful in data file design.

- **`read` and `write`**: When you write numeric data to a stream, you're actually converting them into sequences of characters that represent those numbers. For example, when you print out the four-byte value `78979871`, you're using eight bytes to represent the data on screen or in a file – one for each character. These extra bytes can quickly add up, and it's actually possible to have on-disk representations of data that are more than twice as large as the data stored in memory. To get around this, C++ streams let you directly write data from memory onto disk without any formatting. All `ostream`s support a `write` function that writes unformatted data to a stream, and `istream`s support `read` to read unformatted data from a stream into memory. When used well, these functions can cut file loading times and reduce disk space usage. For example, The CS106B/X `Lexicon` class uses `read` to quickly load its data file into memory.

Practice Problems

Here are some questions to help you play around with the material from this chapter. Try some of these out; you'll be a better coder for the effort.

1. How do you write data to a file in C++?

2. What does the `setw` manipulator do? What does the `setfill` manipulator do? How do you use them?

3. What is stream failure? How do you check for it?

4. What is a `stringstream`?

5. Using a `stringstream`, write a function that converts an `int` into a `string`.

6. Modify the code for `GetInteger` to create a function `GetReal` that reads a real number from the user. How much did you need to modify to make this code work?

7. Using the code for `GetInteger` and the `boolalpha` stream manipulator, write a function `GetBoolean` that waits for the user to enter “true” or “false” and returns the corresponding boolean value.

8. In common usage, numbers are written in *decimal* or *base 10*. This means that a string of digits is interpreted as a sum of multiples of powers of ten. For example, the number 137 is $1 \cdot 100 + 3 \cdot 10 + 7 \cdot 1$, which is the same as $1 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0$. However, it is possible to write numbers in other bases as well. For example, *octal*, or base 8, encodes numbers as sums of multiples of powers of eight. For example, 137 in octal would be $1 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 64 + 24 + 7 = 95$ in decimal.* Similarly, *binary*, or base 2, uses powers of two.

   When working in a particular base, we only use digits from 0 up to that base. Thus in base 10 we use the digits zero through nine, while in base five the only digits would be 0, 1, 2, 3, and 4. This means that 57 is not a valid base-five number and 93 is not a valid octal number. When working in bases numbered higher than ten, it is customary to use letters from the beginning of the alphabet as digits. For example, in *hexadecimal*, or base 16, one counts 0, 1, 2, ..., 9, A, B, C, D, E, F, 10. This means that 3D45E is a valid hexadecimal number, as is DEADBEEF or DEFACED.

   Write a function `HasHexLetters` that accepts an `int` and returns whether or not that integer's hexadecimal representation contains letters. *(Hint: you'll need to use the `hex` and `dec` stream manipulators in conjunction with a `stringstream`. Try to solve this problem without brute-forcing it: leverage off the streams library instead of using loops.)* ♦

9. Although the console does not naturally lend itself to graphics programming, it is possible to draw rudimentary approximations of polygons by printing out multiple copies of a character at the proper location. For example, we can draw a triangle by drawing a single character on one line, then three on the next, five on the line after that, etc. For example:

```
    #
   ###
  #####
 #######
#########
```

   Using the `setw` and `setfill` stream manipulators, write a function `DrawTriangle` that takes in an `int` corresponding to the height of the triangle and a `char` representing a character to print, then draws a triangle of the specified height using that character. The triangle should be aligned so that the bottom row starts at the beginning of its line.

10. Write a function `OpenFile` that accepts as input an `ifstream` by reference and prompts the user for the name of a file. If the file can be found, `OpenFile` should return with the `ifstream` opened to read that file. Otherwise, `OpenFile` should print an error message and reprompt the user. *(Hint: If you try to open a nonexistent file with an `ifstream`, the stream goes into a fail state and you will need to use `.clear()` to restore it before trying again)*.

* Why do programmers always confuse Halloween and Christmas? Because 31 Oct = 25 Dec. ☺

Chapter 4: Multi-File Programs, Abstraction, and the Preprocessor

All of the programs we saw in the previous chapter were fairly short – the most complex of them ran at just under one hundred lines of code. In industrial settings, though, programs are far bigger, and in fact it is common for programs to be tens of millions of lines of code. When code becomes this long, it is simply infeasible to store all of the source code in a single file. Were all the code to be stored in a single file, it would be next to impossible to find a particular function or constant declaration, and it would be incredibly difficult to discern any of the high-level structure of the program. Consequently, most large programs are split across multiple files.

When splitting a program into multiple files, there are many considerations to take into account. First, what support does C++ have for partitioning a program across multiple files? That is, how do we communicate to the C++ compiler that several source files are all part of the same program? Second, what is the best way to logically partition the program into multiple files? In other words, of all of the many ways we could break the program apart, which is the most sensible?

In this chapter, we will address these questions, plus several related problems that arise. First, we will talk about the C++ compilation model – the way that C++ source files are compiled and linked together. Next, we will explore the most common means for splitting a project across files by seeing how to write custom header and implementation files. Finally, we will see how header files work by discussing the *preprocessor*, a program that assists the compiler in generating C++ code.

The C++ Compilation Model

C++ is a *compiled language*, meaning that before a C++ program executes, a special program called the *compiler* converts the C++ program directly to machine code. Once the program is compiled, the resulting executable can be run any number of times, even if the source code is nowhere to be found.

C++ compilation is a fairly complex process that involves numerous small steps. However, it can generally be broken down into three larger processes:

- *Preprocessing*, in which code segments are spliced and inserted,
- *Compilation*, in which code is converted to object code, and
- *Linking*, in which compiled code is joined together into a final executable.

During the preprocessing step, a special program called the preprocessor scans over the C++ source code and applies various transformations to it. For example, `#include` directives are resolved to make various libraries available, special tokens like `__FILE__` and `__LINE__` (covered later) are replaced by the file and line number in the source file, and `#define`-d constants and macros (also covered later) are replaced by their appropriate values.

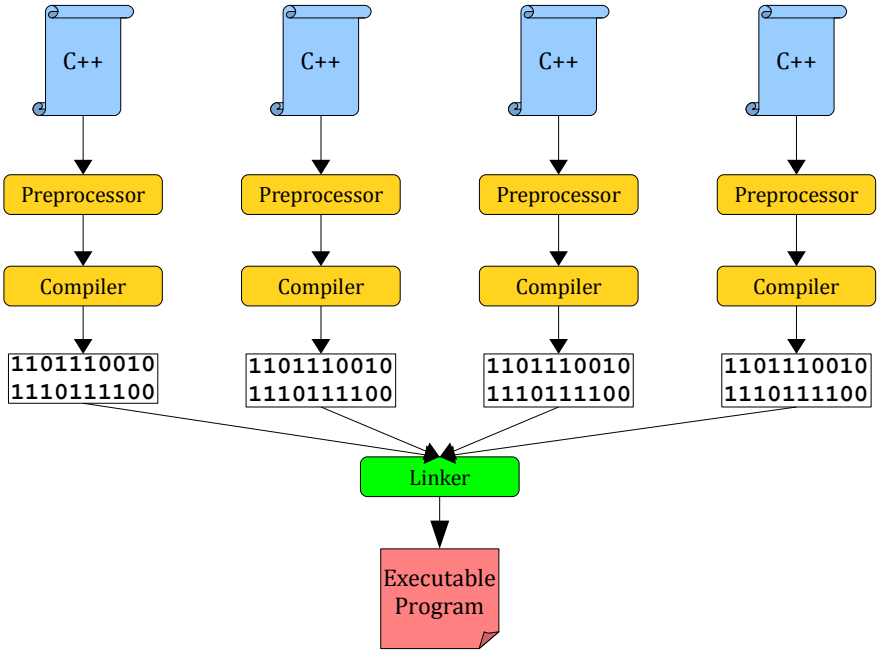
In the compilation step, the C++ source file is read in by the compiler, optimized, and transformed into an *object file*. These object files are machine-specific, but usually contain machine code which executes the instructions specified in the C++ file, along with some extra information. It's at this stage where the compiler will report any syntax errors you make, such as omitting semicolons, referencing undefined variables, or passing arguments of the wrong types into functions.

Finally, in the linking phase, a program called the *linker* gathers together all of the object files necessary to build the final executable, bundles them together with OS-specific information, and finally produces an

executable file that you can run and distribute. During this phase, the linker may report some final errors that prevent it from generating a working C++ program. For example, consider the following C++ program:

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

int Factorial(int n); // Prototype for a function to compute n!

int main() {
    cout << Factorial(10) << endl;
    return 0;
}
```

This program prototypes a function called `Factorial`, calls it in `main`, but never actually defines it. Consequently, this program is erroneous and will not run. However, the error is not detected by the compiler; rather, it shows up as a linker error. During linking, the linker checks to see that every function that was prototyped and called has a corresponding implementation. If it finds that some function has no implementation, it reports an error. In order to understand why this is, consider the following diagram, which portrays the relationships between the three main phases of compilation:



Notice that during compilation, each C++ source file is treated independently the others, but during linking all of the files are glued together. Consequently, it's possible (and, in fact, extremely common) for a function to be *prototyped* in one C++ file but *implemented* in another. For this reason, if the compiler sees a prototype for a function but no implementation, it doesn't report an error – the definition might just be in a different file it hasn't seen yet. Only when all of the files are pulled together by the linker is there an opportunity to check that all of the prototyped functions have some sort of implementation.

What does this mean for you as a C++ programmer? In practice, this distinction usually only manifests itself in the types of error messages you may get compilation. In particular, a program may compile perfectly well but fail to link because you prototyped a function that was never defined. Understanding the source of these errors and why they are reported during linking will help you diagnose these errors more handily.

As an example, consider the following C++ program, which contains a subtle error:

```
#include <iostream>
#include <string>
#include <cctype> // For tolower
using namespace std;

/* Prototype a function called ConvertToLowerCase, which returns a lower-case
 * version of the input string.
 */
string ConvertToLowerCase(string input);

int main() {
    string myString = "THIS IS A STRING!";
    cout << ConvertToLowerCase(myString);
}

/* Implementation of ConvertToLowerCase. */
string ConvertToLowerCase(string& input) { // Error: Doesn't link; see below
    for (int k = 0; k < input.size(); ++k)
        input[k] = tolower(input[k]); // tolower converts a char to lower-case

    return input;
}
```

If you compile this program in `g++`, the program compiles but the linker will produce this mysterious error:

```
main.cpp:(.text+0x14d): undefined reference to
    `ConvertToLowerCase(std::basic_string<char, std::char_traits<char>,
    std::allocator<char> >)'
```

If you compile this program in Microsoft Visual Studio 2005, it will similarly compile and produce this monstrosity of an error:

```
error LNK2019: unresolved external symbol "class std::basic_string<char,struct
     std::char_traits<char>,class std::allocator<char> > __cdecl
     ConvertToLowerCase(class std::basic_string<char,struct
     std::char_traits<char>,class std::allocator<char> >)"
     (?ConvertToLowerCase@@YA?AV?$
     basic_string@DU?$char_traits@D@std@@V?$allocator@D@2@@std@@V12@@Z) referenced
     in function _main
```

What's going on here? This error is tricky to decipher, but as you can see from the highlighting has something to do with `ConvertToLowerCase`. Let's try to see if we can get to the root of the problem. Since this is a linker error, we can immediately rule out any sort of syntax error. If we had made a syntactic mistake, the *compiler*, not the *linker*, would have caught it. Moreover, since this is a linker error, it means that we somehow prototyped a function that we never got around to implementing. This seems strange though – we prototyped the function `ConvertToLowerCase` and it seems like we implemented it later on in the program. The problem, though, is that the function we implemented doesn't match the prototype. Here are the prototype and the implementation, reprinted right next to each other:

```
string ConvertToLowerCase(string input); // Prototype

string ConvertToLowerCase(string& input) { // Implementation
    for (int k = 0; k < input.size(); ++k)
        input[k] = tolower(input[k]); // tolower converts a char to lower-case

    return input;
}
```

Notice that the function we've prototyped takes in a `string` as a parameter, while the implementation takes in a `string&`. That is, the prototype takes its argument by *value*, and the implementation by *reference*. Because these are different parameter-passing schemes, the compiler treats the implementation as a completely different function than the one we've prototyped. Consequently, during linking, the linker can't locate an implementation of the prototyped function, which takes in a `string` by value. Although the functions have the same name, their signatures are different, and they are treated as entirely different entities.

To fix this problem, we must either update the prototype to match the implementation or the implementation to match the prototype. In this case, we'll change the implementation so that it no longer takes in the parameter by reference. This results in the following program, which compiles and links without error:

```
#include <iostream>
#include <string>
#include <cctype> // For tolower
using namespace std;

/* Prototype a function called ConvertToLowerCase, which returns a lower-case
 * version of the input string.
 */
string ConvertToLowerCase(string input);

int main() {
    string myString = "THIS IS A STRING!";
    cout << ConvertToLowerCase(myString);
}

/* Implementation of ConvertToLowerCase. */
string ConvertToLowerCase(string input) { // Now corrected.
    for (int k = 0; k < input.size(); ++k)
        input[k] = tolower(input[k]); // tolower converts a char to lower-case

    return input;
}
```

Running this program produces the output

```
this is a string!
```

If you ever write a program and discover that it produces a linker error, always check to make sure that you've implemented all functions you've prototyped and that those implementations match the prototypes. Otherwise, you might be directing your efforts toward catching a nonexistent syntax error.

Modularity and Abstraction

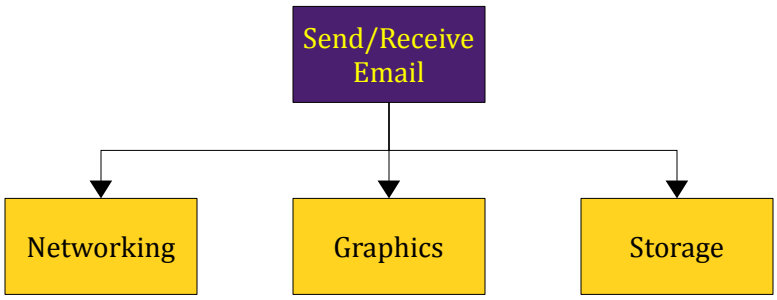
Because compilation and linking are separate steps in C++, it is possible to split up a C++ program across multiple files. To do so, we must first answer two questions:

1. ***How* do you split a program up?** That is, syntactically, how do you communicate to the C++ compiler that you want to build a single program from a collection of files?

2. **What is the *best way* to split a program up?** In other words, given how a single C++ program can be built from many files, what is the best way to logically partition the program code across those files?

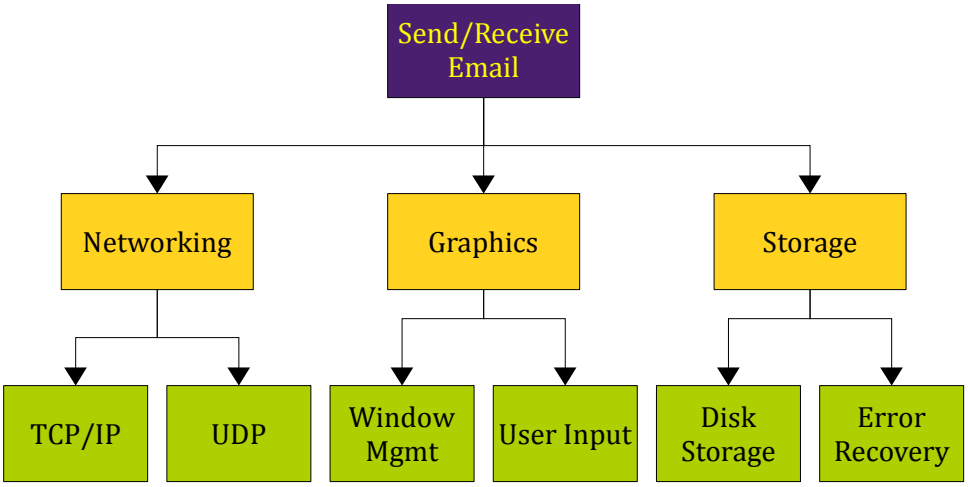
To answer these questions, we first must take a minute to reflect on the structure of most C++ programs.* When writing a C++ program to perform a particular task or solve a particular problem, one usually begins by starting with a large, difficult problem and then solves that problem by breaking it down into smaller and smaller pieces. For example, suppose we want to write a program that allows the user to send and receive emails. Initially, we can think of this as one, enormous task:

How might we go about building such a program? Well, we might begin by realizing that to write an email client, we will need to be able to communicate over a network, since we'll be transmitting and receiving data. Also, we will need some way to store the emails we've received on the user's hard disk so that she can read messages while offline. We'll also need to be able to display graphics contained in those emails, as well as create windows for displaying content. Each one of these tasks is itself a fairly complex problem which needs to be solved, and so if we rethink our strategy for writing the email client, we might be able to visualize it as follows:



Of course, these tasks in of themselves might have some related subproblems. For example, when reading and writing from disk, we will need some tools to allow us to read and write general data from disk, another set of libraries to structure the data stored on disk, another to recover gracefully from errors, etc. Here is one possible way of breaking each of the subproblems down into smaller units:

* In fact, programs in virtually *any* language will have the structure we're about to describe.



In general, we write programs to solve large, complicated tasks by breaking the task down into successively smaller and smaller pieces, then combining all of those pieces back together into a coherent whole.* When using this setup, though, one must be extremely careful. Refer to the above diagram and notice that the abstract problem at the top depends directly on three subproblems. Each of those subproblems depends in turn on even *more* subproblems, etc. If the top-level program needed to explicitly know how each of these sub-subproblems were to work, it would be all but impossible to write. A programmer tasked with designing the email program shouldn't have to understand exactly how the networking module works, but should instead only need to know how to use it. Similarly, in order to use the windowing module, the programmer shouldn't have to understand all of its internal workings.

In order for each part of the program to use its subcomponents without getting overwhelmed by complexity, there must be some way to separate out how each subproblem is *solved* from the way that each subproblem is *used*. For example, think back to the streams library from the previous chapter. As you saw, you can use the `ifstream` and `ofstream` classes to read and write files. But how exactly are `ifstream` and `ofstream` put together behind the scenes? Internally, these classes are incredibly complicated and are composed of numerous different pieces of C++ code that fit together in intricate ways. From your perspective, though, all of this detail is irrelevant; you only care about how you use these stream classes to do input and output.

This distinction between the inner workings of a module (a collection of source code that solves a problem) and the way in which a client use it is an example of *abstraction*. An abstraction is a simplification of a complex object – whether physical or in software – that allows it to be used without an understanding of its underlying mechanism. For example, the iPhone is an incredibly complex piece of hardware with billions of transistors and gates. Even the simplest of tasks, such as making a phone call or sending email, triggers a flurry of electrical activity in the underlying device. But despite the implementation complexity, the iPhone is incredibly easy to use because the interface works at a high level, with tasks like "send text message" or "play music." In other words, the complexity of the implementation is hidden behind a very simple interface.

When designing software, you should strive to structure your software in a similar manner. Whenever you write code to solve a particular task, you should try to package that code so that it communicates primarily *what* it does, rather than *how* it does it. This has several advantages:

---

* For those of you familiar with recursion, you might recognize that this general structure follows a simple recursive formulation: if the problem is simple enough, solve it; otherwise break it down into smaller pieces, solve those pieces, and then glue them all together again.

- **Simplicity**. If you package your code by giving it a simple interface, you make it easier for yourself and other programmers to use. Moreover, if you take a break from a project and then return to it later, it is significantly easier to resume if the interface clearly communicates its intention.

- **Extensibility**. If you design a simple, elegant interface, then you can change the implementation as the program evolves over time without breaking client code. We'll see examples of this later in the chapter.

- **Reusability**. If your interface is sufficiently generic, then you may be able to reuse the code you've written in multiple projects. As an example, the streams library is sufficiently flexible that you can use it to write both a simple "Hello, World!" and a complex program with detailed file-processing requirements.

A Sample Module: String Utilities

To give you a sense for how interfaces and implementations look in software, let's take a quick diversion to build a sample C++ module to simplify common string operations. In particular, we'll write a collection of functions that simplify conversion of several common types to strings and vice-versa, along with conversions to lower- and upper-case.*

In C++, to create a module, we create two files – a *header file* saying what functions and classes a module exports, and an *implementation file* containing the implementations of those functions and classes. Header files usually have the extension .h, though the extension .hh is also sometimes used. Implementation files are regular C++ files, so they often use the extensions .cpp, .cc, or (occasionally) .C or .cpp. Traditionally, a header file and its associated implementation file will have the same name, ignoring the extension. For example, in our string processing library, we might name the header file `strutils.h` and the implementation file `strutils.cpp`.

To give you a sense for what a header file looks like, consider the following code for `strutils.h`:

File: `strutils.h`

```
#ifndef StrUtils_Included
#define StrUtils_Included

#include <string>
using namespace std;

string ConvertToUpperCase(string input);
string ConvertToLowerCase(string input);

string IntegerToString(int value);
string DoubleToString(double value);

#endif
```

Notice that the highlighted part of this file looks just like a regular C++ file. There's a `#include` directive to import the `string` type, followed by several prototypes for functions. However, none of these functions are implemented – the purpose of this file is simply to say what the module exports, not to provide the implementations of those functions.

However, this header file contains some code that you have not yet seen in C++ programs: the lines

* In other words, we'll be writing the `strutils.h` library from CS106B/X.

```
#ifndef StrUtils_Included
#define StrUtils_Included
```

and the line

```
#endif
```

These lines are called an *include guard*. Later in this chapter, we will see exactly why they are necessary and how they work. In the meantime, though, you should note that whenever you create a header file, you should surround that file using an include guard. There are many ways to write include guards, but one simple approach is as follows. When creating a file named ***file***.h, you should surround the file with the lines

```
#ifndef File_Included
#define File_Included

#endif
```

Now that you've seen how to write a header file, let's write the matching implementation file. This is shown here:

File: strutils.cpp

```
#include "strutils.h"
#include <cctype>  // For tolower, toupper
#include <sstream> // For stringstream

string ConvertToUpperCase(string input) {
    for (size_t k = 0; k < input.size(); ++k)
        input[k] = toupper(input[k]);
    return input;
}

string ConvertToUpperCase(string input) {
    for (size_t k = 0; k < input.size(); ++k)
        input[k] = toupper(input[k]);
    return input;
}

string IntegerToString(int input) {
    stringstream converter;
    converter << input;
    return converter.str();
}

string DoubleToString(double input) {
    stringstream converter;
    converter << input;
    return converter.str();
}
```

This C++ source file does not contain any new language constructs – it's just your standard, run-of-the-mill C++ file. However, do note that it provides an implementation of every file exported in the header file. Moreover, the file begins with the line

```
#include "strutils.h"
```

Traditionally, an implementation file `#include`s its corresponding header file. When we discuss the preprocessor in the latter half of this chapter, the rationale behind this should become more clear.

Now that we've written the `strutils.h/.cpp` pair, we can use these functions in other C++ source files. For example, consider the following simple C++ program:

```
#include <iostream>
#include <string>
#include "strutils.h"
using namespace std;

int main() {
    cout << ConvertToLowerCase("THIS IS A STRING!");
    return 0;
}
```

This program produces the output

```
this is a string!
```

Notice that nowhere in this file did we implement or define the `ConvertToLowerCase` function. It suffices to `#include "strutils.h"` to gain access to this functionality.

Behind the Curtain: The Preprocessor

One of the most exciting parts of writing a C++ program is pressing the "compile" button and watching as your code transforms from static text into dynamic software. As mentioned earlier, this process proceeds in several steps. One of the first of these steps is *preprocessing*, where a special program called the *preprocessor* reads in commands called *directives* and modifies your code before handing it off to the compiler for further analysis. You have already seen one of the more common preprocessor directives, `#include`, which imports additional code into your program. However, the preprocessor has far more functionality and is capable of working absolute wonders on your code. But while the preprocessor is powerful, it is difficult to use correctly and can lead to subtle and complex bugs. The rest of this chapter introduces the preprocessor, highlights potential sources of error, and concludes with advanced preprocessor techniques.

A word of warning: the preprocessor was developed in the early days of the C programming language, before many of the more modern constructs of C and C++ had been developed. Since then, both C and C++ have introduced new language features that have obsoleted or superseded much of the preprocessor's functionality and consequently you should attempt to minimize your use of the preprocessor. This is not to say, of course, that you should never use the preprocessor – there are times when it's an excellent tool for the job, as you'll see later in the chapter – but do consider other options before adding a hastily-crafted directive.

`#include` Explained

In both CS106B/X and CS106L, every program you've encountered has begun with several lines using the `#include` directive; for example, `#include <iostream>` or `#include "genlib.h"`. Intuitively, these directives tell the preprocessor to import library code into your programs. Literally, `#include` instructs the preprocessor to locate the specified file and to insert its contents in place of the directive itself. Thus, when you write `#include "genlib.h"` at the top of your CS106B/X assignments, it is as if you had copied and pasted the contents of `genlib.h` into your source file. These header files usually contain

prototypes or implementations of the functions and classes they export, which is why the directive is necessary to access other libraries.

You may have noticed that when `#include`-ing CS106B/X-specific libraries, you've surrounded the name of the file in double quotes (e.g. `"genlib.h"`), but when referencing C++ standard library components, you surround the header in angle brackets (e.g. `<iostream>`). These two different forms of `#include` instruct the preprocessor where to look for the specified file. If a filename is surrounded in angle brackets, the preprocessor searches for it a compiler-specific directory containing C++ standard library files. When filenames are in quotes, the preprocessor will look in the current directory.

`#include` is a preprocessor directive, not a C++ statement, and is subject to a different set of syntax restrictions than normal C++ code. For example, to use `#include` (or any preprocessor directive, for that matter), the directive must be the first non-whitespace text on its line. For example, the following is illegal:

```
cout << #include <iostream> << endl; // Error: #include must start a line.
```

Second, because `#include` is a preprocessor directive, not a C++ statement, it must not end with a semicolon. That is, `#include <iostream>;` will probably cause a compiler error or warning. Finally, the entire `#include` directive must appear on a single line, so the following code will not compile:

```
#include
<iostream> // Error: Multi-line preprocessor directives are illegal.
```

The `#define` Directive

One of the most commonly used (and abused) preprocessor directives is `#define`, the equivalent of a "search and replace" operation on your C++ source files. While `#include` splices new text into an existing C++ source file, `#define` replaces certain text strings in your C++ file with other values. The syntax for `#define` is

```
#define phrase replacement
```

After encountering a `#define` directive, whenever the preprocessor find ***`phrase`*** in your source code, it will replace it with ***`replacement`***. For example, consider the following program:

```
#define MY_CONSTANT 137

int main() {
    int x = MY_CONSTANT - 3;
    return 0;
}
```

The first line of this program tells the preprocessor to replace all instances of `MY_CONSTANT` with the phrase `137`. Consequently, when the preprocessor encounters the line

```
int x = MY_CONSTANT - 3;
```

It will transform it to read

```
int x = 137 - 3;
```

So `x` will take the value 134.

Because `#define` is a preprocessor directive and not a C++ statement, its syntax can be confusing. For example, `#define` determines the stop of the ***phrase*** portion of the statement and the start of the ***replacement*** portion by the position of the first whitespace character. Thus, if you write

```
#define TWO WORDS 137
```

The preprocessor will interpret this as a directive to replace the phrase `TWO` with `WORDS 137`, which is probably not what you intended. The ***replacement*** portion of the `#define` directive consists of all text after ***phrase*** that precedes the newline character. Consequently, it is legal to write statements of the form `#define` ***phrase*** without defining a replacement. In that case, when the preprocessor encounters the specified phrase in your code, it will replace it with nothingness, effectively removing it.

Note that the preprocessor treats C++ source code as sequences of strings, rather than representations of higher-level C++ constructs. For example, the preprocessor treats `int x = 137` as the strings "`int`," "`x`," "`=`," and "`137`" rather than a statement creating a variable `x` with value `137`.* It may help to think of the preprocessor as a scanner that can read strings and recognize characters but which has no understanding whatsoever of their meanings, much in the same way a native English speaker might be able to split Czech text into individual words without comprehending the source material.

That the preprocessor works with text strings rather than language concepts is a source of potential problems. For example, consider the following `#define` statements, which define margins on a page:

```
#define LEFT_MARGIN 100
#define RIGHT_MARGIN 100
#define SCALE .5

/* Total margin is the sum of the left and right margins, multiplied by some
 * scaling factor.
 */
#define TOTAL_MARGIN LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE
```

What happens if we write the following code?

```
int x = 2 * TOTAL_MARGIN;
```

Intuitively, this should set `x` to twice the value of `TOTAL_MARGIN`, but unfortunately this is not the case. Let's trace through how the preprocessor will expand out this expression. First, the preprocessor will expand `TOTAL_MARGIN` to `LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE`, as shown here:

```
int x = 2 * LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE;
```

Initially, this may seem correct, but look closely at the operator precedence. C++ interprets this statement as

```
int x = (2 * LEFT_MARGIN * SCALE) + RIGHT_MARGIN * SCALE;
```

Rather the expected

```
int x = 2 * (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE);
```

---

* Technically speaking, the preprocessor operates on "preprocessor tokens," which are slightly different from the whitespace-differentiated pieces of your code. For example, the preprocessor treats string literals containing whitespace as a single object rather than as a collection of smaller pieces.

And the computation will be incorrect. The problem is that the preprocessor treats the replacement for `TOTAL_MARGIN` as a string, not a mathematical expression, and has no concept of operator precedence. This sort of error – where a `#define`d constant does not interact properly with arithmetic expressions – is a common mistake. Fortunately, we can easily correct this error by adding additional parentheses to our `#define`. Let's rewrite the `#define` statement as

```
#define TOTAL_MARGIN (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE)
```

We've surrounded the replacement phrase with parentheses, meaning that any arithmetic operators applied to the expression will treat the replacement string as a single mathematical value. Now, if we write

```
int x = 2 * TOTAL_MARGIN;
```

It expands out to

```
int x = 2 * (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE);
```

Which is the computation we want. In general, if you `#define` a constant in terms of an expression applied to other `#define`d constants, make sure to surround the resulting expression in parentheses.

Although this expression is certainly more correct than the previous one, it too has its problems. What if we redefine `LEFT_MARGIN` as shown below?

```
#define LEFT_MARGIN 200 - 100
```

Now, if we write

```
int x = 2 * TOTAL_MARGIN
```

It will expand out to

```
int x = 2 * (LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE);
```

Which in turn expands to

```
int x = 2 * (200 - 100 * .5 + 100 * .5)
```

Which yields the incorrect result because `(200 - 100 * .5 + 100 * .5)` is interpreted as

```
(200 - (100 * .5) + 100 * .5)
```

Rather than the expected

```
((200 - 100) * .5 + 100 * .5)
```

The problem is that the `#define`d statement itself has an operator precedence error. As with last time, to fix this, we'll add some additional parentheses to the expression to yield

```
#define TOTAL_MARGIN ((LEFT_MARGIN) * (SCALE) + (RIGHT_MARGIN) * (SCALE))
```

This corrects the problem by ensuring that each `#define`d subexpression is treated as a complete entity when used in arithmetic expressions. When writing a `#define` expression in terms of other `#define`s,

make sure that you take this into account, or chances are that your constant will not have the correct value.

Another potential source of error with `#define` concerns the use of semicolons. If you terminate a `#define` statement with a semicolon, the preprocessor will treat the semicolon as part of the replacement phrase, rather than as an "end of statement" declaration. In some cases, this may be what you want, but most of the time it just leads to frustrating debugging errors. For example, consider the following code snippet:

```
#define MY_CONSTANT 137; // Oops-- unwanted semicolon!

int x = MY_CONSTANT * 3;
```

During preprocessing, the preprocessor will convert the line `int x = MY_CONSTANT * 3` to read

```
int x = 137; * 3;
```

This is not legal C++ code and will cause a compile-time error. However, because the problem is in the preprocessed code, rather than the original C++ code, it may be difficult to track down the source of the error. Almost all C++ compilers will give you an error about the statement `* 3` rather than a malformed `#define`.

As you can tell, using `#define` to define constants can lead to subtle and difficult-to-track bugs. Consequently, it's strongly preferred that you define constants using the `const` keyword. For example, consider the following `const` declarations:

```
const int LEFT_MARGIN = 200 - 100;
const int RIGHT_MARGIN = 100;
const int SCALE = .5;
const int TOTAL_MARGIN = LEFT_MARGIN * SCALE + RIGHT_MARGIN * SCALE;
int x = 2 * TOTAL_MARGIN;
```

Even though we've used mathematical expressions inside the `const` declarations, this code will work as expected because it is interpreted by the C++ compiler rather than the preprocessor. Since the compiler understands the *meaning* of the symbols `200 - 100`, rather than just the characters themselves, you will not need to worry about strange operator precedence bugs.

Include Guards Explained

Earlier in this chapter when we covered header files, you saw that when creating a header file, you should surround the header file using an *include guard*. What is the purpose of the include guard? And how does it work? To answer this question, let's see what happens when a header file lacks an include guard.

Suppose we make the following header file, `mystruct.h`, which defines a `struct` called `MyStruct`:

File: `mystruct.h`

```
struct MyStruct {
    int x;
    double y;
    char z;
};
```

What happens when we try to compile the following program?

```
#include "mystruct.h"
#include "mystruct.h" // #include the same file twice

int main() {
    return 0;
}
```

This code looks innocuous, but produces a compile-time error complaining about a redefinition of `struct MyStruct`. The reason is simple – when the preprocessor encounters each `#include` statement, it copies the contents of `mystruct.h` into the program without checking whether or not it has already included the file. Consequently, it will copy the contents of `mystruct.h` into the code *twice*, and the resulting code looks like this:

```
struct MyStruct {
    int x;
    double y;
    char z;
};
struct MyStruct {// <-- Error occurs here
    int x;
    double y;
    char z;
};

int main() {
    return 0;
}
```

The indicated line is the source of our compiler error – we've doubly-defined `struct MyStruct`. To solve this problem, you might think that we should simply have a policy of not `#include`-ing the same file twice. In principle this may seem easy, but in a large project where several files each `#include` each other, it may be possible for a file to indirectly `#include` the same file twice. Somehow, we need to prevent this problem from happening.

The problem we're running into stems from the fact that the preprocessor has no memory about what it has done in the past. Somehow, we need to give the preprocessor instructions of the form "if you haven't already done so, `#include` the contents of this file." For situations like these, the preprocessor supports conditional expressions. Just as a C++ program can use `if` ... `else if` ... `else` to change program flow based on variables, the preprocessor can use a set of preprocessor directives to conditionally include a section of code based on `#define`d values.

There are several conditional structures built into the preprocessor, the most versatile of which are `#if`, `#elif`, `#else`, and `#endif`. As you might expect, you use these directives according to the pattern

```
#if statement
    ...
#elif another-statement
    ...
#elif yet-another-statement
    ...
#else
    ...
#endif
```

There are two details we need to consider here. First, what sorts of expressions can these preprocessor directives evaluate? Because the preprocessor operates before the rest of the code has been compiled,

preprocessor directives can only refer to `#define`d constants, integer values, and arithmetic and logical expressions of those values. Here are some examples, supposing that some constant `MY_CONSTANT` is defined to 42:

```
#if MY_CONSTANT > 137                // Legal
#if MY_CONSTANT * 42 == MY_CONSTANT // Legal
#if sqrt(MY_CONSTANT) < 4            // Illegal, cannot call function sqrt
#if MY_CONSTANT == 3.14              // Illegal, can only use integral values
```

In addition to the above expressions, you can use the `defined` predicate, which takes as a parameter the name of a value that may have previously been `#define`d. If the constant has been `#define`d, `defined` evaluates to `1`; otherwise it evaluates to `0`. For example, if `MY_CONSTANT` has been previously `#define`d and `OTHER_CONSTANT` has not, then the following expressions are all legal:

```
#if defined(MY_CONSTANT)    // Evaluates to true.
#if defined(OTHER_CONSTANT) // Evaluates to false.
#if !defined(MY_CONSTANT)   // Evaluates to false.
```

Now that we've seen what sorts of expressions we can use in preprocessor conditional expressions, what is the *effect* of these constructs? Unlike regular `if` statements, which change control flow at execution, preprocessor conditional expressions determine whether pieces of code are included in the resulting source file. For example, consider the following code:

```
#if defined(A)
    cout << "A is defined." << endl;
#elif defined(B)
    cout << "B is defined." << endl;
#elif defined(C)
    cout << "C is defined." << endl;
#else
    cout << "None of A, B, or C is defined." << endl;
#endif
```

Here, when the preprocessor encounters these directives, whichever of the conditional expressions evaluates to true will have its corresponding code block included in the final program, and the rest will be ignored. For example, if `A` is defined, this entire code block will reduce down to

```
cout << "A is defined." << endl;
```

And the rest of the code will be ignored.

One interesting use of the `#if ... #endif` construct is to comment out blocks of code. Since C++ interprets all nonzero values as true and zero as false, surrounding a block of code in a `#if 0 ... #endif` block causes the preprocessor to eliminate that block. Moreover, unlike the traditional `/* ... */` comment type, preprocessor directives can be nested, so removing a block of code using `#if 0 ... #endif` doesn't run into the same problems as commenting the code out with `/* ... */`.

In addition to the above conditional directives, C++ provides two shorthand directives, `#ifdef` and `#ifndef`. `#ifdef` (**if def**ined) is a directive that takes as an argument a symbol and evaluates to true if the symbol has been `#define`d. Thus the directive `#ifdef` ***`symbol`*** is completely equivalent to `#if defined(`***`symbol`***`)`. C++ also provides `#ifndef` (**if n**ot **def**ined), which acts as the opposite of `#ifdef`; `#ifndef` ***`symbol`*** is equivalent to `#if !defined(`***`symbol`***`)`. Although these directives are strictly weaker than the more generic `#if`, it is far more common in practice to see `#ifdef` and `#ifndef` rather than `#if defined` and `#if !defined`, primarily because they are more concise.

Using the conditional preprocessor directives, we can solve the problem of double-including header files. Let's return to our example with `#include "mystruct .h"` appearing twice in one file. Here is a slightly modified version of the `mystruct.h` file that introduces some conditional directives:

File: `mystruct.h` (version 2)

```
#ifndef MyStruct_Included
#define MyStruct_Included

struct MyStruct {
    int x;
    double y;
    char z;
};

#endif
```

Here, we've surrounded the entire file in a block `#ifndef MyStruct_Included` ... `#endif`. The specific name `MyFile_Included` is not particularly important, other than the fact that it is unique to the `myfile.h` file. We could have just as easily chosen something like `#ifndef sdf39527dkb2` or another unique name, but the custom is to choose a name determined by the file name. Immediately after this `#ifndef` statement, we `#define` the constant we are considering inside the `#ifndef`. To see exactly what effect this has on the code, let's return to our original source file, reprinted below:

```
#include "mystruct.h"
#include "mystruct.h" // #include the same file twice

int main() {
    return 0;
}
```

With the modified version of `mystruct.h`, this code expands out to

```
#ifndef MyStruct_Included
#define MyStruct_Included

struct MyStruct {
    int x;
    double y;
    char z;
};

#endif
#ifndef MyStruct_Included
#define MyStruct_Included

struct MyStruct {
     int x;
    double y;
    char z;
};

#endif
int main() {
    return 0;
}
```

Now, as the preprocessor begins evaluating the `#ifndef` statements, the first `#ifndef` ... `#endif` block from the header file will be included since the constant `MyStruct_Included` hasn't been defined yet. The code then `#define`s `MyStruct_Included`, so when the program encounters the second `#ifndef` block, the code inside the `#ifndef` ... `#endif` block will not be included. Effectively, we've ensured that the contents of a file can only be `#include`d once in a program. The net program thus looks like this:

```
struct MyStruct {
    int x;
    double y;
    char z;
};
int main() {
    return 0;
}
```

Which is exactly what we wanted. This technique, known as an *include guard*, is used throughout professional C++ code, and, in fact, the boilerplate `#ifndef` / `#define` / `#endif` structure is found in virtually every header file in use today. Whenever writing header files, be sure to surround them with the appropriate preprocessor directives.

Macros

One of the most common and complex uses of the preprocessor is to define *macros*, compile-time functions that accepts parameters and output code. Despite the surface similarity, however, preprocessor macros and C++ functions have little in common. C++ functions represent code that executes at runtime to manipulate data, while macros expand out into newly-generated C++ code during preprocessing.

To create macros, you use an alternative syntax for `#define` that specifies a parameter list in addition to the constant name and expansion. The syntax looks like this:

```
#define macroname(parameter1, parameter2, ... , parameterN) macro-body*
```

Now, when the preprocessor encounters a call to a function named ***macroname***, it will replace it with the text in ***macro-body***. For example, consider the following macro definition:

```
#define PLUS_ONE(x) ((x) + 1)
```

Now, if we write

```
int x = PLUS_ONE(137);
```

The preprocessor will expand this code out to

```
int x = ((137) + 1);
```

So `x` will have the value 138.

If you'll notice, unlike C++ functions, preprocessor macros do not have a return value. Macros expand out into C++ code, so the "return value" of a macro is the result of the expressions it creates. In the case of `PLUS_ONE`, this is the value of the parameter plus one because the replacement is interpreted as a

* Note that when using `#define`, the opening parenthesis that starts the argument list must not be preceded by whitespace. Otherwise, the preprocessor will treat it as part of the replacement phrase for a `#define`d constant.

mathematical expression. However, macros need not act like C++ functions. Consider, for example, the following macro:

```
#define MAKE_FUNCTION(fnName) void fnName()
```

Now, if we write the following C++ code:

```
MAKE_FUNCTION(MyFunction) {
    cout << "This is a function!" << endl;
}
```

The `MAKE_FUNCTION` macro will convert it into the function definition

```
void MyFunction() {
    cout << "This is a function!" << endl;
}
```

As you can see, this is entirely different than the `PLUS_ONE` macro demonstrated above. In general, a macro can be expanded out to any text and that text will be treated as though it were part of the original C++ source file. This is a mixed blessing. In many cases, as you'll see later in the chapter, it can be exceptionally useful. However, as with other uses of `#define`, macros can lead to incredibly subtle bugs that can be difficult to track down. Perhaps the most famous example of macros gone wrong is this `MAX` macro:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Here, the macro takes in two parameters and uses the `?:` operator to choose the larger of the two. If you're not familiar with the `?:` operator, the syntax is as follows:

```
expression ? result-if-true : result-if-false
```

In our case, `((a) > (b) ? (a) : (b))` evaluates the expression `(a) > (b)`. If the statement is true, the value of the expression is `(a)`; otherwise it is `(b)`.

At first, this macro might seem innocuous and in fact will work in almost every situation. For example:

```
int x = MAX(100, 200);
```

Expands out to

```
int x = ((100) > (200) ? (100) : (200));
```

Which assigns `x` the value `200`. However, what happens if we write the following?

```
int x = MAX(MyFn1(), MyFn2());
```

This expands out to

```
int x = ((MyFn1()) > (MyFn2()) ? (MyFn1()) : (MyFn2()));
```

While this will assign `x` the larger of `MyFn1()` and `MyFn2()`, it will not evaluate the parameters only once, as you would expect of a regular C++ function. As you can see from the expansion of the `MAX` macro, the functions will be called once during the comparison and possibly twice in the second half of the statement.

If `MyFn1()` or `MyFn2()` are slow, this is inefficient, and if either of the two have side effects (for example, writing to disk or changing a global variable), the code will be incorrect.

The above example with `MAX` illustrates an important point when working with the preprocessor – in general, C++ functions are safer, less error-prone, and more readable than preprocessor macros. If you ever find yourself wanting to write a macro, see if you can accomplish the task at hand with a regular C++ function. If you can, use the C++ function instead of the macro – you'll save yourself hours of debugging nightmares.

Inline Functions

One of the motivations behind macros in pure C was program efficiency from *inlining*. For example, consider the `MAX` macro from earlier, which was defined as

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

If we call this macro, then the code for selecting the maximum element is directly inserted at the spot where the macro is used. For example, the following code:

```
int myInt = MAX(one, two);
```

Expands out to

```
int myInt = ((one) > (two) ? (one) : (two));
```

When the compiler sees this code, it will generate machine code that directly performs the test. If we had instead written `MAX` as a regular function, the compiler would probably implement the call to `MAX` as follows:

1. Call the function called `MAX` (which actually performs the comparison)
2. Store the result in the variable `myInt`.

This is considerably less efficient than the macro because of the time required to set up the function call. In computer science jargon, the macro is *inlined* because the compiler places the contents of the "function" at the call site instead of inserting an indirect jump to the code for the function. Inlined functions can be considerably more efficient that their non-inline counterparts, and so for many years macros were the preferred means for writing utility routines.

Bjarne Stroustrup is particularly opposed to the preprocessor because of its idiosyncrasies and potential for errors, and to entice programmers to use safer language features developed the `inline` keyword, which can be applied to functions to suggest that the compiler automatically inline them. Inline functions are not treated like macros – they're actual functions and none of the edge cases of macros apply to them – but the compiler will try to safely inline them if at all possible. For example, the following `Max` function is marked `inline`, so a reasonably good compiler should perform the same optimization on the `Max` function that it would on the `MAX` macro:

```
inline int Max(int one, int two) {
    return one > two ? one : two;
}
```

The `inline` keyword is only a suggestion to the compiler and may be ignored if the compiler deems it either too difficult or too costly to inline the function. However, when writing short functions it sometimes helps to mark the function `inline` to improve performance.

A `#define` Cautionary Tale

`#define` is a powerful directive that enables you to completely transform C++. However, many C/C++ experts agree that you should not use `#define` unless it is absolutely necessary. Preprocessor macros and constants obfuscate code and make it harder to debug, and with a few cryptic `#define`s veteran C++ programmers will be at a loss to understand your programs. As an example, consider the following code, which references an external file `mydefines.h`:

```
#include "mydefines.h"

Once upon a time a little boy took a walk in a park
He (the child) found a small stone and threw it (the stone) in a pond
The end
```

Surprisingly, and worryingly, it is possible to make this code compile and run, provided that `mydefines.h` contains the proper `#define`s. For example, here's one possible `mydefines.h` file that makes the code compile:

File: `mydefines.h`

```
#ifndef mydefines_included
#define mydefines_included

#include <iostream>
using namespace std;

#define Once
#define upon
#define a
#define time upon
#define little
#define boy
#define took upon
#define walk
#define in walk
#define the
#define park a
#define He(n) n MyFunction(n x)
#define child int
#define found {
#define small return
#define stone x;
#define and in
#define threw }
#define it(n) int main() {
#define pond cout << MyFunction(137) << endl;
#define end return 0; }
#define The the

#endif
```

After preprocessing (and some whitespace formatting), this yields the program

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

int MyFunction(int x) {
    return x;
}

int main() {
    cout << MyFunction(137) << endl;
    return 0;
}
```

While this example is admittedly a degenerate case, it should indicate exactly how disastrous it can be for your programs to misuse `#define`d symbols. Programmers expect certain structures when reading C++ code, and by obscuring those structures behind walls of `#define`s you will confuse people who have to read your code. Worse, if you step away from your code for a short time (say, a week or a month), you may very well return to it with absolutely no idea how your code operates. Consequently, when working with `#define`, always be sure to ask yourself whether or not you are improving the readability of your code.

Advanced Preprocessor Techniques

The previous section ended on a rather grim note, giving an example of preprocessor usage gone awry. But to entirely eschew the preprocessor in favor of other language features would also be an error. In several circumstances, the preprocessor can perform tasks that other C++ language features cannot accomplish. The remainder of this chapter explores where the preprocessor can be an invaluable tool for solving problems and points out several strengths and weaknesses of preprocessor-based approaches.

Special Preprocessor Values

The preprocessor has access to several special values that contain information about the state of the file currently being compiled. The values act like `#define`d constants in that they are replaced whenever encountered in a program. For example, the values `__DATE__` and `__TIME__` contain string representations of the date and time that the program was compiled. Thus, you can write an automatically-generated "about this program" function using syntax similar to this:

```
string GetAboutInformation() {
    stringstream result;
    result << "This program was compiled on " << __DATE__;
    result << " at time " << __TIME__;
    return result.str();
}
```

Similarly, there are two other values, `__LINE__` and `__FILE__`, which contain the current line number and the name of the file being compiled. We'll see an example of where `__LINE__` and `__FILE__` can be useful later in this chapter.

String Manipulation Functions

While often dangerous, there are times where macros can be more powerful or more useful than regular C++ functions. Since macros work with source-level text strings instead of at the C++ language level, some pieces of information are available to macros that are not accessible using other C++ techniques. For example, let's return to the `MAX` macro we used in the previous chapter:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

Here, the arguments `a` and `b` to `MAX` are passed by *string* – that is, the arguments are passed as the strings that compose them. For example, `MAX(10, 15)` passes in the value `10` not as a numeric value ten, but as the character `1` followed by the character `0`. The preprocessor provides two different operators for manipulating the strings passed in as parameters. First is the *stringizing operator*, represented by the `#` symbol, which returns a quoted, C string representation of the parameter. For example, consider the following macro:

```
#define PRINTOUT(n) cout << #n << " has value  " << (n) << endl
```

Here, we take in a single parameter, `n`. We then use the stringizing operator to print out a string representation of `n`, followed by the value of the expression `n`. For example, given the following code snippet:

```
int x = 137;
PRINTOUT(x * 42);
```

After preprocessing, this yields the C++ code

```
int x = 137;
cout << "x * 42" << " has value " << (x * 42) << endl;
```

Note that after the above program has been compiled from C++ to machine code, any notions of the original variable `x` or the individual expressions making up the program will have been completely eliminated, since variables exist only at the C++ level. However, through the stringizing operator, it is possible to preserve a string version of portions of the C++ source code in the final program, as demonstrated above. This is useful when writing diagnostic functions, as you'll see later in this chapter.

The second preprocessor string manipulation operator is the *string concatenation* operator, also known as the *token-pasting* operator. This operator, represented by `##`, takes the string value of a parameter and concatenates it with another string. For example, consider the following macro:

```
#define DECLARE_MY_VAR(type) type my_##type
```

The purpose of this macro is to allow the user to specify a type (for example, `int`), and to automatically generate a variable declaration of that type whose name is `my_`***`type`***, where ***`type`*** is the parameter type. Here, we use the `##` operator to take the name of the type and concatenate it with the string `my_`. Thus, given the following macro call:

```
DECLARE_MY_VAR(int);
```

The preprocessor would replace it with the code

```
int my_int;
```

Note that when working with the token-pasting operator, if the result of the concatenation does not form a single C++ token (a valid operator or name), the behavior is undefined. For example, calling `DECLARE_MY_VAR(const int)` will have undefined behavior, since concatenating the strings `my_` and `const int` does not yield a single string (the result is `const int my_const int`).

Advanced Preprocessor Techniques: The X Macro Trick

Because the preprocessor gives C++ programs access to their own source code at compile-time, it is possible to harness the preprocessor to do substantial code generation at compile-time. One uncommon

programming technique that uses the preprocessor is known as the *X Macro trick*, a way to specify data in one format but have it available in several formats.

Before exploring the X Macro trick, we need to cover how to redefine a macro after it has been declared. Just as you can define a macro by using `#define`, you can also undefine a macro using `#undef`. The `#undef` preprocessor directive takes in a symbol that has been previously `#define`d and causes the preprocessor to ignore the earlier definition. If the symbol was not already defined, the `#undef` directive has no effect but is not an error. For example, consider the following code snippet:

```
#define MY_INT 137
int x = MY_INT;   // MY_INT is replaced
#undef MY_INT;
int MY_INT = 42;  // MY_INT not replaced
```

The preprocessor will rewrite this code as

```
int x = 137;
int MY_INT = 42;
```

Although `MY_INT` was once a `#define`d constant, after encountering the `#undef` statement, the preprocessor stopped treating it as such. Thus, when encountering `int MY_INT = 42`, the preprocessor made no replacements and the code compiled as written.

To introduce the X Macro trick, let's consider a common programming problem and see how we should go about solving it. Suppose that we want to write a function that, given as an argument an enumerated type, returns the string representation of the enumerated value. For example, given the `enum`

```
enum Color {Red, Green, Blue, Cyan, Magenta, Yellow};
```

We want to write a function called `ColorToString` that returns a string representation of the color. For example, passing in the constant `Red` should hand back the string `"Red"`, `Blue` should yield `"Blue"`, etc. Since the names of enumerated types are lost during compilation, we would normally implement this function using code similar to the following:

```
string ColorToString(Color c) {
    switch(c) {
        case Red: return "Red";
        case Blue: return "Blue";
        case Green: return "Green";
        case Cyan: return "Cyan";
        case Magenta: return "Magenta";
        case Yellow: return "Yellow";
        default: return "<unknown>";
    }
}
```

Now, suppose that we want to write a function that, given a color, returns the opposite color.* We'd need another function, like this one:

* For the purposes of this example, we'll work with additive colors. Thus red is the opposite of cyan, yellow is the opposite of blue, etc.

```
Color GetOppositeColor(Color c) {
    switch(c) {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result
    }
}
```

These two functions will work correctly, and there's nothing functionally wrong with them as written. The problem, though, is that these functions are not *scalable*. If we want to introduce new colors, say, `White` and `Black`, we'd need to change both `ColorToString` and `GetOppositeColor` to incorporate these new colors. If we accidentally forget to change one of the functions, the compiler will give no warning that something is missing and we will only notice problems during debugging. The problem is that a color encapsulates more information than can be expressed in an enumerated type. Colors also have names and opposites, but the C++ `enum Color` knows only a unique ID for each color and relies on correct implementations of `ColorToString` and `GetOppositeColor` for the other two. Somehow, we'd like to be able to group all of this information into one place. While we might be able to accomplish this using a set of C++ `struct` constants (e.g. defining a color `struct` and making `const` instances of these `struct`s for each color), this approach can be bulky and tedious. Instead, we'll choose a different approach by using X Macros.

The idea behind X Macros is that we can store all of the information needed above inside of calls to preprocessor macros. In the case of a color, we need to store a color's name and opposite. Thus, let's suppose that we have some macro called `DEFINE_COLOR` that takes in two parameters corresponding to the name and opposite color. We next create a new file, which we'll call `color.h`, and fill it with calls to this `DEFINE_COLOR` macro that express all of the colors we know (let's ignore the fact that we haven't actually defined `DEFINE_COLOR` yet; we'll get there in a moment). This file looks like this:

File: `color.h`

```
DEFINE_COLOR(Red, Cyan)
DEFINE_COLOR(Cyan, Red)
DEFINE_COLOR(Green, Magenta)
DEFINE_COLOR(Magenta, Green)
DEFINE_COLOR(Blue, Yellow)
DEFINE_COLOR(Yellow, Blue)
```

Two things about this file should jump out at you. First, we haven't surrounded the file in the traditional `#ifndef ... #endif` boilerplate, so clients can `#include` this file multiple times. Second, we haven't provided an implementation for `DEFINE_COLOR`, so if a caller *does* include this file, it will cause a compile-time error. For now, don't worry about these problems – you'll see why we've structured the file this way in a moment.

Let's see how we can use the X Macro trick to rewrite `GetOppositeColor`, which for convenience is reprinted below:

```
Color GetOppositeColor(Color c) {
    switch(c) {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result
    }
}
```

Here, each one of the `case` labels in this switch statement is written as something of the form

```
case color: return opposite;
```

Looking back at our `color.h` file, notice that each `DEFINE_COLOR` macro has the form `DEFINE_COLOR(color, opposite)`. This suggests that we could somehow convert each of these `DEFINE_COLOR` statements into `case` labels by crafting the proper `#define`. In our case, we'd want the `#define` to make the first parameter the argument of the `case` label and the second parameter the return value. We can thus write this `#define` as

```
#define DEFINE_COLOR(color, opposite) case color: return opposite;
```

Thus, we can rewrite `GetOppositeColor` using X Macros as

```
Color GetOppositeColor(Color c) {
    switch(c) {
        #define DEFINE_COLOR(color, opposite) case color: return opposite;
        #include "color.h"
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

This is pretty cryptic, so let's walk through it one step at a time. First, let's simulate the preprocessor by replacing the line `#include "color.h"` with the full contents of `color.h`:

```
Color GetOppositeColor(Color c) {
    switch(c) {
        #define DEFINE_COLOR(color, opposite) case color: return opposite;
        DEFINE_COLOR(Red, Cyan)
        DEFINE_COLOR(Cyan, Red)
        DEFINE_COLOR(Green, Magenta)
        DEFINE_COLOR(Magenta, Green)
        DEFINE_COLOR(Blue, Yellow)
        DEFINE_COLOR(Yellow, Blue)
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

Now, we replace each `DEFINE_COLOR` by instantiating the macro, which yields the following:

```
Color GetOppositeColor(Color c) {
    switch(c) {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        #undef DEFINE_COLOR
        default: return c; // Unknown color, undefined result.
    }
}
```

Finally, we `#undef` the `DEFINE_COLOR` macro, so that the next time we need to provide a definition for `DEFINE_COLOR`, we don't have to worry about conflicts with the existing declaration. Thus, the final code for `GetOppositeColor`, after expanding out the macros, yields

```
Color GetOppositeColor(Color c) {
    switch(c) {
        case Red: return Cyan;
        case Blue: return Yellow;
        case Green: return Magenta;
        case Cyan: return Red;
        case Magenta: return Green;
        case Yellow: return Blue;
        default: return c; // Unknown color, undefined result.
    }
}
```

Which is exactly what we wanted.

The fundamental idea underlying the X Macros trick is that all of the information we can possibly need about a color is contained inside of the file `color.h`. To make that information available to the outside world, we embed all of this information into calls to some macro whose name and parameters are known. We do not, however, provide an implementation of this macro inside of `color.h` because we cannot anticipate every possible use of the information contained in this file. Instead, we expect that if another part of the code wants to use the information, it will provide its own implementation of the `DEFINE_COLOR` macro that extracts and formats the information. The basic idiom for accessing the information from these macros looks like this:

```
#define macroname(arguments) /* some use for the arguments */
#include "filename"
#undef macroname
```

Here, the first line defines the mechanism we will use to extract the data from the macros. The second includes the file containing the macros, which supplies the macro the data it needs to operate. The final step clears the macro so that the information is available to other callers. If you'll notice, the above technique for implementing `GetOppositeColor` follows this pattern precisely.

We can also use the above pattern to rewrite the `ColorToString` function. Note that inside of `ColorToString`, while we can ignore the second parameter to `DEFINE_COLOR`, the macro we define to extract the information still needs to have two parameters. To see how to implement `ColorToString`, let's first revisit our original implementation:

```
string ColorToString(Color c) {
    switch(c) {
        case Red: return "Red";
        case Blue: return "Blue";
        case Green: return "Green";
        case Cyan: return "Cyan";
        case Magenta: return "Magenta";
        case Yellow: return "Yellow";
        default: return "<unknown>";
    }
}
```

If you'll notice, each of the `case` labels is written as

```
case color: return "color";
```

Thus, using X Macros, we can write `ColorToString` as

```
string ColorToString(Color c) {
    switch(c) {
        /* Convert something of the form DEFINE_COLOR(color, opposite)
         * into something of the form 'case color: return "color"';
         */
        #define DEFINE_COLOR(color, opposite) case color: return #color;
        #include "color.h"
        #undef DEFINE_COLOR

        default: return "<unknown>";
    }
}
```

In this particular implementation of `DEFINE_COLOR`, we use the stringizing operator to convert the `color` parameter into a string for the return value. We've used the preprocessor to generate both `GetOppositeColor` and `ColorToString`!

There is one final step we need to take, and that's to rewrite the initial `enum Color` using the X Macro trick. Otherwise, if we make any changes to `color.h`, perhaps renaming a color or introducing new colors, the `enum` will not reflect these changes and might result in compile-time errors. Let's revisit `enum Color`, which is reprinted below:

```
enum Color {Red, Green, Blue, Cyan, Magenta, Yellow};
```

While in the previous examples of `ColorToString` and `GetOppositeColor` there was a reasonably obvious mapping between `DEFINE_COLOR` macros and `case` statements, it is less obvious how to generate this `enum` using the X Macro trick. However, if we rewrite this `enum` as follows:

```
enum Color {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow
};
```

It should be slightly easier to see how to write this `enum` in terms of X Macros. For each `DEFINE_COLOR` macro we provide, we'll simply extract the first parameter (the color name) and append a comma. In code, this looks like

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
};
```

This, in turn, expands out to

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color,
    DEFINE_COLOR(Red, Cyan)
    DEFINE_COLOR(Cyan, Red)
    DEFINE_COLOR(Green, Magenta)
    DEFINE_COLOR(Magenta, Green)
    DEFINE_COLOR(Blue, Yellow)
    DEFINE_COLOR(Yellow, Blue)
    #undef DEFINE_COLOR
};
```

Which in turn becomes

```
enum Color {
    Red,
    Green,
    Blue,
    Cyan,
    Magenta,
    Yellow,
};
```

Which is exactly what we want. You may have noticed that there is a trailing comma at after the final color (`Yellow`), but this is not a problem – it turns out that it's totally legal C++ code.

Analysis of the X Macro Trick

The X Macro-generated functions have several advantages over the hand-written versions. First, the X macro trick makes the code considerably shorter. By relying on the preprocessor to perform the necessary expansions, we can express all of the necessary information for an object inside of an X Macro file and only need to write the syntax necessary to perform some task once. Second, and more importantly, this approach means that adding or removing `Color` values is simple. We simply need to add another `DEFINE_COLOR` definition to `color.h` and the changes will automatically appear in all of the relevant functions. Finally, if we need to incorporate more information into the `Color` object, we can store that information in one location and let any callers that need it access it without accidentally leaving one out.

That said, X Macros are not a perfect technique. The syntax is considerably trickier and denser than in the original implementation, and it's less clear to an outside reader how the code works. Remember that readable code is just as important as correct code, and make sure that you've considered all of your options before settling on X Macros. If you're ever working in a group and plan on using the X Macro trick,

be sure that your other group members are up to speed on the technique and get their approval before using it.*

More to Explore / Practice Problems

I've combined the "More to Explore" and "Practice Problems" sections because many of the topics we didn't cover in great detail in this chapter are best understood by playing around with the material. Here's a sampling of different preprocessor tricks and techniques, mixed in with some programming puzzles:

1. List three major differences between `#define` and the `const` keyword for defining named constants.

2. Give an example, besides preventing problems from `#include`-ing the same file twice, where `#ifdef` and `#ifndef` might be useful. *(Hint: What if you're working on a project that must run on Windows, Mac OS X, and Linux and want to use platform-specific features of each?)*

3. Write a regular C++ function called `Max` that returns the larger of two `int` values. Explain why it does not have the same problems as the macro `MAX` covered earlier in this chapter.

4. Give one advantage of the macro `MAX` over the function `Max` you wrote in the previous problem. *(Hint: What is the value of `Max(1.37, 1.24)`? What is the value of `MAX(1.37, 1.24)`?)*

5. The following C++ code is illegal because the `#if` directive cannot call functions:

```
bool IsPositive(int x) {
    return x < 0;
}

#if IsPositive(MY_CONSTANT) // <-- Error occurs here
    #define result true
#else
    #define result false
#endif
```

   Given your knowledge of how the preprocessor works, explain why this restriction exists. ♦

6. Compilers rarely inline recursive functions, even if they are explicitly marked `inline`. Why do you think this is?

7. Most of the STL algorithms are inlined. Considering the complexity of the implementation of `accumulate` from the chapter on STL algorithms, explain why this is.

8. Modify the earlier definition of `enum Color` such that after all of the colors defined in `color.h`, there is a special value, `NOT_A_COLOR`, that specifies a nonexistent color. *(Hint: Do you actually need to change `color.h` to do this?)* ♦

* The X Macro trick is a special case of a more general technique known as *preprocessor metaprogramming.* If you're interested in learning more about preprocessor metaprogramming, consider looking into the Boost Metaprogramming Library (MPL), a professional C++ package that simplifies common metaprogramming tasks.

9. Using X Macros, write a function `StringToColor` which takes as a parameter a `string` and returns the `Color` object whose name *exactly* matches the input string. If there are no colors with that name, return `NOT_A_COLOR` as a sentinel. For example, calling `StringToColor("Green")` would return the value `Green`, but calling `StringToColor("green")` or `StringToColor("Olive")` should both return `NOT_A_COLOR`.

10. Suppose that you want to make sure that the enumerated values you've made for `Color` do not conflict with other enumerated types that might be introduced into your program. Modify the earlier definition of `DEFINE_COLOR` used to define `enum Color` so that all of the colors are prefaced with the identifier `eColor_`. For example, the old value `Red` should change to `eColor_Red`, the old `Blue` would be `eColor_Blue`, etc. Do not change the contents of `color.h`. *(Hint: Use one of the preprocessor string-manipulation operators)* ♦

11. The `#error` directive causes a compile-time error if the preprocessor encounters it. This may sound strange at first, but is an excellent way for detecting problems during preprocessing that might snowball into larger problems later in the code. For example, if code uses compiler-specific features (such as the OpenMP library), it might add a check to see that a compiler-specific `#define` is in place, using `#error` to report an error if it isn't. The syntax for `#error` is `#error` ***message***, where ***message*** is a message to the user explaining the problem. Modify `color.h` so that if a caller `#include`s the file without first `#define`-ing the `DEFINE_COLOR` macro, the preprocessor reports an error containing a message about how to use the file.

12. If you're up for a challenge, consider the following problem. Below is a table summarizing various units of length:

Unit Name	**#meters / unit**	**Suffix**	**System**
Meter	1.0	m	Metric
Centimeter	0.01	cm	Metric
Kilometer	1000.0	km	Metric
Foot	0.3048	ft	English
Inch	0.0254	in	English
Mile	1609.344	mi	English
Astronomical Unit	1.496×10^{11}	AU	Astronomical
Light Year	9.461×10^{15}	ly	Astronomical
Cubit*	0.55	cubit	Archaic

a) Create a file called `units.h` that uses the X macro trick to encode the above table as calls to a macro `DEFINE_UNIT`. For example, one entry might be `DEFINE_UNIT(Meter, 1.0, m, Metric)`.
b) Create an enumerated type, `LengthUnit`, which uses the suffix of the unit, preceded by `eLengthUnit_`, as the name for the unit. For example, a cubit is `eLengthUnit_cubit`, while a mile would be `eLengthUnit_mi`. Also define an enumerated value `eLengthUnit_ERROR` that serves as a sentinel indicating that the value is invalid.
c) Write a function called `SuffixStringToLengthUnit` that accepts a `string` representation of a suffix and returns the `LengthUnit` corresponding to that string. If the `string` does not match the suffix, return `eLengthUnit_ERROR`.
d) Create a `struct`, `Length`, that stores a `double` and a `LengthUnit`. Write a function `ReadLength` that prompts the user for a `double` and a `string` representing an amount and a unit suffix and stores data in a `Length`. If the string does not correspond to a suffix, reprompt the user. You can modify the code for `GetInteger` from the chapter on streams to make an implementation of `GetReal`.
e) Create a function, `GetUnitType`, that takes in a `Length` and returns the unit system in which it occurs (as a `string`)
f) Create a function, `PrintLength`, that prints out a `Length` in the format ***`amount suffix`*** (***`amount unitname`***`s`). For example, if a `Length` stores 104.2 miles, it would print out `104.2mi (104.2 Miles)`
g) Create a function, `ConvertToMeters`, which takes in a `Length` and converts it to an equivalent length in meters.

Surprisingly, this problem is not particularly long – the main challenge is the user input, not the unit management!

* There is no agreed-upon standard for this unit, so this is an approximate average of the various lengths.

Chapter 5: STL Sequence Containers

> *In October of 1976 I observed that a certain algorithm – parallel reduction – was associated with monoids: collections of elements with an associative operation. That observation led me to believe that it is possible to associate every useful algorithm with a mathematical theory and that such association allows for both widest possible use and meaningful taxonomy. As mathematicians learned to lift theorems into their most general settings, so I wanted to lift algorithms and data structures.*
>
> – Alex Stepanov, inventor of the STL. [Ste07]

The Standard Template Library (STL) is a programmer's dream. It offers efficient ways to store, access, manipulate, and view data and is designed for maximum extensibility. Once you've gotten over the initial syntax hurdles, you will quickly learn to appreciate the STL's sheer power and flexibility.

To give a sense of exactly where we're going, here are a few quick examples of code using the STL:

- We can create a list of random numbers, sort it, and print it to the console *in four lines of code!*

```
vector<int> myVector(NUM_INTS);
generate(myVector.begin(), myVector.end(), rand);
sort(myVector.begin(), myVector.end());
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, "\n"));
```

- We can open a file and print its contents *in two lines of code!*

```
ifstream input("my-file.txt");
copy(istreambuf_iterator<char>(input), istreambuf_iterator<char>(),
     ostreambuf_iterator<char>(cout));
```

- We can convert a string to upper case *in one line of code!*

```
transform(s.begin(), s.end(), s.begin(), ::toupper);
```

If you aren't already impressed by the possibilities this library entails, keep reading. You will not be disappointed.

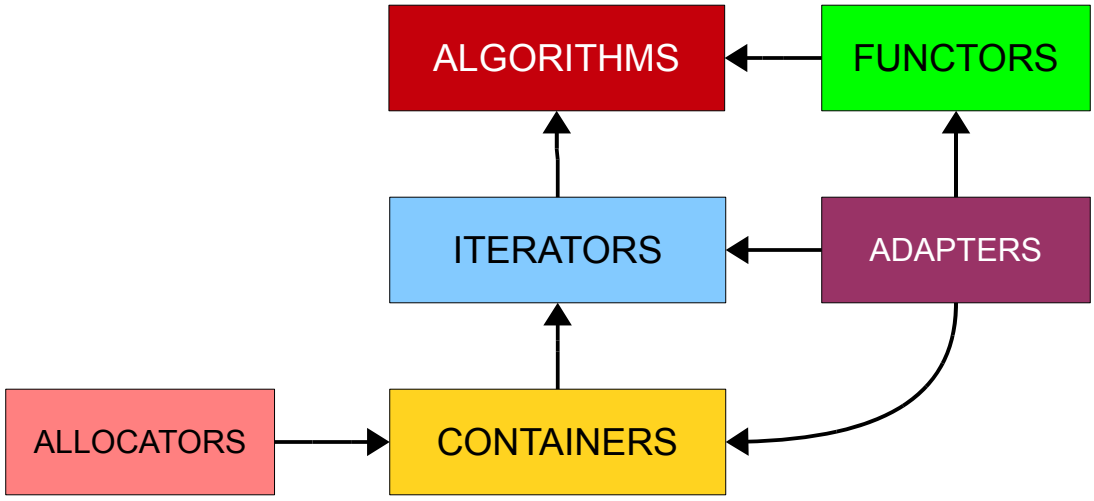
Overview of the STL

The STL is logically divided into six pieces, each consisting of generic components that interoperate with the rest of the library:

- **Containers**. At the heart of the STL are a collection of container classes, standard C++'s analog to the CS106B/X ADTs. For example, you can store an associative collection of key/value pairs in an STL `map`, or a growing list of elements in an STL `vector`.
- **Iterators**. Each STL container exports iterators, objects that view and modify ranges of stored data. Iterators have a common interface, allowing you to write code that operates on data stored in arbitrary containers.
- **Algorithms**. STL algorithms are functions that operate over ranges of data specified by iterators. The scope of the STL algorithms is staggering – there are algorithms for searching, sorting, reordering, permuting, creating, and destroying sets of data.

- **Adapters**. STL *adapters* are objects which transform an object from one form into another. For example, the stack adapter transforms a regular vector or list into a LIFO container, while the `istream_iterator` transforms a standard C++ stream into an STL iterator.
- **Functors**. Because so much of the STL relies on user-defined callback functions, the STL provides facilities for creating and modifying functions at runtime. We will defer our discussion of functors to much later in this text, as they require a fairly nuanced understanding of C++.
- **Allocators**. The STL allows clients of the container classes to customize how memory is allocated and deallocated, either for diagnostic or performance reasons. While allocators are fascinating and certainly worthy of discussion, they are beyond the scope of this text and we will not cover them here.

Diagrammatically, these pieces are related as follows:



Here, the containers rely on the allocators for memory and produce iterators. Iterators can then be used in conjunction with the algorithms. Functors provide special functions for the algorithms, and adapters can produce functors, iterators, and containers. If this seems a bit confusing now, don't worry, you'll understand this relationship well by the time you've finished the next few chapters.

Why the STL is Necessary

Up until this point, all of the programs you've encountered have declared a fixed number of variables that correspond to a fixed number of values. If you declare an `int`, you get a single variable back. If you need to create multiple different `int`s for some reason, you have to declare each of the variables independently. This has its drawbacks. For starters, it means that you need to know how exactly how much data your program will be manipulating before the program begins running. Here's a quick programming challenge for you:

> Write a program that reads in three integers from the user, then prints them in sorted order.

How could you go about writing a program to do this? There are two main technical hurdles to overcome. First, how would we store the numbers the user enters? Second, how do we sort them? Because we know that the user will be entering three distinct values, we could simply store each of them in their own int variable. Thus the first step of writing code for this program might look like this:

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

int GetInteger(); // From the streams chapter

int main() {
    int val1 = GetInteger();
    int val2 = GetInteger();
    int val3 = GetInteger();

    /* Sort, then print out. */
}
```

We now have our three values; how can we sort them? It turns out that sorting a list of integers is a classic algorithms question and there are many elegant solutions to the problem. Some algorithms like *quicksort* and *heapsort* run extremely quickly, but are rather difficult to implement. Instead, we'll use a variant algorithm called *selection sort*. The idea behind selection sort is as follows. If we want to sort a list of numbers, we can find the smallest element in the list, then move it to the front of the list. We can then find the second-smallest value, then put it in the second position, find the third-smallest and put it in the third position, etc. Amazingly, with just the tools we've seen so far this simple algorithm is extremely hard to implement. Here's some code for the operation:

```
int main() {
    int val1 = GetInteger();
    int val2 = GetInteger();
    int val3 = GetInteger();

    /* Three cases: Either val1 is the smallest, val2 is the smallest, or
     * val3 is the smallest.  Whichever ends up being the case, we'll put
     * the smallest value into val1.  This uses the swap() function, which is
     * defined in the <algorithm> header and simply swaps the values of
     * two variables.
     */
    if (val2 <= val1 && val2 <= val3) // val2 is smallest
        swap (val1, val2);
    else if (val3 <= val1 && val3 <= val2) // val3 is smallest
        swap (val1, val3);
    // Otherwise, val1 is smallest, and can remain at the front.

    /* Now, sort val2 and val3.  Since there's just two elements, we can do a
     * simple comparison to determine which is the smaller of the two.
     */
    if (val3 <= val2) // val3 is smaller
       swap (val2, val3);
    // Otherwise, val2 is smallest and we don't need to do anything.

    cout << val1 << ' ' << val2 << ' ' << val3 << endl;
}
```

This code is incredibly dense. Don't panic if you don't understand it – part of the purpose of this example is to illustrate how difficult it can be to write good code for this problem without the STL!

But of course, there's another major problem with this code. Let's modify the problem statement a bit by allowing the user to enter *four* numbers. If we make this change, using just the techniques we've covered so far we'll be forced to write code along the following lines:

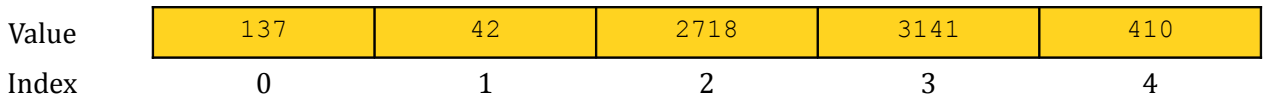
```
    int main() {
        int val1 = GetInteger();
        int val2 = GetInteger();
        int val3 = GetInteger();
        int val4 = GetInteger();

        /* Find the smallest. */
        if (val2 <= val1 && val2 <= val3 && val2 <= val4) // val2 is smallest
            swap (val1, val2);
        else if (val3 <= val1 && val3 <= val2 && val3 <= val4) // val3 is smallest
            swap (val1, val3);
        else if (val4 <= val1 && val4 <= val2 && val4 <= val3) // val4 is smallest
            swap (val1, val4);
        // Otherwise, val1 is smallest, and can remain at the front.

        /* Find the second-smallest. */
        if (val3 <= val2 && val3 <= val4) // val3 is smallest
            swap (val2, val3);
        else if (val4 <= val2 && val4 <= val3) // val4 is smallest
            swap (val2, val4);
        // Otherwise, val2 is smallest, and can remain at the front.

        /* Find the third-smallest. */
        if (val4 <= val3) // val4 is smaller
           swap (val3, val4);
        // Otherwise, val3 is smallest and we don't need to do anything.

        cout << val1 << ' ' << val2 << ' ' << val3 << ' ' << val4 << endl;
}
```

This code is just downright awful! It's cryptic, difficult to read, and not the sort of code you'd like to bring home to mom and dad. Now imagine what the code would look like if we had *five* numbers instead of four. It will keep growing and growing, becoming progressively more impossible to read until eventually we'd give up in frustration. What's worse, though, is that each of these programs is a special case of a general problem – read in *n* integers from the user and print them in sorted order – but the code for each case bears little to no resemblance to the code for each other case.

Introducing the `vector`

What's missing from our programming repertoire right now is the ability to create and access a *variable number* of objects. Right now, if we want to store a sequence of five integers, we must create five independent integer variables and have no way of accessing them uniformly. Fortunately, the STL provides us a versatile tool called the *`vector`* that allows us to store sequences of elements using a single variable. As you'll see, the `vector` is nothing short of extraordinary and will arise time and again throughout your programming career.

At a high level, a `vector` is an object that represents a sequence of elements. This means that you can use a vector to store a grocery list, a list of Olympic figure skating scores, or a set of files to read. The elements in a `vector` are *indexed*, meaning that they have a well-defined position in the sequence. For example, given the sequence



The first element (137) is at position zero, the second (42) at position one, etc. Notice that elements in the sequence appear in the order 0, 1, 2, etc. rather than the more intuitive 1, 2, 3, ... You have seen this already in your exploration of the string class, so hopefully this notation isn't too startling.

The major difference between the vector and the string is that the `vector` can be configured to store elements of *any* type. That is, you can have a `vector` of `int`s, a `vector` of `string`s, or even a `vector` of `vector`s of `string`s. However, while the `vector` can store elements of any type, any single `vector` can only store elements of a single type. It is illegal to create a `vector` that stores a sequence of many different types of elements, meaning that you can't represent the list 0, Apple, 2.71828 because each of the list elements has a different type. This may seem like a rather arbitrary restriction – theoretically speaking, there's no reason that you shouldn't be able to have lists of all sorts of elements – but fortunately in most cases this restriction does not pose too much of a problem.

Because vectors can only store elements of a fixed type, when you create a `vector` in your programs you will need to explicitly indicate to the compiler what type of elements you aim to store in the `vector`. As an example, to create a `vector` of `int`s, you would write

```
vector<int> myVector;
```

Here, we declare a local variable called `myVector` that has type `vector<int>`. The type inside of the angle brackets is called a *template argument* and indicates to C++ what type of elements are stored in the `vector`. Here, the type is `int`, but it's legal to put pretty much whatever type you would like in the brackets. For example, all of the following declarations are legal:

```
vector<int>    intVector;  // Stores ints
vector<string> strVector;  // Stores strings
vector<double> realVector; // Stores real numbers
```

It is also perfectly legal to store your own custom structs in a `vector`, as seen here:

```
struct MyStruct {
    int myInt;
    double myDouble;
    string myString;
};

vector<MyStruct> myStructVector; // Stores MyStructs
```

In order to use the `vector` type, you will need to `#include <vector>` at the top of your program. As we explore some of the other STL containers, this pattern will also apply.

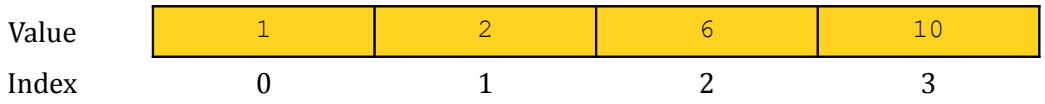
We now know how to *create* `vector`s, but how do we *use* them? To give you a feel for how the `vector` works, let's return to our previous example of reading in numbers and printing them out in sorted order. Using a `vector`, this can be accomplished very elegantly. We'll begin by defining a constant, `kNumValues`, which will represent the number of elements to read in. This is shown here:

```
#include <iostream>
#include <vector>   // Necessary to use vector
#include <string>
#include <sstream>
using namespace std;

string GetLine(); // As defined in the previous chapter
int GetInteger(); // As defined in the previous chapter

const int kNumValues = 10;

int main() {
    /* ... still more work to come ... */
}
```

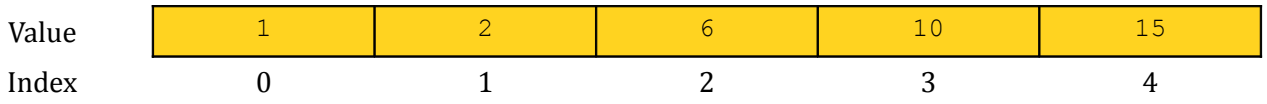
Now, we'll show to how to read in `kNumValues` values from the user and store them inside a `vector`. First, we'll have to create the `vector`, as shown here:

```
int main() {
    vector<int> values;

    /* ... */
}
```

A freshly-constructed `vector`, like a freshly-constructed `string`, is initially empty. Consequently, we'll need to get some values from the user, then store them inside the `vector`. Reading the values is fairly easy; we simply sit in a `for` loop reading data from the user. But how do we store them in the `vector`? There are several ways to do this, of which the simplest is the `push_back` function. The `push_back` function can be invoked on a `vector` to add a new element to the end of `vector`'s sequence. For example, given a `vector` managing the following sequence:



Then calling push_back on the vector to store the value fifteen would cause the vector to manage the new sequence



Syntactically, `push_back` can be used as follows:

```
int main() {
    vector<int> values;

    for (int i = 0; i < kNumValues; ++i) {
        cout << "Enter another value: ";
        int val = GetInteger();

        values.push_back(val);
    }
}
```

Notice that we write `values.push_back(val)` to append the number `val` to the sequence stored inside the `vector`.

Now that we have read in our sequence of values, let's work on the next part of the program, sorting the elements in the `vector` and printing them to the user. For this we'll still use the selection sort algorithm, but it will be miraculously easier to follow when implemented on top of the `vector`. Recall that the general description of the selection sort algorithm is as follows:

- Find the smallest element of the list.
- Put that element at the front of the list.
- Repeat until all elements are in place.

Let's see how to implement this function. We'll begin simply by writing out the function signature, which looks like this:

```
void SelectionSort(vector<int>& v) {
    /* ... */
}
```

This function is named `SelectionSort` and takes as a parameter a `vector<int>` by reference. There are two key points to note here. First, we still have to explicitly indicate what type of elements are stored in the `vector` parameter. That is, it's illegal to write code of this sort:

```
void SelectionSort(vector& v) { // Error: What kind of vector?
    /* ... */
}
```

As a general rule, you will *never* see `vector` unless it's immediately followed with a type in angle brackets. The reason is simple – every `vector` can only store one type of element, and unless you explicitly indicate what that type is the compiler will have no way of knowing.

The other important detail about this function is that it takes its parameter by reference. We will be sorting the `vector` in-place, meaning that we will be reordering the elements of the `vector` rather than creating a new `vector` containing a sorted copy of the input.

We now have the function prototype set up, so let's get into the meat of the algorithm. To implement selection sort, we'll need to find the smallest element and put it in front, the the second-smallest element and put it in the second position, etc. The code for this is as follows:

```
void SelectionSort(vector<int>& v) {
    for (size_t i = 0; i < v.size(); ++i) {
        size_t smallestIndex = GetSmallestIndex(v, i); // We'll write this
                                                       // function momentarily
        swap (v[i], v[smallestIndex]);
    }
}
```

This code is fairly dense and introduces some syntax you have not yet seen before, so let's take a few moments to walk through exactly how it works. The first detail that might have caught your eye is this one:

```
for (size_t i = 0; i < v.size(); ++i)
```

This for loop looks strange for two reasons. First, instead of creating an `int` variable for the iteration, we create a variable of type `size_t`. A `size_t` is a special type of variable that can hold values that represent *sizes* of things (`size_t` stands for "size type"). In many aspects `size_t` is like a regular `int` – it holds an integer value, can be incremented with `++`, compared using the relational operators, etc. However, unlike regular `int`s, `size_t`s cannot store negative values. The intuition behind this idea is that no collection of elements can have a negative size. You may have a list of no elements, or a list of billions of elements, but you'll *never* encounter a list with -1 elements in it. Consequently, when iterating over an STL container, it is customary to use the special type `size_t` to explicitly indicate that your iteration variable should always be nonnegative. The other detail of this for loop is that the loop iterates from 0 to `v.size()`. The `size()` member function on the STL `vector` returns the number of elements stored in the sequence, just like the `size()` and `length()` functions on the standard `string` class. In this particular case we could have iterated from 0 to `kNumValues`, since we're guaranteed that the main function will produce a `vector` of that many elements, but in general when iterating over a vector it's probably a wise idea to use the `vector`'s size as an upper bound so that the code works for vectors of arbitrarily many elements.

Let's continue onward through the code. The body of the loop is this code here:

```
size_t smallestIndex = GetSmallestIndex(v, i);
swap (v[i], v[smallestIndex]);
```

This calls some function called `GetSmallestIndex` (which we have not yet defined) which takes in the vector and the current index. We will implement this function shortly, and its job will be to return the index of the smallest element in the vector occurring no earlier than position `i`. We then use the `swap` function to exchange the values stored in at positions `i` and `smallestIndex` of the `vector`. Notice that, as with the standard string class, the syntax `v[i]` means "the element in `vector` `v` at position `i`." Let's take a minute to think about how this code works. The variable `i` counts up from `0` to `v.size() - 1` and visits every element of the `vector` exactly once. On each iteration, the code finds the smallest element in the `vector` occurring no earlier than position `i`, then exchanges it with the element at position `i`. This means that the code will

1. Find the smallest element of the `vector` and put it in position 0.
2. Find the smallest element of the remainder of the `vector` and put it in position 1.
3. Find the smallest element of the remainder of the `vector` and put it in position 2.

...

This is precisely what we set out to do, and so the code will work marvelously. Of course, this assumes that we have a function called `GetSmallestIndex` which returns the index of the smallest element in the `vector` occurring no earlier than position `i`. To finalize our implementation of `SelectionSort`, let's go implement this function. Again, we'll start with the prototype, which is shown here:

```
size_t GetSmallestIndex(vector<int>& v, size_t startIndex) {
    /* ... */
}
```

This function accepts as input a `vector<int>` by reference and a `size_t` containing the start index, then returns a `size_t` containing the result. There's an important but subtle detail to note here. At a high level, the `GetSmallestIndex` function has no business modifying the `vector` it takes as input. Its task is to find the smallest element and return it, no more. So why exactly does this function take the `vector<int>` parameter by reference? The answer is *efficiency*. In C++, if you pass a parameter to a function by value, then whenever that function is called C++ will make a full copy of the argument. When working with the STL containers, which can contain thousands (if not millions or tens of millions) of elements, the cost of copying the container can be staggering. Consequently, it is considered good

programming practice to pass STL containers into functions by reference rather than by value, since this avoids an expensive copy.*

The implementation of this function is rather straightforward:

```
size_t GetSmallestIndex(vector<int>& v, size_t startIndex) {
    size_t bestIndex = startIndex;

    for (size_t i = startIndex; i < v.size(); ++i)
        if (v[i] < v[bestIndex])
            bestIndex = i;

    return bestIndex;
}
```

This function iterates over the elements of the `vector` starting with position `i`, checking whether the element at the current position is less than the smallest element we've seen so far. If so, the function updates where in the `vector` that element appears. At the end of the function, once we've looked at every element in range, the `bestIndex` variable will hold the index of the smallest element in `vector v` occurring no earlier than `startIndex`, and so we return the value. We've implemented the `GetSmallestIndex` function, meaning that we have a working implementation of `SelectionSort`.

To finalize our program, let's update the main function to print out the sorted vector. This is shown here:

```
int main() {
    vector<int> values;

    for (int i = 0; i < kNumValues; ++i) {
        cout << "Enter another value: ";
        int val = GetInteger();
        values.push_back(val);
    }

    SelectionSort(values);

    for (size_t i = 0; i < kNumValues; ++i)
        cout << values[i] << endl;
}
```

Compare this implementation of the program to the previous version, which did not have the luxury of using `vector`. As you can see, the code in this program is significantly clearer than before. Moreover, it's much more *scalable*. If we want to read in a different number of values from the user, we can do so simply by adjusting the value of the `kNumValues` constant, and the rest of the code will update automatically.

An Alternative Implementation

In the previous section, we wrote a program which reads in some number of values from the user, sorts them, and then prints them out. Of course, the program we wrote was just one method for solving the problem. In particular, there is another implementation strategy we could have considered that lends itself to a substantially shorter implementation. Notice that in the above program, we read in a list of values from the user and blindly added them to the end of the list we wanted to sort. These values weren't necessarily in sorted order, and so we had to run a postprocessing step to sort the vector before displaying

* In practice you would almost certainly pass the parameter by *reference-to-`const`*, which indicates that the parameter cannot be modified. We will take this issue up in a later chapter, but for now pass-by-reference should be good enough for our purposes.

it to the user. But what if we opt for a different approach? In particular, suppose that whenever we read a value from the user, instead of putting that value at the end of the sequence, we find where in the vector the element should go, then insert the value at that point? For example, suppose that the user has entered the following four values:

Value	100	200	300	400
Index	0	1	2	3

Now suppose that she enters the number 137. If we append 137 to the `vector`, then the numbers will not all be in sorted order. Instead, we'll find the location where 137 should go in the sorted `vector`, then insert it directly into that location. This is shown here:

Value	100	137	200	300	400
Index	0	1	2	3	4

This strategy ends up being a bit simpler to implement than our previous program, which relied on selection sort. Of course, to implement the program using the above strategy, we need to answer two questions. First, how do we find out where in the `vector` the user's element should go? Second, how do we insert an element into a `vector` directly at that position? This first question is algorithmic; the second is simply a question of what operations are legal on the `vector`. Consequently, we'll begin our discussion with how to find the insertion point, and will then see how to add an element to a `vector` at a particular point.

Suppose that we are given a sorted list of integers and some value to insert into the list. We are curious to find at what index the new value should go. This is an interesting algorithmic challenge, since there are many valid solutions. However, there's one particular simple way to find where the element should go. In order for a list to be in sorted order, every element has to be smaller than the element that comes one position after it. Therefore, if we find the first element in the `vector` that is *bigger* than the element we want to insert, we know that the element we want to insert must come directly before that element. This suggests the following algorithm:

```
/* Watch out!  This code contains a bug! */
size_t InsertionIndex(vector<int>& v, int toInsert) {
    for(size_t i = 0; i < v.size(); ++i)
        if (toInsert < v[i])
            return i;
}
```

This code is *mostly* correct, but contains a pretty significant flaw. In particular, what happens if the element we want to insert is bigger than every element in the `vector`? In that case, the `if` statement inside of the `for` loop will never evaluate to true, and so the function will not return a value. If a function finishes without returning a value, the program has *undefined behavior*. This means that the function might return zero, it might return garbage, or your program might immediately crash outright. This certainly isn't what we want to have happen, so how can we go about fixing it? Notice that the only way that the above function never returns a value is if the element to be inserted is at least as big as every element in the `vector`. In that case, the correct behavior should be to put the element on the end of the `vector`. We'll signal this by having the function return `v.size()` if the element is bigger than every element in the sequence. This is shown here:

```
size_t InsertionIndex(vector<int>& v, int toInsert) {
    for(size_t i = 0; i < v.size(); ++i)
        if (toInsert < v[i])
            return i;
    return v.size();
}
```

All that's left to do now is use this function to build a sleek implementation of the program. But before we can do that, we have to see how to insert an element into a `vector` at an arbitrary position. The good news is that the `vector` supports this operation naturally, and in fact you can insert an element into a `vector` at any position. The bad news is that the syntax for doing so is nothing short of cryptic and without an understanding of STL iterators will look entirely alien. We'll talk about iterators more next chapter, but in the meantime you can just take it for granted that the following syntax is legal. Given a `vector` `v` and an element `e`, to insert `e` into `v` at position `n`, we use the syntax

```
v.insert(v.begin() + n, e);
```

For example, to insert the element 137 at position zero in the vector, you would write

```
v.insert(v.begin(), 137);
```

Similarly, to insert the element 42 at position five, we could write

```
v.insert(v.begin() + 5, 42);
```

One of the trickier parts of the `insert` function is determining exactly where the element will be inserted. Recall that `vector`s are zero-indexed, the above statement will insert the number 42 as the *sixth* element of the sequence, not the fifth. When an element is inserted at a position, all of the elements after it are shuffled down one spot to make room, so calling insert will never overwrite a value.

Given this syntax and the above implementation of `InsertionIndex`, we can write a program to read in a list of values and print them out in sorted order as follows:

```
int main() {
    vector<int> values;

    /* Read the values. */
    for (int i = 0; i < kNumValues; ++i) {
        cout << "Enter an integer: ";
        int val = GetInteger();

        /* Insert the element at the correct position. */
        values.insert(values.begin() + InsertionIndex(values, val), val);
    }

    /* Print out the sorted list. */
    for (size_t i = 0; i < values.size(); ++i)
        cout << values[i] << endl;
}
```

This code is much shorter than before, even when you factor in the code for `InsertionIndex`.

Additional `vector` Operations

The previous section on sorting numbers with the `vector` showcased many of operations that can be performed on a `vector`, but was by no means a complete survey of what the `vector` can do. While some `vector` operations require a nuanced understanding of *iterators*, which we will cover next chapter, there are a few common operations on the `vector` that we will address in this section before moving on to other container classes.

One of the key distinctions between the `vector` and other data types we've seen so far is that the `vector` has a variable size. It can contain no elements, dozens of elements, or even millions of elements. In the preceding examples, we explored two ways to change the number of elements in the `vector`: `push_back`, which appends new elements to the back of the `vector`, and `insert`, which adds an element to the `vector` at an arbitrary position. However, there are several more ways to add and remove elements from the `vector`, some of which are discussed here.

When creating a new vector to represent a list of values, by default C++ will make the `vector` store an empty list. That is, a newly-created `vector` is by default empty. However, at times you might want to initialize the `vector` to a certain size. C++ allows you to do this by specifying the starting size of the `vector` at the point where the vector is created. For example, to create a `vector` of integers that initially holds fifteen elements, you could write this as

```
vector<int> myVector(15);
```

That is, you declare the `vector` as normal, and put the default size in parentheses afterwards. Note that this only changes the starting size of the `vector`, and you are free to add additional elements to the `vector` later in your program.

When creating a `vector` that holds primitive types, such as `int` or `double`, the elements in the `vector` will default to zero (or `false` in the case of `bool`s). This means that the above line of code means "create a `vector` of integers called `myVector` that initially holds fifteen entries, all zero." Similarly, this line of code:

```
vector<string> myStringVector(10);
```

Will create a `vector` of `string`s that initially holds ten copies of the empty string.

In some cases, you may want to initialize the `vector` to a certain size where each element holds a value other than zero. You may wish, for example, to construct a `vector<string>` holding five copies of the string "`(none)`," or a `vector<double>` holding twenty copies of the value 137. In these cases, C++ lets you specify both the number and default value for the elements in the vector using the following syntax:

```
vector<double> myReals(20, 137.0);
vector<string> myStrings(5, "(none)");
```

Notice that we've enclosed in parentheses both the *number* of starting elements in the vector and the *value* of these starting elements.

An important detail is that this syntax is only legal when initially creating a `vector`. If you have an existing `vector` and try to use this syntax, you will get a compile-time error. That is, the following code is illegal:

```
vector<double> myReals;
myReals(20, 137.0); // Error: Only legal to do this when the object is created
```

If you want to change the number of elements in a `vector` after it has already been created, you can always use the `push_back` and `insert` member functions. However, if you'd like to make an abrupt change in the number of elements in the `vector` (perhaps by adding or deleting a large number of elements all at once), you can use the `vector`'s `resize` member function. The `resize` function is very similar to the syntax we've just encountered: you can specify either a number of elements or a number of elements and a value, and the `vector` will be resized to hold that many elements. However, `resize` behaves somewhat differently from the previous construct because when using `resize`, the `vector` might already contain elements. Consequently, `resize` works by adding or removing elements from the end of the `vector` until the desired size is reached. To get a better feel for how `resize` works, let's suppose that we have a function called `PrintVector` that looks like this:

```
void PrintVector(vector<int>& elems) {
    for (size_t i = 0; i < elems.size(); ++i)
        cout << elems[i] << ' ';
    cout << endl;
}
```

This function takes in a `vector<int>`, then prints out the elements in the `vector` one at a time, followed by a newline. Given this function, consider the following code snippet:

```
vector<int> myVector;     // Defaults to empty vector
PrintVector(myVector);    // Output: [nothing]

myVector.resize(10);      // Grow the vector, setting new elements to 0
PrintVector(myVector);    // Output: 0 0 0 0 0 0 0 0 0 0

myVector.resize(5);       // Shrink the vector
PrintVector(myVector);    // Output: 0 0 0 0 0

myVector.resize(7, 1);    // Grow the vector, setting new elements to 1
PrintVector(myVector);    // Output: 0 0 0 0 0 1 1

myVector.resize(1, 7);    // The second parameter is effectively ignored.
PrintVector(myVector);    // Output: 0
```

In the first line, we construct a new `vector`, which is by default empty. Consequently, the call to `PrintVector` will produce no output. We then invoke `resize` to add ten elements to the `vector`. These elements are added to the end of the `vector`, and because we did not specify a default value are all initialized to zero. On our next call to `resize`, we shrink the `vector` down to five elements. Next, we use `resize` to expand the vector to hold seven elements. Because we specified a default value, the newly-created elements default to 1, and so the sequence is now 0, 0, 0, 0, 0, 1, 1. Finally, we use `resize` to trim the sequence. Because the second argument to `resize` is only considered if new elements are added, it is effectively ignored.

We've seen several vector operations so far, but there is a wide class of operations we have not yet considered – operations which remove elements from the `vector`. As you saw, the `push_back` and `insert` functions can be used to splice new elements into the `vector`'s sequence. These two functions are balanced by the `pop_back` and `erase` functions. `pop_back` is the opposite of `push_back`, and removes the last element from the vector's sequence. `erase` is the deletion counterpart to `insert`, and removes an element at a particular position from the `vector`. As with `insert`, the syntax for `erase` is a bit tricky. To remove a single element from a random point in a `vector`, use the `erase` method as follows:

```
myVector.erase(myVector.begin() + n);
```

where `n` represents the index of the element to erase.

In some cases, you may feel compelled to completely erase the contents of the `vector`. In that case, you can use the `clear` function, which completely erases the `vector` contents. `clear` can be invoked as follows:

```
myVector.clear();
```

Summary of `vector`

The following table lists some of the more common operations that you can perform on a `vector`. We have not talked about `iterator`s or the `const` keyword yet, so don't worry if you're confused by those terms. This table is designed as a reference for any point in your programming career, so feel free to skip over entries that look too intimidating.

Constructor: `vector<T> ()`	`vector<int> myVector;` Constructs an empty vector.
Constructor: `vector<T> (size_type size)`	`vector<int> myVector(10);` Constructs a vector of the specified size where all elements use their default values (for integral types, this is zero).
Constructor: `vector<T> (size_type size, const T& default)`	`vector<string> myVector(5, "blank");` Constructs a vector of the specified size where each element is equal to the specified default value.
`size_type size() const;`	`for(int i = 0; i < myVector.size(); ++i) { ... }` Returns the number of elements in the vector.
`bool empty() const;`	`while(!myVector.empty()) { ... }` Returns whether the vector is empty.
`void clear();`	`myVector.clear();` Erases all the elements in the vector and sets the size to zero.

`T& operator [] (size_type position);`
`const T& operator [] (size_type position) const;`
 	
`T& at(size_type position);`
`const T& at(size_type position) const;`
	`myVector[0] = 100;`
	`int x = myVector[0];`
	`myVector.at(0) = 100;`
	`int x = myVector.at(0);`
	
	Returns a reference to the element at the specified position. The bracket notation [] does not do any bounds checking and has undefined behavior past the end of the data. The `at` member function will throw an exception if you try to access data beyond the end. We will cover exception handling in a later chapter.

`void resize(size_type newSize);`
`void resize(size_type newSize, T fill);`
	`myVector.resize(10);`
	`myVector.resize(10, "default");`
	
	Resizes the vector so that it's guaranteed to be the specified size. In the second version, the `vector` elements are initialized to the value specified by the second parameter. Elements are added to and removed from the end of the vector, so you can't use `resize` to add elements to or remove elements from the start of the vector.

`void push_back();`	`myVector.push_back(100);` Appends an element to the `vector`.

`T& back();`
`const T& back() const;`
	`myVector.back() = 5;`
	`int lastElem = myVector.back();`
	
	Returns a reference to the last element in the `vector`.

`T& front();`
`const T& front() const;`
	`myVector.front() = 0;`
	`int firstElem = myVector.front();`
	
	Returns a reference to the first element in the vector.

`void pop_back();`	`myVector.pop_back();` Removes the last element from the vector.
`iterator begin();` `const_iterator begin() const;`	`vector<int>::iterator itr = myVector.begin();` Returns an iterator that points to the first element in the vector.
`iterator end();` `const_iterator end() const;`	`while(itr != myVector.end());` Returns an iterator to the element *after* the last. The iterator returned by end does not point to an element in the vector.

`iterator insert(iterator position,`
`const T& value);`
`void insert(iterator start,`
`size_type numCopies,`
`const T& value);`
	`myVector.insert(myVector.begin() + 4, "Hello");`
	`myVector.insert(myVector.begin(), 2, "Yo!");`
	
	The first version inserts the specified value into the vector, and the second inserts `numCopies` copies of the value into the vector. Both calls invalidate all outstanding iterators for the vector.

`iterator erase(iterator position);`
`iterator erase(iterator start,`
`iterator end);`
	`myVector.erase(myVector.begin());`
	`myVector.erase(startItr, endItr);`
	
	The first version erases the element at the position pointed to by position. The second version erases all elements in the range [`startItr`, `endItr`). Note that this does **not** erase the element pointed to by `endItr`. All iterators after the remove point are invalidated. If using this member function on a `deque` (see below), all iterators are invalidated.

`deque`: A New Kind of Sequence

For most applications where you need to represent a sequence of elements, the `vector` is an ideal tool. It is fast, lightweight, and intuitive. However, there are several aspects of the vector that can be troublesome in certain applications. In particular, the `vector` is only designed to grow in one direction; calling

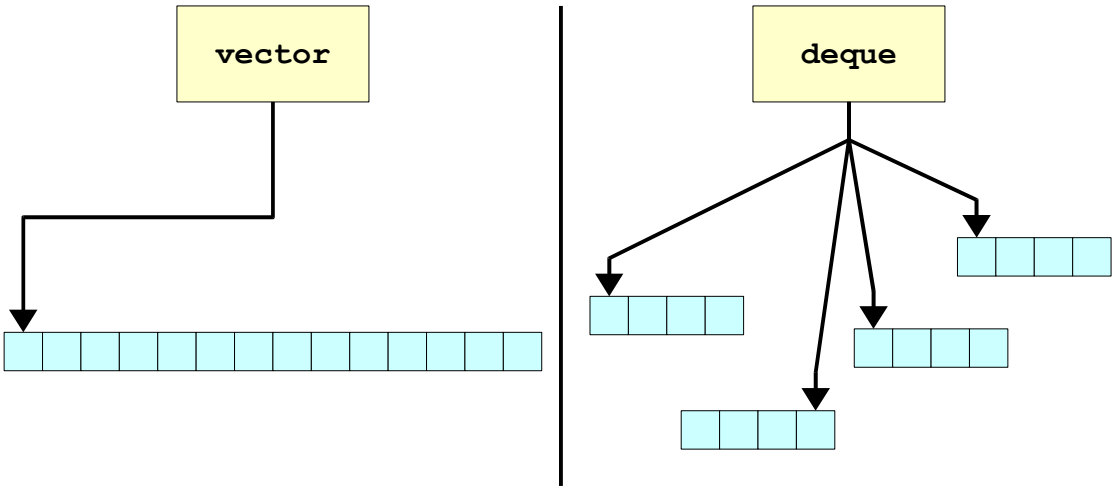
`push_back` inserts elements at the end of the vector, and `resize` always appends elements to the end. While it's possible to insert elements into other positions of the `vector` using the `insert` function, doing so is fairly inefficient and relying on this functionality can cause a marked slowdown in your program's performance. For most applications, this is not a problem, but in some situations you will need to manage a list of elements that will grow and shrink on both ends. Doing this with a `vector` would be prohibitively costly, and we will need to introduce a new container class: the `deque`.

`deque` is a strange entity. It is pronounced "deck," as in a deck of cards, and is named as a contraction of "**d**ouble-**e**nded **que**ue." It is similar to the `vector` in almost every way, but supports a few operations that the `vector` has trouble with. Because of its similarity to `vector`, many C++ programmers don't even know that the `deque` exists. In fact, of all of the standard STL containers, `deque` is probably the least-used. But this is not to say that it is not useful. The `deque` packs significant firepower, and in this next section we'll see some of the basic operations that you can perform on it.

What's interesting about the `deque` is that all operations supported by `vector` are also provided by `deque`. Thus we can `resize` a `deque`, use the bracket syntax to access individual elements, and erase elements at arbitrary positions. In fact, we can rewrite the number-sorting program to use a `deque` simply by replacing all instances of vector with `deque`. For example:

```
int main() {
    deque<int> values; // Use deque instead of vector

    /* Read the values. */
    for (int i = 0; i < kNumValues; ++i)
    {
        cout << "Enter an integer: ";
        int val = GetInteger();

        /* Insert the element at the correct position. */
        values.insert(values.begin() + InsertionIndex(values, val), val);
    }

    /* Print out the sorted list. */
    for (size_t i = 0; i < values.size(); ++i)
        cout << values[i] << endl;
}
```

However, `deque`s also support two more functions, `push_front` and `pop_front`, which work like the `vector`'s `push_back` and `pop_back` except that they insert and remove elements at the front of the `deque`. But this raises an interesting question: if `deque` has strictly more functionality than `vector`, why use `vector`? The main reason is speed. `deque`s and `vector`s are implemented in two different ways. Typically, a `vector` stores its elements in contiguous memory addresses. `deque`s, on the other hand, maintain a list of different "pages" that store information. This is shown here:



These different implementations impact the efficiency of the `vector` and `deque` operations. In a `vector`, because all elements are stored in consecutive locations, it is possible to locate elements through simple arithmetic: to look up the nth element of a `vector`, find the address of the first element in the vector, then jump forward n positions. In a `deque` this lookup is more complex: the `deque` has to figure out which page the element will be stored in, then has to search that page for the proper item. However, inserting elements at the front of a `vector` requires the `vector` to shuffle all existing elements down to make room for the new element (slow), while doing the same in the `deque` only requires the `deque` to rearrange elements in a single page (fast).

If you're debating about whether to use a `vector` or a `deque` in a particular application, you might appreciate this advice from the C++ ISO Standard (section 23.1.1.2):

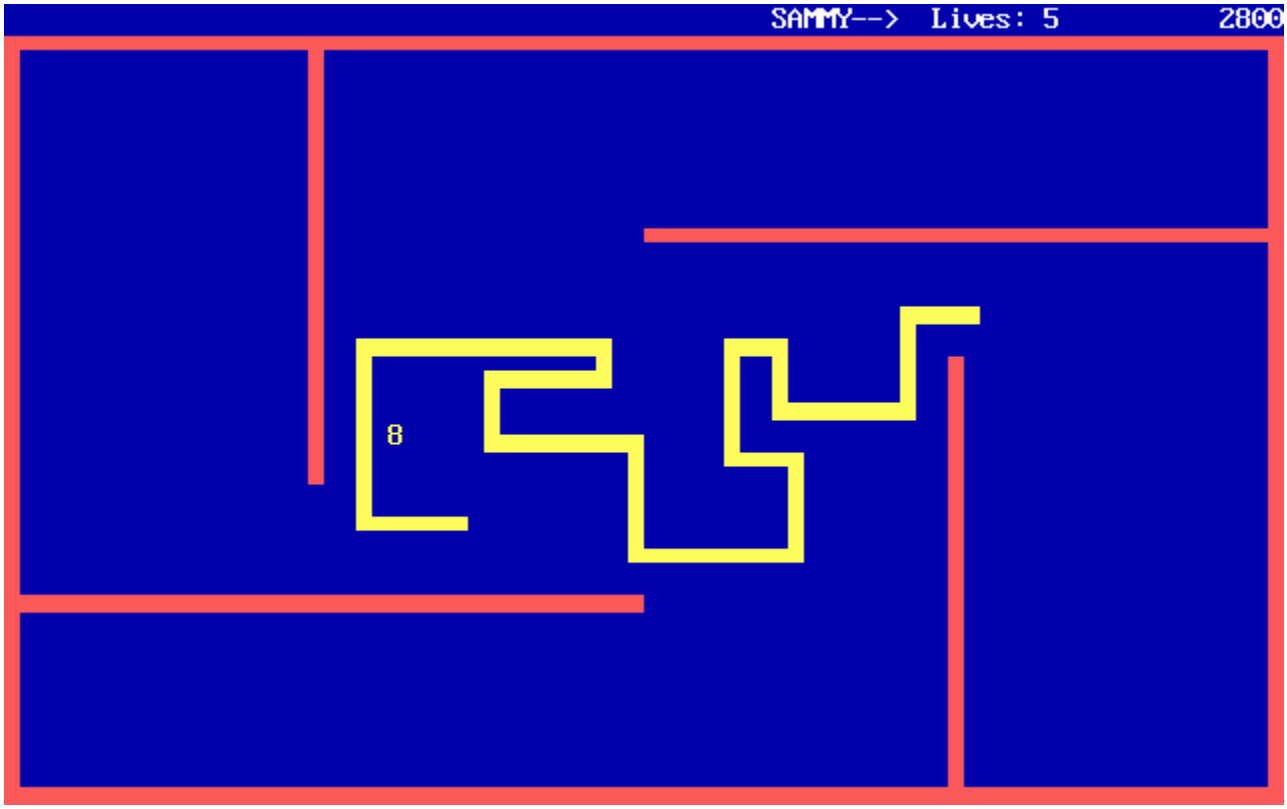
> *`vector` is the type of sequence that should be used by default... `deque` is the data structure of choice when most insertions and deletions take place at the beginning or at the end of the sequence.*

If you ever find yourself about to use a vector, check to see what you're doing with it. If you need to optimize for fast access, keep using a `vector`. If you're going to be inserting or deleting elements at the beginning or end of the container frequently, consider using a `deque` instead.

Extended Example: Snake

Few computer games can boast the longevity or addictive power of *Snake*. Regardless of your background, chances are that you have played Snake or one of its many variants. The rules are simple – you control a snake on a two-dimensional grid and try to eat food pellets scattered around the grid. You lose if you crash into the walls or into your own body. True to Newton's laws, the snake continues moving in a single direction until you explicitly change its bearing by ninety degrees. Every time the snake eats food, a new piece of food is randomly placed on the grid and the snake's length increases. Over time, the snake's body grows so long that it becomes an obstacle, and if the snake collides with itself the player loses.

Here's a screenshot from QBasic *Nibbles*, a Microsoft implementation of Snake released with MS-DOS version 5.0. The snake is the long yellow string, and the number 8 is the food:



Because the rules of Snake are so simple, it's possible to implement the entire game in only a few hundred lines of C++ code. In this extended example, we'll write a Snake program in which the *computer* controls the snake according to a simple AI. In the process, we'll gain experience with the STL `vector` and `deque`, the streams library, and a sprinkling of C library functions. Once we've finished, we'll have a rather snazzy program that can serve as a launching point for further C++ exploration.

Our Version of Snake

There are many variants of Snake, so to avoid confusion we'll explicitly spell out the rules of the game we're implementing:

1. The snake moves by extending its head in the direction it's moving and pulling its tail in one space.
2. The snake wins if it eats twenty pieces of food.
3. The snake loses if it crashes into itself or into a wall.
4. If the snake eats a piece of food, its length grows by one and a new piece of food is randomly placed.
5. There is only one level, the starting level.

While traditionally Snake is played by a human, our Snake will be computer-controlled so that we can explore some important pieces of the C runtime library. We'll discuss the AI we'll use when we begin implementing it.

Representing the World

In order to represent the Snake world, we need to keep track of the following information:

1. The size and layout of the world.
2. The location of the snake.
3. How many pieces of food we've consumed.

Let's consider this information one piece at a time. First, how should we represent the world? The world is two-dimensional, but all of the STL containers we've seen so far only represent lists, which are inherently one-dimensional. Unfortunately, the STL doesn't have a container class that encapsulates a multidimensional array, but we can emulate this functionality with an STL `vector` of `vector`s. For example, if we represent each square with an object of type `WorldTile`, we could use a `vector<vector<WorldTile> >`. Note that there is a space between the two closing angle brackets – this is deliberate and is an unfortunate bug in the C++ specification. If we omit the space, C++ would interpret the closing braces on `vector<vector<WorldTile>>` as the stream extraction operator `>>`, as in `cin >> myValue`. Although most compilers will accept code that uses two adjacent closing braces, it's bad practice to write it this way.

While we could use a `vector<vector<WorldTile> >`, there's actually a simpler option. Since we need to be able to display the world to the user, we can instead store the world as a `vector<string>` where each `string` encodes one row of the board. This also simplifies displaying the world; given a `vector<string>` representing all the world information, we can draw the board by outputting each string on its own line. Moreover, since we can use the bracket operator [] on both `vector` and `string`, we can use the familiar syntax `world[row][col]` to select individual locations. The first brackets select the `string` out of the `vector` and the second the character out of the `string`.

We'll use the following characters to encode game information:

- A space character (`' '`) represents an empty tile.
- A pound sign (`'#'`) represents a wall.
- A dollar sign (`'$'`) represents food.
- An asterisk (`'*'`) represents a tile occupied by a snake.

For simplicity, we'll bundle all the game data into a single `struct` called `gameT`. This will allow us to pass all the game information to functions as a single parameter. Based on the above information, we can begin writing this `struct` as follows:

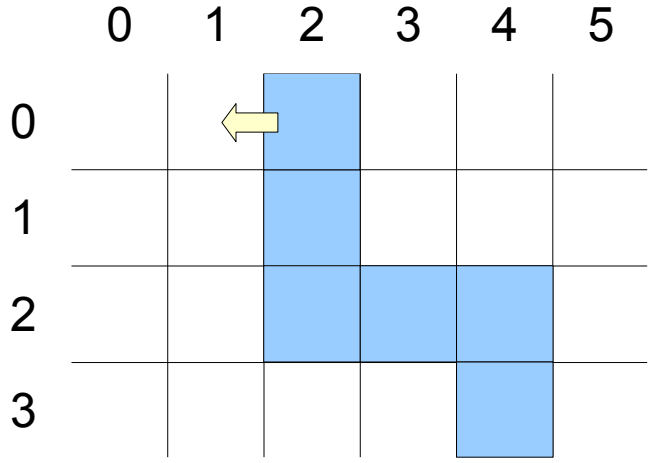
```
struct gameT {
    vector<string> world;
};
```

We also will need quick access to the dimensions of the playing field, since we will need to be able to check whether the snake is out of bounds. While we could access this information by checking the dimensions of the `vector` and the strings stored in it, for simplicity we'll store this information explicitly in the `gameT` struct, as shown here:

```
struct gameT {
    vector<string> world;
    int numRows, numCols;
};
```

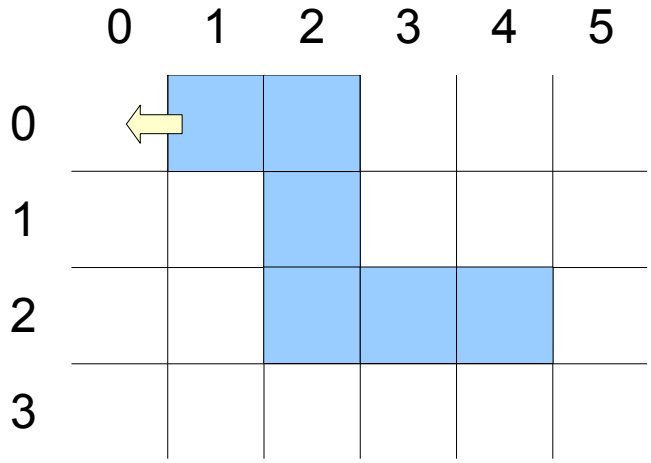
For consistency, we'll access elements in the `vector<string>` treating the first index as the row and the second as the column. Thus `world[3][5]` is row three, column five (where indices are zero-indexed).

Now, we need to settle on a representation for the snake. The snake lives on a two-dimensional grid and moves at a certain velocity. Because the grid is discrete, we can represent the snake as a collection of its points along with its velocity vector. For example, we can represent the following snake:



As the points (2, 0), (2, 1), (2, 2), (3, 2), (4, 2), (4, 3) and the velocity vector (-1, 0).

The points comprising the snake body are ordered to determine how the snake moves. When the snake moves, the first point (the *head*) moves one step in the direction of the velocity vector. The second piece then moves into the gap left by the first, the third moves into the gap left by the second piece, etc. This leaves a gap where the tail used to be. For example, after moving one step, the above snake looks like this:



To represent the snake in memory, we thus need to keep track of its velocity and an ordered list of the points comprising it. The former can be represented using two `int`s, one for the Δx component and one for the Δy component. But how should we represent the latter? We've just learned about the `vector` and `deque`, each of which could represent the snake. To see what the best option is, let's think about how we might implement snake motion. We can think of snake motion in one of two ways – first, as the head moving forward a step and the rest of the points shifting down one spot, and second as the snake getting a new point in front of its current head and losing its tail. The first approach requires us to update every element in the body and is not particularly efficient. The second approach can easily be implemented with a `deque` through an appropriate combination of `push_front` and `pop_back`. We will thus use a `deque` to encode the snake body.

If we want to have a `deque` of points, we'll first need some way of encoding a point. This can be done with this struct:

```
struct pointT {
    int row, col;
};
```

Taking these new considerations into account, our new `gameT` struct looks like this:

```
struct gameT {
    vector<string> world;
    int numRows, numCols;

    deque<pointT> snake;
    int dx, dy;
};
```

Finally, we need to keep track of how many pieces of food we've munched so far. That can easily be stored in an `int`, yielding this final version of `gameT`:

```
struct gameT {
    vector<string> world;
    int numRows, numCols;

    deque<pointT> snake;
    int dx, dy;

    int numEaten;
};
```

The Skeleton Implementation

Now that we've settled on a representation for our game, we can start thinking about how to organize the program. There are two logical steps – setup and gameplay – leading to the following skeleton implementation:

```
#include <iostream>
#include <string>
#include <deque>
#include <vector>
using namespace std;

/* Number of food pellets that must be eaten to win. */
const int kMaxFood = 20;

/* Constants for the different tile types. */
const char kEmptyTile = ' ';
const char kWallTile  = '#';
const char kFoodTile  = '$';
const char kSnakeTile = '*';

/* A struct encoding a point in a two-dimensional grid. */
struct pointT {
    int row, col;
};

/* A struct containing relevant game information. */
struct gameT {
    vector<string> world; // The playing field
    int numRows, numCols; // Size of the playing field

    deque<pointT> snake;  // The snake body
    int dx, dy;           // The snake direction

    int numEaten;         // How much food we've eaten.
};

/* The main program.  Initializes the world, then runs the simulation. */
int main() {
    gameT game;
    InitializeGame(game);
    RunSimulation(game);
    return 0;
}
```

Atop this program are the necessary `#include`s for the functions and objects we're using, followed by a list of constants for the game. The `pointT` and `gameT` structs are identical to those described above. `main` creates a `gameT` object, passes it into `InitializeGame` for initialization, and finally hands it to `RunSimulation` to play the game.

We'll begin by writing `InitializeGame` so that we can get a valid `gameT` for `RunSimulation`. But how should we initialize the game board? Should we use the same board every time, or let the user specify a level of their choosing? Both of these are resaonable, but for the this extended example we'll choose the latter. In particular, we'll specify a level file format, then let the user specify which file to load at runtime.

There are many possible file formats to choose from, but each must contain at least enough information to populate a `gameT` struct; that is, we need the world dimensions and layout, the starting position of the snake, and the direction of the snake. While I encourage you to experiment with different structures, we'll use a simple file format that encodes the world as a list of strings and the rest of the data as integers in a particular order. Here is one possible file:

File: *level.txt*

```
15 15
1 0
###############
#$             $#
#    #      #    #
#    #      #    #
#    #  $   #    #
#    #      #    #
#    #      #    #
#        *        #
#    #      #    #
#    #      #    #
#    #  $   #    #
#    #      #    #
#    #      #    #
#$             $#
###############
```

The first two numbers encode the number of rows and columns in the file, respectively. The next line contains the initial snake velocity as Δx, Δy. The remaining lines encode the game board, using the same characters we settled on for the world `vector`. We'll assume that the snake is initially of length one and its position is given by a `*` character.

There are two steps necessary to let the user choose the level layout. First, we need to prompt the user for the name of the file to open, reprompting until she chooses an actual file. Second, we need to parse the contents of the file into a `gameT` struct. In this example we won't check that the file is formatted correctly, though in professional code we would certainly need to check this. If you'd like some additional practice with the streams library, this would be an excellent exercise.

Let's start writing the function responsible for loading the file from disk, `InitializeGame`. Since we need to prompt the user for a filename until she enters a valid file, we'll begin writing:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        /* ... */
    }
    /* ... */
}
```

The `while(true)` loop will continuously prompt the user until she enters a valid file. Here, we assume that `GetLine()` is the version defined in the chapter on streams. Also, since we're now using the `ifstream` type, we'll need to `#include <fstream>` at the top of our program.

Now that the user has given us the a filename, we'll try opening it using the `.open()` member function. If the file opens successfully, we'll break out of the loop and start reading level data:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for info on .c_str().
        if(input.is_open()) break;

        /* ... */
    }
    /* ... */
}
```

If the file did *not* open, however, we need to report this to the user. Additionally, we have to make sure to reset the stream's error state, since opening a nonexistent file causes the stream to fail. Code for this is shown here:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for info on .c_str().
        if(input.is_open()) break;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
    /* ... */
}
```

Now we need to parse the file data into a `gameT` struct. Since this is rather involved, we'll decompose it into a helper function called `LoadWorld`, then finish `InitializeGame` as follows:

```
void InitializeGame(gameT& game) {
    ifstream input;
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for info on .c_str().
        if(input.is_open()) break;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
    LoadWorld(game, input);
}
```

Notice that except for the call to `LoadWorld`, nothing in the code for `InitializeGame` actually pertains to our Snake game. In fact, the code we've written is a generic routine for opening a file specified by the user. We'll thus break this function down into two functions – `OpenUserFile`, which prompts the user for a filename, and `InitializeGame`, which opens the specified file, then hands it off to `LoadWorld`. This is shown here:

```
void OpenUserFile(ifstream& input) {
    while(true) {
        cout << "Enter filename: ";
        string filename = GetLine();

        input.open(filename.c_str()); // See Chapter 3 for .c_str().
        if(input.is_open()) return;

        cout << "Sorry, I can't find the file " << filename << endl;
        input.clear();
    }
}

void InitializeGame(gameT& game) {
    ifstream input;
    OpenUserFile(input);
    LoadWorld(game, input);
}
```

Let's begin working on `LoadWorld`. The first line of our file format encodes the number of rows and columns in the world, and we can read this data directly into the `gameT` struct, as seen here:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    /* ... */
}
```

We've also resized the `vector` to hold `game.numRows` `string`s, guaranteeing that we have enough strings to store the entire world. This simplifies the implementation, as you'll see momentarily.

Next, we'll read the starting velocity for the snake, as shown here:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    /* ... */
}
```

At this point, we've read in the parameters of the world, and need to start reading in the actual world data. Since each line of the file contains one row of the grid, we'll use `getline` for the remaining read operations. There's a catch, however. Recall that `getline` does not mix well with the stream extraction operator (`>>`), which we've used exclusively so far. In particular, the first call to `getline` after using the stream extraction operator will return the empty string because the newline character delimiting the data is still waiting to be read. To prevent this from gumming up the rest of our input operations, we'll call `getline` here on a dummy string to flush out the remaining newline:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    /* ... */
}
```

Now we're ready to start reading in world data. We'll read in `game.numRows` lines from the file directly into the `game.world vector`. Since earlier we `resize`d the `vector`, there already are enough `string`s to hold all the data we'll read. The reading code is shown below:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row) {
        getline(input, game.world[row]);
        /* ... */
    }

    /* ... */
}
```

Recall that somewhere in the level file is a single `*` character indicating where the snake begins. To make sure that we set up the snake correctly, after reading in a line of the world data we'll check to see if it contains a star and, if so, we'll populate the `game.snake deque` appropriately. Using the `.find()` member function on the `string` simplifies this task, as shown here:

```
    void LoadWorld(gameT& game, ifstream& input) {
        input >> game.numRows >> game.numCols;
        game.world.resize(game.numRows);

        input >> game.dx >> game.dy;

        string dummy;
        getline(input, dummy);

        for(int row = 0; row < game.numRows; ++row) {
            getline(input, game.world[row]);
            int col = game.world[row].find(kSnakeTile);
            if(col != string::npos) {
                pointT head;
                head.row = row;
                head.col = col;
                game.snake.push_back(head);
            }
        }

        /* ... */
    }
```

The syntax for creating and filling in the `pointT` data is a bit bulky here. When we cover classes in the second half of this course you'll see a much better way of creating this `pointT`. In the meantime, we can write a helper function to clean this code up, as shown here:

```
pointT MakePoint(int row, int col) {
  pointT result;
  result.row = row;
  result.col = col;
  return result;
}

    void LoadWorld(gameT& game, ifstream& input) {
        input >> game.numRows >> game.numCols;
        game.world.resize(game.numRows);

        input >> game.dx >> game.dy;

        string dummy;
        getline(input, dummy);

        for(int row = 0; row < game.numRows; ++row) {
            getline(input, game.world[row]);
            int col = game.world[row].find(kSnakeTile);
            if(col != string::npos)
                game.snake.push_back(MakePoint(row, col));
        }

        /* ... */
    }
```

There's one last step to take care of, and that's to ensure that we set the `numEaten` field to zero. This edit completes `LoadWorld` and the final version of the code is shown here:

```
void LoadWorld(gameT& game, ifstream& input) {
    input >> game.numRows >> game.numCols;
    game.world.resize(game.numRows);

    input >> game.dx >> game.dy;

    string dummy;
    getline(input, dummy);

    for(int row = 0; row < game.numRows; ++row) {
        getline(input, game.world[row]);
        int col = game.world[row].find(kSnakeTile);
        if(col != string::npos)
            game.snake.push_back(MakePoint(row, col));
    }

    game.numEaten = 0;
}
```

Great! We've just finished setup and it's now time to code up the actual game. We'll begin by coding a skeleton of `RunSimulation` which displays the current state of the game, runs the AI, and moves the snake:

```
void RunSimulation(gameT& game) {
    /* Keep looping while we haven't eaten too much. */
    while(game.numEaten < kMaxFood) {
        PrintWorld(game);     // Display the board
        PerformAI(game);      // Have the AI choose an action

        if(!MoveSnake(game)) // Move the snake and stop if we crashed.
            break;

        Pause();              // Pause so we can see what's going on.
    }
    DisplayResult(game);      // Tell the user what happened
}
```

We'll implement the functions referenced here out of order, starting with the simplest and moving to the most difficult. First, we'll begin by writing `Pause`, which stops for a short period of time to make the game seem more fluid. The particular implementation of `Pause` we'll use is a *busy loop*, a `while` loop that does nothing until enough time has elapsed. Busy loops are frowned upon in professional code because they waste CPU power, but for our purposes are perfectly acceptable.

The `<ctime>` header exports a function called `clock()` that returns the number of "clock ticks" that have elapsed since the program began. The duration of a clock tick varies from system to system, so C++ provides the constant `CLOCKS_PER_SEC` to convert clock ticks to seconds. We can use `clock` to implement a busy loop as follows:

1. Call `clock()` to get the current time in clock ticks and store the result.
2. Continuously call `clock()` and compare the result against the cached value. If enough time has passed, stop looping.

This can be coded as follows:

```
const double kWaitTime = 0.1; // Pause 0.1 seconds between frames
void Pause() {
    clock_t startTime = clock(); // clock_t is a type which holds clock ticks.

    /* This loop does nothing except loop and check how much time is left.
     * Note that we have to typecast startTime from clock_t to double so
     * that the division is correct.  The static_cast<double>(...) syntax
     * is the preferred C++ way of performing a typecast of this sort;
     * see the chapter on
     * inheritance for more information.
     */
    while(static_cast<double>(clock() - startTime) / CLOCKS_PER_SEC <
              kWaitTime);
}
```

Next, let's implement the `PrintWorld` function, which displays the current state of the world. We chose to represent the world as a `vector<string>` to simplify this code, and as you can see this design decision pays off well:

```
void PrintWorld(gameT& game) {
    for(int row = 0; row < game.numRows; ++row)
        cout << game.world[row] << endl;
    cout << "Food eaten: " << game.numEaten << endl;
}
```

This implementation of `PrintWorld` is fine, but every time it executes it adds more text to the console instead of clearing what's already there. This makes it tricky to see what's happening. Unfortunately, standard C++ does not export a set of routines for manipulating the console. However, every major operating system exports its own console manipulation routines, primarily for developers working on a command line. For example, on a Linux system, typing `clear` into the console will clear its contents, while on Windows the command is `CLS`.

C++ absorbed C's standard library, including the `system` function (header file `<cstdlib>`). `system` executes an operating system-specific instruction as if you had typed it into your system's command line. This function can be very dangerous if used incorrectly,* but also greatly expands the power of C++. We will not cover how to use `system` in detail since it is platform-specific, but one particular application of system is to call the appropriate operating system function to clear the console. We can thus upgrade our implementation of `PrintWorld` as follows:

```
/* The string used to clear the display before printing the game board.
 * Windows systems should use "CLS"; Mac OS X or Linux users should use
 * "clear" instead.
 */
const string kClearCommand = "CLS";

void PrintWorld(gameT& game) {
    system(kClearCommand.c_str());
    for(int row = 0; row < game.numRows; ++row)
        cout << game.world[row] << endl;
    cout << "Food eaten: " << game.numEaten << endl;
}
```

Because `system` is from the days of pure C, we have to use `.c_str()` to convert the string parameter into a C-style string before we can pass it into the function.

---

* In particular, calling `system` without checking that the parameters have been sanitized can let malicious users completely compromise your system. Take CS155 for more information on what sorts of attacks are possible.

The final quick function we'll write is `DisplayResult`, which is called after the game has ended to report whether the computer won or lost. This function is shown here:

```
void DisplayResult(gameT& game) {
    PrintWorld(game);
    if(game.numEaten == kMaxFood)
        cout << "The snake ate enough food and wins!" << endl;
    else
        cout << "Oh no!  The snake crashed!" << endl;
}
```

Now, on to the two tricky functions – `PerformAI`, which determines the snake's next move, and `MoveSnake`, which moves the snake and processes collisions. We'll begin with `PerformAI`.

Designing an AI that plays Snake intelligently is far beyond the scope of this class. However, it is feasible to build a rudimentary AI that plays reasonably well. Our particular AI works as follows: if the snake is about to collide with an object, the AI will turn the snake out of danger. Otherwise, the snake will continue on its current path, but has a percent chance to randomly change direction.

Let's begin by writing the code to check whether the snake will turn; that is, whether we're about to hit a wall or if the snake randomly decides to veer in a direction. We'll write a skeletal implementation of this code, then will implement the requisite functions. Our initial code is

```
const double kTurnRate = 0.2; // 20% chance to turn each step.
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game);

    /* If that hits a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        /* ... */
    }
}
```

Here we're calling three functions we haven't written yet – `GetNextPosition`, which computes the position of the head on the next iteration; `Crashed`, which returns whether the snake would crash if its head was in the given position; and `RandomChance`; which returns true with probability equal to the parameter. Before implementing the rest of `PerformAI`, let's knock these functions out so we can focus on the rest of the task at hand. We begin by implementing `GetNextPosition`. This function accepts as input the game state and returns the point that we will occupy on the next frame if we continue moving in our current direction. This function isn't particularly complex and is shown here:

```
pointT GetNextPosition(gameT& game) {
    /* Get the head position. */
    pointT result = game.snake.front();

    /* Increment the head position by the current direction. */
    result.row += game.dy;
    result.col += game.dx;
    return result;
}
```

The implementation of `Crashed` is similarly straightforward. The snake has crashed if it has gone out of bounds or if its head is on top of a wall or another part of the snake:

```
bool Crashed(pointT headPos, gameT& game) {
    return !InWorld(headPos, game) ||
           game.world[headPos.row][headPos.col] == kSnakeTile ||
           game.world[headPos.row][headPos.col] == kWallTile;
}
```

Here, `InWorld` returns whether the point is in bounds and is defined as

```
bool InWorld(pointT& pt, gameT& game) {
    return pt.col >= 0 &&
           pt.row >= 0 &&
           pt.col < game.numCols &&
           pt.row < game.numRows;
}
```

Next, we need to implement `RandomChance`. In CS106B/X we provide you a header file, `random.h`, that exports this function. However, `random.h` is not a standard C++ header file and thus we will not use it here. Instead, we will use C++'s `rand` and `srand` functions, also exported by `<cstdlib>`, to implement `RandomChance`. `rand()` returns a pseudorandom number in the range [0, `RAND_MAX`], where `RAND_MAX` is usually $2^{15} - 1$. srand seeds the random number generator with a value that determines which values are returned by `rand`. One common technique is to use the `time` function, which returns the current system time, as the seed for `srand` since different runs of the program will yield different random seeds. Traditionally, you will only call `srand` once per program, preferably during initialization. We'll thus modify `InitializeGame` so that it calls `srand` in addition to its other functionality:

```
void InitializeGame(gameT& game) {
    /* Seed the randomizer.  The static_cast converts the result of time(NULL)
     * from time_t to the unsigned int required by srand.  This line is
     * idiomatic C++.
     */
    srand(static_cast<unsigned int>(time(NULL)));

    ifstream input;
    OpenUserFile(input);
    LoadWorld(game, input);
}
```

Now, let's implement `RandomChance`. To write this function, we'll call rand to obtain a value in the range [0, `RAND_MAX`], then divide it by `RAND_MAX + 1.0` to get a value in the range [0, 1). We can then return whether this value is less than the input probability. This yields true with the specified probability; try convincing yourself that this works if it doesn't immediately seem obvious. This is a common technique and in fact is how the CS106B/X `RandomChance` function is implemented.

`RandomChance` is shown here:

```
bool RandomChance(double probability) {
    return (rand() / (RAND_MAX + 1.0)) < probability;
}
```

Notice that we added 1.0 to `RAND_MAX`. This both adds the +1 necessary from the above discussion and implicitly converts the denominator into a `double`, which is necessary to avoid integer truncation.

Phew! Apologies for the lengthy detour – let's get back to writing the AI! Recall that we've written this code so far:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        /* ... */
    }
}
```

We now need to implement the logic for turning the snake left or right. First, we'll figure out in what positions the snake's head would be if we turned left or right. Then, based on which of these positions are safe, we'll pick a direction to turn. To avoid code duplication, we'll modify our implementation of `GetNextPosition` so that the caller can specify the direction of motion, rather than relying on the `gameT`'s stored direction. The modified version of `GetNextPosition` is shown here:

```
pointT GetNextPosition(gameT& game, int dx, int dy) {
    /* Get the head position. */
    gameT result = game.snake.front();

    /* Increment the head position by the specified direction. */
    result.row += dy;
    result.col += dx;
    return result;
}
```

We'll need to modify `PerformAI` to pass in the proper parameters to `GetNextPosition`, as shown here:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        /* ... */
    }
}
```

Now, let's write the rest of this code. Given that the snake's velocity is (`game.dx`, `game.dy`), what velocities would we move at if we were heading ninety degrees to the left or right? Using some basic linear algebra,* if our current heading is along dx and dy, then the headings after turning left and right from our current heading are be given by

$$dx_{left} = -dy$$
$$dy_{left} = dx$$

$$dx_{right} = dy$$
$$dy_{right} = -dx$$

Using these equalities, we can write the following code, which determines what bearings are available and whether it's safe to turn left or right:

---

* This is the result of multiplying the vector $(dx, dy)^T$ by a rotation matrix for either $+\pi/2$ or $-\pi/2$ radians.

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),
                                 game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy),
                                 game);

        /* ... */
    }
}
```

Now, we'll decide which direction to turn. If we can only turn one direction, we will choose that direction. If we can't turn at all, we will do nothing. Finally, if we can turn either direction, we'll pick a direction randomly. We will store which direction to turn in a boolean variable called `willTurnLeft` which is `true` if we will turn left and `false` if we will turn right. This is shown here:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),
                                 game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy),
                                 game);

        bool willTurnLeft = false;
        if(!canLeft && !canRight)
            return; // If we can't turn, don't turn.
        else if(canLeft && !canRight)
            willTurnLeft = true; // If we must turn left, do so.
        else if(!canLeft && canRight)
            willTurnLeft = false; // If we must turn right, do so.
        else
            willTurnLeft = RandomChance(0.5); // Else pick randomly

        /* ... */
    }
}
```

Finally, we'll update our direction vector based on our choice:

```
void PerformAI(gameT& game) {
    /* Figure out where we will be after we move this turn. */
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);

    /* If that puts us into a wall or we randomly decide to, turn the snake. */
    if(Crashed(nextHead, game) || RandomChance(kTurnRate)) {
        int leftDx = -game.dy;
        int leftDy =  game.dx;

        int rightDx =  game.dy;
        int rightDy = -game.dx;

        /* Check if turning left or right will cause us to crash. */
        bool canLeft  = !Crashed(GetNextPosition(game, leftDx,  leftDy),
                                 game);
        bool canRight = !Crashed(GetNextPosition(game, rightDx, rightDy),
                                 game);

        bool willTurnLeft = false;
        if(!canLeft && !canRight)
            return; // If we can't turn, don't turn.
        else if(canLeft && !canRight)
            willTurnLeft = true; // If we must turn left, do so.
        else if(!canLeft && canRight)
            willTurnLeft = false; // If we must turn right, do so.
        else
            willTurnLeft = RandomChance(0.5); // Else pick randomly

        game.dx = willTurnLeft? leftDx : rightDx;
        game.dy = willTurnLeft? leftDy : rightDy;
    }
}
```

If you're not familiar with the `?:` operator, the syntax is as follows:

```
expression ? result-if-true : result-if-false
```

Here, this means that we'll set `game.dx` to `leftDx` if `willTurnLeft` is true and to `rightDx` otherwise.

We now have a working version of `PerformAI`. Our resulting implementation is not particularly dense, and most of the work is factored out into the helper functions.

There is one task left – implementing `MoveSnake`. Recall that `MoveSnake` moves the snake one step forward on its path. If the snake crashes, the function returns `false` to indicate that the game is over. Otherwise, the function returns `true`.

The first thing to do in `MoveSnake` is to figure out where the snake's head will be after taking a step. Thanks to `GetNextPosition`, this has already been taken care of for us:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    /* ... */
}
```

Now, if we crashed into something (either by falling off the map or by hitting an object), we'll return `false` so that the main loop can terminate:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    /* ... */
}
```

Next, we need to check to see if we ate some food. We'll store this in a `bool` variable for now, since the logic for processing food will come a bit later:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    /* ... */
}
```

Now, let's update the snake's head. We need to update the `world vector` so that the user can see that the snake's head is in a new square, and also need to update the `snake deque` so that the snake's head is now given by the new position. This is shown here:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    game.world[nextHead.row][nextHead.col] = kSnakeTile;
    game.snake.push_front(nextHead);

    /* ... */
}
```

Finally, it's time to move the snake's tail forward one step. However, if we've eaten any food, we will leave the tail as-is so that the snake grows by one tile. We'll also put food someplace else on the map so the snake has a new objective. The code for this is shown here:

```
bool MoveSnake(gameT& game) {
    pointT nextHead = GetNextPosition(game, game.dx, game.dy);
    if(Crashed(nextHead, game))
        return false;

    bool isFood = (game.world[nextHead.row][nextHead.col] == kFoodTile);

    game.world[nextHead.row][nextHead.col] = kSnakeTile;
    game.snake.push_front(nextHead);

    if(!isFood) {
        game.world[game.snake.back().row][game.snake.back().col] = kEmptyTile;
        game.snake.pop_back();
    } else {
        ++game.numEaten;
        PlaceFood(game);
    }
    return true;
}
```

We're nearing the home stretch – all that's left to do is to implement `PlaceFood` and we're done! This function is simple – we'll just sit in a loop picking random locations on the board until we find an empty spot, then will put a piece of food there. To generate a random location on the board, we'll scale `rand()` down to the proper range using the modulus (`%`) operator. For example, on a world with four rows and ten columns, we'd pick as a row `rand() % 4` and as a column `col() % 10`. The code for this function is shown here:

```
void PlaceFood(gameT& game) {
    while(true) {
        int row = rand() % game.numRows;
        int col = rand() % game.numCols;

        /* If the specified position is empty, place the food there. */
        if(game.world[row][col] == kEmptyTile) {
            game.world[row][col] = kFoodTile;
            return;
        }
    }
}
```

More To Explore

There's so much to explore with the STL that we could easily fill the rest of the course reader with STL content. If you're interested in some more advanced topics relating to this material and the STL in general, consider reading on these topics:

1. **`stack` and `queue`**: The `vector` and `deque` pack a lot of firepower and can solve a wide array of problems. However, in some cases, you may want to use a container with a more restricted set of operations. For these purposes, the STL exports two *container adapters*, containers that export functionality similar to a `vector` or `deque` but with a slight reduction in power. The first of these is the `stack`, which only lets you view the final element of a sequence; the second is the `queue`, which is similar to line at a ticket counter. If you plan on pursuing C++ more seriously, you should take the time to look over what these container adapters have to offer.

2. **`valarray`**: The `valarray` class is similar to a `vector` in that it's a managed array that can hold elements of any type. However, unlike `vector`, `valarray` is designed for numerical computations. `valarray`s are fixed-size and have intrinsic support for mathematical operators. For example, you can use the syntax `myValArray *= 2` to multiply all of the entries in a `valarray` by two. If you're interested in numeric or computational programming, consider looking into the `valarray`.

3. There's an excellent article online comparing the performances of the `vector` and `deque` containers. If you're interested, you can see it at http://www.codeproject.com/vcpp/stl/vector_vs_deque.asp.

Practice Problems

1. List two differences between the `vector`'s `push_back` and `resize` member functions.

2. What header files do you need to `#include` to use `vector`? `deque`?

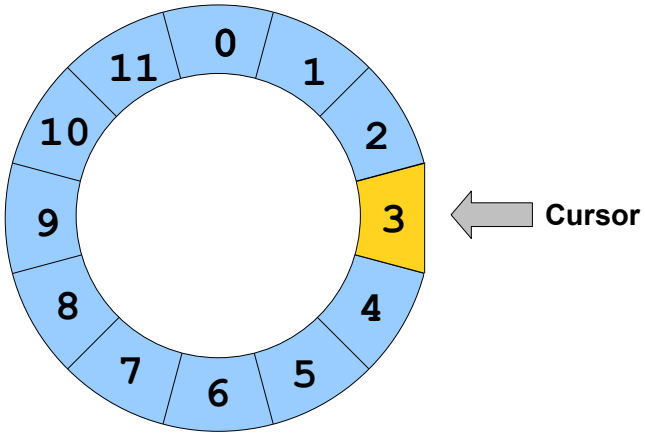
3. How do you tell how many elements are in a `vector`? In a `deque`?

4. How do you remove the first element from a `vector`? From a `deque`?

5. Write a function called `LinesFromFile` which takes in a `string` containing a filename and returns a `vector<string>` containing all of the lines of text in the file in the order in which they appear. If the file does not exist, you can return an empty `vector`. *(Hint: look at the code for reading the world file in the Snake example and see if you can modify it appropriately)*

6. Update the code for the sorting program so that it sorts elements in *descending* order instead of *ascending* order.

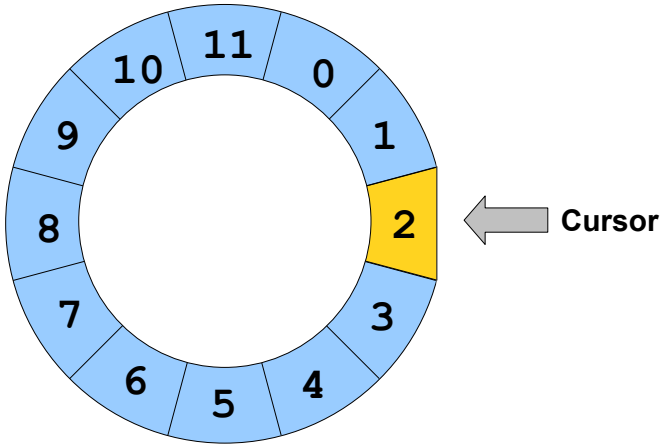
7. One use for the `deque` container is to create a *ring buffer*. Unlike the linear `vector` and `deque` containers you saw in this chapter, a ring buffer is circular. Here's an illustration of a ring buffer:



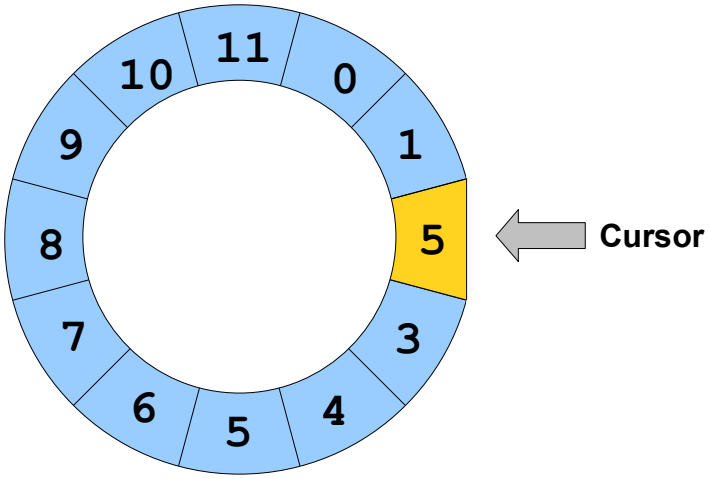
A ring buffer consists of a circular ring of elements with a *cursor* which selects some particular element out of the buffer. The four main operations on a ring buffer are as follows:

- Rotate the ring clockwise
- Rotate the ring counterclockwise
- Read the value at the cursor.
- Write the value at the cursor.

For example, given the above ring buffer, the result of rotating the ring clockwise would be

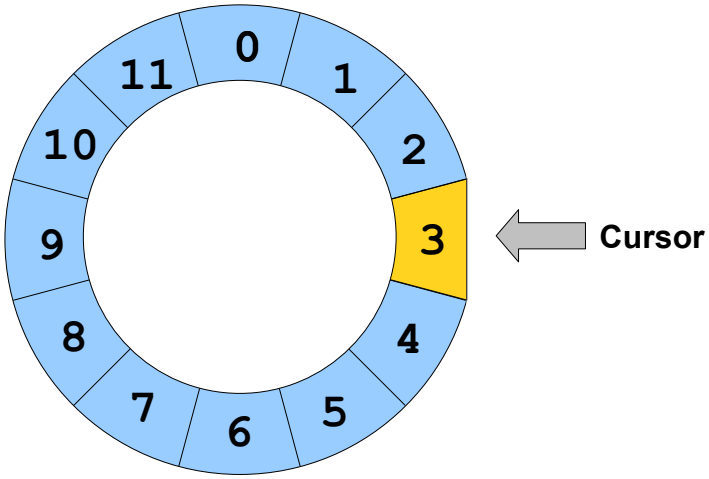


If we then wrote the value 5 to the location specified by the cursor, the buffer would look like this:

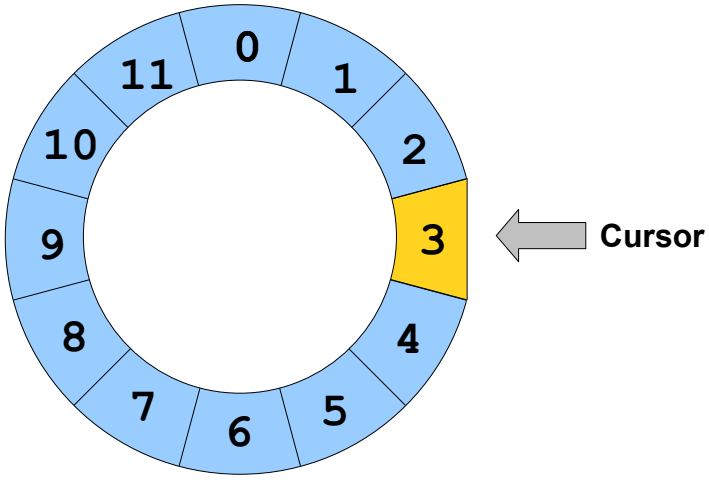


There is a particularly elegant construction which enables us to build a ring buffer out of a `deque`. The basic idea is to "unroll" the ring buffer into a linear sequence, then use a combination of `push_front`, `pop_front`, `push_back`, and `pop_back` to simulate moving the cursor to the left or to the right. This technique of simulating one data structure using another is ubiquitous in computer science, and many important results in computability theory use constructions of this form.

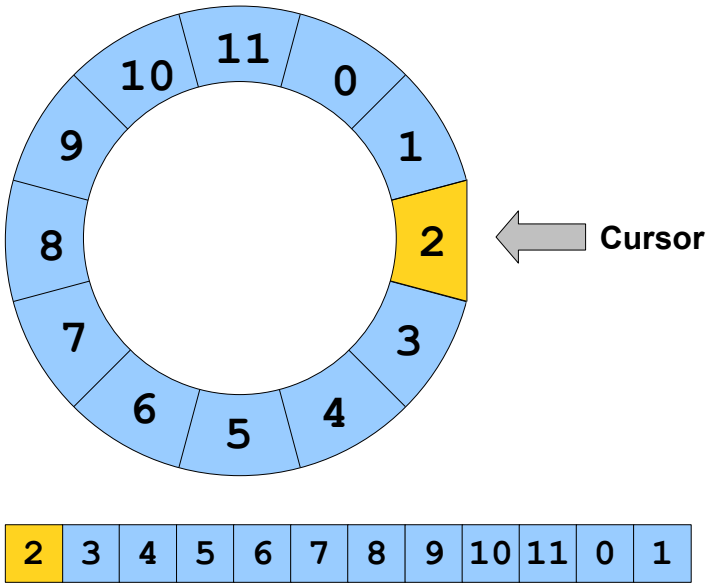
To see exactly how the construction works, suppose that we have the following ring buffer:



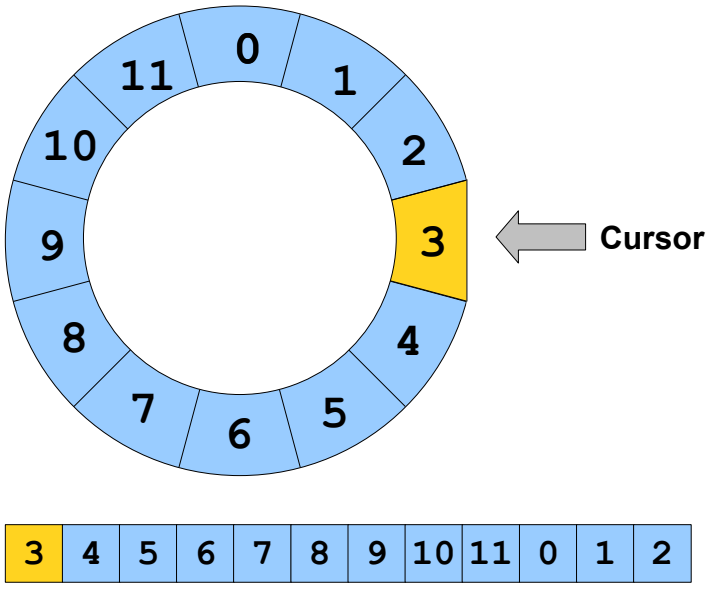
We can then represent this using a `deque` as follows:



That is, the first element of the `deque` is the element under the cursor, and the rest of the elements in the `deque` are the elements in the ring buffer formed by walking clockwise around the ring buffer. Given this construction, we can simulate rotating the ring one position clockwise by moving the last element of the `deque` onto the front, as shown here:



Similarly, to move the cursor one step counterclockwise, we move the element at the end of the `deque` onto the front, as shown here:



Write a pair of functions `CursorClockwise` and `CursorCounterClockwise` which take in a `deque` representing a ring buffer and update the `deque` by simulating a cursor move in either direction. Then write functions `CursorRead` and `CursorWrite` which read and write the element stored at the cursor. You've just shown how to represent one data structure using another!

8. As mentioned earlier, the `deque` outperforms the vector when inserting and removing elements at the end of the container. However, the `vector` has a useful member function called `reserve` that can be used to increase its performance against the `deque` in certain circumstances. The `reserve` function accepts an integer as a parameter and acts as a sort of "size hint" to the `vector`. Behind the scenes, `reserve` works by allocating additional storage space for the `vector` elements, reducing the number of times that the `vector` has to ask for more storage space. Once you have called `reserve`, as long as the size of the `vector` is less than the number of elements you have reserved, calls to `push_back` and `insert` on the `vector` will execute more quickly than normal. Once the `vector` hits the size you reserved, these operations revert to their original speed.*

   Write a program that uses `push_back` to insert a large number of `string`s into two different `vector`s – one which has had `reserve` called on it and one which hasn't – as well as a `deque`. The exact number and content of strings is up to you, but large numbers of long strings will give the most impressive results. Use the `clock()` function exported by `<ctime>` to compute how long it takes to finish inserting the `string`s. Now repeat this trial, but insert the elements at the beginning of the container rather than the end. Did calling `reserve` help to make the `vector` more competitive against the `deque`?

9. In this next problem we'll explore a simple encryption algorithm called the *Vigenère cipher* and how to implement it using the STL containers.

   One of the oldest known ciphers is the *Caesar cipher*, named for Julius Caesar, who allegedly employed it. The idea is simple. We pick a secret number between 1 and 26, inclusive, then encrypt the input string by replacing each letter with the letter that comes that many spaces after it. If this pushes us off the end of the alphabet, we wrap around to the start of the alphabet. For example, if we were given the string "The cookies are in the fridge" and picked the number 1, we would end up with the resulting string "Uif dppljft bsf jo uif gsjehf." To decrypt the string, we simply need to push each letter backwards by one spot.

   The Caesar cipher is an extremely weak form of encryption; it was broken in the ninth century by the Arab polymath al-Kindi. The problem is that the cipher preserves the relative frequencies of each of the letters in the source text. Not all letters appear in English with equal frequency – e and t are far more common than q or w, for example – and by looking at the relative letter frequencies in the encrypted text it is possible to determine which letter in the encrypted text corresponds to a letter in the source text and to recover the key.

   The problem with the Caesar cipher is that it preserves letter frequencies because each letter is transformed using the same key. But what if we were to use *multiple* keys while encrypting the message? That is, we might encrypt the first letter with one key, the second with another, the third with yet another, etc. One way of doing this is to pick a sequence of numbers, then cycle through them while encrypting the text. For example, let's suppose that we want to encrypt the above message using the key string 1, 3, 7. Then we would do the following:

T	H	E	C	O	O	K	I	E	S	A	R	E	I	N	T	H	E	F	R	I	D	G	E
1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7	1	3	7
U	**K**	**L**	**D**	**R**	**V**	**L**	**L**	**L**	**T**	**D**	**Y**	**F**	**L**	**U**	**U**	**K**	**L**	**G**	**U**	**P**	**E**	**J**	**L**

   Notice that the letters KIE from COOKIES are all mapped to the letter L, making cryptanalysis much more difficult. This particular encryption system is the Vigenère cipher.

* Calling `push_back` n times always takes O(n) time, whether or not you call `reserve`. However, calling `reserve` reduces the constant term in the big-O to a smaller value, meaning that the overall execution time is lower.

Now, let's consider what would happen if we wanted to implement this algorithm in C++ to work on arbitrary strings. Strings in C++ are composed of individual chars, which can take on (typically) one of 256 different values. If we had a list of integer keys, we could encrypt a string using the Vigenère cipher by simply cycling through those keys and incrementing the appropriate letters of the string. In fact, the algorithm is quite simple. We iterate over the characters of the string, at each point incrementing the character by the current key and then rotating the keys one cycle.

a. Suppose that we want to represent a list of integer keys that can easily be cycled; that is, we want to efficiently support moving the first element of the list to the back. Of the containers covered in this chapter (`vector` and `deque`), which have the best support for this operation? ♦

b. Based on your decision, implement a function `VigenereEncrypt` that accepts a string and a list of `int` keys stored in the container of your choice, then encrypts the string using the Vigenère cipher. ♦

Chapter 6: STL Associative Containers and Iterators

In the previous chapter, we explored two of the STL's sequence containers, the `vector` and `deque`. These containers are ideally suited for situations where we need to keep track of an ordered list of elements, such as an itinerary, shopping list, or mathematical vector. However, representing data in ordered lists is not optimal in many applications. For example, when keeping track of what merchandise is sold in a particular store, it does not make sense to think of the products as an ordered list. Storing merchandise in a list would imply that the merchandise could be ordered as "this is the first item being sold, this is the second item being sold, etc." Instead, it makes more sense to treat the collection of merchandise as an *unordered collection*, where *membership* rather than *ordering* is the defining characteristic. That is, we are more interested in answers to the question "is item X being sold here?" than answers to the question "where in the sequence is the element X?" Another scenario in which ordered lists are suboptimal arises when trying to represent *relationships* between sets of data. For example, we may want to encode a mapping from street addresses to buildings, or from email addresses to names. In this setup, the main question we are interested in answering is "what value is associated with X?," not "where in the sequence is element X?"

In this chapter, we will explore four new STL container classes – `map`, `set`, `multimap`, and `multiset` – that provide new abstractions for storing data. These containers will represent allow us to ask different questions of our data sets and will make it possible to write programs to solve increasingly complex problems. As we explore those containers, we will introduce *STL iterators*, tools that will pave the way for more advanced STL techniques.

Storing Unordered Collections with `set`

To motivate the STL `set` container, let's consider a simple probability question. Recall from last chapter's *Snake* example that the C++ `rand()` function can be used to generate a pseudorandom integer in the range [0, `RAND_MAX`]. (Recall that the notation [*a*, *b*] represents all real numbers between *a* and *b*, inclusive). Commonly, we are interested not in values from zero to `RAND_MAX`, but instead values from 0 to some set upper bound *k*. To get values in this range, we can use the value of

```
rand() % (k + 1)
```

This computes the remainder when dividing `rand()` by *k* + 1, which must be in the range [0, *k*].*

Now, consider the following question. Suppose that we have a six-sided die. We roll the die, then record what number we rolled. We then keep rolling the die and record what number came up, and keep repeating this process. The question is as follows: how many times, on average, will we roll the die before the same number comes up twice? This is actually a special case of a more general problem: if we continuously generate random integers in the range [0, *k*], how many numbers should we expect to generate before we generate some number twice? With some fairly advanced probability theory, this value can be calculated exactly. However, this is a textbook on C++ programming, not probability theory, and so we'll write a short program that will simulate this process and report the average number of die rolls.

---

* This process will not always yield uniformly-distributed values, because `RAND_MAX` will not always be a multiple of *k*. For a fun math exercise, think about why this is.

There are many ways that we can write this program. In the interest of simplicity, we'll break the program into two separate tasks. First, we'll write a function that rolls the die over and over again, then reports how many die rolls occurred before some number came up twice. Second, we'll write our `main` function to call this function multiple times to get a good sample, then will have it print out the average.

Let's think about how we can write a function that rolls a die until the same number comes up twice. At a high level, this function needs to generate a random number from 1 to 6, then check if it has been generated before. If so, it should stop and report the number of dice rolled. Otherwise, it should remember that this number has been rolled, then generate a new number. A key step of this process is remembering what numbers have come up before, and using the techniques we've covered so far we could do this using either a `vector` or a `deque`. For simplicity, we'll use a `vector`. One implementation of this function looks like this:

```
/* Rolls a six-sided die and returns the number that came up. */
int DieRoll() {
    /* rand() % 6 gives back a value between 0 and 5, inclusive.  Adding one to
     * this gives us a valid number for a die roll.
     */
    return (rand() % 6) + 1;
}

/* Rolls the dice until a number appears twice, then reports the number of die
 * rolls.
 */
size_t RunProcess() {
    vector<int> generated;

    while (true) {
        /* Roll the die. */
        int nextValue = DieRoll();

        /* See if this value has come up before.  If so, return the number of
         * rolls required.  This is equal to the number of dice that have been
         * rolled up to this point, plus one for this new roll.
         */
        for (size_t k = 0; k < generated.size(); ++k)
            if (generated[k] == nextValue)
                return generated.size() + 1;

        /* Otherwise, remember this die roll. */
        generated.push_back(nextValue);
    }
}
```

Now that we have the `RunProcess` function written, we can run through one simulation of this process. However, it would be silly to give an estimate based on just one iteration. To get a good estimate, we'll need to run this process multiple times to control for randomness. Consequently, we can write the following `main` function, which runs the process multiple times and reports the average value:

```
const size_t kNumIterations = 10000; // Number of iterations to run

int main() {
    /* Seed the randomizer.  See the last chapter for more information on this
     * line.
     */
    srand(static_cast<unsigned>(time(NULL)));

    size_t total = 0; // Total number of dice rolled

    /* Run the process kNumIterations times, accumulating the result into
     * total.
     */
    for (size_t k = 0; k < kNumIterations; ++k)
        total += RunProcess();

    /* Finally, report the result. */
    cout << "Average number of steps: "
         << double(total) / kNumIterations << endl;
}
```

If you compile and run this program, you'll see output that looks something like this:

```
Average number of steps: 3.7873
```

You might see a different number displayed on your system, since the program involves a fundamentally random process.

Now, let's make a small tweak to this program. Suppose that instead of rolling a six-sided die, we roll a twenty-sided die.* How many steps should we expect this to take now? If we change our implementation of `DieRoll` to the following:

```
int DieRoll() {
    return (rand() % 20) + 1;
}
```

Then running the program will produce output along the following lines:

```
Average number of steps: 6.2806
```

This is interesting – we more than tripled the number of sides on the die (from six to twenty), but the total number of expected rolls increased by less than a factor of two! Is this a coincidence, or is there some fundamental law of probability at work here? To find out, let's assume that we're now rolling a die with 365 sides (i.e. one side for every day of the year). This means our new implementation of `DieRoll` is

```
int DieRoll() {
    return (rand() % 365) + 1;
}
```

Running this program produces output that looks like this:

```
Average number of steps: 24.6795
```

* If you haven't seen a twenty-sided die (or *D20* in gamer-speak), you're really missing out. They're very fun to play with.

Now *that's* weird! In increasing the number of sides on the die from 20 to 365, we increased the number of sides on the die by a factor of (roughly) eighteen. However, the number of expected rolls went up only by a factor of four! But more importantly, think about what this result means. If you have a roomful of people with twenty-five people, then you should expect at least two people in that room to have the same birthday! This is sometimes called the *birthday paradox*, since it seems counterintuitive that such a small sample of people would cause this to occur. The more general result, for those of you who are interested, is that you will need to roll an n sided die roughly $\sqrt{n}$ times before the same number will come up twice.

This has been a fun diversion into the realm of probability theory, but what does it have to do with C++ programming? The answer lies in the implementation of the `RunProcess` function. The heart of this function is a `for` loop that checks whether a particular value is contained inside of a `vector`. This loop is reprinted here for simplicity:

```
for (size_t k = 0; k < generated.size(); ++k)
    if (generated[k] == nextValue)
        return generated.size() + 1;
```

Notice that there is a disparity between the high-level operation being modeled here ("check if the number has already been generated") and the actual implementation ("loop over the `vector`, checking, for each element, whether that element is equal to the most-recently generated number"). There is a tension here between what the code accomplishes and the way in which it accomplishes it. The reason for this is that we're using the *wrong abstraction*. Intuitively, a `vector` maintains an *ordered sequence* of elements. The main operations on a `vector` maintain that sequence by adding and removing elements from that sequence, looking up elements at particular positions in that sequence, etc. For this application, we want to store a collection of numbers that is *unordered*. We don't care when the elements were added to the `vector` or what position they occupy. Instead, we are interested *what* elements are in the `vector`, and in particular whether a given element is in the `vector` at all.

For situations like these, where the *contents* of a collection of elements are more important than the actual *sequence* those elements are in, the STL provides a special container called the `set`. The `set` container represents an arbitrary, unordered collection of elements and has good support for the following operations:

- Adding elements to the collection.
- Removing elements from the collection.
- Determining whether a particular element is in the collection.

To see the `set` in action, let's consider a modified version of the `RunProcess` function which uses a `set` instead of a `vector` to store its elements. This code is shown here (though you'll need to `#include <set>` for it to compile):

```
size_t RunProcess() {
    set<int> generated;

    while (true) {
        int nextValue = DieRoll();

        /* Check if this value has been rolled before. */
        if (generated.count(nextValue)) return generated.size() + 1;

        /* Otherwise, add this value to the set. */
        generated.insert(nextValue);
    }
}
```

Take a look at the changes we made to this code. To determine whether the most-recently-generated number has already been produced, we can use the simple syntax `generated.count(nextValue)` rather than the clunkier `for` loop from before. Also notice that to insert the new element into the `set`, we used the `insert` function rather than `push_back`.

The names of the functions on the `set` are indicative of the differences between the `set` and the `vector` and `deque`. When inserting an element into a `vector` or `deque`, we needed to specify where to put that element: at the end using `push_back`, at the beginning with `push_front`, or at some arbitrary position using `insert`. The `set` has only one function for adding elements – `insert` – which does not require us to specify where in the `set` the element should go. This makes sense, since the `set` is an inherently unordered collection of elements. Additionally, the `set` has no way to query elements at specific positions, since the elements of a `set` don't *have* positions. However, we can check whether an element exists in a `set` very simply using the `count` function, which returns `true` if the element exists and `false` otherwise.*

If you rerun this program using the updated code, you'll find that the program produces almost identical output (the randomness will mean that you're unlikely to get the same output twice). The only difference between the old code and the new code is the internal structure. Using the `set`, the code is easier to read and understand. In the next section, we'll probe the `set` in more detail and explore some of its other uses.

A Primer on `set`

The STL `set` container represents an unordered collection of elements that does not permit duplicates. Logically, a `set` is a collection of unique values that efficiently supports inserting and removing elements, as well as checking whether a particular element is contained in the `set`. Like the `vector` and `deque`, the `set` is a parameterized class. Thus we can speak of a `set<int>`, a `set<double>`, `set<string>`, etc. As with `vector` and `deque`, `set`s can only hold one type of element, so you cannot have a `set` that mixes and matches between `int`s and `string`s, for example. However, unlike the `vector` or `deque`, `set` can only store objects that can be compared using the `<` operator. This means that you can store all primitive types in a `set`, along with `string`s and other STL containers. However, you cannot store custom `struct`s inside of an STL `set`. For example, the following is illegal:

```
struct Point {
    double x, y;
};

set<Point> mySet; // Illegal, Point cannot be compared with <
```

This may seem like a somewhat arbitrary restriction. Logically, we could be able to gather up anything into an unordered collection. Why does it matter that those elements be comparable using `<`? The answer has to do with how the `set` is implemented behind the scenes. Internally, the `set` is layered on top of a *balanced binary tree*, a special data structure that naturally supports the `set`'s main operations. However, balanced binary trees can only be constructed on data sets where elements can be compared to one another, hence the restriction. Later in this text we'll see how to use a technique called *operator overloading* to make it possible to store objects of any type in an STL `set`, but for now you will need to confine yourself to primitives and other STL containers.

As we saw in the previous example, one of the most basic `set` operations is insertion using the `insert` function. Unlike the `deque` and `vector` `insert` functions, you do not need to specify a location for the new element. After all, a `set` represents an unordered collection, and specifying where an element should go in a `set` does not make any sense. Here is some sample code using `insert`:

* Technically speaking, `count` returns 1 if the element exists and 0 otherwise. For most purposes, though, it's safe to treat the function as though it returns a boolean true or false.

```
set<int> mySet;
mySet.insert(137);  // Now contains: 137
mySet.insert(42);   // Now contains: 42 137
mySet.insert(137);  // Now contains: 42 137
```

Notice in this last line that inserting a second copy of 137 into the `set` did not change the contents of the `set`. `set`s do not allow for duplicate elements.

To check whether a particular element is contained in an STL `set`, you can also use the `count` function, which returns 1 if the element is contained in the set and 0 otherwise. Using C++'s automatic conversion of nonzero values into `true` and zero values to `false`, you usually do not need to explicitly check whether `count` yields a one or zero and can rely on implicit conversions instead. For example:

```
if(mySet.count(137))
    cout << "137 is in the set." << endl;  // Printed
if(!mySet.count(500))
    cout << "500 is not in the set." << endl; // Printed
```

To remove an element from a `set`, you use the `erase` function. `erase` is a mirror to `insert`, and the two have very similar syntax. For example:

```
  mySet.erase(137); // Removes 137, if it exists.
```

The STL `set` also supports several operations common to all STL containers. You can remove all elements from a `set` using `clear`, check how many elements are present using `size`, etc. A full table of all `set` operations is presented later in this chapter.

Traversing Containers with Iterators

One of the most common operations we've seen in the course of working with the STL containers is *iteration*, traversing the contents of a container and performing some task on every element. For example, the following loop iterates over the contents of a `vector`, printing each element:

```
for (size_t h = 0; h < myVector.size(); ++h)
    cout << myVector[h] << endl;
```

We can similarly iterate over a `deque` as follows:

```
for (size_t h = 0; h < myDeque.size(); ++h)
    cout << myDeque[h] << endl;
```

The reason that we can use this convenient syntax to traverse the contents of the `vector` and `deque` is because the `vector` and `deque` represent linear sequences, and so it is possible to enumerate all possible indices in the container using the standard `for` loop. That is, we can iterate so easily over a `vector` or `deque` because we can look up the zeroth element, then the first element, then the second, etc. Unfortunately, this logic does not work on the STL `set`. Because the `set` does not have an ordering on its elements, it does not make sense to speak of the "zeroth element of a set," nor the "first element of a set," etc. To traverse the elements of a `set`, we will need to use a new concept, the *iterator*.

Every STL container presents a different means of storing data. `vector` and `deque` store data in an ordered list. `set` stores its data as an unordered collection. As you'll soon see, `map` encodes data as a collection of key/value pairs. But while each container stores its data in a different format, fundamentally, each container still stores data. Iterators provide a clean, consistent mechanism for accessing data stored in containers, irrespective of how that data may be stored. That is, the syntax for looking at `vector` data

with iterators is almost identical to the syntax for examining `set` and `deque` data with iterators. This fact is extremely important. For starters, it implies that once you've learned how to use iterators to traverse *any* container, you can use them to traverse *all* containers. Also, as you'll see, because iterators can traverse data stored in any container, they can be used to specify a collection of values in a way that masks how those values are stored behind-the-scenes.

So what exactly is an iterator? At a high level, an iterator is like a cursor in a text editor. Like a cursor, an iterator has a well-defined position inside a container, and can move from one character to the next. Also like a cursor, an iterator can be used to read or write a range of data one element at a time.

It's difficult to get a good feel for how iterators work without having a sense of how all the pieces fit together. Therefore, we'll get our first taste of iterators by jumping head-first into the idiomatic "loop over the elements of a container" `for` loop, then will clarify all of the pieces individually. Here is a sample piece of code that will traverse the elements of a `vector<int>`, printing each element out on its own line:

```
vector<int> myVector = /* ... some initialization ... */
for (vector<int>::iterator itr = myVector.begin();
     itr != myVector.end(); ++itr)
    cout << *itr << endl;
```

This code is perhaps the densest C++ we've encountered yet, so let's take a few minutes to dissect exactly what's going on here. The first part of the `for` loop is the statement

```
vector<int>::iterator itr = myVector.begin();
```

This line of code creates an object of type `vector<int>::iterator`, an iterator variable named `itr` that can traverse a `vector<int>`. Note that a `vector<int>::iterator` can only iterate over a `vector<int>`. If we wanted to iterate over a `vector<string>`, we would need to use a `vector<string>::iterator`, and if we wanted to traverse a `set<int>` we would have to use a `set<int>::iterator`. We then initialize the iterator to `myVector.begin()`. Every STL container class exports a member function `begin()` which yields an iterator pointing to the first element of that container. By initializing the iterator to `myVector.begin()`, we indicate to the C++ compiler that the `itr` iterator will be traversing elements of the container `myVector`.

Inside the body of the `for` loop, we have the line

```
cout << *itr << endl;
```

The strange-looking entity `*itr` is known as an *iterator dereference* and means "the element being iterated over by `itr`." As `itr` traverses the elements of the `vector`, it will proceed from one element to the next in sequence until all of the elements of the `vector` have been visited. At each step, the element being iterated over can be yielded by prepending a star to the name of the iterator. In the above context, we dereference the iterator to yield the current element of `myVector` being traversed, then print it out. We will discuss the nuances of iterator dereferences in more detail shortly.

Returning up to the `for` loop itself, notice that after each iteration we execute

```
++itr;
```

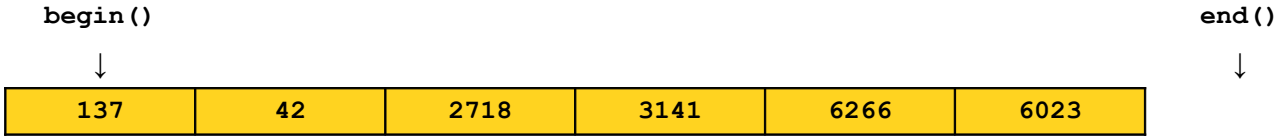
When applied to `int`s, the `++` operator is the increment operator; writing `++myInt` means "increment the value of the `myInt` variable." When applied to iterators, the `++` operator means "advance the iterator one step forward." Because the step condition of the `for` loop is `++itr`, this means that each iteration of the `for` loop will advance the iterator to the next element in the container, and eventually all elements will be

visited. Of course, at some point, we will have visited all of the elements in the `vector` and will need to stop iterating. To detect when an iterator has visited all of the elements, we loop on the condition that

```
itr != myVector.end();
```

Each STL container exports a special function called `end()` that returns an iterator to the element *one past the end of the container*. For example, consider the following `vector`:

In this case, the iterators returned by that `vector`'s `begin()` and `end()` functions would point to the following locations:



Notice that the `begin()` iterator points to the first element of the `vector`, while the `end()` iterator points to the slot one position past the end of the `vector`. This may seem strange at first, but is actually an excellent design decision. Recall the `for` loop from above, which iterates over the elements of a `vector`. This is reprinted below:

```
for (vector<int>::iterator itr = myVector.begin();
     itr != myVector.end(); ++itr)
    cout << *itr << endl;
```

Compare this to the more traditional loop you're used to, which also iterates over a `vector`:

```
for (size_t h = 0; h < myVector.size(); ++h)
    cout << myVector[h] << endl;
```

Because the `vector` is zero-indexed, if you were to look up the element in the `vector` at position `myVector.size()`, you would be reading a value not actually contained in the `vector`. For example, in a `vector` of five elements, the elements are stored at positions 0, 1, 2, 3, and 4. There is no element at position five, and trying to read an element there will result in undefined behavior. However, in the `for` loop to iterate over the contents of the `vector`, we still use the value of `myVector.size()` as the upper bound for the iteration, since the loop will cut off as soon as the iteration index reaches the value `myVector.size()`. This is identical to the behavior of the `end()` iterator in the iterator-based `for` loop. `myVector.end()` is never a valid iterator, but we use it as the loop upper bound because as soon as the `itr` iterator reaches `myVector.end()` the loop will terminate.

Part of the beauty of iterators is that the above `for` loop for iterating over the contents of a `vector` can trivially be adapted to iterate over just about any STL container class. For instance, if we want to iterate over the contents of a `deque<int>`, we could do so as follows:

```
deque<int> myDeque = /* ... some initialization ... */
for (deque<int>::iterator itr = myDeque.begin(); itr != myDeque.end(); ++itr)
    cout << *itr << endl;
```

This is *exactly* the same loop structure, though some of the types have changed (i.e. we've replaced `vector<int>::iterator` with `deque<int>::iterator`). However, the behavior is identical. This loop will traverse the contents of the `deque` in sequence, printing each element out as it goes.

Of course, at this point iterators may seem like a mere curiosity. Sure, we can use them to iterate over a `vector` or `deque`, but we already could do that using a more standard `for` loop. The beauty of iterators is that they work on any STL container, including the `set`. If we have a `set` of elements we wish to traverse, we can do so using the following syntax:

```
set<int> mySet = /* ... some initialization ... */
for (set<int>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << endl;
```

Again, notice that the structure of the loop is the same as before. Only the types have changed.

One crucial detail we've ignored up to this point is in what order the elements of a `set` will be traversed. When using the `vector` or `deque` there is a natural iteration order (from the start of the sequence to the end), but when using the STL `set` the idea of ordering is a bit more vague. However, iteration order over a `set` is well-specified. When traversing `set` elements via an iterator, the elements will be visited in sorted order, starting with the smallest element and ending with the largest. This is in part why the STL `set` can only store elements comparable using the less-than operator: there is no well-defined "smallest" or "biggest" element of a `set` if the elements cannot be compared. To see this in action, consider the following code snippet:

```
/* Generate ten random numbers */
set<int> randomNumbers;
for (size_t k = 0; k < 10; ++k)
    randomNumbers.insert(rand());

/* Print them in sorted order. */
for (set<int>::iterator itr = randomNumbers.begin();
     itr != randomNumbers.end(); ++itr)
    cout << *itr << endl;
```

This will print different outputs on each run, since the program generates and stores random numbers. However, the values will always be in sorted order. For example:

```
137 2718 3141 4103 5422 6321 8938 10299 12003 16554
```

Spotlight on Iterators

As you just saw, there are three major operations on iterators:

- *Dereferencing* the iterator to read a value.
- *Advancing* the iterator from one position to the next.
- *Comparing* two iterators for equality.

Iterator dereferencing is a particularly important operation, and so before moving on we'll take a few minutes to explore this in more detail.

As you've seen so far, iterators can be used to read the values of a container indirectly. However, iterators can also be used to *write* the values of a container indirectly as well. For example, here is a simple `for` loop to set all of the elements of a `vector<int>` to 137:

```
for (vector<int>::iterator itr = myVector.begin();
     itr != myVector.end(); ++itr)
    *itr = 137;
```

This is your first glimpse of the true power of iterators. Because iterators give a means for reading and writing container elements indirectly, it is possible to write functions that operate on data from any container class by manipulating iterators from that container class. These functions are called *STL algorithms* and will be discussed in more detail next chapter.

Up to this point, when working with iterators, we have restricted ourselves to STL containers that hold primitive types. That is, we've talked about `vector<int>` and `set<int>`, but not, say, `vector<string>`. All of the syntax that we have seen so far for containers holding primitive types are applicable to containers holding objects. For example, this loop will correctly print out all of the `string`s in a `set<string>`:

```
for (set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << endl;
```

However, let's suppose that we want to iterate over a `set<string>` printing out the *lengths* of the `string`s in that `set`. Unfortunately, the following syntax will *not* work:

```
for (set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr.length() << endl; // Error: Incorrect syntax!
```

The problem with this code is that the C++ compiler interprets it as

```
*(itr.length())
```

Instead of

```
(*itr).length()
```

That is, the compiler tries to call the nonexistent `length()` function on the iterator and to dereference *that*, rather than dereferencing the iterator and then invoking the `length()` function on the resulting value. This is a subtle yet important difference, so make sure that you take some time to think it through before moving on.

To fix this problem, all STL iterators support and operator called the *arrow operator* that allows you to invoke member functions on the element currently being iterated over. For example, to print out the lengths of all of the `string`s in a `set<string>`, the proper syntax is

```
for (set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << itr->length() << endl;
```

We will certainly encounter the arrow operator more as we continue our treatment of the material, so make sure that you understand its usage before moving on.

Defining Ranges with Iterators

Recall for a moment the standard "loop over a container" `for` loop:

```
set<int> mySet = /* ... some initialization ... */
for (set<int>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << endl;
```

If you'll notice, this loop is bounded by two iterators – `mySet.begin()`, which specifies the first element to iterate over, and `mySet.end()`, which defines the element one past the end of the iteration range. This raises an interesting point about the *duality* of iterators. A *single* iterator points to a single position in a container class and represents a way to read or write that value indirectly. A *pair* of iterators defines two positions and consequently defines a *range* of elements. In particular, given two iterators `start` and `stop`, these iterators define the range of elements beginning with `start` and ending one position before `stop`. Using mathematical notation, the range of elements defined by `start` and `stop` spans [`start`, `stop`).

So far, the only ranges we've considered have been those of the form [`begin()`, `end()`) consisting of all of the elements of a container. However, as we begin moving on to progressively more complicated programs, we will frequently work on ranges that do not span all of a container. For example, we might be interested in iterating over only the first half of a container, or perhaps just a slice of elements in a container meeting some property.

If you'll recall, the STL `set` stores its elements in sorted order, a property that guarantees efficient lookup and insertion. Serendipitously, this allows us to efficiently iterate over a slice out of a `set` whose values are bounded between some known limits. The `set` exports two functions, `lower_bound` and `upper_bound`, that can be used to iterate over the elements in a `set` that are within a certain range. `lower_bound` accepts a value, then returns an iterator to the first element in the `set` greater than or equal to that value. `upper_bound` similarly accepts a value and returns an iterator to the first element in the `set` that is strictly greater than the specified element. Given a closed range [`lower`, `upper`], we can iterate over that range by using `lower_bound` to get an iterator to the first element no less than `lower` and iterating until we reach the value returned by `upper_bound`, the first element strictly greater than `upper`. For example, the following loop iterates over all elements in the set in the range [10, 100]:

```
set<int>::iterator stop = mySet.upper_bound(100);
for(set<int>::iterator itr = mySet.lower_bound(10); itr != stop; ++itr)
     /* ... perform tasks... */
```

Part of the beauty of `upper_bound` and `lower_bound` is that it doesn't matter whether the elements specified as arguments to the functions actually exist in the `set`. For example, suppose that we run the above `for` loop on a `set` containing all the odd numbers between 3 and 137. In this case, neither 10 nor 100 are contained in the `set`. However, the code will still work correctly. The `lower_bound` function returns an iterator to the first element *at least as large* as its argument, and in the `set` of odd numbers would return an iterator to the element 11. Similarly, `upper_bound` returns an iterator to the first element *strictly greater* than its argument, and so would return an iterator to the element 101.

Summary of set

The following table lists some of the most important `set` functions. Again, we haven't covered `const` yet, so for now it's safe to ignore it. We also haven't covered `const_iterator`s, but for now you can just treat them as iterators that can't write any values.

Constructor: `set<T>()`	`set<int> mySet;` Constructs an empty `set`.
Constructor: `set<T>(const set<T>& other)`	`set<int> myOtherSet = mySet;` Constructs a `set` that's a copy of another `set`.

Constructor: `set<T>(InputIterator start, InputIterator stop)`	`set<int> mySet(myVec.begin(), myVec.end());` Constructs a `set` containing copies of the elements in the range [`start`, `stop`). Any duplicates are discarded, and the elements are sorted. Note that this function accepts iterators from any source.
`size_type size() const`	`int numEntries = mySet.size();` Returns the number of elements contained in the `set`.
`bool empty() const`	`if(mySet.empty()) { ... }` Returns whether the `set` is empty.
`void clear()`	`mySet.clear();` Removes all elements from the `set`.
`iterator begin()` `const_iterator begin() const`	`set<int>::iterator itr = mySet.begin();` Returns an iterator to the start of the `set`. Be careful when modifying elements in-place.
`iterator end()` `const_iterator end()`	`while(itr != mySet.end()) { ... }` Returns an iterator to the element one past the end of the final element of the `set`.

`pair<iterator, bool>`
`insert(const T& value)`
`void insert(InputIterator begin,`
`InputIterator end)`
	`mySet.insert(4);`
	`mySet.insert(myVec.begin(), myVec.end());`
	
	The first version inserts the specified value into the `set`. The return type is a `pair` containing an iterator to the element and a `bool` indicating whether the element was inserted successfully (`true`) or if it already existed (`false`). The second version inserts the specified range of elements into the `set`, ignoring duplicates.

`iterator find(const T& element)`
`const_iterator`
`find(const T& element) const`
	`if(mySet.find(0) != mySet.end()) { ... }`
	
	Returns an iterator to the specified element if it exists, and `end` otherwise.

`size_type count(const T& item) const`	`if(mySet.count(0)) { ... }` Returns 1 if the specified element is contained in the `set`, and 0 otherwise.

`size_type erase(const T& element)`
`void erase(iterator itr);`
`void erase(iterator start,`
`iterator stop);`
	`if(mySet.erase(0)) {...} // 0 was erased`
	`mySet.erase(mySet.begin());`
	`mySet.erase(mySet.begin(), mySet.end());`
	
	Removes an element from the `set`. In the first version, the specified element is removed if found, and the function returns 1 if the element was removed and 0 if it wasn't in the `set`. The second version removes the element pointed to by `itr`. The final version erases elements in the range [`start`, `stop`).

`iterator lower_bound(const T& value)`	`itr = mySet.lower_bound(5);` Returns an iterator to the first element that is greater than or equal to the specified value. This function is useful for obtaining iterators to a range of elements, especially in conjunction with `upper_bound`.
`iterator upper_bound(const T& value)`	`itr = mySet.upper_bound(100);` Returns an iterator to the first element that is greater than the specified value. Because this element must be strictly greater than the specified value, you can iterate over a range until the iterator is equal to `upper_bound` to obtain all elements less than or equal to the parameter.

A Useful Helper: `pair`

We have just finished our treatment of the `set` and are about to move on to one of the STL's most useful containers, the `map`. However, before we can cover the `map` in any detail, we must first make a quick diversion to a useful helper class, the `pair`.

`pair` is a parameterized class that simply holds two values of arbitrary type. `pair`, defined in `<utility>`, accepts two template arguments and is declared as

```
pair<TypeOne, TypeTwo>
```

`pair` has two fields, named `first` and `second`, which store the values of the two elements of the pair; `first` is a variable of type `TypeOne`, `second` of type `TypeTwo`. For example, to make a `pair` that can hold an `int` and a `string`, we could write

```
pair<int, string> myPair;
```

We could then access the `pair`'s contents as follows

```
pair<int, string> myPair;
myPair.first  =  137;
myPair.second = "C++ is awesome!";
```

In some instances, you will need to create a `pair` on-the-fly to pass as a parameter (especially to the `map`'s `insert`). You can therefore use the `make_pair` function as follows:

```
pair<int, string> myPair = make_pair(137, "string!");
```

Interestingly, even though we didn't specify what type of `pair` to create, the `make_pair` function was able to deduce the type of the `pair` from the types of the elements. This has to do with how C++ handles function templates and we'll explore this in more detail later.

Representing Relationships with `map`

One of the most important data structures in modern computer programming is the *map*, a way of tagging information with some other piece of data. The inherent idea of a *mapping* should not come as a surprise to you. Almost any entity in the real world has extra information associated with it. For example, days of the year have associated events, items in your refrigerator have associated expiration dates, and people you know have associated titles and nicknames. The `map` STL container manages a relationship between a

set of *keys* and a set of *values*. For example, the keys in the `map` might be email addresses, and the values the names of the people who own those email addresses. Alternatively, the keys might be longitude/latitude pairs, and the values the name of the city that resides at those coordinates.

Data in a `map` is stored in key/value pairs. Like the `set`, these elements are unordered. Also like the `set`, it is possible to query the map for whether a particular *key* exists in the `map` (note that the check is "does key X exist?" rather than "does *key/value pair* X exist?"). Unlike the `set`, however, the `map` also allows clients to ask "what is the value associated with key X?" For example, in a `map` from longitude/latitude pairs to city names, it is possible to give a properly-constructed pair of coordinates to the map, then get back which city is at the indicated location (if such a city exists).

The `map` is unusual as an STL container because unlike the `vector`, `deque`, and `set`, the `map` is parameterized over two types, the type of the key and the type of the value. For example, to create a `map` from `string`s to `int`s, you would use the syntax

```
map<string, int> myMap;
```

Like the STL `set`, behind the scenes the `map` is implemented using a balanced binary tree. This means that the *keys* in the `map` must be comparable using the less-than operator. Consequently, you won't be able to use your own custom `struct`s as keys in an STL `map`. However, the *values* in the map needn't be comparable, so it's perfectly fine to map from `string`s to custom `struct` types. Again, when we cover operator overloading later in this text, you will see how to store arbitrary types as keys in an STL `map`.

The `map` supports many different operations, of which four are key:

- Inserting a new key/value pair.
- Checking whether a particular key exists.
- Querying which value is associated with a given key.
- Removing an existing key/value pair.

We will address each of these in turn.

In order for a `map` to be useful, we will need to populate it with a collection of key/value pairs. There are two ways to insert key/value pairs into the map. The simplest way to insert key/value pairs into a `map` is to user the element selection operator (square brackets) to implicitly add the pair, as shown here:

```
map<string, int> numberMap;
numberMap["zero"] = 0;
numberMap["one"] = 1;
numberMap["two"] = 2;
/* ... etc. ... */
```

This code creates a new `map` from `string`s to `int`s. It then inserts the key `"zero"` which maps to the number zero, the key `"one"` which maps to the number one, etc. Notice that this is a major way in which the `map` differs from the `vector`. Indexing into a `vector` into a nonexistent position will cause undefined behavior, likely a full program crash. Indexing into a `map` into a nonexistent key implicitly creates a key/value pair.

The square brackets can be used both to insert new elements into the `map` and to query the `map` for the values associated with a particular key. For example, assuming that `numberMap` has been populated as above, consider the following code snippet:

```
cout << numberMap["zero"] << endl;
cout << numberMap["two"] * numberMap["two"] << endl;
```

The output of this program is

```
0
4
```

On the first line, we query the `numberMap` map for the value associated with the key `"zero"`, which is the number zero. The second line looks up the value associated with the key `"two"` and multiplies it with itself. Since `"two"` maps to the number two, the output is four.

Because the square brackets both query and create key/value pairs, you should use care when looking values up with square brackets. For example, given the above number map, consider this code:

```
cout << numberMap["xyzzy"] << endl;
```

Because `"xyzzy"` is not a key in the `map`, this implicitly creates a key/value pair with `"xyzzy"` as the key and zero as the value. (Like the `vector` and `deque`, the `map` will zero-initialize any primitive types used as values). Consequently, this code will output

```
0
```

and will change the `numberMap` map so that it now has `"xyzzy"` as a key. If you want to look up a key/value pair without accidentally adding a new key/value pair to the `map`, you can use the `map`'s `find` member function. `find` takes in a key, then returns an `iterator` that points to the key/value pair that has the specified key. If the key does not exist, `find` returns the `map`'s `end()` iterator as a sentinel. For example:

```
map<string, int>::iterator itr = numberMap.find("xyzzy");
if (itr == numberMap.end())
    cout << "Key does not exist." << endl;
else
    /* ... */
```

When working with an STL `vector`, `deque`, or `set`, iterators simply iterated over the contents of the container. That is, a `vector<int>::iterator` can be dereferenced to yield an `int`, while a `set<string>::iterator` dereferences to a `string`. `map` iterators are slightly more complicated because they dereference to a key/value pair. In particular, if you have a `map<KeyType, ValueType>`, then the iterator will dereference to a value of type

```
pair<const KeyType, ValueType>
```

This is a `pair` of an immutable key and a mutable value. We have not talked about the `const` keyword yet, but it means that keys in a `map` cannot be changed after they are set (though they can be removed). The values associated with a key, on the other hand, can be modified.

Because `map` iterators dereference to a `pair`, you can access the keys and values from an iterator as follows:

```
map<string, int>::iterator itr = numberMap.find("xyzzy");
if (itr == numberMap.end())
    cout << "Key does not exist." << endl;
else
    cout << "Key " << itr->first << " has value " << itr->second << endl;
```

That is, to access the key from a `map` iterator, you use the arrow operator to select the `first` field of the `pair`. The value is stored in the `second` field. This naturally segues into the stereotypical "iterate over the elements of a `map` loop," which looks like this:

```
for (map<string, int>::iterator itr = myMap.begin(); itr != myMap.end(); ++itr)
    cout << itr->first << ": " << itr->second << endl;
```

When iterating over a `map`, the key/value pairs will be produced sorted by key from lowest to highest. This means that if we were to iterate over the `numberMap` map from above printing out key/value pairs, the output would be

```
one: 1
two: 2
zero: 0
```

Since the keys are `string`s which are sorted in alphabetical order.

You've now seen how to insert, query, and iterate over key/value pairs. Removing key/value pairs from a `map` is also fairly straightforward. To do so, you use the `erase` function as follows:

```
myMap.erase("key");
```

That is, the erase function accepts a *key*, then removes the key/value pair from the `map` that has that key (if it exists).

As with all STL containers, you can remove all key/value pairs from a `map` using the `clear` function, determine the number of key/value pairs using the `size` function, etc. There are a few additional operations on a `map` beyond these basic operations, some of which are covered in the next section.

`insert` and How to Avoid It

As seen above, you can use the square brackets operator to insert and update key/value pairs in the `map`. However, there is another mechanism for inserting key/value pairs: `insert`. Like the `set`'s `insert` function, you need only specify what to insert, since the `map`, like the `set`, does not store values in a particular order. However, because the `map` stores elements as key/value pairs, the parameter to the `insert` function should be a `pair` object containing the key and the value. For example, the following code is an alternative means of populating the `numberMap` map:

```
map<string, int> numberMap;
numberMap.insert(make_pair("zero", 0));
numberMap.insert(make_pair("one", 1));
numberMap.insert(make_pair("two", 2));
/* ... */
```

There is one key difference between the `insert` function and the square brackets. Consider the following two code snippets:

```
/* Populate a map using [ ] */
map<string, string> one;
one["C++"] = "sad";
one["C++"] = "happy";

/* Populate a map using insert */
map<string, string> two;
two.insert(make_pair("C++", "sad"));
two.insert(make_pair("C++", "happy"));
```

In the first code snippet, we create a `map` from `string`s to `string`s called `one`. We first create a key/value pair mapping `"C++"` to `"sad"`, and then overwrite the value associated with `"C++"` to `"happy"`. After this code executes, the `map` will map the key "C++" to the value "happy", since in the second line the value was overwritten. In the second code snippet, we call `insert` twice, once inserting the key `"C++"` with the value `"sad"` and once inserting the key `"C++"` with the value `"happy"`. When this code executes, the `map` will end up holding one key/value pair: `"C++"` mapping to `"sad"`. Why is this the case?

Like the STL `set`, the `map` stores a unique set of keys. While multiple keys may map to the same value, there can only be one key/value pair for any given key. When inserting and updating keys with the square brackets, any updates made to the `map` are persistent; writing code to the effect of `myMap[key] = value` ensures that the map contains the key `key` mapping to value `value`. However, the `insert` function is not as forgiving. If you try to insert a key/value pair into a `map` using the `insert` function and the `key` already exists, the `map` will not insert the key/value pair, nor will it update the value associated with the existing key. To mitigate this, the `map`'s `insert` function returns a value of type `pair<iterator, bool>`. The `bool` value in the `pair` indicates whether the `insert` operation succeeded; a result of `true` means that the key/value pair was added, while a result of `false` means that the key already existed. The `iterator` returned by the `insert` function points to the key/value `pair` in the `map`. If the key/value pair was newly-added, this iterator points to the newly-inserted value, and if a key/value pair already exists the iterator points to the existing key/value pair that prevented the operation from succeeding. If you want to use `insert` to insert key/value pairs, you can write code to the following effect:

```
/* Try to insert normally. */
pair<map<string, int>::iterator, bool> result =
    myMap.insert(make_pair("STL", 137));

/* If insertion failed, manually set the value. */
if(!result.second)
    result.first->second = 137;
```

In the last line, the expression `result.first->second` is the value of the existing entry, since `result.first` yields an iterator pointing to the entry, so `result.first->second` is the value field of the iterator to the entry. As you can see, the `pair` can make for tricky, unintuitive code.

If `insert` is so inconvenient, why even bother with it? Usually, you won't, and will use the square brackets operator instead. However, when working on an existing codebase, you are extremely likely to run into the `insert` function, and being aware of its somewhat counterintuitive semantics will save you many hours of frustrating debugging.

`map` Summary

The following table summarizes the most important functions on the STL `map` container. Feel free to ignore `const` and `const_iterator`s; we haven't covered them yet.

Constructor: `map<K, V>()`	`map<int, string> myMap;` Constructs an empty `map`.
Constructor: `map<K, V>(const map<K, V>& other)`	`map<int, string> myOtherMap = myMap;` Constructs a `map` that's a copy of another `map`.
Constructor: `map<K, V>(InputIterator start, InputIterator stop)`	`map<string, int> myMap(myVec.begin(), myVec.end());` Constructs a `map` containing copies of the elements in the range [`start`, `stop`). Any duplicates are discarded, and the elements are sorted. Note that this function accepts iterators from any source, but they must be iterators over pairs of keys and values.
`size_type size() const`	`int numEntries = myMap.size();` Returns the number of elements contained in the `map`.
`bool empty() const`	`if(myMap.empty()) { ... }` Returns whether the `map` is empty.
`void clear()`	`myMap.clear();` Removes all elements from the `map`.
`iterator begin()` `const_iterator begin() const`	`map<int>::iterator itr = myMap.begin();` Returns an iterator to the start of the `map`. Remember that iterators iterate over pairs of keys and values.
`iterator end()` `const_iterator end()`	`while(itr != myMap.end()) { ... }` Returns an iterator to the element one past the end of the final element of the `map`.

`pair<iterator, bool>`
`insert(const pair<const K, V>& value)`
`void insert(InputIterator begin,`
`InputIterator end)`
	`myMap.insert(make_pair("STL", 137));`
	`myMap.insert(myVec.begin(), myVec.end());`
	
	The first version inserts the specified key/value pair into the `map`. The return type is a `pair` containing an iterator to the element and a `bool` indicating whether the element was inserted successfully (`true`) or if it already existed (`false`). The second version inserts the specified range of elements into the `map`, ignoring duplicates.

`V& operator[] (const K& key)`	`myMap["STL"] = "is awesome";` Returns the value associated with the specified key, if it exists. If not, a new key/value pair will be created and the value initialized to zero (if it is a primitive type) or the default value (for non-primitive types).

`iterator find(const K& element)` `const_iterator find(const K& element) const`	`if(myMap.find(0) != myMap.end()) { ... }` Returns an iterator to the key/value pair having the specified key if it exists, and `end` otherwise.
`size_type count(const K& item) const`	`if(myMap.count(0)) { ... }` Returns 1 if some key/value pair in the `map` has specified key and 0 otherwise.

`size_type erase(const K& element)`
`void erase(iterator itr);`
`void erase(iterator start,`
`iterator stop);`
	`if(myMap .erase(0)) {...}`
	`myMap.erase(myMap.begin());`
	`myMap.erase(myMap.begin(), myMap.end());`
	
	Removes a key/value pair from the `map`. In the first version, the key/value pair having the specified key is removed if found, and the function returns 1 if a pair was removed and 0 otherwise. The second version removes the element pointed to by `itr`. The final version erases elements in the range [`start`, `stop`).

`iterator lower_bound(const K& value)`	`itr = myMap.lower_bound(5);` Returns an iterator to the first key/value pair whose key is greater than or equal to the specified value. This function is useful for obtaining iterators to a range of elements, especially in conjunction with `upper_bound`.
`iterator upper_bound(const K& value)`	`itr = myMap.upper_bound(100);` Returns an iterator to the first key/value pair whose key is greater than the specified value. Because this element must be strictly greater than the specified value, you can iterate over a range until the iterator is equal to `upper_bound` to obtain all elements less than or equal to the parameter.

Extended Example: Keyword Counter

To give you a better sense for how `map` and `set` can be used in practice, let's build a simple application that brings them together: a *keyword counter*. C++, like most programming languages, has a set of *reserved words*, keywords that have a specific meaning to the compiler. For example, the keywords for the primitive types `int` and `double` are reserved words, as are the `switch`, `for`, `while`, `do`, and `if` keywords used for control flow. For your edification, here's a complete list of the reserved words in C++:

```
and             continue        goto            public          try
and_eq          default         if              register        typedef
asm             delete          inline          reinterpret_cast typeid
auto            do              int             return          typename
bitand          double          long            short           union
bitor           dynamic_cast    mutable         signed          unsigned
bool            else            namespace       sizeof          using
break           enum            new             static          virtual
case            explicit        not             static_cast     void
catch           export          not_eq          struct          volatile
char            extern          operator        switch          wchar_t
class           false           or              template        while
compl           float           or_eq           this            xor
const           for             private         throw           xor_eq
const_cast      friend          protected       true
```

We are interested in answering the following question: given a C++ source file, how many times does each reserved word come up? This by itself might not be particularly enlightening, but in some cases it's interesting to see how often (or infrequently) the keywords come up in production code.

We will suppose that we are given a file called `keywords.txt` containing all of C++'s reserved words. This file is structured such that every line of the file contains one of C++'s reserved words. Here's the first few lines:

***File:* `keywords.txt`**

```
and
and_eq
asm
auto
bitand
bitor
bool
break
...
```

Given this file, let's write a program that prompts the user for a filename, loads the file, then reports the frequency of each keyword in that file. For readability, we'll only print out a report on the keywords that actually occurred in the file. To avoid a major parsing headache, we'll count keywords wherever they appear, even if they're in comments or contained inside of a string.

Let's begin writing this program. We'll use a top-down approach, breaking the task up into smaller subtasks which we will implement later on. Here is one possible implementation of the `main` function:

```
#include <iostream>
#include <string>
#include <fstream>
#include <map>
using namespace std;

/* Function: OpenUserFile(ifstream& fileStream);
 * Usage: OpenUserFile(myStream);
 * ------------------------------------------------
 * Prompts the user for a filename until a valid filename
 * is entered, then sets fileStream to read from that file.
 */
void OpenUserFile(ifstream& fileStream);

/* Function: GetFileContents(ifstream& file);
 * Usage: string contents = GetFileContents(ifstream& file);
 * ------------------------------------------------
 * Returns a string containing the contents of the file passed
 * in as a parameter.
 */
string GetFileContents(ifstream& file);

/* Function: GenerateKeywordReport(string text);
 * Usage: map<string, size_t> keywords = GenerateKeywordReport(contents);
 * ------------------------------------------------
 * Returns a map from keywords to the frequency at which those keywords
 * appear in the input text string.  Keywords not contained in the text will
 * not appear in the map.
 */
map<string, size_t> GenerateKeywordReport(string contents);
```

```
    int main() {
        /* Prompt the user for a valid file and open it as a stream. */
        ifstream input;
        OpenUserFile(input);

        /* Generate the report based on the contents of the file. */
        map<string, size_t> report = GenerateKeywordReport(GetFileContents(input));

        /* Print a summary. */
        for (map<string, size_t>::iterator itr = report.begin();
             itr != report.end(); ++itr)
            cout << "Keyword " << itr->first << " occurred "
                 << itr->second << " times." << endl;
}
```

The breakdown of this program is as follows. First, we prompt the user for a file using the `OpenUserFile` function. We then obtain the file contents as a string and pass it into `GenerateKeywordReport`, which builds us a map from `string`s of the keywords to `size_t`s representing the frequencies. Finally, we print out the contents of the `map` in a human-readable format. Of course, we haven't implemented any of the major functions that this program will use, so this program won't link, but at least it gives a sense of where the program is going.

Let's begin implementing this code by writing the `OpenUserFile` function. Fortunately, we've already written this function last chapter in the *Snake* example. The code for this function is reprinted below:

```
    void OpenUserFile(ifstream& input) {
        while(true) {
            cout << "Enter filename: ";
            string filename = GetLine();

            input.open(filename.c_str()); // See Chapter 3 for .c_str().
            if(input.is_open()) return;

            cout << "Sorry, I can't find the file " << filename << endl;
            input.clear();
        }
    }
```

Here, the `GetLine()` function is from the chapter on streams, and is implemented as

```
    string GetLine() {
        string line;
        getline(input, line);
        return line;
    }
```

Let's move on to the next task, reading the file contents into a `string`. This can be done in a few lines of code using the streams library. The idea is simple: we'll maintain a `string` containing all of the file contents encountered so far, and continuously concatenate on the next line of the file (which we'll read with the streams library's handy `getline` function). This is shown here:

```
string GetFileContents(ifstream& input) {
    /* String which will hold the file contents. */
    string result;

    /* Keep reading a line of the file until no data remains. *
    string line;
    while (getline(input, line))
        result += line + "\n"; // Add the newline character; getline removes it

    return result;
}
```

All that remains at this point is the `GenerateKeywordReport` function, which ends up being most of the work. The basic idea behind this function is as follows:

- Load in the list of keywords.
- For each word in the file:
  - If it's a keyword, increment the keyword count appropriately.
  - Otherwise, ignore it.

We'll take this one step at a time. First, let's load in the list of keywords. But how should we store those keywords? We'll be iterating over words from the user's file, checking at each step whether the given word is a keyword. This means that we will want to store the keywords in a way where we can easily query whether a string is or is not contained in the list of keywords. This is an ideal spot for a `set`, which is optimized for these operations. We can therefore write a function that looks like this to read the reserved words list into a `set`:

```
set<string> LoadKeywords() {
    ifstream input("keywords.txt"); // No error checking for brevity's sake
    set<string> result;

    /* Keep reading strings out of the file until we cannot read any more.
     * After reading each string, store it in the result set.  We can either
     * use getline or the stream extraction operator here, but the stream
     * extraction operator is a bit more general.
     */
    string keyword;
    while (input >> keyword)
        result.insert(keyword);

    return result;
}
```

We now have a way to read in the set of keywords, and can move on to our next task: reading all of the words out of the file and checking whether any of them are reserved words. This is surprisingly tricky. We are given a string, a continuous sequence of characters, and from this string want to identify where each "word" is. How are we to do this? There are many options at our disposal (we'll see a heavy-duty way to do this at the end of the chapter), but one particularly elegant method is to harness a `stringstream`. If you'll recall, the `stringstream` class is a stream object that can build and parse strings using standard stream operations. Further recall that when reading string data out of a stream using the stream extraction operator, the read operation will proceed up until it encounters whitespace or the end of the stream. That is, if we had a stream containing the text

```
This, dear reader, is a string.
```

If we were to read data from the stream one string at a time, we would get back the strings

```
This,
dear
reader,
is
a
string.
```

In that order. As you can see, the input is broken apart at whitespace boundaries, rather than word boundaries. However, whenever we encounter a word that does not have punctuation immediately on either side, the string is parsed correctly. This suggests a particularly clever trick. We will modify the full text of the file by replacing all punctuation characters with whitespace characters. Having performed this manipulation, if we parse the file contents using a `stringstream`, each string handed back to us will be a complete word.

Let's write a function, `PreprocessString`, which accepts as input a `string` by reference, then replaces all punctuation characters in that string with the space character. To help us out, we have the `<cctype>` header, which exports the `ispunct` function. `ispunct` takes in a single character, then returns whether or not it is a punctuation character. Unfortunately, `ispunct` treats underscores as punctuation, which will cause problems for some reserved words (for example, `static_cast`), and so we'll need to special-case it. The `PreprocessString` function is as follows:

```
void PreprocessString(string& text) {
    for (size_t k = 0; k < text.size(); ++k)
        if (ispunct(text[k]) && text[k] != '_') // If we need to change it...
            text[k] = ' '; // ... replace it with a space.
}
```

Combining this function with `LoadKeywords` gives us this partial implementation of `GenerateKeywordReport`:

```
map<string, size_t> GenerateKeywordReport(string fileContents) {
    /* Load the set of keywords from disk. */
    set<string> keywords = LoadKeywords();

    /* Preprocess the string to allow for easier parsing. */
    PreprocessString(fileContents);

    /* ... need to fill this in ... */
}
```

All that's left to do now is tokenize the string into individual words, then build up a frequency map of each keyword. To do this, we'll funnel the preprocessed file contents into a `stringstream` and use the prototypical stream reading loop to break it apart into individual words. This can be done as follows:

```
map<string, size_t> GenerateKeywordReport(string fileContents) {
    /* Load the set of keywords from disk. */
    set<string> keywords = LoadKeywords();

    /* Preprocess the string to allow for easier parsing. */
    PreprocessString(fileContents);

    /* Populate a stringstream with the file contents. */
    stringstream tokenizer;
    tokenizer << fileContents;

    /* Loop over the words in the file, building up the report. */
    map<string, size_t> result;

    string word;
    while (tokenizer >> word)
        /* ... process word here ... */
}
```

Now that we have a loop for extracting single words from the input, we simply need to check whether each word is a reserved word and, if so, to make a note of it. This is done here:

```
map<string, size_t> GenerateKeywordReport(string fileContents) {
    /* Load the set of keywords from disk. */
    set<string> keywords = LoadKeywords();

    /* Preprocess the string to allow for easier parsing. */
    PreprocessString(fileContents);

    /* Populate a stringstream with the file contents. */
    stringstream tokenizer;
    tokenizer << fileContents;

    /* Loop over the words in the file, building up the report. */
    map<string, size_t> result;

    string word;
    while (tokenizer >> word)
        if (keywords.count(word))
            ++ result[word];

    return result;
}
```

Let's take a closer look at what this code is doing. First, we check whether the current word is a keyword by using the `set`'s `count` function. If so, we increment the count of that keyword in the file by writing `++result[word]`. This is a surprisingly compact line of code. If the keyword has not been counted before, then `++result[word]` will implicitly create a new key/value pair using that keyword as the key and initializing the associated value to zero. The `++` operator then kicks in, incrementing the value by one. Otherwise, if the key already existed in the `map`, the line of code will retrieve the value, then increment it by one. Either way, the count is updated appropriately, and the `map` will be populated correctly.

We now have a working implementation of the `GenerateKeywordReport` function, and, combined with the rest of the code we've written, we now have a working implementation of the keyword counting program. As an amusing test, the result of running this program on itself is as follows:

```
Keyword for occurred 3 times.
Keyword if occurred 3 times.
Keyword int occurred 1 times.
Keyword namespace occurred 1 times.
Keyword return occurred 6 times.
Keyword true occurred 1 times.
Keyword using occurred 1 times.
Keyword void occurred 2 times.
Keyword while occurred 4 times.
```

How does this compare to production code? For reference, here is the output of the program when run on the monster source file `nsCSSFrameConstructor.cpp`, an 11,000+ line file from the Mozilla Firefox source code:*

```
Keyword and occurred 268 times.
Keyword auto occurred 2 times.
Keyword break occurred 58 times.
Keyword case occurred 66 times.
Keyword catch occurred 2 times.
Keyword char occurred 4 times.
Keyword class occurred 10 times.
Keyword const occurred 149 times.
Keyword continue occurred 11 times.
Keyword default occurred 8 times.
Keyword delete occurred 6 times.
Keyword do occurred 99 times.
Keyword else occurred 135 times.
Keyword enum occurred 1 times.
Keyword explicit occurred 4 times.
Keyword extern occurred 4 times.
Keyword false occurred 12 times.
Keyword float occurred 15 times.
Keyword for occurred 292 times.
Keyword friend occurred 3 times.
Keyword if occurred 983 times.
Keyword inline occurred 86 times.
Keyword long occurred 5 times.
Keyword namespace occurred 5 times.
Keyword new occurred 59 times.
Keyword not occurred 145 times.
Keyword operator occurred 1 times.
Keyword or occurred 108 times.
Keyword private occurred 2 times.
Keyword protected occurred 1 times.
Keyword public occurred 5 times.
Keyword return occurred 452 times.
Keyword sizeof occurred 3 times.
Keyword static occurred 118 times.
Keyword static_cast occurred 20 times.
Keyword struct occurred 8 times.
Keyword switch occurred 4 times.
Keyword this occurred 205 times.
Keyword true occurred 14 times.
Keyword try occurred 10 times.
Keyword using occurred 6 times.
Keyword virtual occurred 1 times.
Keyword void occurred 82 times.
Keyword while occurred 53 times.
```

As you can see, we have quite a lot of C++ ground to cover – just look at all those keywords we haven't covered yet!

* As of April 12, 2010

Multicontainers

The STL provides two special "multicontainer" classes, `multimap` and `multiset`, that act as `map`s and `set`s except that the values and keys they store are not necessarily unique. That is, a `multiset` could contain several copies of the same value, while a `multimap` might have duplicate keys associated with different values.

`multimap` and `multiset` (declared in `<map>` and `<set>`, respectively) have identical syntax to `map` and `set`, except that some of the functions work slightly differently. For example, the `count` function will return the number of copies of an element an a multicontainer, not just a binary zero or one. Also, while `find` will still return an iterator to an element if it exists, the element it points to is not guaranteed to be the only copy of that element in the multicontainer. Finally, the `erase` function will erase *all* copies of the specified key or element, not just the first it encounters.

One important distinction between the `multimap` and regular `map` is the lack of square brackets. On a standard STL `map`, you can use the syntax `myMap[key] = value` to add or update a key/value pair. However, this operation only makes sense because keys in a `map` are unique. When writing `myMap[key]`, there is only one possible key/value pair that could be meant. However, in a `multimap` this is not the case, because there may be multiple key/value pairs with the same key. Consequently, to insert key/value pairs into a `multimap`, you will need to use the `insert` function. Fortunately, the semantics of the `multimap` `insert` function are much simpler than the `map`'s `insert` function, since insertions never fail in a `multimap`. If you try to insert a key/value pair into a `multimap` for which the key already exists in the `multimap`, the new key/value pair will be inserted without any fuss. After all, `multimap` exists to allow single keys to map to multiple values!

One function that's quite useful for the multicontainers is `equal_range`. `equal_range` returns a `pair<iterator, iterator>` that represents the span of entries equal to the specified value. For example, given a `multimap<string, int>`, you could use the following code to iterate over all entries with key "STL":

```
/* Store the result of the equal_range */
pair<multimap<string, int>::iterator, multimap<string, int>::iterator>
   myPair = myMultiMap.equal_range("STL");

/* Iterate over it! */
for(multimap<string, int>::iterator itr = myPair.first;
    itr != myPair.second; ++itr)
   cout << itr->first << ": " << itr->second << endl;
```

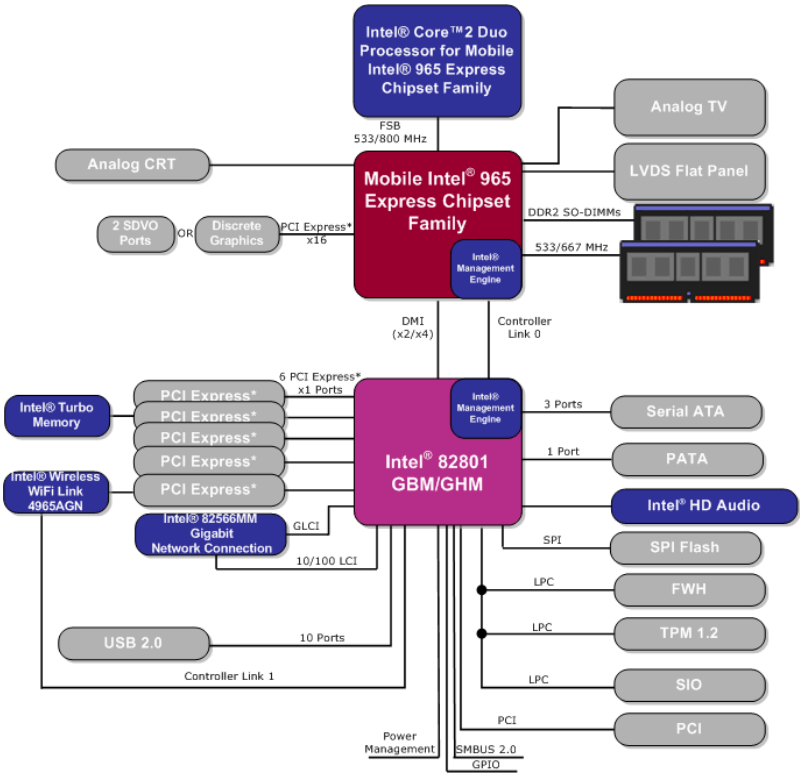
The multicontainers are fairly uncommon in practice partially because they can easily be emulated using the regular `map` or `set`. For example, a `multimap<string, int>` behaves similarly to a `map<string, vector<int> >` since both act as a map from `string`s to some number of `int`s. However, in many cases the multicontainers are exactly the tool for the job; we'll see them used later in this chapter.

Extended Example: Finite Automata

Computer science is often equated with programming and software engineering. Many a computer science student has to deal with the occasional "Oh, you're a computer science major! Can you make me a website?" or "Computer science, eh? Why isn't my Internet working?" This is hardly the case and computer science is a much broader discipline that encompasses many fields, such as artificial intelligence, graphics, and biocomputation.

One particular subdiscipline of computer science is *computability theory*. Since computer science involves so much programming, a good question is exactly *what* we can command a computer to do. What sorts of problems can we solve? How efficiently? What problems *can't* we solve and why not? Many of the most important results in computer science, such as the undecidability of the halting problem, arise from computability theory.

But how exactly can we determine what can be computed with a computer? Modern computers are phenomenally complex machines. For example, here is a high-level model of the chipset for a mobile Intel processor: [Intel]



Modeling each of these components is exceptionally tricky, and trying to devise any sort of proof about the capabilities of such a machine would be all but impossible. Instead, one approach is to work with *automata*, abstract mathematical models of computing machines (the singular of *automata* is the plural of *automaton*). Some types of automata are realizable in the physical world (for example, deterministic and nondeterministic finite automata, as you'll see below), while others are not. For example, the *Turing machine*, which computer scientists use as an overapproximation of modern computers, requires infinite storage space, as does the weaker *pushdown automaton*.

Although much of automata theory is purely theoretical, many automata have direct applications to software engineering. For example, most production compilers simulate two particular types of automata (called *pushdown automata* and *nondeterministic finite automata*) to analyze and convert source code into a form readable by the compiler's semantic analyzer and code generator. Regular expression matchers, which search through text strings in search of patterned input, are also frequently implemented using an automaton called a *deterministic finite automaton*.

In this extended example, we will introduce two types of automata, *deterministic finite automata* and *nondeterministic finite automata*, then explore how to represent them in C++. We'll also explore how these automata can be used to simplify difficult string-matching problems.

Deterministic Finite Automata

Perhaps the simplest form of an automaton is a *deterministic finite automaton*, or DFA. At a high-level, a DFA is similar to a flowchart – it has a collection of *states* joined by various *transitions* that represent how the DFA should react to a given input. For example, consider the following DFA:

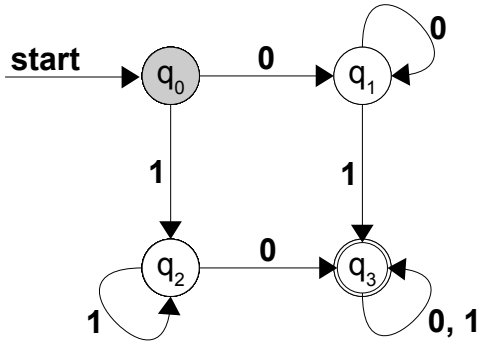
This DFA has four states, labeled q_0, q_1, q_2, and q_3, and a set of labeled transitions between those states. For example, the state q_0 has a transition labeled **0** to q_1 and a transition labeled **1** to q_2. Some states have transitions to themselves; for example, q_2 transitions to itself on a **1**, while q_3 transitions to itself on either a **0** or **1**. Note that as shorthand, the transition labeled **0, 1** indicates two different transitions, one labeled with a **0** and one labeled with a **1**. The DFA has a designated state state, in this case q_0, which is indicated by the arrow labeled **start**.

Notice that the state q_3 has two rings around it. This indicates that q_3 is an *accepting state*, which will have significance in a moment when we discuss how the DFA processes input.

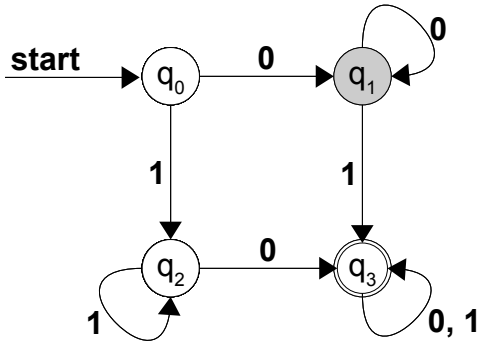
Since all of the transitions in this DFA are labeled either **0** or **1**, this DFA is said to have an *alphabet* of {**0**, **1**}. A DFA can use any nonempty set of symbols as an alphabet; for example, the Latin or Greek alphabets are perfectly acceptable for use as alphabets in a DFA, as is the set of integers between 42 and 137. By definition, every state in a DFA is required to have a transition for each symbol in its alphabet. For example, in the above DFA, each state has exactly two transitions, one labeled with a **0** and the other with a **1**. Notice that state q_3 has only one transition explicitly drawn, but because the transition is labeled with two symbols we treat it as two different transitions.

The DFA is a simple computing machine that accepts as input a string of characters formed from its alphabet, processes the string, and then halts by either *accepting* the string or *rejecting* it. In essence, the DFA is a device for discriminating between two types of input – input for which some criterion is true and input for which it is false. The DFA starts in its designated start state, then processes its input character-by-character by transitioning from its current state to the state indicated by the transition. Once the DFA has finished consuming its input, it accepts the string if it ends in an accepting state; otherwise it rejects the input.

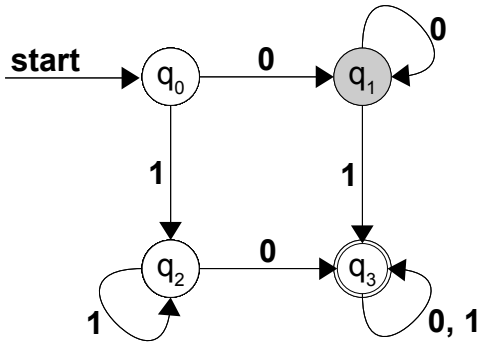
To see exactly how a DFA processes input, let us consider the above DFA simulated on the input **0011**. Initially, we begin in the start state, as shown here:



Since the first character of our string is a **0**, we follow the transition to state q_1, as shown here:

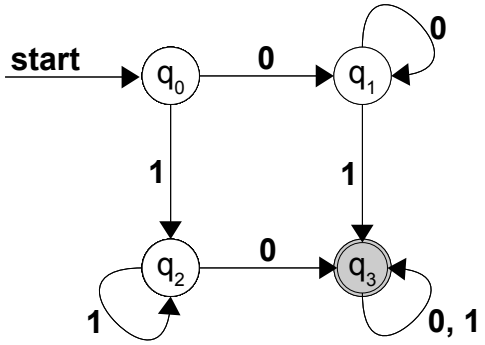


The second character of input is also a **0**, so we follow the transition labeled with a **0** and end up back in state q_1, leaving us in this state:



Next, we consume the next input character, a **1**, which causes us to follow the transition labeled **1** to state q_3:

The final character of input is also a **1**, so we follow the transition labeled **1** and end back up in q_3:



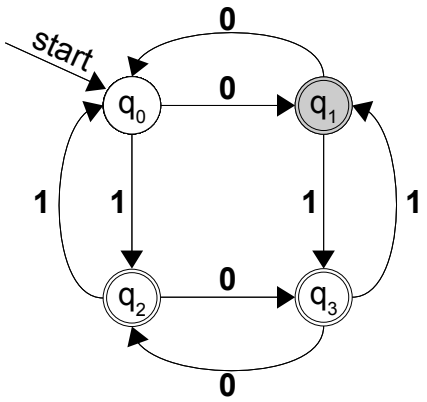
We are now done with our input, and since we have ended in an accepting state, the DFA accepts this input.

We can similarly consider the action of this DFA on the string **111**. Initially the machine will start in state q_0, then transition to state q_2 on the first input. The next two inputs each cause the DFA to transition back to state q_2, so once the input is exhausted the DFA ends in state q_2, so the DFA rejects the input. We will not prove it here, but this DFA accepts all strings that have at least one **0** and at least one **1**.

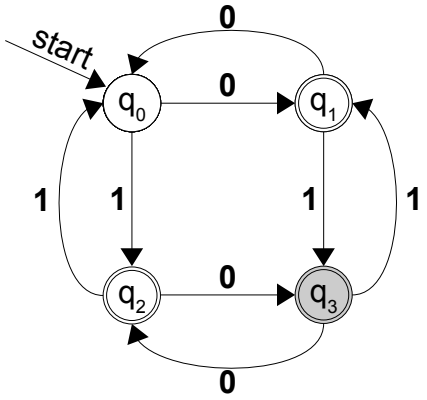
Two important details regarding DFAs deserve some mention. First, it is possible for a DFA to have multiple accepting states, as is the case in this DFA:

As with the previous DFA, this DFA has four states, but notice that three of them are marked as accepting. This leads into the second important detail regarding DFAs – the DFA only accepts its input if the DFA ends in an accepting state *when it runs out of input*. Simply transitioning into an accepting state does not cause the DFA to accept. For example, consider the effect of running this DFA on the input **0101**. We begin in the start state, as shown here:

We first consume a **0**, sending us to state q_1:

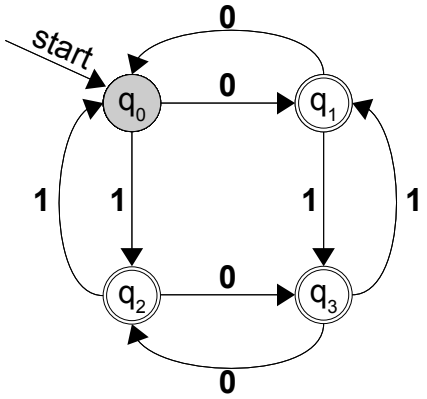


Next, we read a **1**, sending us to state q_3:



The next input is a **0**, sending us to q_2:

Finally, we read in a **1**, sending us back to q_0:



Since we are out of input and are not in an accepting state, this DFA rejects its input, even though we transitioned through every single accepting state. If you want a fun challenge, convince yourself that this DFA accepts all strings that contain an odd number of **0**s or an odd number of **1**s (inclusive OR).

Representing a DFA

A DFA is a simple model of computation that can easily be implemented in software or hardware. For any DFA, we need to store five pieces of information:*

1. The set of states used by the DFA.
2. The DFA's alphabet.
3. The start state.
4. The state transitions.
5. The set of accepting states.

Of these five, the one that deserves the most attention is the fourth, the set of state transitions. Visually, we have displayed these transitions as arrows between circles in the graph. However, another way to treat state transitions is as a table with states along one axis and symbols of the alphabet along the other. For example, here is a transition table for the DFA described above:

State	**0**	**1**
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

To determine the state to transition to given a current state and an input symbol, we look up the row for the current state, then look at the state specified in the column for the current input.

If we want to implement a program that simulates a DFA, we can represent almost all of the necessary information simply by storing the transition table. The two axes encode all of the states and alphabet symbols, and the entries of the table represent the transitions. The information not stored in this table is the set of accepting states and the designated start state, so provided that we bundle this information with the table we have a full description of a DFA.

To concretely model a DFA using the STL, we must think of an optimal way to model the transition table. Since transitions are associated with pairs of states and symbols, one option would be to model the table as an STL `map` mapping a state-symbol pair to a new state. If we represent each symbol as a `char` and each state as an `int` (i.e. q_0 is 0, q_1 is 1, etc.), this leads to a state transition table stored as a `map<pair<int, char>, int>`. If we also track the set of accepting states as a `set<int>`, we can encode a DFA as follows:

```
struct DFA {
    map<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

For the purposes of this example, assume that we have a function which fills this `DFA` struct will relevant data. Now, let's think about how we might go about simulating the DFA. To do this, we'll write a function `SimulateDFA` which accepts as input a `DFA struct` and a string representing the input, simulates the DFA when run on the given input, and then returns whether the input was accepted. We'll begin with the following:

```
bool SimulateDFA(DFA& d, string input) {
```

* In formal literature, a DFA is often characterized as a quintuple $(Q, \Sigma, q_0, \delta, F)$ of the states, alphabet, start state, transition table, and set of accepting states, respectively. Take CS154 if you're interested in learning more about these wonderful automata, or CS143 if you're interested in their applications.

```
    /* ... */
}
```

We need to maintain the state we're currently in, which we can do by storing it in an `int`. We'll initialize the current state to the starting state, as shown here:

```
bool SimulateDFA(DFA& d, string input) {
    int currState = d.startState;
    /* ... */
}
```

Now, we need to iterate over the string, following transitions from state to state. Since the transition table is represented as a `map` from `pair<int, char>`s, we can look up the next state by using `make_pair` to construct a pair of the current state and the next input, then looking up its corresponding value in the `map`. As a simplifying assumption, we'll assume that the input string is composed only of characters from the DFA's alphabet.

This leads to the following code:

```
bool SimulateDFA(DFA& d, string input) {
    int currState = d.startState;
    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
        currState = d.transitions[make_pair(currState, *itr)];
    /* ... */
}
```

You may be wondering how we're iterating over the contents of a `string` using iterators. Surprisingly, the `string` is specially designed like the STL container classes, and so it's possible to use all of the iterator tricks you've learned on the STL containers directly on the `string`.

Once we've consumed all the input, we need to check whether we ended in an accepting state. We can do this by looking up whether the `currState` variable is contained in the `acceptingStates set` in the `DFA` struct, as shown here:

```
bool SimulateDFA(DFA& d, string input) {
    int currState = d.startState;
    for(string::iterator itr = input.begin(); itr != input.end(); ++itr)
        currState = d.transitions[make_pair(currState, *itr)];
    return d.acceptingStates.find(currState) != d.acceptingStates.end();
}
```

This function is remarkably simple but correctly simulates the DFA run on some input. As you'll see in the next section on applications of DFAs, the simplicity of this implementation lets us harness DFAs to solve a suite of problems surprisingly efficiently.

Applications of DFAs

The C++ `string` class exports a handful of searching functions (`find`, `find_first_of`, `find_last_not_of`, etc.) that are useful for locating specific strings or characters. However, it's surprisingly tricky to search strings for specific patterns of characters. The canonical example is searching for email addresses in a string of text. All email addresses have the same structure – a name field followed by an at sign (`@`) and a domain name. For example, htiek@cs.stanford.edu and this.is.not.my.real.address@example.com are valid email addresses. In general, we can specify the formatting of an email address as follows:*

- The name field, which consists of nonempty alphanumeric strings separated by periods. Periods can only occur between alphanumeric strings, never before or after. Thus hello.world@example.com and cpp.is.really.cool@example.com are legal but .oops@example.com, oops.@example.com, and oops..oops@example.com are not.

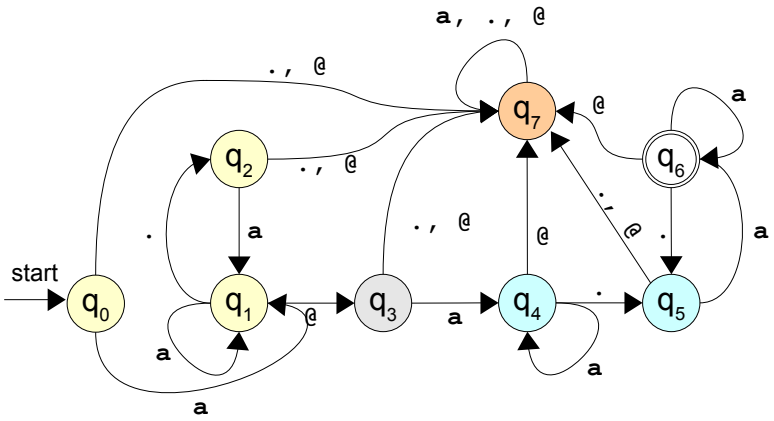
- The host field, which is structured similarly to the above except that there must be at least two sequences separated by a dot.

Now, suppose that we want to determine whether a string is a valid email address. Using the searching functions exported by the `string` class this would be difficult, but the problem is easily solved using a DFA. In particular, we can design a DFA over a suitable alphabet that accepts a string if and only if the string has the above formatting.

The first question to consider is what alphabet this DFA should be over. While we could potentially have the DFA operate over the entire ASCII alphabet, it's easier if we instead group together related characters and use a simplified alphabet. For example, since email addresses don't distinguish between letters and numbers, we can have a single symbol in our alphabet that encodes any alphanumeric character. We would need to maintain the period and at-sign in the alphabet since they have semantic significance. Thus our alphabet will be {**a**, `.`, `@`}, where **a** represents alphanumeric characters, `.` is the period character, and `@` is an at-sign.

Given this alphabet, we can represent all email addresses using the following DFA:

* This is a simplified version of the formatting of email addresses. For a full specification, refer to RFCs 5321 and 5322.



This DFA is considerably trickier than the ones we've encountered previously, so let's take some time to go over what's happening here. The machine starts in state q_0, which represents the beginning of input. Since all email addresses have to have a nonempty name field, this state represents the beginning of the first string in the name. The first character of an email address must be an alphanumeric character, which if read in state q_0 cause us to transition to state q_1. States q_1 and q_2 check that the start of the input is something appropriately formed from alphanumeric characters separated by periods. Reading an alphanumeric character while in state q_1 keeps the machine there (this represents a continuation of the current word), and reading a dot transitions the machine to q_2. In q_2, reading anything other than an alphanumeric character puts the machine into the "trap state," state q_7, which represents that the input is invalid. Note that once the machine reaches state q_7 no input can get the machine out of that state and that q_7 isn't accepting. Thus any input that gets the machine into state q_7 will be rejected.

State q_3 represents the state of having read the at-sign in the email address. Here reading anything other than an alphanumeric character causes the machine to enter the trap state.

States q_4 and q_5 are designed to help catch the name of the destination server. Like q_1, q_4 represents a state where we're reading a "word" of alphanumeric characters and q_5 is the state transitioned to on a dot. Finally, state q_6 represents the state where we've read at least one word followed by a dot, which is the accepting state. As an exercise, trace the action of this machine on the inputs valid.address@email.com and invalid@not.com@ouch.

Now, how can we use this DFA in code? Suppose that we have some way to populate a `DFA` struct with the information for this DFA. Then we could check if a string contains an email address by converting each character in the string into its appropriate character in the DFA alphabet, then simulating the DFA on the input. If the DFA rejects the input or the string contains an invalid character, we can signal that the string is invalid, but otherwise the string is a valid email address.

This can be implemented as follows:

```
bool IsEmailAddress(string input) {
    DFA emailChecker = LoadEmailDFA(); // Implemented elsewhere

    /* Transform the string one character at a time. */
    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        /* isalnum is exported by <cctype> and checks if the input is an
         * alphanumeric character.
         */
        if(isalnum(*itr))
            *itr = 'a';
        /* If we don't have alphanumeric data, we have to be a dot or at-sign
         * or the input is invalid.
         */
        else if(*itr != '.' && *itr != '@')
            return false;
    }
    return SimulateDFA(emailChecker, input);
}
```

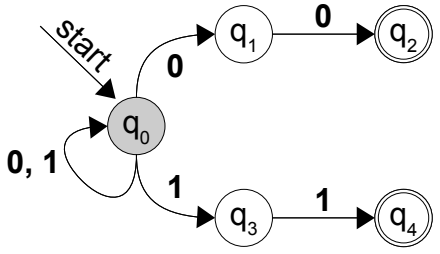
This code is remarkably concise, and provided that we have an implementation of `LoadEmailDFA` the function will work correctly. I've left out the implementation of `LoadEmailDFA` since it's somewhat tedious, but if you're determined to see that this actually works feel free to try your hand at implementing it.

Nondeterministic Finite Automata

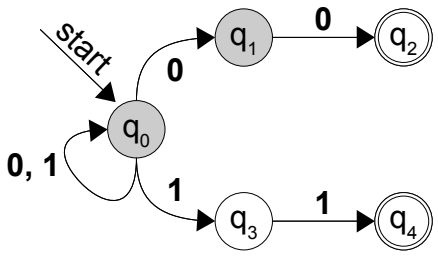
A generalization of the DFA is the *nondeterministic finite automaton*, or NFA. At a high level, DFAs and NFAs are quite similar – they both consist of a set of states connected by labeled transitions, of which some states are designated as accepting and others as rejecting. However, NFAs differ from DFAs in that a state in an NFA can have any number of transitions on a given input, including zero. For example, consider the following NFA:

Here, the start state is q_0 and accepting states are q_2 and q_4. Notice that the start state q_0 has two transitions on **0** – one to q_1 and one to itself – and two transitions on **1**. Also, note that q_3 has no defined transitions on **0**, and states q_2 and q_4 have no transitions at all.

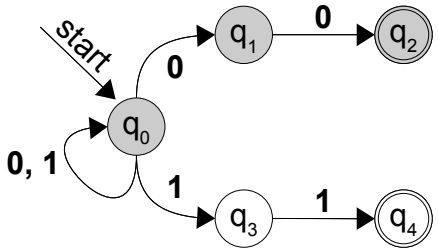
There are several ways to interpret a state having multiple transitions. The first is to view the automaton as choosing one of the paths nondeterministically (hence the name), then accepting the input if *some* set of choices results in the automaton ending in an accepting state. Another, more intuitive way for modeling multiple transitions is to view the NFA as being in several different states simultaneously, at each step following every transition with the appropriate label in each of its current states. To see this, let's consider what happens when we run the above NFA on the input **0011**. As with a DFA, we begin in the start state, as shown here:



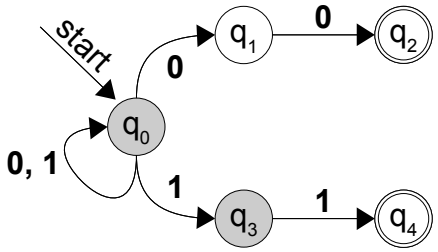
We now process the first character of input (**0**) and find that there are two transitions to follow – the first to q_0 and the second to q_1. The NFA thus ends up in both of these states simultaneously, as shown here:



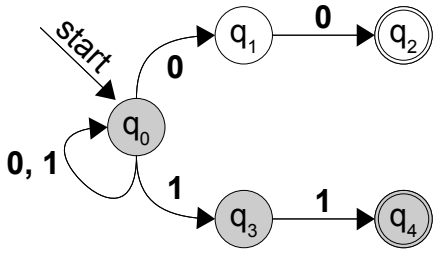
Next, we process the second character (**0**). From state q_0, we transition into q_0 and q_1, and from state q_1 we transition into q_2. We thus end up in states q_0, q_1, and q_2, as shown here:



We now process the third character of input, which is a **1**. From state q_0 we transition to states q_0 and q_3. We are also currently in states q_1 and q_2, but neither of these states has a transition on a **1**. When this happens, we simply drop the states from the set of current states. Consequently, we end up in states q_0 and q_3, leaving us in the following configuration:



Finally, we process the last character, a **1**. State q_0 transitions to q_0 and q_1, and state q_1 transitions to state q_4. We thus end up in this final configuration:



Since the NFA ends in a configuration where at least one of the active states is an accepting state (q_4), the NFA accepts this input. Again as an exercise, you might want to convince yourself that this NFA accepts all and only the strings that end in either **00** or **11**.

Implementing an NFA

Recall from above the definition of the `DFA` struct:

```
struct DFA {
    map<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

Here, the transition table was encoded as a `map<pair<int, char>, int>` since for every combination of a state and an alphabet symbol there was exactly one transition. To generalize this to represent an NFA, we need to be able to associate an arbitrary number of possible transitions. This is an ideal spot for an STL `multimap`, which allows for duplicate key/value pairs. This leaves us with the following definition for an NFA type:

```
struct NFA {
    multimap<pair<int, char>, int> transitions;
    set<int> acceptingStates;
    int startState;
};
```

How would we go about simulating this NFA? At any given time, we need to track the set of states that we are currently in, and on each input need to transition from the current set of states to some other set of states. A natural representation of the current set of states is (hopefully unsurprisingly) as a `set<int>`. Initially, we start with this set of states just containing the start state. This is shown here:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    /* ... */
}
```

Next, we need to iterate over the string we've received as input, following transitions where appropriate. This at least requires a simple `for` loop, which we'll write here:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        /* ... */
    }

    /* ... */
}
```

Now, for each character of input in the string, we need to compute the set of next states (if any) to which we should transition. To simplify the implementation of this function, we'll create a second `set<int>` corresponding to the next set of states the machine will be in. This eliminates problems caused by adding elements to our set of states as we're iterating over the set and updating it. We thus have

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        set<int> nextStates;
        /* ... */
    }
```

```
    /* ... */
}
```

Now that we have space to put the next set of machine states, we need to figure out what states to transition to. Since we may be in multiple different states, we need to iterate over the set of current states, computing which states they transition into. This is shown here:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state) {
            /* ... */
        }
    }

    /* ... */
}
```

Given the state being iterated over by `state` and the current input character, we now want to transition to each state indicated by the `multimap` stored in the `NFA` struct. If you'll recall, the STL `multimap` exports a function called `equal_range` which returns a `pair` of iterators into the `multimap` that delineate the range of elements with the specified key. This function is exactly what we need to determine the set of new states we'll be entering for each given state – we simply query the `multimap` for all elements whose key is the pair of the specified state and the current input, then add all of the destination states to our next set of states. This is shown here:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state) {
            /* Get all states that we transition to from this current state. */
            pair<multimap<pair<int, char>, int>::iterator,
                 multimap<pair<int, char>, int>::iterator>
            transitions = nfa.transitions.equal_range(make_pair(*state, *itr));

            /* Add these new states to the nextStates set. */
            for(; transitions.first != transitions.second; ++transitions.first)
                /* transitions.first is the current iterator, and its second
                 * field is the value (new state) in the STL multimap.
                 */
                nextStates.insert(transitions.first->second);
        }
    }

    /* ... */
}
```

Finally, once we've consumed all input, we need to check whether the set of states contains any states that are also in the set of accepting states. We can do this by simply iterating over the set of current states, then checking if any of them are in the accepting set. This is shown here and completes the implementation of the function:

```
bool SimulateNFA(NFA& nfa, string input) {
    /* Track our set of states.  We begin in the start state. */
    set<int> currStates;
    currStates.insert(nfa.startState);

    for(string::iterator itr = input.begin(); itr != input.end(); ++itr) {
        set<int> nextStates;
        for(set<int>::iterator state = currStates.begin();
            state != currStates.end(); ++state) {
            /* Get all states that we transition to from this current state. */
            pair<multimap<pair<int, char>, int>::iterator,
                 multimap<pair<int, char>, int>::iterator>
            transitions = nfa.transitions.equal_range(make_pair(*state, *itr));

            /* Add these new states to the nextStates set. */
            for(; transitions.first != transitions.second; ++transitions.first)
                /* transitions.first is the current iterator, and its second
                 * field is the value (new state) in the STL multimap.
                 */
                nextStates.insert(transitions.first->second);
        }
    }

    for(set<int>::iterator itr = currStates.begin();
        itr != currStates.end(); ++itr)
        if(nfa.acceptingStates.count(*itr)) return true;
    return false;
}
```

Compare this function to the implementation of the DFA simulation. There is substantially more code here, since we have to track multiple different states rather than just a single state. However, this extra complexity is counterbalanced by the simplicity of designing NFAs compared to DFAs. Building a DFA to match a given pattern can be much trickier than building an equivalent NFA because it's difficult to model "guessing" behavior with a DFA. However, both functions are a useful addition to your programming arsenal, so it's good to see how they're implemented.

More to Explore

In this chapter we covered `map` and `set`, which combined with `vector` and `deque` are the most commonly-used STL containers. However, there are several others we didn't cover, a few of which might be worth looking into. Here are some topics you might want to read up on:

1. **`list`**: `vector` and `deque` are sequential containers that mimic built-in arrays. The `list` container, however, models a sequence of elements without indices. `list` supports several nifty operations, such as merging, sorting, and splicing, and has quick insertions at almost any point. If you're planning on using a linked list for an operation, the `list` container is perfect for you.

2. **The Boost Containers**: The Boost C++ Libraries are a collection of functions and classes developed to augment C++'s native library support. Boost offers several new container classes that might be worth looking into. For example, `multi_array` is a container class that acts as a `Grid` in any number of dimensions. Also, the `unordered_set` and `unordered_map` act as replacements to the `set` and `map` that use hashing instead of tree structures to store their data. If you're interested in exploring these containers, head on over to www.boost.org.

Practice Problems

1. How do you check whether an element is contained in an STL `set`?

2. What is the restriction on what types can be stored in an STL `set`? Do the `vector` or `deque` have this restriction?

3. How do you insert an element into a `set`? How do you remove an element from a `set`?

4. How many copies of a single element can exist in a `set`? How about a `multiset`?

5. How do you iterate over the contents of a `set`?

6. How do you check whether a key is contained in an STL `map`?

7. List two ways that you can insert key/value pairs into an STL `map`.

8. What happens if you look up the value associated with a nonexistent key in an STL `map` using square brackets? What if you use the `find` function?

9. Recall that when iterating over the contents of an STL `multiset`, the elements will be visited in sorted order. Using this property, rewrite the program from last chapter that reads a list of numbers from the user, then prints them in sorted order. Why is it necessary to use a `multiset` instead of a regular `set`?

10. The *union* of two sets is the collection of elements contained in at least one of the `set`s. For example, the union of {1, 2, 3, 5, 8} and {2, 3, 5, 7, 11} is {1, 2, 3, 5, 7, 8, 11}. Write a function `Union` which takes in two `set<int>`s and returns their union.

11. The *intersection* of two sets is the collection of elements contained in *both* of the `set`s. For example, the intersection of {1, 2, 3, 5, 8} and {2, 3, 5, 7, 11} is {2, 3, 5}. Write a function `Intersection` that takes in two `set<int>`s and returns their intersection.

12. Earlier in this chapter, we wrote a program that rolled dice until the same number was rolled twice, then printed out the number of rolls made. Rewrite this program so that the same number must be rolled *three* times before the process terminates. How many times do you expect this process to take when rolling twenty-sided dice? *(Hint: you will probably want to switch from using a `set` to using a `multiset`. Also, remember the difference between the `set`'s `count` function and the `multiset`'s `count` function).*

13. As mentioned in this chapter, you can use a combination of `lower_bound` and `upper_bound` to iterate over elements in the closed interval [min, max]. What combination of these two functions could you use to iterate over the interval [min, max)? What about (min, max] and (min, max)?

14. Write a function `NumberDuplicateEntries` that accepts a `map<string, string>` and returns the number of duplicate *values* in the `map` (that is, the number of key/value pairs in the map with the same value).

15. Write a function `InvertMap` that accepts as input a `map<string, string>` and returns a `multimap<string, string>` where each pair (key, value) in the source map is represented by (value, key) in the generated `multimap`. Why is it necessary to use a `multimap` here? How could you use the `NumberDuplicateEntries` function from the previous question to determine whether it is possible to invert the `map` into another `map`?

16. Suppose that we have two `map<string, string>`s called `one` and `two`. We can define the *composition* of `one` and `two` (denoted `two ○ one`) as follows: for any string `r`, if `one[r]` is `s` and `two[s]` is `t`, then `(two ○ one)[r] = t`. That is, looking up an element `x` in the composition of the maps is equivalent to looking up the value associated with `x` in `one` and then looking up its associated value in `two`. If `one` does not contain `r` as a key or if `one[r]` is not a key in `two`, then `(two ○ one)[r]` is undefined.

    Write a function `ComposeMaps` that takes in two `map<string, string>`s and returns a `map<string, string>` containing their composition.

17. (Challenge problem!) Write a function `PrintMatchingPrefixes` that accepts a `set<string>` and a `string` containing a prefix and prints out all of the entries of the `set` that begin with that prefix. Your function should only iterate over the entires it finally prints out. You can assume the prefix is nonempty, consists only of alphanumeric characters, and should treat prefixes case-sensitively. *(Hint: In a `set<string>`, strings are sorted lexicographically, so all strings that start with "abc" will come before all strings that start with "abd.")*

Chapter 7: STL Algorithms

Consider the following problem: suppose that we want to write a program that reads in a list of integers from a file (perhaps representing grades on an assignment), then prints out the average of those values. For simplicity, let's assume that this data is stored in a file called `data.txt` with one integer per line. For example:

***File:* `data.txt`**

```
100
95
92
98
87
88
100
...
```

Here is one simple program that reads in the contents of the file, stores them in an STL `multiset`, computes the average, then prints it out:

```
#include <iostream>
#include <fstream>
#include <set>
using namespace std;

int main() {
    ifstream input("data.txt");
    multiset<int> values;

    /* Read the data from the file. */
    int currValue;
    while (input >> currValue)
        values.insert(currValue);

    /* Compute the average. */
    double total = 0.0;
    for (multiset<int>::iterator itr = values.begin();
         itr != values.end(); ++itr)
        total += *itr;
    cout << "Average is: " << total / values.size() << endl;
}
```

As written, this code is perfectly legal and will work as intended. However, there's something slightly odd about it. If you were to describe what this program needs to do in plain English, it would probably be something like this:

1. Read the contents of the file.
2. Add the values together.
3. Divide by the number of elements.

In some sense, the above code matches this template. The first loop of the program reads in the contents of the file, the second loop sums together the values, and the last line divides by the number of elements. However, the code we've written is somewhat unsatisfactory. Consider this first loop:

```
int currValue;
while (input >> currValue)
    values.insert(currValue);
```

Although the intuition behind this loop is "read the contents of the file into the `multiset`," the way the code is actually written is "create an integer, and then while it's possible to read another element out of the file, do so and insert it into the `multiset`." This is a very *mechanical* means for inserting the values into the `multiset`. Our English description of this process is "read the file contents into the `multiset`," but the actual code is a step-by-step process for extracting data from the file one step at a time and inserting it into the `multiset`.

Similarly, consider this second loop, which sums together the elements of the `multiset`:

```
double total = 0.0;
for (multiset<int>::iterator itr = values.begin(); itr != values.end(); ++itr)
    total += *itr;
```

Again, we find ourselves taking a very mechanical view of the operation. Our English description "sum the elements together" is realized here as "initialize the total to zero, then iterate over the elements of the `multiset`, increasing the total by the value of the current element at each step."

The reason that we must issue commands to the computer in this mechanical fashion is precisely because the computer *is* mechanical – it's a machine for efficiently computing functions. The challenge of programming is finding a way to translate a high-level set of commands into a series of low-level instructions that control the machine. This is often a chore, as the basic operations exported by the computer are fairly limited. But programming doesn't have to be this difficult. As you've seen, we can define new functions in terms of old ones, and can build complex programs out of these increasingly more powerful subroutines. In theory, you could compile an enormous library containing solutions to all nontrivial programming problems. With this library in tow, you could easily write programs by just stitching together these prewritten components.

Unfortunately, there is no one library with the solutions to every programming problem. However, this hasn't stopped the designers of the STL from trying their best to build one. These are the STL algorithms, a library of incredibly powerful routines for processing data. The STL algorithms can't do everything, but what they can do they do fantastically. In fact, using the STL algorithms, it will be possible to rewrite the program that averages numbers in *four lines of code*. This chapter details many common STL algorithms, along with applications. Once you've finished this chapter, you'll have one of the most powerful standard libraries of any programming language at your disposal, and you'll be ready to take on increasingly bigger and more impressive software projects.

Your First Algorithm: `accumulate`

Let's begin our tour of the STL algorithms by jumping in head-first. If you'll recall, the second loop from the averaging program looks like this:

```
double total = 0.0;
for (multiset<int>::iterator itr = values.begin(); itr != values.end(); ++itr)
    total += *itr;
cout << "Average is: " << total / values.size() << endl;
```

This code is entirely equivalent to the following:

```
cout << accumulate(values.begin(), values.end(), 0.0) / values.size() << endl;
```

We've replaced the entire `for` loop with a single call to `accumulate`, eliminating about a third of the code from our original program.

The `accumulate` function, defined in the `<numeric>` header, takes three parameters – two iterators that define a range of elements, and an initial value to use in the summation. It then computes the sum of all of the elements contained in the range of iterators, plus the base value.* What's beautiful about `accumulate` (and the STL algorithms in general) is that `accumulate` can take in iterators of any type. That is, we can sum up iterators from a `multiset`, a `vector`, or `deque`. This means that if you ever find yourself needing to compute the sum of the elements contained in a container, you can pass the `begin()` and `end()` iterators of that container into `accumulate` to get the sum. Moreover, `accumulate` can accept any valid iterator range, not just an iterator range spanning an entire container. For example, if we want to compute the sum of the elements of the `multiset` that are between 42 and 137, inclusive, we could write

```
accumulate(values.lower_bound(42), values.upper_bound(137), 0);
```

Behind the scenes, `accumulate` is implemented as a template function that accepts two iterators and simply uses a loop to sum together the values. Here's one possible implementation of `accumulate`:

```
template <typename InputIterator, typename Type> inline
Type accumulate(InputIterator start, InputIterator stop, Type initial) {
    while(start != stop) {
        initial += *start;
        ++start;
    }
    return initial;
}
```

While some of the syntax specifics might be a bit confusing (notably the template header and the `inline` keyword), you can still see that the heart of the code is just a standard iterator loop that continuously advances the start iterator forward until it reaches the destination. There's nothing magic about `accumulate`, and the fact that the function call is a single line of code doesn't change that it still uses a loop to sum all the values together.

If STL algorithms are just functions that use loops behind the scenes, why even bother with them? There are several reasons, the first of which is *simplicity*. With STL algorithms, you can leverage off of code that's already been written for you rather than reinventing the code from scratch. This can be a great time-saver and also leads into the second reason, *correctness*. If you had to rewrite all the algorithms from scratch every time you needed to use them, odds are that at some point you'd slip up and make a mistake. You might, for example, write a sorting routine that accidentally uses < when you meant > and consequently does not work at all. Not so with the STL algorithms – they've been thoroughly tested and will work correctly for any given input. The third reason to use algorithms is *speed*. In general, you can assume that if there's an STL algorithm that performs a task, it's going to be faster than most code you could write by hand. Through advanced techniques like template specialization and template metaprogramming, STL algorithms are transparently optimized to work as fast as possible. Finally, STL algorithms offer *clarity*. With algorithms, you can immediately tell that a call to `accumulate` adds up numbers in a range. With a for loop that sums up values, you'd have to read each line in the loop before you understood what the code did.

---

* There is also a version of `accumulate` that accepts four parameters, as you'll see in the chapter on functors.

Algorithm Naming Conventions

There are over fifty STL algorithms (defined either in `<algorithm>` or in `<numeric>`), and memorizing them all would be a chore, to say the least. Fortunately, many of them have common naming conventions so you can recognize algorithms even if you've never encountered them before.

The suffix `_if` on an algorithm (`replace_if`, `count_if`, etc.) means the algorithm will perform a task on elements only if they meet a certain criterion. Functions ending in `_if` require you to pass in a predicate function that accepts an element and returns a `bool` indicating whether the element matches the criterion. For example consider the `count` algorithm and its counterpart `count_if`. `count` accepts a range of iterators and a value, then returns the number of times that the value appears in that range. If we have a `vector<int>` of several integer values, we could print out the number of copies of the number 137 in that `vector` as follows:

```
cout << count(myVec.begin(), myVec.end(), 137) << endl;
```

`count_if`, on the other hand, accepts a range of iterators and a predicate function, then returns the number of times the predicate evaluates to `true` in that range. If we were interested in how number of even numbers are contained in a `vector<int>`, we could could obtain the value as follows. First, we write a predicate function that takes in an `int` and returns whether it's even, as shown here:

```
bool IsEven(int value) {
    return value % 2 == 0;
}
```

We could then use `count_if` as follows:

```
cout << count_if(myVec.begin(), myVec.end(), IsEven) << endl;
```

Algorithms containing the word `copy` (`remove_copy`, `partial_sort_copy`, etc.) will perform some task on a range of data and store the result in the location pointed at by an extra iterator parameter. With `copy` functions, you'll specify all the normal data for the algorithm plus an extra iterator specifying a destination for the result. We'll cover what this means from a practical standpoint later.

If an algorithm ends in `_n` (`generate_n`, `search_n`, etc**)**, then it will perform a certain operation `n` times. These functions are useful for cases where the number of times you perform an operation is meaningful, rather than the range over which you perform it. To give you a better feel for what this means, consider the `fill` and `fill_n` algorithms. Each of these algorithms sets a range of elements to some specified value. For example, we could use `fill` as follows to set every element in a `deque` to have value 0:

```
fill(myDeque.begin(), myDeque.end(), 0);
```

The `fill_n` algorithm is similar to `fill`, except that instead of accepting a range of iterators, it takes in a start iterator and a number of elements to write. For instance, we could set the first ten elements of a `deque` to be zero by calling

```
fill_n(myDeque.begin(), 10, 0);
```

Iterator Categories

If you'll recall from the discussion of the `vector` and `deque` `insert` functions, to specify an iterator to the nth element of a `vector`, we used the syntax `myVector.begin() + n`. Although this syntax is legal in conjunction with `vector` and `deque`, it is illegal to use `+` operator with iterators for other container classes

like `map` and `set`. At first this may seem strange – after all, there's nothing intuitively wrong with moving a `set` iterator forward multiple steps, but when you consider how the `set` is internally structured the reasons become more obvious. Unlike `vector` and `deque`, the elements in a `map` or `set` are not stored sequentially (usually they're kept in a balanced binary tree). Consequently, to advance an iterator *n* steps forward, the `map` or `set` iterator must take *n* individual steps forward. Contrast this with a `vector` iterator, where advancing forward *n* steps is a simple addition (since all of the `vector`'s elements are stored contiguously). Since the runtime complexity of advancing a `map` or `set` iterator forward *n* steps is linear in the size of the jump, whereas advancing a `vector` iterator is a constant-time operation, the STL disallows the + operator for `map` and `set` iterators to prevent subtle sources of inefficiency.

Because not all STL iterators can efficiently or legally perform all of the functions of every other iterator, STL iterators are categorized based on their relative power. At the high end are *random-access iterators* that can perform all of the possible iterator functions, and at the bottom are the *input* and *output* iterators which guarantee only a minimum of functionality. There are five different types of iterators, each of which is discussed in short detail below.

- **Output Iterators**. Output iterators are one of the two weakest types of iterators. With an output iterator, you can write values using the syntax `*myItr = value` and can advance the iterator forward one step using the ++ operator. However, you cannot read a value from an output iterator using the syntax `value = *myItr`, nor can you use the += or – operators.

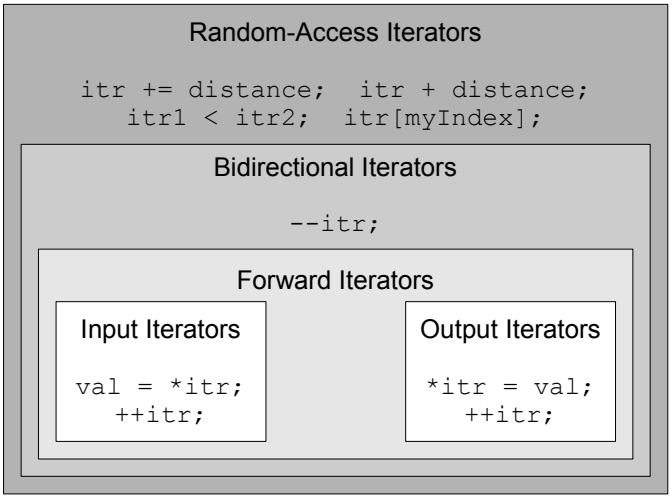
- **Input Iterators**. Input iterators are similar to output iterators except that they read values instead of writing them. That is, you can write code along the lines of `value = *myItr`, but not `*myItr = value`. Moreover, input iterators cannot iterate over the same range twice.

- **Forward Iterators**. Forward iterators combine the functionality of input and output iterators so that most intuitive operations are well-defined. With a forward iterator, you can write both `*myItr = value` and `value = *myItr`. Forward iterators, as their name suggests, can only move forward. Thus `++myItr` is legal, but `--myItr` is not.

- **Bidirectional Iterators**. Bidirectional iterators are the iterators exposed by `map` and `set` and encompass all of the functionality of forward iterators. Additionally, they can move backwards with the decrement operator. Thus it's possible to write `--myItr` to go back to the last element you visited, or even to traverse a list in reverse order. However, bidirectional iterators cannot respond to the + or += operators.

- **Random-Access Iterators**. Don't get tripped up by the name – random-access iterators don't move around randomly. Random-access iterators get their name from their ability to move forward and backward by arbitrary amounts at any point. These are the iterators employed by `vector` and `deque` and represent the maximum possible functionality, including iterator-from-iterator subtraction, bracket syntax, and incrementation with + and +=.

If you'll notice, each class of iterators is progressively more powerful than the previous one – that is, the iterators form a functionality hierarchy. This means that when a library function requires a certain class of iterator, you can provide it any iterator that's at least as powerful. For example, if a function requires a forward iterator, you can provide either a forward, bidirectional, or random-access iterator. The iterator hierarchy is illustrated below:



Why categorize iterators this way? Why not make them all equally powerful? There are several reasons. First, in some cases, certain iterator operations cannot be performed efficiently. For instance, the STL `map` and `set` are layered on top of balanced binary trees, a structure in which it is simple to move from one element to the next but significantly more complex to jump from one position to another arbitrarily. By disallowing the `+` operator on `map` and `set` iterators, the STL designers prevent subtle sources of inefficiency where simple code like `itr + 5` is unreasonably inefficient. Second, iterator categorization allows for better classification of the STL algorithms. For example, suppose that an algorithm takes as input a pair of input iterators. From this, we can tell that the algorithm will not modify the elements being iterated over, and so can feel free to pass in iterators to data that must not be modified under any circumstance. Similarly, if an algorithm has a parameter that is labeled as an output iterator, it should be clear from context that the iterator parameter defines where data generated by the algorithm should be written.

Reordering Algorithms

There are a large assortment of STL algorithms at your disposal, so for this chapter it's useful to discuss the different algorithms in terms of their basic functionality. The first major grouping of algorithms we'll talk about are the *reordering algorithms*, algorithms that reorder but preserve the elements in a container.

Perhaps the most useful of the reordering algorithms is `sort`, which sorts elements in a range in ascending order. For example, the following code will sort a `vector<int>` from lowest to highest:

```
sort(myVector.begin(), myVector.end());
```

`sort` requires that the iterators you pass in be random-access iterators, so you cannot use `sort` to sort a `map` or `set`. However, since `map` and `set` are always stored in sorted order, this shouldn't be a problem.

By default, `sort` uses the < operator for whatever element types it's sorting, but you can specify a different comparison function if you wish. Whenever you write a comparison function for an STL algorithm, it should accept two parameters representing the elements to compare and return a `bool` indicating whether the first element is strictly less than the second element. In other words, your callback should mimic the < operator. For example, suppose we had a `vector<placeT>`, where `placeT` was defined as

```
struct placeT {
    int x;
    int y;
};
```

Then we could `sort` the `vector` only if we wrote a comparison function for `placeT`s.* For example:

```
bool ComparePlaces(placeT one, placeT two) {
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}

sort(myPlaceVector.begin(), myPlaceVector.end(), ComparePlaces);
```

You can also use custom comparison functions even if a default already exists. For example, here is some code that sorts a `vector<string>` by length, ignoring whether the strings are in alphabetical order:

```
bool CompareStringLength(string one, string two) {
    return one.length() < two.length();
}

sort(myVector.begin(), myVector.end(), CompareStringLength);
```

One last note on comparison functions is that they should either accept the parameters by value or by "reference to `const`." Since we haven't covered `const` yet, for now your comparison functions should accept their parameters by value. Otherwise you can get some pretty ferocious compiler errors.

Another useful reordering function is `random_shuffle`, which randomly scrambles the elements of a container. Because the scrambling is random, there's no need to pass in a comparison function. Here's some code that uses `random_shuffle` to scramble a `vector`'s elements:

```
random_shuffle(myVector.begin(), myVector.end());
```

As with `sort`, the iterators must be random-access iterators, so you can't scramble a `set` or `map`. Then again, since they're sorted containers, you shouldn't want to do this in the first place.

Internally, `random_shuffle` uses the built-in `rand()` function to generate random numbers. Accordingly, you should use the `srand` function to seed the randomizer before using `random_shuffle`.

The last major algorithm in this category is `rotate`, which cycles the elements in a container. For example, given the input container (0, 1, 2, 3, 4, 5), rotating the container around position 3 would result in the container (2, 3, 4, 5, 0, 1). The syntax for `rotate` is anomalous in that it accepts three iterators delineating the range and the new front, but in the order *begin*, *middle*, *end*. For example, to rotate a `vector` around its third position, we would write

```
rotate(v.begin(), v.begin() + 2, v.end());
```

Searching Algorithms

Commonly you're interested in checking membership in a container. For example, given a `vector`, you might want to know whether or not it contains a specific element. While the `map` and `set` naturally support `find`, `vector`s and `deque`s lack this functionality. Fortunately, you can use STL algorithms to correct this problem.

---

* When we cover operator overloading in the second half of this text, you'll see how to create functions that `sort` will use automatically.

To search for an element in a container, you can use the `find` function. `find` accepts two iterators delineating a range and a value, then returns an iterator to the first element in the range with that value. If nothing in the range matches, `find` returns the second iterator as a sentinel. For example:

```
if(find(myVector.begin(), myVector.end(), 137) != myVector.end())
    /* ... vector contains 137 ... */
```

Although you can legally pass `map` and `set` iterators as parameters to `find`, you should avoid doing so. If a container class has a member function with the same name as an STL algorithm, you should use the member function instead of the algorithm because member functions can use information about the container's internal data representation to work much more quickly. Algorithms, however, must work for all iterators and thus can't make any optimizations. As an example, with a `set` containing one million elements, the `set`'s `find` member function can locate elements in around twenty steps using binary search, while the STL `find` function could take up to one million steps to linearly iterate over the entire container. That's a staggering difference and really should hit home how important it is to use member functions over STL algorithms.

Just as a sorted `map` and `set` can use binary search to outperform the linear STL `find` algorithm, if you have a sorted linear container (for example, a sorted `vector`), you can use the STL algorithm `binary_search` to perform the search in a fraction of the time. For example:

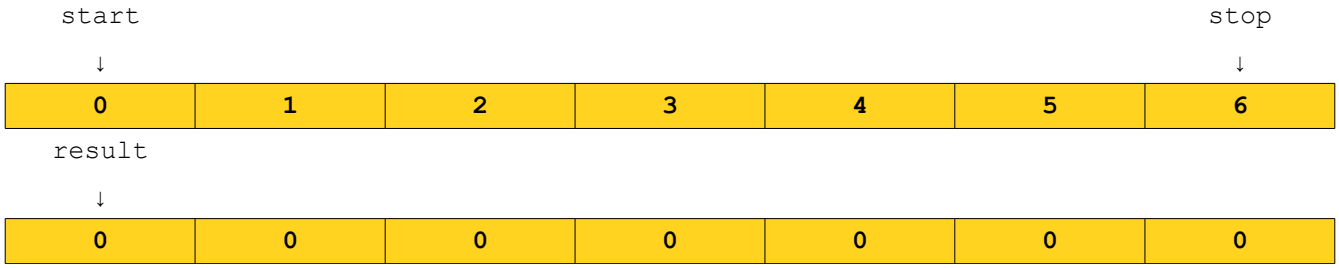
```
/* Assume myVector is sorted. */
if (binary_search(myVector.begin(), myVector.end(), 137)) {
   /* ... Found 137 ... */
}
```

Also, as with `sort`, if the container is sorted using a special comparison function, you can pass that function in as a parameter to `binary_search`. However, make sure you're consistent about what comparison function you use, because if you mix them up `binary_search` might not work correctly.

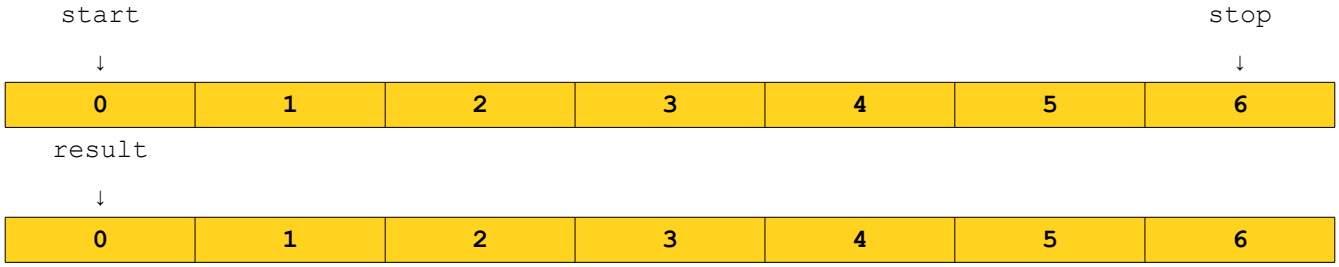
Note that `binary_search` doesn't return an iterator to the element – it simply checks to see if it's in the container. If you want to do a binary search in order to get an iterator to an element, you can use the `lower_bound` algorithm which, like the `map` and `set` `lower_bound` functions, returns an iterator to the first element greater than or equal to the specified value. Note that `lower_bound` might hand back an iterator to a different element than the one you searched for if the element isn't in the range, so be sure to check the return value before using it. As with `binary_search`, the container must be in sorted order for `lower_bound` algorithm to work correctly.

Iterator Adaptors

The algorithms that we've encountered so far do not produce any new data ranges. The `sort` algorithm rearranges data without generating new values. `binary_search` and `accumulate` scan over data ranges, but yield only a single value. However, there are a great many STL algorithms that take in ranges of data and produce new data ranges at output. As a simple example, consider the `copy` algorithm. At a high level, `copy` takes in a range of data, then duplicates the values in that range at another location. Concretely, `copy` takes in three parameters – two input iterators defining a range of values to copy, and an output iterator indicating where the data should be written. For example, given the following setup:



After calling `copy(start, stop, result)`, the result is as follows:



When using algorithms like `copy` that generate a range of data, you *must* make sure that the destination has enough space to hold the result. Algorithms that generate data ranges work by *overwriting* elements in the range beginning with the specified iterator, and if your output iterator points to a range that doesn't have enough space the algorithms will write off past the end of the range, resulting in undefined behavior. But here we reach a wonderful paradox. When running an algorithm that generates a range of data, you must make sure that sufficient space exists to hold the result. However, in some cases you can't tell how much data is going to be generated until you actually run the algorithm. That is, the only way to determine how much space you'll need is to run the algorithm, which might result in undefined behavior because you didn't allocate enough space.

To break this cycle, we'll need a special set of tools called *iterator adaptors*. Iterator adaptors (defined in the `<iterator>` header) are objects that act like iterators – they can be dereferenced with `*` and advanced forward with `++` – but which don't actually point to elements of a container. To give a concrete example, let's consider the `ostream_iterator`. `ostream_iterator`s are objects that look like output iterators. That is, you can dereference them using the `*` operator, advance them forward with the `++` operator, etc. However, `ostream_iterator`s don't actually point to elements in a container. Whenever you dereference an `ostream_iterator` and assign a value to it, that value is printed to a specified output stream, such as `cout` or an `ofstream`. Here's some code showing off an `ostream_iterator`; the paragraph after it explores how it works in a bit more detail:

```
/* Declare an ostream_iterator that writes ints to cout. */
ostream_iterator<int> myItr(cout, " ");

/* Write values to the iterator.  These values will be printed to cout. */
*myItr = 137; // Prints 137 to cout
++myItr;

*myItr = 42;  // Prints 42 to cout
++myItr
```

If you compile and run this code, you will notice that the numbers 137 and 42 get written to the console, separated by spaces. Although it *looks* like you're manipulating the contents of a container, you're actually writing characters to the `cout` stream.

Let's consider this code in a bit more detail. If you'll notice, we declared the `ostream_iterator` by writing

```
ostream_iterator<int> myItr(cout, " ");
```

There are three important pieces of data in this line of code. First, notice that `ostream_iterator` is a parameterized type, much like the `vector` or `set`. In the case of `ostream_iterator`, the template argument indicates what sorts of value will be written to this iterator. That is, an `ostream_iterator<int>` writes `int`s into a stream, while an `ostream_iterator<string>` would write strings. Second, notice that when we created the `ostream_iterator`, we passed it two pieces of information. First, we gave the `ostream_iterator` a stream to write to, in this case `cout`. Second, we gave it a *separator string*, in our case a string holding a single space. Whenever a value is written to an `ostream_iterator`, that value is pushed into the specified stream, followed by the separator string.

At this point, iterator adaptors might seem like little more than a curiosity. Sure, we can use an `ostream_iterator` to write values to `cout`, but we could already do that directly with `cout`. So what makes the iterator adaptors so useful? The key point is that iterator adaptors are *iterators*, and so they can be used in conjunction with the STL algorithms. Whenever an STL algorithm expects a regular iterator, you can supply an iterator adaptor instead to "trick" the algorithm into performing some complex task when it believes it's just writing values to a range. For example, let's revisit the `copy` algorithm now that we have `ostream_iterator`s. What happens if we use `copy` to copy values from a container to an `ostream_iterator`? That is, what is the output of the following code:

```
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
```

This code copies all of the elements from the `myVector` container to the range specified by the `ostream_iterator`. Normally, `copy` would duplicate the values from `myVector` at another location, but since we've written the values to an `ostream_iterator`, this code will instead print all of the values from the `vector` to `cout`, separated by spaces. This means that this single line of code prints out `myVector`!

Of course, this is just one of many iterator adaptors. We initially discussed iterator adaptors as a way to break the "vicious cycle" where algorithms need space to hold their results, but the amount of space needed can only be calculated by running the algorithm. To resolve this issue, the standard library provides a collection of special iterator adapters called *insert iterators*. These are output iterators that, when written to, insert the value into a container using one of the `insert`, `push_back`, or `push_front` functions. As a simple example, let's consider the `back_insert_iterator`. `back_insert_iterator` is an iterator that, when written to, calls `push_back` on a specified STL sequence containers (i.e. `vector` or `deque`) to store the value. For example, consider the following code snippet:

```
vector<int> myVector; /* Initially empty */

/* Create a back_insert_iterator that inserts values into myVector. */
back_insert_iterator< vector<int> > itr(myVector);

for (int i = 0; i < 10; ++i) {
    *itr = i; // "Write" to the back_insert_iterator, appending the value.
    ++itr;
}

/* Print the vector contents; this displays 0 1 2 3 4 5 6 7 8 9 */
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
```

This code is fairly dense, so let's go over it in some more detail. The first line simply creates an empty `vector<int>`. The next line is

```
back_insert_iterator< vector<int> > itr(myVector);
```

This code creates a `back_insert_iterator` which inserts into a `vector<int>`. This syntax might be a bit strange, since the iterator type is parameterized over the type of the *container* it inserts into, not the type of the elements stored in that container. Moreover, notice that we indicated to the iterator that it should insert into the `myVector` container by surrounding the container name in parentheses. From this point, any values written to the `back_insert_iterator` will be stored inside of `myVector` by calling `push_back`.

We then have the following loop, which indirectly adds elements to the `vector`:

```
for (int i = 0; i < 10; ++i) {
    *itr = i; // "Write" to the back_insert_iterator, appending the value.
    ++itr;
}
```

Here, the line `*itr = i` will implicitly call `myVector.push_back(i)`, adding the value to the `vector`. Thus, when we encounter the final line:

```
copy(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, " "));
```

the call to `copy` will print out the numbers 0 through 9, inclusive, since they've been stored in the `vector`.

In practice, it is rare to see `back_insert_iterator` used like this. This type of iterator is almost exclusively used as a parameter to STL algorithms that need a place to store a result. For example, consider the `reverse_copy` algorithm. Like `copy`, `reverse_copy` takes in three iterators, two delineating an input range and one specifying a destination, then copies the elements from the input range to the destination. However, unlike the regular `copy` algorithm, `reverse_copy` copies the elements in reverse order. For example, using `reverse_copy` to copy the sequence 0, 1, 2, 3, 4 to a destination would cause the destination range to hold the sequence 4, 3, 2, 1, 0. Suppose that we are interested in using the `reverse_copy` algorithm to make a copy of a `vector` with the elements in reverse order as the original. Then we could do so as follows:

```
vector<int> original = /* ... */
vector<int> destination;
reverse_copy(original.begin(), original.end(),
             back_insert_iterator< vector<int> >(destination));
```

The syntax `back_insert_iterator<vector<int> >` is admittedly bit clunky, and fortunately there's a shorthand. To create a `back_insert_iterator` that inserts elements into a particular container, you can write

```
back_inserter(container);
```

Thus the above code with `reverse_copy` could be rewritten as

```
vector<int> original = /* ... */
vector<int> destination;
reverse_copy(original.begin(), original.end(), back_inserter(destination));
```

This is much cleaner than the original and is likely to be what you'll see in practice.

The `back_inserter` is a particularly useful container when you wish to store the result of an operation in a `vector` or `deque`, but cannot be used in conjunction with `map` or `set` because those containers do not

support the `push_back` member function. For those containers, you can use the more general `insert_iterator`, which insert elements into arbitrary positions in a container. A great example of `insert_iterator` in action arises when computing the union, intersection, or difference of two sets. Mathematically speaking, the *union* of two sets is the set of elements contained in *either* of the sets, the *intersection* of two sets is the set of elements contained in *both* of the sets, and the *difference* of two sets is the set of elements contained in the first set but not in the second. These operations are exported by the STL algorithms as `set_union`, `set_intersection`, and `set_difference`. These algorithms take in five parameters – two pairs of iterator ranges defining what ranges to use as the input sets, along with one final iterator indicating where the result should be written. As with all STL algorithms, the set algorithms assume that the destination range has enough space to store the result of the operation, and again we run into a problem because we cannot tell how many elements will be produced by the algorithm. This is an ideal spot for an `insert_iterator`. Given two sets `one` and `two`, we can compute the union of those two `set`s as follows:

```
set<int> result;
set_union(setOne.begin(), setOne.end(),      // All of the elements in setOne
          setTwo.begin(), setTwo.end(),      // All of the elements in setTwo
          inserter(result, result.begin())); // Store in result.
```

Notice that the last parameter is `inserter(result, result.begin())`. This is an insert iterator that inserts its elements into the `result` set. For somewhat technical reasons, when inserting elements into a `set`, you must specify both the container and the container's `begin` iterator as parameters, though the generated elements will be stored in sorted order.

All of the iterator adaptors we've encountered so far have been used to channel the output of an algorithm to a location other than an existing range of elements. `ostream_iterator` writes values to streams, `back_insert_iterator` invokes `push_back` to make space for its elements, etc. However, there is a particularly useful iterator adapter, the `istream_iterator`, which is an *input* iterator. That is, `istream_iterator`s can be used to provide data as inputs to particular STL algorithms. As its name suggests, `istream_iterator` can be used to read values from a stream as if it were a container of elements. To illustrate `istream_iterator`, let's return to the example from the start of this chapter. If you'll recall, we wrote a program that read in a list of numbers from a file, then computed their average. In this program, we read in the list of numbers using the following `while` loop:

```
int currValue;
while (input >> currValue)
    values.insert(currValue);
```

Here, `values` is a `multiset<int>`. This code is equivalent to the following, which uses the STL `copy` algorithm in conjunction with an `inserter` and two `istream_iterator`s:

```
copy(istream_iterator<int>(input), istream_iterator<int>(),
     inserter(values, values.begin()));
```

This is perhaps the densest single line of code we've encountered yet, so let's dissect it to see how it works. Recall that the `copy` algorithm copies the values from an iterator range and stores them in the range specified by the destination iterator. Here, our destination is an `inserter` that adds elements into the `values` `multiset`. Our input is the pair of iterators

```
istream_iterator<int>(input), istream_iterator<int>()
```

What exactly does this mean? Whenever a value is read from an `istream_iterator`, the iterator uses the stream extraction operator `>>` to read a value of the proper type from the input stream, then returns it.

Consequently, the iterator `istream_iterator<int>(input)` is an iterator that reads `int` values out of the stream `input`. The second iterator, `istream_iterator<int>()`, is a bit stranger. This is a special `istream_iterator` called the *end-of-stream iterator*. When defining ranges with STL iterators, it is always necessary to specify two iterators, one for the beginning of the range and one that is one past the end of it. When working with STL containers this is perfectly fine, since the size of the container is known. However, when working with streams, it's unclear exactly how many elements that stream will contain. If the stream is an `ifstream`, the number of elements that can be read depends on the contents of the file. If the stream is `cin`, the number of elements that can be read depends on how many values the user decides to enter. To get around this, the STL designers used a bit of a hack. When reading values from a stream with an `istream_iterator`, whenever no more data is available in the stream (either because the stream entered a fail state, or because the end of the file was reached), the `istream_iterator` takes on a special value which indicates "there is no more data in the stream." This value can be formed by constructing an `istream_iterator` without specifying what stream to read from. Thus in the code

```
copy(istream_iterator<int>(input), istream_iterator<int>(),
     inserter(values, values.begin()));
```

the two `istream_iterator`s define the range from the beginning of the input stream up until no more values can be read from the stream.

The following table lists some of the more common iterator adapters and provides some useful context. You'll likely refer to this table most when writing code that uses algorithms.

`back_insert_iterator<Container>`	`back_insert_iterator<vector<int> > itr(myVector);`
	`back_insert_iterator<deque<char> > itr = back_inserter(myDeque);`
	
	An output iterator that stores elements by calling `push_back` on the specified container. You can declare `back_insert_iterator`s explicitly, or can create them with the function `back_inserter`.

`front_insert_iterator<Container>`	`front_insert_iterator<deque<int> > itr(myIntDeque);`
	`front_insert_iterator<deque<char> > itr = front_inserter(myDeque);`
	
	An output iterator that stores elements by calling `push_front` on the specified container. Since the container must have a `push_front` member function, you cannot use a `front_insert_iterator` with a `vector`. As with `back_insert_iterator`, you can create `front_insert_iterator`s with the the `front_inserter` function.

`insert_iterator<Container>`	`insert_iterator<set<int> > itr(mySet, mySet.begin());`
	`insert_iterator<set<int> > itr = inserter(mySet, mySet.begin());`
	
	An output iterator that stores its elements by calling `insert` on the specified container to insert elements at the indicated position. You can use this iterator type to insert into any container, especially `set`. The special function `inserter` generates `insert_iterator`s for you.

`ostream_iterator<Type>`	`ostream_iterator<int> itr(cout, " ");`
	`ostream_iterator<char> itr(cout);`
	`ostream_iterator<double> itr(myStream, "\n");`
	
	An output iterator that writes elements into an output stream. In the constructor, you must initialize the `ostream_iterator` to point to an `ostream`, and can optionally provide a separator string written after every element.

`istream_iterator<Type>`	`istream_iterator<int> itr(cin);  // Reads from cin`
	`istream_iterator<int> endItr;  // Special end value`
	
	An input iterator that reads values from the specified `istream` when dereferenced. When `istream_iterator`s reach the end of their streams (for example, when reading from a file), they take on a special "end" value that you can get by creating an `istream_iterator` with no parameters. `istream_iterator`s are susceptible to stream failures and should be used with care.

`ostreambuf_iterator<char>`	`ostreambuf_iterator<char> itr(cout); // Write to cout` An output iterator that writes raw character data to an output stream. Unlike `ostream_iterator`, which can print values of any type, `ostreambuf_iterator` can only write individual characters. `ostreambuf_iterator` is usually used in conjunction with `istreambuf_iterator`.

`istreambuf_iterator<char>`	`istreambuf_iterator<char> itr(cin); // Read data from cin`
	`istreambuf_iterator<char> endItr;  // Special end value`
	
	An input iterator that reads unformatted data from an input stream. `istreambuf_iterator` always reads in character data and will not skip over whitespace. Like `istream_iterator`, `istreambuf_iterator`s have a special iterator constructed with no parameters which indicates "end of stream." `istreambuf_iterator` is used primarily to read raw data from a file for processing with the STL algorithms.

Removal Algorithms

The STL provides several algorithms for removing elements from containers. However, removal algorithms have some idiosyncrasies that can take some time to adjust to.

Despite their name, removal algorithms **do not** actually remove elements from containers. This is somewhat counterintuitive but makes sense when you think about how algorithms work. Algorithms accept *iterators*, not *containers*, and thus do not know how to erase elements from containers. Removal functions work by shuffling down the contents of the container to overwrite all elements that need to be erased. Once finished, they return iterators to the first element not in the modified range. So for example, if you have a `vector` initialized to 0, 1, 2, 3, 3, 3, 4 and then `remove` all instances of the number 3, the resulting `vector` will contain 0, 1, 2, 4, 3, 3, 4 and the function will return an iterator to one spot past the first 4. If you'll notice, the elements in the iterator range starting at `begin` and ending with the element one past the four are the sequence 0, 1, 2, 4 – exactly the range we wanted.

To truly remove elements from a container with the removal algorithms, you can use the container class member function `erase` to erase the range of values that aren't in the result. For example, here's a code snippet that removes all copies of the number 137 from a `vector`:

```
myVector.erase(remove(myVector.begin(), myVector.end(), 137), myVector.end());
```

Note that we're erasing elements in the range [*, end), where * is the value returned by the remove algorithm.

There is another useful removal function, remove_if, that removes all elements from a container that satisfy a condition specified as the final parameter. For example, using the ispunct function from the header file <cctype>, we can write a StripPunctuation function that returns a copy of a string with all the punctuation removed:*

```
string StripPunctuation(string input) {
    input.erase(remove_if(input.begin(), input.end(), ispunct), input.end());
    return input;
}
```

(Isn't it amazing how much you can do with a single line of code? That's the real beauty of STL algorithms.)

If you're shaky about how to actually remove elements in a container using remove, you might want to consider the remove_copy and remove_copy_if algorithms. These algorithms act just like remove and remove_if, except that instead of modifying the original range of elements, they copy the elements that aren't removed into another container. While this can be a bit less memory efficient, in some cases it's exactly what you're looking for.

Other Noteworthy Algorithms

The past few sections have focused on common genera of algorithms, picking out representatives that illustrate the behavior of particular algorithm classes. However, there are many noteworthy algorithms that we have not discussed yet. This section covers several of these algorithms, including useful examples.

A surprisingly useful algorithm is transform, which applies a function to a range of elements and stores the result in the specified destination. transform accepts four parameters – two iterators delineating an input range, an output iterator specifying a destination, and a callback function, then stores in the output destination the result of applying the function to each element in the input range. As with other algorithms, transform assumes that there is sufficient storage space in the range pointed at by the destination iterator, so make sure that you have sufficient space before transforming a range.

transform is particularly elegant when combined with functors, but even without them is useful for a whole range of tasks. For example, consider the tolower function, a C library function declared in the header <cctype> that accepts a char and returns the lowercase representation of that character. Combined with transform, this lets us write ConvertToLowerCase from strutils.h in two lines of code, one of which is a return statement:

```
string ConvertToLowerCase(string text) {
    transform(text.begin(), text.end(), text.begin(), tolower);
    return text;
}
```

Note that after specifying the range text.begin(), text.end() we have another call to text.begin(). This is because we need to provide an iterator that tells transform where to put its output. Since we want to overwrite the old contents of our container with the new values, we specify text.begin() another time to indicate that transform should start writing elements to the beginning of the string as it generates them.

---

* On some compilers, this code will not compile as written. See the later section on compatibility issues for more information.

There is no requirement that the function you pass to `transform` return elements of the same type as those stored in the container. It's legal to `transform` a set of `string`s into a set of `double`s, for example.

Most of the algorithms we've seen so far operate on entire ranges of data, but not all algorithms have this property. One of the most useful (and innocuous-seeming) algorithms is `swap`, which exchanges the values of two variables. We first encountered `swap` two chapters ago when discussing sorting algorithms, but it's worth repeating. Several advanced C++ techniques hinge on `swap`'s existence, and you will almost certainly encounter it in your day-to-day programming even if you eschew the rest of the STL.

Two last algorithms worthy of mention are the `min_element` and `max_element` algorithms. These algorithms accept as input a range of iterators and return an iterator to the largest element in the range. As with other algorithms, by default the elements are compared by `<`, but you can provide a binary comparison function to the algorithms as a final parameter to change the default comparison order.

The following table lists some of the more common STL algorithms. It's by no means an exhaustive list, and you should consult a reference to get a complete list of all the algorithms available to you.

`Type accumulate(InputItr start,`
`InputItr stop,`
`Type value)`
	Returns the sum of the elements in the range [`start`, `stop`) plus the value of `value`.

`bool binary_search(RandomItr start,`
`RandomItr stop,`
`const Type& value)`
	Performs binary search on the sorted range specified by [`start`, `stop`) and returns whether it finds the element `value`. If the elements are sorted using a special comparison function, you must specify the function as the final parameter.

`OutItr copy(InputItr start,`
`InputItr stop,`
`OutItr outputStart)`
	Copies the elements in the range [`start`, `stop`) into the output range starting at `outputStart`. `copy` returns an iterator to one past the end of the range written to.

`size_t count(InputItr start,`
`InputItr end,`
`const Type& value)`
	Returns the number of elements in the range [`start`, `stop`) equal to `value`.

`size_t count_if(InputItr start,`
`InputItr end,`
`PredicateFunction fn)`
	Returns the number of elements in the range [`start`, `stop`) for which `fn` returns true. Useful for determining how many elements have a certain property.

`bool equal(InputItr start1,`
`InputItr stop1,`
`InputItr start2)`
	Returns whether elements contained in the range defined by [`start1`, `stop1`) and the range beginning with start2 are `equal`. If you have a special comparison function to compare two elements, you can specify it as the final parameter.

`pair<RandomItr, RandomItr>`
`equal_range(RandomItr start,`
`RandomItr stop,`
`const Type& value)`
	Returns two iterators as a `pair` that defines the sub-range of elements in the sorted range [`start`, `stop`) that are equal to `value`. In other words, every element in the range defined by the returned iterators is equal to `value`. You can specify a special comparison function as a final parameter.

`void fill(ForwardItr start,`
`ForwardItr stop,`
`const Type& value)`
	Sets every element in the range [`start`, `stop`) to `value`.

`void fill_n(ForwardItr start,`
`size_t num,`
`const Type& value)`
	Sets the first `num` elements, starting at `start`, to `value`.

`InputItr find(InputItr start,`
`InputItr stop,`
`const Type& value)`
	Returns an iterator to the first element in [`start`, `stop`) that is equal to `value`, or `stop` if the value isn't found. The range doesn't need to be sorted.

`InputItr find_if(InputItr start, InputItr stop, PredicateFunc fn)`	Returns an iterator to the first element in [`start`, `stop`) for which `fn` is true, or `stop` otherwise.
`Function for_each(InputItr start, InputItr stop, Function fn)`	Calls the function `fn` on each element in the range [`start`, `stop`).
`void generate(ForwardItr start, ForwardItr stop, Generator fn);`	Calls the zero-parameter function `fn` once for each element in the range [`start`, `stop`), storing the return values in the range.
`void generate_n(OutputItr start, size_t n, Generator fn);`	Calls the zero-parameter function `fn` `n` times, storing the results in the range beginning with `start`.
`bool includes(InputItr start1, InputItr stop1, InputItr start2, InputItr stop2)`	Returns whether every element in the sorted range [`start2`, `stop2`) is also in [`start1`, `stop1`). If you need to use a special comparison function, you can specify it as the final parameter.
`Type inner_product(InputItr start1, InputItr stop1, InputItr start2, Type initialValue)`	Computes the inner product of the values in the range [start1, stop1) and [start2, start2 + (stop1 – start1)). The inner product is the value $\sum_{i=1}^{n} a_i b_i + initialValue$, where a_i and b_i denote the ith elements of the first and second range.
`bool lexicographical_compare(InputItr s1, InputItr s2, InputItr t1, InputItr t2)`	Returns whether the range of elements defined by [`s1`, `s2`) is lexicographically less than [`t1`, `t2`); that is, if the first range precedes the second in a "dictionary ordering."
`InputItr lower_bound(InputItr start, InputItr stop, const Type& elem)`	Returns an iterator to the first element greater than or equal to the element `elem` in the sorted range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`InputItr max_element(InputItr start, InputItr stop)`	Returns an iterator to the largest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.
`InputItr min_element(InputItr start, InputItr stop)`	Returns an iterator to the smallest value in the range [`start`, `stop`). If you need to use a special comparison function, you can specify it as the final parameter.

`bool next_permutation(BidirItr start, BidirItr stop)`	Given a range of elements [`start`, `stop`), modifies the range to contain the next lexicographically higher permutation of those elements. The function then returns whether such a permutation could be found. It is common to use this algorithm in a `do ... while` loop to iterate over all permutations of a range of data, as shown here:
	
	`sort(range.begin(), range.end());`
	`do {`
	`  /* ... process ... */`
	`}while(next_permutation(range.begin(), range.end()));`

`bool prev_permutation(BidirItr start, BidirItr stop)`	Given a range of elements [`start`, `stop`), modifies the range to contain the next lexicographically lower permutation of those elements. The function then returns whether such a permutation could be found.
`void random_shuffle(RandomItr start, RandomItr stop)`	Randomly reorders the elements in the range [`start`, `stop`).

`ForwardItr remove(ForwardItr start, ForwardItr stop, const Type& value)`	Removes all elements in the range [`start`, `stop`) that are equal to `value`. This function will **not** remove elements from a container. To shrink the container, use the container's `erase` function to erase all values in the range [`retValue`, `end()`), where `retValue` is the return value of `remove`.
`ForwardItr remove_if(ForwardItr start, ForwardItr stop, PredicateFunc fn)`	Removes all elements in the range [`start`, `stop`) for which `fn` returns true. See `remove` for information about how to actually remove elements from the container.
`void replace(ForwardItr start, ForwardItr stop, const Type& toReplace, const Type& replaceWith)`	Replaces all values in the range [`start`, `stop`) that are equal to `toReplace` with `replaceWith`.
`void replace_if(ForwardItr start, ForwardItr stop, PredicateFunction fn, const Type& with)`	Replaces all elements in the range [`start`, `stop`) for which `fn` returns true with the value `with`.
`ForwardItr rotate(ForwardItr start, ForwardItr middle, ForwardItr stop)`	Rotates the elements of the container such that the sequence [`middle`, `stop`) is at the front and the range [`start`, `middle`) goes from the new middle to the end. `rotate` returns an iterator to the new position of `start`.
`ForwardItr search(ForwardItr start1, ForwardItr stop1, ForwardItr start2, ForwardItr stop2)`	Returns whether the sequence [`start2`, `stop2`) is a subsequence of the range [`start1`, `stop1`). To compare elements by a special comparison function, specify it as a final parameter.
`InputItr set_difference( InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutItr dest)`	Stores all elements that are in the sorted range [`start1`, `stop1`) but not in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
`InputItr set_intersection( InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutItr dest)`	Stores all elements that are in both the sorted range [`start1`, `stop1`) and the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
`InputItr set_union( InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutItr dest)`	Stores all elements that are in either the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`) in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
`InputItr set_symmetric_difference( InputItr start1, InputItr stop1, InputItr start2, InputItr stop2, OutItr dest)`	Stores all elements that are in the sorted range [`start1`, `stop1`) or in the sorted range [`start2`, `stop2`), but not both, in the destination pointed to by `dest`. If the elements are sorted according to a special comparison function, you can specify the function as the final parameter.
`void swap(Value& one, Value& two)`	Swaps the values of `one` and `two`.
`ForwardItr swap_ranges(ForwardItr start1, ForwardItr stop1, ForwardItr start2)`	Swaps each element in the range [`start1`, `stop1`) with the correspond elements in the range starting with `start2`.

`OutputItr transform(InputItr start, InputItr stop, OutputItr dest, Function fn)`	Applies the function `fn` to all of the elements in the range [`start`, `stop`) and stores the result in the range beginning with `dest`. The return value is an iterator one past the end of the last value written.
`RandomItr upper_bound(RandomItr start, RandomItr stop, const Type& val)`	Returns an iterator to the first element in the sorted range [`start`, `stop`) that is strictly greater than the value `val`. If you need to specify a special comparison function, you can do so as the final parameter.

A Word on Compatibility

The STL is ISO-standardized along with the rest of C++. Ideally, this would mean that all STL implementations are uniform and that C++ code that works on one compiler should work on any other compiler. Unfortunately, this is not the case. No compilers on the market fully adhere to the standard, and almost universally compiler writers will make minor changes to the standard that decrease portability.

Consider, for example, the `ConvertToLowerCase` function from earlier in the section:

```
string ConvertToLowerCase(string text) {
    transform(text.begin(), text.end(), text.begin(), tolower);
    return text;
}
```

This code will compile in Microsoft Visual Studio, but not in Xcode or the popular Linux compiler g++. The reason is that there are *two* `tolower` functions – the original C `tolower` function exported by `<cctype>` and a more modern `tolower` function exported by the `<locale>` header. Unfortunately, Xcode and g++ cannot differentiate between the two functions, so the call to `transform` will result in a compiler error. To fix the problem, you must explicitly tell C++ which version of `tolower` you want to call as follows:

```
string ConvertToLowerCase(string text) {
    transform(text.begin(), text.end(), text.begin(), ::tolower);
    return text;
}
```

Here, the strange-looking `::` syntax is the *scope-resolution operator* and tells C++ that the `tolower` function is the original C function rather than the one exported by the `<locale>` header. Thus, if you're using Xcode or g++ and want to use the functions from `<cctype>`, you'll need to add the `::`.

Another spot where compatibility issues can lead to trouble arises when using STL algorithms with the STL `set`. Consider the following code snippet, which uses `fill` to overwrite all of the elements in an STL `set` with the value 137:

```
fill(mySet.begin(), mySet.end(), 137);
```

This code will compile in Visual Studio, but will not under g++. Recall from the second chapter on STL containers that manipulating the contents of an STL `set` in-place can destroy the set's internal ordering. Visual Studio's implementation of `set` will nonetheless let you modify `set` contents, even in situations like the above where doing so is unsafe. g++, however, uses an STL implementation that treats all `set` iterators as read-only. Consequently, this code won't compile, and in fact will cause some particularly nasty compiler errors.

When porting C++ code from one compiler to another, you might end up with inexplicable compiler errors. If you find some interesting C++ code online that doesn't work on your compiler, it doesn't necessarily

mean that the code is invalid; rather, you might have an overly strict compiler or the online code might use an overly lenient one.

Extended Example: Palindromes

> *A man, a plan, a caret, a ban, a myriad, a sum, a lac, a liar, a hoop, a pint, a catalpa, a gas, an oil, a bird, a yell, a vat, a caw, a pax, a wag, a tax, a nay, a ram, a cap, a yam, a gay, a tsar, a wall, a car, a luger, a ward, a bin, a woman, a vassal, a wolf, a tuna, a nit, a pall, a fret, a watt, a bay, a daub, a tan, a cab, a datum, a gall, a hat, a tag, a zap, a say, a jaw, a lay, a wet, a gallop, a tug, a trot, a trap, a tram, a torr, a caper, a top, a tonk, a toll, a ball, a fair, a sax, a minim, a tenor, a bass, a passer, a capital, a rut, an amen, a ted, a cabal, a tang, a sun, an ass, a maw, a sag, a jam, a dam, a sub, a salt, an axon, a sail, an ad, a wadi, a radian, a room, a rood, a rip, a tad, a pariah, a revel, a reel, a reed, a pool, a plug, a pin, a peek, a parabola, a dog, a pat, a cud, a nu, a fan, a pal, a rum, a nod, an eta, a lag, an eel, a batik, a mug, a mot, a nap, a maxim, a mood, a leek, a grub, a gob, a gel, a drab, a citadel, a total, a cedar, a tap, a gag, a rat, a manor, a bar, a gal, a cola, a pap, a yaw, a tab, a raj, a gab, a nag, a pagan, a bag, a jar, a bat, a way, a papa, a local, a gar, a baron, a mat, a rag, a gap, a tar, a decal, a tot, a led, a tic, a bard, a leg, a bog, a burg, a keel, a doom, a mix, a map, an atom, a gum, a kit, a baleen, a gala, a ten, a don, a mural, a pan, a faun, a ducat, a pagoda, a lob, a rap, a keep, a nip, a gulp, a loop, a deer, a leer, a lever, a hair, a pad, a tapir, a door, a moor, an aid, a raid, a wad, an alias, an ox, an atlas, a bus, a madam, a jag, a saw, a mass, an anus, a gnat, a lab, a cadet, an em, a natural, a tip, a caress, a pass, a baronet, a minimax, a sari, a fall, a ballot, a knot, a pot, a rep, a carrot, a mart, a part, a tort, a gut, a poll, a gateway, a law, a jay, a sap, a zag, a tat, a hall, a gamut, a dab, a can, a tabu, a day, a batt, a waterfall, a patina, a nut, a flow, a lass, a van, a mow, a nib, a draw, a regular, a call, a war, a stay, a gam, a yap, a cam, a ray, an ax, a tag, a wax, a paw, a cat, a valley, a drib, a lion, a saga, a plat, a catnip, a pooh, a rail, a calamus, a dairyman, a bater, a canal – Panama!*
>
> – Dan Hoey [Pic96]

It is fitting to conclude our whirlwind tour of the STL with an example showcasing exactly how concise and powerful well-written STL code can be. This example is shorter than the others in this book, but should nonetheless illustrate how the different library pieces all fit together. Once you've finished reading this chapter, you should have a solid understanding of how the STL and streams libraries can come together beautifully to elegantly solve a problem.

Palindromes

A *palindrome* is a word or phrase that is the same when read forwards or backwards, such as “racecar” or “Malayalam.” It is customary to ignore spaces, punctuation, and capitalization when reading palindromes, so the phrase “Mr. Owl ate my metal worm” would count as a palindrome, as would “Go hang a salami! I'm a lasagna hog.”

Suppose that we want to write a function `IsPalindrome` that accepts a `string` and returns whether or not the string is a palindrome. Initially, we'll assume that spaces, punctuation, and capitalization are all significant in the string, so “Party trap” would not be considered a palindrome, though “Part y traP” would. Don't worry – we'll loosen this restriction in a bit. Now, we want to verify that the string is the same when read forwards and backwards. There are many possible ways to do this. Prior to learning the STL, we might have written this function as follows:

```
bool IsPalindrome(string input) {
    for(int k = 0; k < input.size() / 2; ++k)
        if(input[k] != input[input.length() - 1 - k])
            return false;
    return true;
}
```

That is, we simply iterate over the first half of the string checking to see if each character is equal to its respective character on the other half of the string. There's nothing wrong with the approach, but it feels too *mechanical*. The high-level operation we're modeling asks whether the first half of the string is the same forwards as the second half is backwards. The code we've written accomplishes this task, but has to explicitly walk over the characters from start to finish, manually checking each pair. Using the STL, we can accomplish the same result as above without explicitly spelling out the details of how to check each character.

There are several ways we can harness the STL to solve this problem. For example, we could use the STL `reverse` algorithm to create a copy of the string in reverse order, then check if the string is equal to its reverse. This is shown here:

```
bool IsPalindrome(string input) {
    string reversed = input;
    reverse(input.begin(), input.end());
    return reversed == input;
}
```

This approach works, but requires us to create a copy of the string and is therefore less efficient than our original implementation. Can we somehow emulate the functionality of the initial `for` loop using iterators? The answer is yes, thanks to `reverse_iterator`s. Every STL container class exports a type `reverse_iterator` which is similar to an iterator except that it traverses the container backwards. Just as the `begin` and `end` functions define an iterator range over a container, the `rbegin` and `rend` functions define a `reverse_iterator` range spanning a container.

Let's also consider the the STL `equal` algorithm. `equal` accepts three inputs – two iterators delineating a range and a third iterator indicating the start of a second range – then returns whether the two ranges are equal. Combined with `reverse_iterator`s, this yields the following *one-line implementation* of `IsPalindrome`:

```
bool IsPalindrome(string input) {
    return equal(input.begin(), input.begin() + input.size() / 2,
                 input.rbegin());
}
```

This is a remarkably simple approach that is identical to what we've written earlier but much less verbose. Of course, it doesn't correctly handle capitalization, spaces, or punctuation, but we can take care of that with only a few more lines of code. Let's begin by stripping out everything from the string except for alphabetic characters. For this task, we can use the STL `remove_if` algorithm, which accepts as input a range of iterators and a predicate, then modifies the range by removing all elements for which the predicate returns true. Like its partner algorithm `remove`, `remove_if` doesn't actually remove the elements from the sequence (see the last chapter for more details), so we'll need to `erase` the remaining elements afterwards.

Because we want to eliminate all characters from the string that are not alphabetic, we need to create a predicate function that accepts a character and returns whether it is not a letter. The header file

<cctype> exports a helpful function called isalpha that returns whether a character *is* a letter. This is the opposite what we want, so we'll create our own function which returns the negation of isalpha:*

```
bool IsNotAlpha(char ch) {
    return !isalpha(ch);
}
```

We can now strip out nonalphabetic characters from our input string as follows:

```
bool IsPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlpha),
                input.end());
    return equal(input.begin(), input.begin() + input.size() / 2,
                 input.rbegin());
}
```

Finally, we need to make sure that the string is treated case-insensitively, so inputs like "RACEcar" are accepted as palindromes. Using the code developed in the chapter on algorithms, we can convert the string to uppercase after stripping out everything except characters, yielding this final version of IsPalindrome:

```
bool IsPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlpha),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);
    return equal(input.begin(), input.begin() + input.size() / 2,
                 input.rbegin());
}
```

This function is remarkable in its elegance and terseness. In *three lines of code* we've stripped out all of the characters in a string that aren't letters, converted what's left to upper case, and returned whether the string is the same forwards and backwards. This is the STL in action, and I hope that you're beginning to appreciate the power of the techniques you've learned over the past few chapters.

Before concluding this example, let's consider a variant on a palindrome where we check whether the *words* in a phrase are the same forwards and backwards. For example, "Did mom pop? Mom did!" is a palindrome both with respect to its letters and its words, while "This is this" is a phrase that is not a palindrome but is a word-palindrome. As with regular palindromes, we'll ignore spaces and punctuation, so "It's an its" counts as a word-palindrome even though it uses two different forms of the word its/it's. The machinery we've developed above works well for entire strings; can we modify it to work on a word-by-word basis?

In some aspects this new problem is similar to the original. We still to ignore spaces, punctuation, and capitalization, but now need to treat words rather than letters as meaningful units. There are many possible algorithms for checking this property, but one solution stands out as particularly good. The idea is as follows:

1. Clean up the input: strip out everything except letters *and spaces*, then convert the result to upper case.
2. Break up the input into a list of words.
3. Return whether the list is the same forwards and backwards.

---

* When we cover the <functional> library in the second half of this book, you'll see a simpler way to do this.

In the first step, it's important that we preserve the spaces in the original input so that we don't lose track of word boundaries. For example, we would convert the string "Hello? Hello!? HELLO?" into "HELLO HELLO HELLO" instead of "HELLOHELLOHELLO" so that we can recover the individual words in the second step. Using a combination of the `isalpha` and `isspace` functions from `<cctype>` and the convert-to-upper-case code used above, we can preprocess the input as shown here:

```
bool IsNotAlphaOrSpace(char ch) {
    return !isalpha(ch) && !isspace(ch);
}

bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);
    /* ... */
}
```

At this point the string `input` consists of whitespace-delimited strings of uniform capitalization. We now need to tokenize the input into individual words. This would be tricky were it not for `stringstream`. Recall that when reading a `string` out of a stream using the stream extraction operator (>>), the stream treats whitespace as a delimiter. Thus if we funnel our string into a `stringstream` and then read back individual strings, we'll end up with a tokenized version of the input. Since we'll be dealing with an arbitrarily-long list of strings, we'll store the resulting list in a `vector<string>`, as shown here:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    /* ... */
}
```

Now, what is the easiest way to read strings out of the stream until no strings remain? We could do this manually, as shown here:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    string token;
    while(tokenizer >> token)
        tokens.push_back(token);
}
```

This code is correct, but it's bulky and unsightly. The problem is that it's just too *mechanical*. We want to insert all of the tokens from the `stringstream` into the `vector`, but as written it's not clear that this is what's happening. Fortunately, there is a much, *much* easier way to solve this problem thanks to `istream_iterator`. Recall that `istream_iterator` is an iterator adapter that lets you iterate over an input stream as if it were a range of data. Using `istream_iterator` to wrap the stream operations and

the `vector`'s `insert` function to insert a range of data, we can rewrite this entire loop in one line as follows:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    tokens.insert(tokens.begin(),
                  istream_iterator<string>(tokenizer),
                  istream_iterator<string>());
}
```

Recall that two `istream_iterator`s are necessary to define a range, and that an `istream_iterator` constructed with no arguments is a special "end of stream" iterator. This one line of code replaces the entire loop from the previous implementation, and provided that you have some familiarity with the STL this second version is also easier to read.

The last step in this process is to check if the sequence of strings is the same forwards and backwards. But we already know how to do this – we just use `equal` and a `reverse_iterator`. Even though the original implementation applied this technique to a `string`, we can use the same pattern here on a `vector<string>` because all the container classes are designed with a similar interface. Remarkable, isn't it?

The final version of `IsWordPalindrome` is shown here:

```
bool IsWordPalindrome(string input) {
    input.erase(remove_if(input.begin(), input.end(), IsNotAlphaOrSpace),
                input.end());
    transform(input.begin(), input.end(), input.begin(), ::toupper);

    stringstream tokenizer(input);
    vector<string> tokens;

    tokens.insert(tokens.begin(),
                  istream_iterator<string>(tokenizer),
                  istream_iterator<string>());
    return equal(tokens.begin(), tokens.begin() + tokens.size() / 2,
                 tokens.rbegin());
}
```

More to Explore

While this chapter lists some of the more common algorithms, there are many others that are useful in a variety of contexts. Additionally, there are some useful C/C++ library functions that work well with algorithms. If you're interested in maximizing your algorithmic firepower, consider looking into some of these topics:

1. **`<cctype>`**: This chapter briefly mentioned the `<cctype>` header, the C runtime library's character type library. `<cctype>` include support for categorizing characters (for example, `isalpha` to return if a character is a letter and `isxdigit` to return if a character is a valid hexadecimal digit) and formatting conversions (`toupper` and `tolower`).

2. **<cmath>**: The C mathematics library has all sorts of nifty functions that perform arithmetic operations like `sin`, `sqrt`, and `exp`. Consider looking into these functions if you want to use `transform` on your containers.

3. **Boost Algorithms**: As with most of the C++ Standard Library, the Boost C++ Libraries have a whole host of useful STL algorithms ready for you to use. One of the more useful Boost algorithm sets is the string algorithms, which extend the functionality of the `find` and `replace` algorithms on `string`s from dealing with single characters to dealing with entire strings.

Practice Problems

Algorithms are ideally suited for solving a wide variety of problems in a small space. Most of the following programming problems have short solutions – see if you can whittle down the space and let the algorithms do the work for you!

1. Give three reasons why STL algorithms are preferable over hand-written loops.

2. What does the `_if` suffix on an STL algorithm indicate? What about `_n`?

3. What are the five iterator categories?

4. Can an input iterator be used wherever a forward iterator is expected? That is, if an algorithm requires a forward iterator, is it legal to provide it an input iterator instead? What about the other way around?

5. Why do we need `back_insert_iterator` and the like? That is, what would happen with the STL algorithms if these iterator adaptors didn't exist?

6. The `distance` function, defined in the `<iterator>` header, takes in two iterators and returns the number of elements spanned by that iterator range. For example, given a `vector<int>`, calling

   ```
   distance(v.begin(), v.end());
   ```

   returns the number of elements in the container.

   Modify the code from this chapter that prints the average of the values in a file so that it instead prints the average of the values in the file between 25 and 75. If no elements are in this range, you should print a message to this effect. You will need to use a combination of `accumulate` and `distance`.

7. Using `remove_if` and a custom callback function, write a function `RemoveShortWords` that accepts a `vector<string>` and removes all strings of length 3 or less from it. This function can be written in two lines of code if you harness the algorithms correctly.

8. In n-dimensional space, the distance from a point (x_1, x_2, x_3, ..., x_n) to the origin is $\sqrt{x_1^2 + x_2^2 + x_3^2 + \ldots + x_n^2}$. Write a function `DistanceToOrigin` that accepts a `vector<double>` representing a point in space and returns the distance from that point to the origin. Do not use any loops – let the algorithms do the heavy lifting for you. *(Hint: Use the `inner_product` algorithm to compute the expression under the square root.)*

9. Write a function `BiasedSort` that accepts a `vector<string>` by reference and sorts the `vector` lexicographically, except that if the `vector` contains the string "Me First," that string is always at the front of the sorted list. This may seem like a silly problem, but can come up in some circumstances. For example, if you have a list of songs in a music library, you might want songs with the title "Untitled" to always appear at the top.

10. Write a function `CriticsPick` that accepts a `map<string, double>` of movies and their ratings (between 0.0 and 10.0) and returns a `set<string>` of the names of the top ten movies in the `map`. If there are fewer than ten elements in the `map`, then the resulting `set` should contain every string in the `map`. *(Hint: Remember that all elements in a `map<string, double>` are stored internally as `pair<string, double>`)*

11. Implement the `count` algorithm for `vector<int>`s. Your function should have the prototype `int count(vector<int>::iterator start, vector<int>::iterator stop, int element)` and should return the number of elements in the range [`start`, `stop`) that are equal to `element`.

12. Using the `generate_n` algorithm, the `rand` function, and a `back_insert_iterator`, show how to populate a `vector` with a specified number of random values. Then use `accumulate` to compute the average of the range.

13. The *median* of a range of data is the value that is bigger than half the elements in the range and smaller than half the elements in a range. For data sets with odd numbers of elements, this is the middle element when the elements are sorted, and for data sets with an even number of elements it is the average of the two middle elements. Using the `nth_element` algorithm, write a function that computes the median of a set of data.

14. Show how to use a combination of `copy`, `istreambuf_iterator`, and `ostreambuf_iterator` to open a file and print its contents to `cout`.

15. Show how to use a combination of `copy` and iterator adapters to write the contents of an STL container to a file, where each element is stored on its own line.

16. Suppose that you are given two `vector<int>`s with their elements stored in sorted order. Show how to print out the elements those `vector`s have in common in one line of code using the `set_intersection` algorithm and an appropriate iterator adaptor.

17. A *monoalphabetic substitution cipher* is a simple form of encryption. We begin with the letters of the alphabet, as shown here:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

We then scramble these letters randomly, yielding a new ordering of the alphabet. One possibility is as follows:

K	V	D	Q	J	W	A	Y	N	E	F	C	L	R	H	U	X	I	O	G	T	Z	P	M	S	B

This new ordering thus defines a mapping from each letter in the alphabet to some other letter in the alphabet, as shown here:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
K	V	D	Q	J	W	A	Y	N	E	F	C	L	R	H	U	X	I	O	G	T	Z	P	M	S	B

To encrypt a source string, we simply replace each character in the string with its corresponding encrypted character. For example, the string "The cookies are in the fridge" would be encoded as follows:

T	H	E	C	O	O	K	I	E	S	A	R	E	I	N	T	H	E	F	R	I	D	G	E
G	**Y**	**J**	**D**	**H**	**H**	**F**	**N**	**J**	**O**	**K**	**I**	**J**	**N**	**R**	**G**	**Y**	**J**	**W**	**I**	**N**	**Q**	**A**	**J**

Monoalphabetic substitution ciphers are surprisingly easy to break – in fact, most daily newspapers include a daily puzzle that involves deciphering a monoalphabetic substitution cipher – but they are still useful for low-level encryption tasks such as posting spoilers to websites (where viewing the spoiler explicitly requires the reader to decrypt the text).

Using the `random_shuffle` algorithm, implement a function `MonoalphabeticSubstitutionEncrypt` that accepts a source string and encrypts it with a random monoalphabetic substitution cipher.

Part Two

Data Abstraction

It's all just bits and bytes.

Everything on your machine, whether it's your tax return, a picture from a trip, a web page, or a web browser, is stored in memory as a series of ones and zeros encoded as magnetic, optical, or electrical signals. How, then, can a computer do word processing? Or view images? Or check your email? All of this data has *structure* – text documents store words and fonts, images vivid color pictures, and email a mixture of text, headers, and contacts.

Chapter 8: Abstraction and Classes

Software keeps getting bigger. Society keeps digitizing and automating more and more aspects of life, and the scope and complexity of software systems are ever increasing. For computer scientists, this is a thrilling prospect: even after decades of booming growth, the field is still expanding and applications abound. But for software engineers - the brave souls who actually write the code – this can be daunting.

In the early days of programming, software was considerably less complicated because the tasks we used to ask of computers are nowhere near as complex as those we ask today. Operating systems worked on less powerful hardware and with considerably fewer peripherals. The earliest web browsers didn't need to support a wide array of HTML, CSS, JavaScript, XML, SVG, and RSS formats. Video games didn't need to take advantage of the latest-and-greatest 3D hardware and weren't criticized for not having the most up-to-date shading engine. But nowadays, the expectations are higher, and software is growing more complicated.

Unfortunately, increasing the size of a software system greatly increases the system's complexity and opens all sorts of avenues for failure. Combating software complexity is therefore extraordinarily important as it allows software systems to grow robustly. This section of this book is dedicated entirely to techniques for combating complexity through a particular technique called *abstraction*. Abstraction is a subtle but important aspect of software design, and in many ways the difference between good programmers and excellent programmers is the ability to design robust and intuitive abstractions in software.

Many textbooks jump directly into a discussion of what abstraction is all about and how to represent it in software, but I feel that doing so obscures the fundamental reasons underlying abstraction. This chapter discusses how software engineering is different from other engineering disciplines, why software complexity is particularly dangerous, and how abstraction can dramatically reduce the complexity of a software system. It then introduces the `class` keyword and how to represent abstraction directly in source code.

The Complexity of Software

What exactly does it mean to talk about the complexity of a software system? One of the first metrics that may come to mind is the number of lines of code in the program. This is akin to measuring the complexity of a chip by the number of transistors on it or a bridge by the number of welds required: while in general system complexity rises with lines of code, a program with ten thousand lines of code is not necessarily ten times more complicated than a system with one thousand lines of code. However, number of lines of code is still a reasonable metric of software complexity. To give you a sense for how massive some projects can be, here is a list of various software projects and the number of lines of code they contain:

1 – 10	Hello, World!
10 – 100	Most implementation of the STL `stack` or `queue`.
100 – 1,000	Most of the worked examples in this book.
1,000 – 10,000	Intensive team project in a typical computer science curriculum.
10,000 – 100,000	Most Linux command-line utilities.
100,000 – 1,000,000	Linux `g++` Compiler
1,000,000 – 10,000,000	Mozilla Firefox
10,000,000 – 100,000,000	Microsoft Windows 2000 Kernel
100,000,000 - 1,000,000,000	Debian Linux Operating System

The number of lines of code in each of these entries is ten times more than in the previous example. This means that there are ten times as many lines of code in Firefox than in g++, for example. And yes, you did read this correctly – there are many, *many* projects that clock in at over a million lines of code. The Debian Linux kernel is roughly 230 million lines of code as of this writing. It's generally accepted that no single programmer can truly understand more than fifty thousand lines of code, which means that in all but the simplest of programs, no one programmer understands how the entire system works.

So software is complex – so what? That is, why does it matter that modern software systems are getting bigger and more complicated? There are many reasons, of which two specifically stand out.

Every Bit Counts

In a software system, a single incorrect bit can spell disaster for the entire program. For example, suppose you are designing a system that controls a nuclear reactor core. At some point in your program, you have the following control logic:

```
if (MeltdownInProgress()) {
   SetReactorPower(0);
   EmergencyShutoff();
}
```

Let's suppose that `MeltdownInProgress` returns a `bool` that signals whether the reactor is melting down. On most systems, `bool`s are represented as a sequence of ones and zeros called *bits*. The value `false` is represented by those bits all being zero, and `true` represented by any of the bits being nonzero. For example, the value `00010000` would be interpreted as `true`, while `00000000` would be `false`. This means that the difference between `true` and `false` is a single bit. In the above example, this means that the difference between shutting down the reactor in an emergency and continuing normal operation is a *single bit in memory*.

In the above example, our “single incorrect bit” was the difference between the boolean values `true` and `false`, but in most systems the “single incorrect bit” will be something else. It might, for example, be an negative integer where a positive integer was expected, an iterator that is past the end of a container, or an unsorted vector when the data was expected to be sorted. In each of these cases, the erroneous data is likely to be only a handful of bits off from a meaningful piece of data, but the result will be the same – the program won't work as expected.

Interactions Grow Exponentially

Suppose you have a program with *n* lines of code. Let's consider an "interaction" to be where data is manipulated by two different lines of code. For example, one interaction might be creating an `int` in one line and then printing it to the console in the next, or a function passing one of its parameters into another function. Since every line of code might potentially manipulate data created by any of the other *n* lines of code, if you sum up this count for all *n* lines of code there are roughly n^2 pairs of interacting lines. This which means that the number of possible interactions in a software project increases, in the worst case, as the *square* of the number of lines of code.

Let's consider a slightly different take on this. Suppose you are working on a software project with *n* lines of code and you're interested in adding a new feature. As mentioned above, any changes you make might interact with any of the existing *n* lines of code, and those *n* lines of code might interact with all of your new lines of code. If the changes you make somehow violate an assumption that exists in some other module, then changes in your relatively isolated region of the code base might cause catastrophic failures in entirely different parts of the code base.

However, the situation is far worse than this. In this example we considered an interaction to be an interaction between two lines of code. A more realistic model of interactions would consider interactions between *arbitrarily many* lines of code, since changes made in several different points might converge together in a point to form a result not possible if a single one of the changes didn't occur. In this case, if the code base has *n* lines of code, the maximum number of interactions (sets of two or more lines of code) is roughly 2^n. That's a staggeringly huge number. In fact, if we make the liberal assumption that there are 10^{100} atoms in the universe (most estimates put the figure at much less than this), then even a comparably small software system (say, three thousand lines of code) has more possible interactions than there are atoms in the universe.

In short, the larger a software system gets, the greater the likelihood than an error occurs and, consequently, the more difficult it is to make changes. In short, software is *chaotic*, and even minuscule changes to a code base can completely cripple the system.

Abstraction

One of the most powerful techniques available to combat complexity is *abstraction*, a means of simplifying a complex program to a manageable level. Rather than jumping headfirst into a full-on definition of abstraction with examples, let's look at abstraction by means of an example. Consider a standard, run-of-the-mill stapler. Certainly you understand how to use a stapler: you place the papers to staple under the arm of the stapler, then depress the handle to staple the pages together. You've undoubtedly encountered more than one stapler in your life, yet (barring unfortunate circumstances) you've probably figured out how to work all of them without much trouble. Staplers come in all shapes and sizes, and consequently have many different internal mechanisms, yet switching from one type of stapler to another poses little to no problem to you. In fact, you probably don't think much about staplers, even when you're using them.

Now consider the companies that make staplers – Swingline or McGill, for example. These companies expend millions of dollars designing progressively better staplers. They consider all sorts of tradeoffs between different types of springs and different construction materials. In fact, they probably expend more effort in a single day designing staplers than you will ever spend thinking about staplers in your entire life. But nonetheless, at the end of the day, staplers are simple and easy to use and bear no markings to indicate the painstaking labor that has gone into perfecting them. This setup, where a dedicated manufacturer designs a complex but easy-to-use product, is the heart of abstraction.

Formally speaking, an *abstraction* is a description of an object that omits all but a few salient details. For example, suppose we want to describe a particular coffee mug. Here are several different descriptions of the coffee mug, each at different levels of abstraction:

- Matter.
- An object.
- A beverage container.
- A coffee mug.
- A white coffee mug.
- A white ceramic coffee mug.
- A white ceramic coffee mug with a small crack in the handle.
- A white ceramic coffee mug with a small crack in the handle whose manufacturer's logo is emblazoned on the bottom.

Notice how these descriptions move from least specific (matter) to most specific (A white ceramic coffee mug with a small crack in the handle whose manufacturer's logo is emblazoned on the bottom). Each of the descriptions describe the same coffee mug, but each does so at a different level of detail. Depending on the circumstance, different levels of detail might be appropriate. For example, if you were a physicist interested in modeling universal gravitation, the fact that the coffee mug is made of matter might be sufficient for your purposes. However, if you wanted to paint a picture of the mug, you would probably want to pick the last description, since it offers the most detail. If you'll notice, as the descriptions become more and more detailed, more and more information is revealed about the object. Starting from the first of these descriptions and moving downward, the picture of the coffee mug becomes more clear. Describing the coffee mug as “matter” hardly helps you picture the mug, but as you go down the list you begin to notice that the mug is white, has a small crack, and has a logo printed on the bottom.

What does this have to do with our previous discussion on staplers? The answer is simple: the reason that staplers are so easy to use despite the complex mechanisms that make them work is because the very notion of a stapler is an abstraction. There are many ways to build a stapler, some of which have handles to staple documents, and others which use proximity sensors to detect paper and insert the staples automatically. Although these devices have little mechanism or structure in common, we would consider both of them staplers because they staple paper. In other words, what is important to us as stapler users is the fact that staplers fasten paper together, not their inner workings. This may seem like a strange line of reasoning, but it's one of the single most important concepts to grasp as a computer scientist. The rest of this chapter is dedicated to exploring what abstraction means from a programming perspective and how abstraction can combat complexity. But first, let's discuss some of the major concepts and terms pertaining to abstraction at a high level.

The Wall of Abstraction

In our previous example with staplers, there was a clear separation of complexity between the stapler manufacturer and the end user. The manufacturer took painstaking care to ensure that the stapler works correctly, and end users just press a handle or feed paper near a sensor. This separation is fundamental to combating complexity, and is given an appropriately impressive name: *the wall of abstraction*.

The wall of abstraction is the information barrier between a *device* and *how it works*. On one side of the wall is the *manufacturer*, whose task is to provide a device capable of meeting certain requirements. To the manufacturer, the single most important task is producing a device that works correctly and reliably. On the other side of the wall of abstraction is the *end user*, who is interested in using the device but who, unless curious, does not particularly care how the device works. In computer science, we refer to these two roles as the *client*, who uses the device, and the *implementer*, who is tasked with making it work correctly.

When using a stapler, you don't care how the stapler works because it's an unnecessary mental burden. You shouldn't need to know what type of metal the casing is made from, nor should you have to worry about what type of spring pushes the staples up to the front of the stapler. The same is true of almost every device and appliance in use today. Do you know exactly how a microwave works? How about an internal combustion engine? What about an iPhone? Each of these devices is extraordinarily complicated and works on nuanced principles of physics, materials science, chemical engineering, and in some cases electrical and software engineering. The magic of all of these devices is that *we don't need to know how they work*. We can trust that a team of dedicated engineers understand their inner workings, and can focus instead on using them.

The fact that the wall of abstraction separates the implementer in the client necessarily means that an abstraction shields clients from unnecessary implementation details. In that sense, the wall of abstraction is a barrier that prevents information about the device's internals from leaking outside. That the wall of abstraction is an information barrier has profound consequences for software design. Before we discuss how abstraction can reduce system complexity, let us focus on this aspect in more detail.

Abstractions are Imprecise

Earlier in this chapter we discussed abstraction in the context of a coffee mug by exploring descriptions of a cof fee mug at various levels of abstraction. Suppose that we have an actual coffee mug we are interested in designing an abstraction for; for example, this mug here:

The highest-level description of a coffee mug in the original list was the extraordinarily vague “matter.” That is, all of the properties of the coffee mug are ignored except for the fact that it is matter. This means the implementer (us) holds a coffee mug, but the client (the person reading the description “matter”) knows only that our object is composed of matter. Because the wall of abstraction prevents information from leaking to the client, that our object is made of matter is the only information the client has about the coffee mug. This means that the client can't tell if our object is a coffee mug, Jupiter's moon Ganymede, or a fried egg. In other words, our description was so vague that the client knows nothing about what object we have.

Let's now move to a lower level of abstraction, the description that the mug is “a beverage container.” The client can now tell that our mug is not Ganymede, nor is it a fried egg, but we haven't yet excluded other possibilities. Many things are beverage containers – punch bowls, wine glasses, thermoses, etc. – and the client cannot figure out from our limited description that the object is a coffee mug. However, without knowing that the object is a coffee mug, at this level of abstraction the client could safely assume that our object could store water.

Now let's consider an even more precise description: we describe the coffee mug as “a coffee mug.” Now what can the client infer about our mug? Because we have said that the object is a mug, they know that it's a coffee mug, true, but there are many properties of the object that they don't know. For example, how big is the mug? What color is it? What design, if any, adorns the mug? A client on the other side of the wall of abstraction still can't paint a particularly vivid picture of the mug we're describing, but they now know a good deal more about the mug than they did with either of the two previous descriptions.

This above discussion hits on a major point: abstractions are imprecise. When describing the coffee mug at various levels of detail, we always truthfully represented our coffee mug, but our descriptions never were sufficient to unambiguously convince the client of what object we were describing. In fact, if we had instead been describing a different coffee mug, like this one here:

then all of our descriptions would still have been perfectly honest.

Abstractions, by their very nature, make it impossible for the client to know exactly what object is being described. This is an incredible blessing. Suppose, for example, that you bring your car in for routine maintenance. The mechanic informs you that your radiator is nearing the end of its lifespan, so you pay for a replacement. Once the mechanic replaces the radiator and you drive off into the sunset, the car that you are driving is not the same car that you drove in with. One of its fundamental components has been replaced, and so an integral part of the car's system is not the same as it used to be. However, you feel like you are driving the same car when you leave because from your perspective, nothing has changed. The accelerator and brakes still work as they used to, the car handles like it used to, and in fact almost every maneuver you perform with the car will execute exactly the same way that the car used to. Viewing this idea through the lens of abstraction, the reason for this is that your conception of the car has to do with its observable behavior. The car you are now driving has a different “implementation” than the original car, but it adheres to the same abstraction as the old car and is consequently indistinguishable from the original car. Without looking under the hood (breaking the wall of abstraction), you wouldn't be able to notice the difference.

Interfaces

The wall of abstraction sits at the boundary between two worlds – the world of the implementer, where the workings of the mechanism are of paramount importance, and the world of the client, where the observable behavior is all that matters. As mentioned earlier, the wall of abstraction is an information barrier that prevents implementation details and usage information from crossing between the client and implementer. But if the client is unaware of the implementation of the particular device, how can she possibly use it? That is, if there is a logical barrier between the two parties, how can the client actually use the implementer's device?

One typical way to provide the client access to the implementation is via an *interface*. An interface is a set of commands and queries that can be executed on the device, and is the way that the client interacts with the object. For example, an interface might let the client learn some properties of the object in question, or might allow the client to ask the device to perform some task. In software engineering, an interface typically consists of a set of attributes (also called properties) that the object is required to have, along with a set of actions that the object can perform. For example, here's one possible interface for a stapler:

- Attributes:
  - Number of staples left.
  - Size of staples being used.
  - How many sheets of paper are in the stapler.
  - The maximum number of sheets of paper that the stapler can staple.
- Actions:
  - Add more staples.
  - Put paper into the stapler.
  - Staple the papers together.

Interfaces are fascinating because they provide a particularly elegant means for a implementer to expose an object to a client. The implementer is free to build the device in question as she sees fit, provided that all of the operations specified in the interface work correctly. That is, someone implementing a stapler that adheres to the above interface can use whatever sort of mechanism they feel like to build the stapler, so long as it is possible to look up how many staples are left, to add more staples to the stapler, etc. Similarly, the client needs only learn the operations in the interface and should (theoretically) be able to use any device that conforms to that interface. In software engineering terminology, we say that the implementer *implements* the interface by providing a means of transforming any request given to the interface to a request to the underlying device. For example, if the Swingline corporation decided to create a new stapler, they might build a concrete stapler and then implement the interface for staplers as follows:

- Attributes:
  - Number of staples left: **Open the cover and count the number of staples.**
  - Size of staples being used: **Open the cover and look at the size of the staples.**
  - How many sheets of paper are in the stapler: **Count the sheets of paper on the base plate.**
  - The maximum number of sheets of paper that the stapler can staple: **25**
- Actions:
  - Add more staples: **Open the cover and insert more staples.**
  - Put paper into the stapler: **Place the paper over the base plate**.
  - Staple the papers together: **Depress the handle until it clicks, then release the handle.**

However, we could also implement the stapler inteface in a different way if we were using an electronic stapler:

- Attributes:
  - Number of staples left: **Read the digital display**.
  - Size of staples being used: **Read the digital display**.
  - How many sheets of paper are in the stapler: **Read the digital display**.
  - The maximum number of sheets of paper that the stapler can staple: **75**
- Actions:
  - Add more staples: **Open the cover and snap the new roll of staples in place**.
  - Put paper into the stapler: **Place the paper into the loading assembly**.
  - Staple the papers together: **Press the "staple" button**.

Notice that these two staplers have the same interface but entirely different actions associated with each item in the interface. This is a concrete example of abstraction in action - because the interface only describes some specific attributes and actions associated with the stapler, anything that can make these actions work correctly can be treated as a stapler. The actual means by which the interface is implemented may be different, but the general principle is the same.

An extremely important point to note is the relation between interfaces and abstractions. Abstraction is a general term that describes how to simplify systems by separating the role of the client and the implementer. Interfaces are the means by which abstractions are actually modeled in software systems. Whenever one speaks of abstraction in programming, it usually refers to designing an interface. In other words, an object's interface is a concrete description of the abstraction provided for that object.

Encapsulation

When working with interfaces and abstractions, we build a wall of abstraction to prevent implementation details about an object from leaking to the client. This means that the client does not necessarily need to know how the particular object is implemented, and can just rely on the fact that some implementer has implemented the interface correctly. But while an interface captures the idea that a client doesn't *have* to know the particular implementation details, it does not express the idea that a client *shouldn't* know the particular implementation details. To understand this, let's return to our discussion of staplers. If an implementer provides a particular stapler that implements the stapler interface, then anyone using that stapler can just use the interface to the stapler to get all of their required functionality. However, there's nothing stopping them from disassembling the stapler, looking at its component parts, etc. In fact, given a physical stapler, it's possible to do things with that stapler that weren't initially anticipated. You could, for example, replace the stapler handle with a pneumatic compressor to build a stapler gun, which might make the stapler more efficient in a particular application. However, you could also remove the spring inside the stapler which forces the staples to the front of the staple tray, rendering the stapler useless.

In general, allowing clients to bypass interfaces and directly modify the object described by that interface is dangerous. The entire purpose of an interface is to let implementers build arbitrarily complicated systems that can be operated simply, and if a client bypasses the interface he'll be dealing with an object whose workings could easily be far beyond his comprehension. In the case of a stapler, bypassing the interface and looking at the stapler internals isn't likely to cause any problems, but you would certainly be asking for trouble if you were to start poking around the internals of the Space Shuttle. This violation, where a client bypasses an interface, is called *breaking the wall of abstraction*.

The above examples have hinted at why breaking the wall of abstraction is a bad idea, but we haven't explicitly spelled out any reasons why in general it can be dangerous. Let us do this now. First, breaking the wall of abstraction allows clients to severely hurt themselves by tweaking a complex system. In a complex system like a car engine, certain assumptions have to hold about the relationship between the parts of the car in order for the car to work correctly. That is, fuel shouldn't be injected into the engine except in certain parts of the cycle, the transmission shouldn't try shifting gears until the clutch has been released, etc. Consequently, most cars provide an interface into the engine that consists of a gas and brake pedal, whose operation controls all of the relevant parts of the engine. If you were to ignore these controls and instead try to drive the car by manually adjusting fuel intake and the brake pressure, barring special training, you would almost certainly either cause an explosion or irreversibly destroy the engine. Remember that abstraction protects both the client and the implementer – the client doesn't need to know about the inner workings of the object, and the implementer doesn't need to worry that the client can make arbitrary changes to the object; all operations on the object must come through the interface. Breaking the wall of abstraction violates both these assumptions and can hurt both parties.

The second major reason against breaking the wall of abstraction is to ensure system flexibility. As mentioned earlier, abstractions are by nature imprecise, and multiple different implementations might

satisfy a particular interface. If the client is allowed to break the wall of abstraction and look at a particular part of the implementation, then that implementation is in essence "locked in place." For example, suppose that you provide a traditional stapler to a client. That client then decides to use the stapler in a context where the exact position and orientation of the stapler hinge is important; perhaps the client has trained a robot to use the stapler by learning to feed paper into the stapler whenever the hinge is at a particular angle. Earlier in this chapter, we discussed how interfaces make it possible to change an object's implementation without befuddling the user. That is, if the client only uses the operations listed in an interface, then any object implementing that interface should be substitutable for any other. The problem with breaking the wall of abstraction is that this is no longer possible. Consider, for example, what happens if we try to replace the mechanical stapler from this setup with an electric stapler. Electric staplers tend not to have hinges, and so if we swapped staplers the robot designed to feed paper into the stapler would no longer be able to insert paper. In other words, because the robot assumed that some property held true of the implementation that was documented nowhere in the interface, it became impossible to ever change the implementation.

To summarize – peering behind an interface and looking at the underlying implementation is a bad idea. It allows clients to poke systems in ways that were never intended, and it locks the particular implementation in place.

If an abstraction does not allow clients to look at the implementation under any circumstance, that abstraction is said to be *encapsulated*. In other words, it is as though the actual implementation is trapped inside a giant capsule, and the only way to access the object is by issuing queries and commands specified by the interface. Encapsulation is uncommon in the real world, but some analogies exist. For example, if you visit a rare book collection, you cannot just go in and take any book off the shelf. Instead, you have to talk to a librarian who will then get the book for you. You have no idea where the book comes from – perhaps it's sitting on a shelf in the back, or perhaps the librarian has to get a courier to fetch it from some special vault – but this doesn't concern you because at the end of the day you (hopefully) have the book anyway.

Encapsulated interfaces are extraordinarily important in software because they represent a means for entirely containing complexity. The immense amount of implementation detail that might be necessary to implement an interface is abstracted away into a small set of commands that can be executed on that interface, and encapsulation prevents other parts of the program from inadvertently (or deliberately) modifying the implementation in unexpected ways. Later in this chapter, when we discuss classes, you will see how C++ allows you to build encapsulated interfaces.

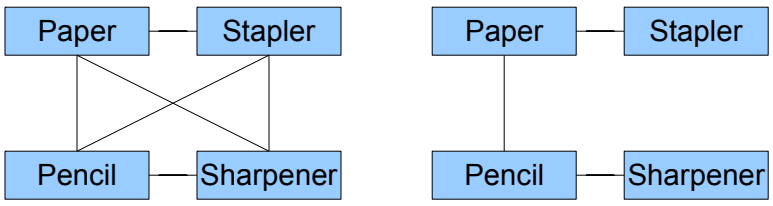
The Math: Why Abstraction Works

We've talked about abstraction and how it lets clients operate with complex objects without knowing their full implementation. The implicit claim throughout this chapter has been that this greatly reduces the complexity of software systems. Amazingly, given a suitable definition of system complexity, we can *prove* that increasing the level of abstraction in a system reduces the maximum complexity possible in the system.

In this discussion, we'll need to settle on a definition of a system's complexity. If a system consists of different interfaces, we will define the maximum complexity of that system to be the maximum number of interactions between these interfaces. For our purposes, we'll consider an interaction between interfaces to be a set of two or more interfaces. It can be shown that there are $2^n - n - 1$ possible interactions between interfaces, which is an absolutely huge number. In fact, in a system with ten interfaces, there are 1,013 possible interactions between those interfaces. The reason for this is the first term in this quantity (2^n), which grows extremely fast. It grows so quickly that we can ignore the last two terms in the sum and approximate the maximum complexity of a system as 2^n.

Now, suppose that we introduce a new abstraction into a system that reduces the total number of interfaces in the system by 10. This means that the new system has n - 10 interfaces, and consequently its maximum complexity is $2^{n - 10}$. If we take the ratio of the maximum complexity of the new system to the maximum complexity of the old system, we get 2^{-10}, which is just under one one-thousandth. That is, the maximum complexity of the new system will be roughly one one-thousandth that of the original system. This result has extremely important implications for software design. It is possible to build reasonably simple software systems that are hundreds of millions of lines of code simply by minimizing the number of interfaces present in the software system. This caps the maximum complexity of the system by limiting the number of possible interactions.

But the above logic is terribly misleading. In practice, software systems rarely get even close to reaching the maximum number of possible interactions. Maximum complexity only occurs if every combination of objects has a well-defined interaction, and this is rarely the case. For example, in a simulation with a stapler, pen, and pencil sharpener, you are unlikely to ever have the stapler and pencil sharpener interact, and if you do it is extremely unlikely that you will have all three objects have some specific behavior when interacting all at the same time. A more realistic measure of complexity is the number of ways in which *pairs* of objects can interact. This is a desirable choice for several reasons. First, it corresponds to an elegant graphical measure of complexity. If we list all of the components in a system and add lines between pairs of objects that interact with each other, the complexity of that system is then the number of lines in the picture. For example, here are two diagrams of ways that common office supplies might interact with one another. The first system is clearly more complex than the second since there are more interactions defined between the components.



Setup one: Everything interacts with everything else.

Setup two: Only meaningful interactions defined.

Second, in practice, interactions between two or more objects can usually be simplified down into multiple instances of interactions between pairs of objects. For example, if three billiard balls all collide, we could consider the interaction between the three balls as three separate interactions of the pairs of balls. Only in unusual circumstances is such a decomposition not possible. Finally, considering interactions only of pairs rather than of triples or quadruples tends to correspond more accurately to how systems are actually built. It is conceptually simpler to think about how a single piece of a system interacts with each of its neighbors in isolation than it is to think about how that pieces interacts with all of its neighbors simultaneously.

Even with this more restrictive definition of complexity, reducing the number of interfaces in a system still produces larger reductions in complexity. It can be shown that the number of possible pairs of interacting objects is slightly less than n^2. This means that if we make a linear reduction in the number of objects in the system, we get a quadratic decrease in the maximum complexity in that system. That is, removing ten interfaces isn't going to drop the maximum complexity by ten interactions – it will be a considerably bigger number.

Classes

The single most important difference between C++ and its predecessor C is the notion of a *class*. Classes are C++'s mechanism for encoding and representing abstraction, pairing interfaces with implementations, and enforcing encapsulation. The entire remainder of this book will be dedicated to exploring how to create, modify, maintain, use, and refine classes and class definitions.

In the previous discussion on abstraction, we discussed abstractly the notions of interfaces and encapsulation. Before we discuss the class mechanism, let's consider an extended example that illustrates exactly why abstraction and interfaces are so important. In particular, we will explore how one might represent an FM radio tuner in C++ code. We won't actually create a working FM radio in software – that would require specialized hardware – but the example should demonstrate many of the reasons why classes are so important.

Designing an FM Radio

In our example, we will create a data structure that stores information about an FM radio. Since the properties of an FM radio can't be represented with a single variable, we'll create a `struct` called `FMRadio` which will hold all of our data. What should this `struct` contain? At a bare minimum, we will need to know what frequency the radio is tuned in to. We also probably want to specify a volume control, so that listeners can turn up high-energy music or turn down shouting news pundits. We can represent this information as follows:

```
struct FMRadio {
    double frequency;
    int    volume;
};
```

Here, the frequency field stores the frequency in MHz. For example, if you were listening to 88.5 KQED San Francisco, this field would have the value 88.5. Similarly, listening to 107.9 The End Sacramento would have the field hold 107.9. I've arbitrarily chosen to store the volume as an int that holds a value between 0 and 10, inclusive. Volume zero completely mutes the radio, while volume ten is as much power as the speaker can deliver. This means that if I wanted to configure my radio to listen to “This American Life” at a reasonably quiet level, I could write

```
FMRadio myRadio;
myRadio.frequency = 88.5; // 88.5 MHz (KQED)
myRadio.volume    = 3;    // Reasonably quiet
```

Now, let's consider one more extension to the radio. Most radios these days let the user configure up to six different “presets,” saved stations that listeners can adjust the radio to quickly. Most car radios have this feature, although older FM radios do not. The presets are numbered one through six, and at any time a particular preset might be empty (the listener hasn't programmed this preset yet) or set to a particular frequency. As an example, I frequently commute between Palo Alto and Sacramento, and enjoy listening to NPR on the drive. Both Sacramento and San Francisco have stations that broadcast NPR content, and about halfway between Palo Alto and Sacramento one of the stations fades out dramatically while the other one comes in much more strongly. To make it easier to switch between the stations, I programmed my car radio's presets so that preset one is the San Francisco station (88.5) and preset two is the Sacramento station (89.3).

We'd like to add this functionality to FMRadio, but what's the best way to do so? If we want to store a list of six different settings, we could do so with a `vector`, but run into a problem because a `vector` always enforces the restriction that there must be an element at every position. Because some of the presets

might not be configured, we might run into trouble if we stored the elements in a `vector` because it would be difficult to determine whether a particular position in the `vector` was empty or filled with a valid station. Instead, we'll implement the `FMRadio` using a `map` that maps from the preset number to the station at the preset. If a particular value between 1 and 6 is not a key in the `map`, then the preset has not been configured; if it is a key, then its value is the preset. This leads to the following version of `FMRadio`:

```
struct FMRadio {
    double frequency;
    int    volume;
    map<int, double> presets;
};
```

If I then wanted to program my radio as described above, I would do so as follows:

```
FMRadio myRadio;
myRadio.presets[1] = 88.5;
myRadio.presets[2] = 89.3;
```

Abusing the FM Radio

The definition of `FMRadio` from above seems reasonably straightforward. It has three fields that correspond to some attribute of the radio. Unfortunately, however, using this `FMRadio` in any complex software system can cause problems. The reason is that there are certain restrictions on what values the fields of the `FMRadio` can and cannot be, but there is no means of enforcing those restrictions. For example, in the United States, all FM radio frequencies are between 87.5 and 108.0 MHz. Consequently, the `frequency` field should never be set to any value out of this range, since doing so would be meaningless. Similarly, we've stated that we don't want the volume field to leave the range 0 to 10, but nothing prevents clients of `FMRadio` from doing so. Finally, the presets field has to obey two restrictions: that the keys are integers between 1 and 6, and that the values are `double`s restricted to the range of valid frequencies.

Now, suppose that someone who does not have this intimate knowledge of the FM radio class we've designed comes along and writes the following code:

```
FMRadio myRadio;
myRadio.frequency  = 110.0; // Problem: Invalid frequency
myRadio.volume     = 11;    // Problem: Volume out of range
myRadio.presets[0] = 85.0;  // Problem: Bad preset index, invalid frequency
```

All of the above operations are illegal on FM radios, but this code compiles and runs just fine. Moreover, there is no indication at runtime that this code isn't going to work correctly. Everything that the client has done is perfectly legal C++, and the compiler has no idea that something bad might happen in the future. To give a context of where things can go wrong, suppose that we have a function that adjusts the power level to some system peripheral to tune in to the proper frequency. Because all legal frequencies are between 87.5 and 108.0 MHz, the code adjusts the power level to a floating-point value such that the power is 0.0 at the lowest possible frequency (87.5 MHz) and 1.0 at the highest frequency (108.0 MHz). This code is shown below, assuming the existence of a `SetDevicePower` function that actually sets the device power:

```
void TuneReceiver(FMRadio radio) {
    /* Compute the fraction of the maximum power that this
     * frequency requires.
     */
    double powerLevel = (radio.frequency - 87.5) / (108.0 - 87.5);
    SetDevicePower(powerLevel);
}
```

I'll leave double-checking that the above computation gives the fraction of the power to the transmitter as an exercise to the reader. In the meantime, think about what will happen if we write the following code:

```
FMRadio myRadio;
myRadio.frequency  = 110.0;
myRadio.volume     = 11;
myRadio.presets[0] = 85.0;
TuneReceiver(myRadio);
```

We now have a fairly serious problem on our hands. Because the radio frequency is 110.0 MHz, a value out of the valid FM radio range, the code inside of `TuneReceiver` is going to set the power level to a nonsensical value. In particular, since (110.0 – 87.5) / (108.0 – 87.5) ≈ 1.095, the code will turn the receiver on at roughly 110% of the maximum power it's supposed to receive. If we're lucky, the code inside `TuneReceiver` will have a check that this value is out of range, and the program will report an error. If we're unlucky and the code actually drives too much power into the receiver, we might overload the device and set it on fire. In other words, because the client of the `FMRadio` struct set a single field to a nonsensical value, it's possible that our program will crash or cause a physical device malfunction. This is clearly unacceptable, and we will need to do something about this.

Modifying the FM Radio

Of course, that's not all of the problems we might encounter when working with the `FMRadio`. Suppose, for example, that we write the following function, which sets the radio's frequency to the preset at the given position if possible, and does not change the frequency otherwise. The code is as follows:

```
void LoadPreset(FMRadio& radio, int preset) {
    /* Check whether this preset exists. */
    map<int, double>::iterator itr = radio.presets.find(preset);

    /* If not, don't do anything. */
    if (itr == radio.presets.end())
        return;

    /* Otherwise, change the radio frequency. */
    radio.frequency = itr->second;
}
```

Now, suppose that for some reason (efficiency, perhaps) that we decide to change the `FMRadio struct` so that the presets are implemented as an array of doubles. We arbitrarily say that any preset that has not been programmed will be represented by having the value 0 stored in a particular slot. That is, given my NPR travel presets, the preset array would look like this:

Value	88.5	89.3	0	0	0	0
Index	0	1	2	3	4	5

This requires us to change the definition of the `FMRadio` interface to use a raw array instead of an STL `map`. The updated definition is shown here:

```
struct FMRadio {
    double frequency;
    int    volume;
    double presets[6];
};
```

We now have a serious problem. Almost of the code that we've written previously that uses the `FMRadio`'s preset field will fail to compile. For example, our earlier code for `LoadPreset` will call `presets.find`, which does not exist in a raw array. This means that this single change might require us to rewrite huge amounts of code. In a small project, this is a mere annoyance, but in a large system on the order of millions of lines of code might be so time-consuming as to render the change impossible.

What Went Wrong?

The above discussion highlighted two problems with the `FMRadio struct`. First, `FMRadio` provides no means for enforcing its invariants. Because the aspects of the FM radio were represented in `FMRadio` by raw variables, any part of the program can modify those variables without the `FMRadio` getting a chance to intervene. In other words, the `FMRadio` expects that certain relations hold between its fields, but has no mechanism for enforcing those relations. Second, because the `FMRadio` is represented in software as a particular implementation of an FM radio, code that uses the FM radio necessarily locks the FM radio into that particular implementation. Using the terminology from the earlier in this chapter, this implementation of the FM radio provides no *abstraction* and no *encapsulation*. The `FMRadio` interface *is* its implementation – that is, the operations that clients can perform on the `FMRadio` are manipulations of the fields that compose the `FMRadio`. Changing the implementation thus changes the interface, which is why changing the fields breaks existing code. Similarly, because the interface of `FMRadio` is the set of all possible manipulations of the data members, clients can tweak the `FMRadio` in any way they see fit, even if such manipulations break internal invariants.

This is the reality of what C++ programming is like without classes. Code bases are more brittle, bugs are more likely, and changes are more difficult. As we now change gears and see how to represent an FM radio using classes, keep this starting point in mind. By the time you finish this chapter, the FM radio will be significantly more robust than it is now.

Introduction to Classes

In C++, a class is an *interface* paired with an *implementation*. Like `struct`s, classes define new types that can be created and used elsewhere in the program.

Because classes pair an implementation and an interface, the structure of an individual class can be partitioned into two halves – the *public interface* specifying how clients interact with the class, and the *private implementation*, which specifies how functions in the public interface are implemented. Rather than diving head-first into a full-blown class definition, we'll investigate each of these parts individually. We will focus first on how to declare the class, and worry about the implementation later.

Defining a Public Interface

Let's return to the example of the FM radio. We are interested in designing an abstraction that represents an FM radio, then expressing the radio in software. In particular, we want our radio to have three pieces of data: the current frequency (in MHz), the volume (from 0 to 10), and the presets. As we saw in the failed experiment with `struct FMRadio`, we cannot simply give clients direct access to the fields that ultimately

implement these properties. How, then, can we design an FM radio that contains some data but which does not allow clients to directly modify the data? The answer is a subtle yet beautiful trick that is ubiquitous in modern software. We will create a set of functions that set and query the value of these data members. We then prevent the client from directly accessing the data members that these functions manipulate. The major advantage of this approach is that every operation that could potentially read or modify the data must go through this small set of functions. Consequently, the implementer retains full control over what operations manipulate the class's implementation.

Now, let's see how one might express this in C++. We will rewrite our `FMRadio struct` from earlier to convert it into a fully-fledged C++ class. To begin, we use the C++ `class` keyword to indicate that we're defining a new class called `FMRadio`. This is shown here:

```
class FMRadio {
};
```

Currently, this class is empty and is useless. We'll thus start defining the public interface by which clients of `FMRadio` will interact with the class. In C++, to define a class's public interface, we use the `public` keyword to indicate the start of the interface, and then list the functions contained in that public interface. This leads us to the following code:

```
class FMRadio {
public:

};
```

That is, the public keyword, followed by a colon. Any definitions that follow the public keyword will be included as part of the class's public interface. But what functions should we put in here? Let's begin by letting the client query and set the radio's frequency. To do this, we'll define two member functions called `getFrequency` and `setFrequency`. This is shown here:

```
class FMRadio {
public:
    double getFrequency();
    void   setFrequency(double newFreq);
};
```

These functions are called member functions of the `FMRadio` class. Although they look like regular function prototypes, because these functions are defined inside of `FMRadio`, they are local to that class. In fact, calling the function `getFrequency` by itself will result in a compile-time error because there is no global function called `getFrequency`. Instead, we've defined a function that can be invoked on an object of type `FMRadio`. To see how this works, let's create a new object of type `FMRadio`. This is syntactically identical to the code for creating instances of a `struct` type – we put the name of the type, followed by the variable name. This is shown here:

```
FMRadio myRadio; // Declare a new variable of type FMRadio
```

Now that we have this `myRadio` object, we can ask it for its frequency by invoking the `getFrequency` member function. This is shown here:

```
FMRadio myRadio;
double f = myRadio.getFrequency(); // Query the radio for its frequency
```

Note that this code will not run as written, because we have not yet implemented the `getFrequency()` function; we'll see how to do that later in this chapter. However, this syntax should seem familiar, as it's

the same syntax we used to invoke functions on STL containers, stream objects, and strings. In fact, all of those objects are instances of classes. You're on the road to learning how these complex objects are put together!

Let's continue designing our interface. We also want a means for the client to set and read the radio volume. Along the same lines as before, we can add a pair of member functions to `FMRadio` to grant access to this data. This is shown here:

```
class FMRadio {
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);
};
```

Clients can then read and write the volume by writing code like this:

```
FMRadio myRadio;

myRadio.setVolume(8);
cout << myRadio.getVolume() << endl;
```

Again, this code will not run because we haven't implemented either of these functions. Don't worry, we're almost at the point where we'll be able to do this.

Let us now consider the final piece of the `FMRadio` interface – the code for manipulating presets. With the previous two properties (volume and frequency) we were working with a single entity, but we now must design an interface to let clients read and write multiple different values. Moreover, some of these values might not exist, since the presets might not yet be programmed in. To design a good interface, we should consider what clients would like to do with presets. We should certainly allow clients to set each of the presets. Additionally, clients should be able to check whether a certain preset has been programmed in. Finally, clients should be able to read back the presets they've programmed in, assuming they exist. We can represent each of these operations with a member function, leading to this interface for the `FMRadio` class:

```
class FMRadio {
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);
};
```

We now have an interface for our `FMRadio` class. Now, let's see how we specify the implementation of this interface.

Writing a Class Implementation

A C++ class represents an abstraction, which consists of an interface into some object. We've just seen how to define the interface for the class, and now we must provide an implementation of that interface.

This implementation consists of two parts. First, we must define what variables we will use to implement the class. This is akin to choosing the fields we put in a `struct` for the information to be useful. Second, we must provide an implementation of each of the member functions we defined in the class's public interface. We will do each of these in a separate step.

If you'll recall, one old version of `FMRadio` was a struct that looked like this:

```
struct FMRadio {
    double frequency;
    int    volume;
    map<int, double> presets;
};
```

This is a perfectly fine implementation of an `FMRadio` since it allows us to store all of the information we could possibly need. We'll therefore modify our implementation of the `FMRadio` class so that it is implemented using these three fields. However, we want to do this in a way that prevents clients of `FMRadio` from accessing the fields directly. For this purpose, C++ provides the `private` keyword, which indicates that certain parts of a class are completely off-limits to clients. This is shown here:

```
class FMRadio {
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    map<int, double> presets;
};
```

When referring to elements of a `struct`, one typically uses the term *field*. In the context of classes, these variables are called *data members*. That is, `frequency` is a *data member* of `FMRadio`, and `getVolume` is a *member function*.

Because we've marked these data members private, the C++ compiler will enforce that no client of the `FMRadio` class can access them. For example, consider the following code:

```
FMRadio myRadio;
myRadio.frequency = 110.0; // Problem: Illegal; frequency is private
```

This code will cause a compile-time error because the frequency data member is private. To write code to this effect, clients would have to use the public interface, in particular the `setFrequency` member function, as shown here:

```
FMRadio myRadio;
myRadio.setFrequency(110.0); // Legal: setFrequency is public
```

All that's left to do now is implement the member functions on the `FMRadio` class. Implementing a member function is syntactically similar to implementing a regular function, though there are a few differences. One obvious syntactic difference is the means by which we specify the name of the function. If we are interested in implementing the `getFrequency` function of `FMRadio`, for example, then we would begin as follows:

```
double FMRadio::getFrequency() {
    /* ... implementation goes here ... */
}
```

Notice that the name of the function is `FMRadio::getFrequency`. The double-colon operator (`::`) is called the *scope resolution operator* and tells C++ where to look for the function we want to implement. You can think of the syntax `X::Y` as meaning "look inside `X` for `Y`." When implementing member functions, it is extremely important that you make sure to use the full name of the function you want to implement. If instead we had written the following:

```
double getFrequency() { // Problem: Legal but incorrect
    /* ... implementation goes here ... */
}
```

Then C++ would think that we were implementing a regular function called `getFrequency` that has no relationship whatsoever to the `getFrequency` function inside of `FMRadio`.

Now that we've seen how to tell C++ that we're implementing the function, what code should we write inside of the function? We know that the function should return the FM radio's current frequency. Moreover, the frequency is stored inside of a data member called `frequency`. Consequently, we can write the following code for `FMRadio::getFrequency`:

```
double FMRadio::getFrequency() {
    return frequency;
}
```

This may look a bit confusing, so let's take a second to think about what's going on here. This function is a single line, `return frequency`. If you'll notice, nowhere in the `getFrequency()` function did we define a variable called `frequency`, but this function still compiles and runs correctly. The reason is as follows – inside of a member function, all of the class's data members can be accessed by name. That is, when implementing the `getFrequency` function, we can freely access and manipulate any or all of the class's data members by referring to them by name. We don't need to indicate that `frequency` is a data member, nor do we have to specify *which* `FMRadio`'s `frequency` data member we're referring to. By default, C++ assumes that all data members are the data members of the receiver object, and so the line return frequency means "return the value of the `frequency` data member of the object on which this function was invoked."

At this point, let us more formally define what the public and private access specifiers actually mean. If a member of a class is marked public, then *any* part of the code can access and manipulate it. Thus if you have a public member function in the class interface, all code can access it. If a class member is marked private, then the only pieces of the code that can access that member are the member functions of the class. That is, private data can only be read and written by the implementations of the class's member functions. In this sense, the public and private keywords are C++'s mechanism for defining interfaces and enforcing encapsulation. A class's interface is defined by all of its public members, and its implementation by the implementations of those public member functions along with any private data members. The compiler enforces encapsulation by disallowing class clients from directly accessing private data, and so

the implementation can assume that any access to the class's private data goes through the public interface.

Let's conclude this section by implementing the remaining pieces of the `FMRadio` class. First, let's implement the `setFrequency` function, which sets the radio's frequency to a particular value. If you'll recall, all FM radio frequencies must be between 87.5 MHz and 108.0 MHz. Thus, we'll have this function verify that the new frequency is in this range, and will then set the frequency to be in that range if so. Here's one possible implementation:

```
void FMRadio::setFrequency(double newFreq) {
    assert(newFreq >= 87.5 && newFreq <= 108.0);
    frequency = newFreq;
}
```

Here, the `assert` function, defined in `<cassert>`, is a function that tests whether the particular condition is true and aborts the program with a useful error message if it isn't. `assert` is useful in testing because it allows you to verify that certain invariants hold in your programs in a means conducive to debugging. Plus, most compilers completely remove `assert` statements from release builds, so there's no runtime overhead when you decide to ship your software.

This is a remarkably simple two lines of code. We first assert that the frequency is in range, and then set the frequency data member of the class to the new value. What's so fantastic about this code is that it allows us to enforce the restriction that the frequency be constrained to the range spanned by 87.5 MHz to 108.0 MHz. Because the only way that clients can change the frequency data member is through the setFrequency function, we can prevent the frequency from ever being set to a value out of range. We'll discuss this in more detail when we talk about class invariants.

Using the implementation of the `get`/`setFrequency` functions as a basis, we can easily implement the `get`/`setVolume` functions. This is shown here:

```
int FMRadio::getVolume() {
    return volume;
}

void FMRadio::setVolume(int newVol) {
    assert(newVol >= 0 && newVol <= 10);
    volume = newVol;
}
```

This pattern of pairing a `get*` function along with a `set*` function is extremely common, and you will undoubtedly see it in any major C++ project you work on. We'll detail exactly why it is such a useful design later in this chapter.

The final three functions we wish to implement are the `setPreset`, `presetExists`, and `getPreset` functions. These functions are in some ways similar to the `get`/`setVolume` functions, but differ in that the values they read and write might not exist. We'll begin with `setPreset`, which is shown here:

```
void FMRadio::setPreset(int index, double freq) {
    assert(index >= 1 && index <= 6);
    assert(freq >= 87.5 && freq <= 108.0);
    presets[index] = freq;
}
```

The `presetExists` function can be implemented quite simply by returning whether the `map` contains the specified key. However, there's one detail we didn't consider – what happens if the index is out of bounds? That is, what do we do if the client asks whether preset 0 exists, or whether preset 137 exists? We could implement `presetExists` to so that it returns false in these cases (since there are no presets in those slots), but it seems more reasonable to have the function assert that the value is in bounds first. The reason is that if a client is querying whether a preset that is out of the desired range exists, it almost certainly represents a logical error. Using `assert` to check that the value is in bounds will let us debug the program more easily. This leads to the following implementation of `presetExists`:

```
bool FMRadio::presetExists(int index) {
    assert(index >= 1 && index <= 6);
    return presets.find(index) != presets.end();
}
```

Finally, we'll implement the `getPreset` function. Since there is no meaningful value to return if the preset doesn't exist, we'll have this function verify that the preset is indeed valid before returning it. This is shown here:

```
double FMRadio::getPreset(int index) {
    assert(presetExists(index));
    return presets[index];
}
```

Notice that in this function, we invoked the `presetExists` member function. As with private data members, C++ lets you call member functions of the receiver object without having to explicitly specify which object you are referring to. That is, the compiler is smart enough to tell that the call to `presetExists(index)` should be interpreted as "call the `presetExists` function on the receiver object, passing in the value `index`." This also brings up another important point: it is perfectly legal to use a class's public interface in its implementation. In fact, doing so is often a wise idea. If we had implemented `getPreset` without calling `presetExists`, we would have to duplicate a reasonable amount of code, which is in general a very bad idea.

Comparing `classes` and `structs`

Earlier in this chapter, we saw how representing the `FMRadio` as a `struct` led to all sorts of problems. The `struct` had no means of enforcing invariants, and any change to the `struct`'s fields could break a potentially unbounded amount of code. We discussed earlier at a high level how abstraction and encapsulation can prevent these problems from occurring. Does the class mechanism, which is designed to represent these ideas in software, prevent the aforementioned problems from happening? Let's take a few minutes to see whether this is the case.

Classes Enforce Invariants

An *invariant* is a property of a set of data that always holds true for that data. For example, one possible invariant might be that a certain value always be even, while another could be that the difference between two values is less than fourteen. In our example with `FMRadio`, our class had several invariants:

- The radio's frequency is always between 87.5 MHz and 108.8 MHz.
- The radio's volume is a value between 0 and 10, inclusive.
- The radio's presets are numbered between 1 and 6, and are valid frequencies.

The `struct` version of `FMRadio` failed to enforce any of these invariants because clients could go in and directly modify the fields responsible for holding the data. In the class version, however, any access to the

data members that represent these quantities must go through the appropriate `set*` and `get*` functions. This allows the implementation to double-check that all of the invariants hold before modifying the class's data members. For example, let's review the implementation of `setPreset`:

```
void FMRadio::setPreset(int index, double freq) {
    assert(index >= 1 && index <= 6);
    assert(freq >= 87.5 && freq <= 108.0);
    presets[index] = freq;
}
```

This is the only function in the interface that allows clients to modify the radio's presets. Before the function writes a value to one of the presets, it verifies that the index and frequency are in range. Consequently, if the data member is written to, it is only after the implementer has had a chance to inspect the value and confirm that it is indeed in range. In other words, by restricting access to the data members and instead providing a set of functions that modify the data members, the implementer can prevent clients from modifying the implementation in a way that violates the class invariants.

Classes Enforce Encapsulation

Recall that when we implemented the `FMRadio` as a `struct`, changing any of the fields would break existing code. The reason for this is that any manipulations of the `struct` required direct access to the fields of the `struct`. When using classes, however, all operations on the class must go through an additional layer – the interface – which is independent of the current implementation. For example, consider the following code:

```
FMRadio myRadio;

myRadio.setVolume(10);
cout << myRadio.getVolume() << endl;
```

This isn't the most exciting code we've written, but it illustrates how a client might read and set the radio's volume. Now, suppose that we are implementing the `FMRadio` class so that it interacts with a real set of speakers. Initially, you might think that the speaker volume is controlled by modifying how much power the speakers receive; at lower power, the speakers output less sound. In reality, though, most speaker volumes are controlled by modifying how much *attenuation* the sound signal receives. That is, when a speaker is at full volume, the attenuation level is zero, and the speaker plays the sound at maximum volume. When the volume is zero (the sound is muted), the attenuation level is 100% and the speakers produce no sound. In other words, the volume control is represented by determining how much attenuation to insert. Consequently, whenever we want to increase the volume, we would decrease the attenuation, and vice-versa. Given this description, we therefore might change the implementation of the `FMRadio` class so that the volume is represented internally as an attenuation amount. Here's the modified class:

```
class FMRadio {
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);
```

```
private:
    double frequency;
    int    attenuation; // 0 is no attenuation, 10 is maximum attenuation
    map<int, double> presets;
};
```

Because we've changed the internal representation of the `FMRadio`, we will need to change the implementation of the `get/setVolume` functions. Now, these functions are designed so that the user inputs an amount of volume, not an amount of attenuation, and so the functions will have to do a quick behind-the-scenes calculation to convert between the two. Here's one possible implementation of `setVolume`:

```
void FMRadio::setVolume(int newVol) {
    assert(newVol >= 0 && newVol <= 10); // Unchanged
    attenuation = 10 - newVol; // Convert from volume to attenuation level
}
```

Here, the code for `setVolume` takes in a volume level from the client, then converts it into an attenuation level by subtracting the volume from ten. This means that volume 10 corresponds to 0 attenuation, volume 7 to 3 attenuation, etc.

Now, how might we go about changing the implementation of `getVolume`? This function must return a volume between 0 and 10 with 0 meaning no volume and 10 meaning maximum volume, but we've implemented the volume level internally as the attenuation. This means that the function must do a quick calculation to convert between the two. The resulting implementation is shown here:

```
int FMRadio::getVolume() {
    return 10 - attenuation;
}
```

I'll leave it as an exercise to the reader to verify that this computation is correct. ☺

In this short discussion, we completely changed the internal implementation of the radio volume. But from a client's perspective, absolutely nothing has changed. Recall the client code we wrote earlier on for changing the radio volume:

```
FMRadio myRadio;

myRadio.setVolume(10);
cout << myRadio.getVolume() << endl;
```

This code is still perfectly legal, and moreover it produces the exact same output as before. Because this code only uses the class's public interface, the client cannot tell that calling `myRadio.setVolume(10)` actually sets an internal field in the `FMRadio` to zero, nor can she tell that calling `myRadio.getVolume()` will perform a conversion behind-the-scenes. In other words, using the public interface allows clients of `FMRadio` to write code that will compile and run correctly even if the entire implementation of the `FMRadio` has changed.

Class Constructors

One of the recurring themes of this chapter has been that classes can enforce invariants. However, using only the techniques we've covered so far, there are some invariants that classes cannot enforce automatically. To see this, let's return to the `FMRadio` class. If you'll recall, when implementing `FMRadio` using a `struct`, we saw that one possible implementation of the preset list was to use an array of six

`double`s, where an unprogrammed preset has value 0.0. Let's modify our original implementation of the `FMRadio` class so that we use this implementation strategy. The new class looks like this:

```
class FMRadio {
public:
    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

This, of course, necessitates that we change our implementation of `presetExists`, since we no longer represent the preset list as a map. The new implementation is shown here:

```
bool FMRadio::presetExists(int index) {
    assert(index >= 1 && index <= 6);
    return presets[index - 1] == 0.0; // -1 maps [1, 6] to [0, 5]
}
```

Given this implementation, what is the result of running the following code snippet?

```
FMRadio myRadio;
if (myRadio.presetExists(1))
    cout << "Preset 1: " << myRadio.getPreset(1) << endl;
else
    cout << "Preset 1 not programmed." << endl;
```

Intuitively, this program should print out that preset one is not programmed, since we just created the radio. Unfortunately, though, this program produces undefined behavior. Here is the output from several different runs of the program on my machine:

```
Preset 1: 3.204e+108
Preset 1 not programmed.
Preset 1: -1.066e-34
Preset 1: 4.334e+20
```

This certainly doesn't seem right! What's going on here?

The problem is that all of the data members of `FMRadio` are primitive types, and unless you explicitly initialize a primitive type, it will hold whatever value happens to be in memory at the time that it is created. In particular, this means that the presets array will be filled with garbage, and so the `presetExists` and `getPreset` functions will be working with garbage data. Garbage data is never a good thing, but it is even more problematic from the standpoint of class invariants. The `FMRadio` assumes that certain constraints hold for its data members, but those data members are initialized randomly. How can `FMRadio` ensure that it behaves consistently when it does not have control over its implementation? The answer is simple: it can't, and we're going to need to refine our approach to make everything work correctly.

A Step in the Right Direction: `init()`

One way that we could fix this problem is to create a new member function called `init` that initializes all of the data members. We then require all clients of the `FMRadio` class to call this `init` function before using the other member functions of the `FMRadio` class. Assuming that clients ensure to call `init` before using the `FMRadio`, this should solve all of our problems.

Let's take a minute to see how we might implement the `init` function. First, we need to modify the class's public interface, as shown here:

```
class FMRadio {
public:
    void   init();

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

We could then implement `init` as follows:

```
void FMRadio::init() {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
}
```

This is certainly a step in the right direction. We no longer have to worry about the presets array containing uninitialized values. But what of the other data members, frequency and volume? They too must be initialized to some meaningful value. We can therefore update the `init` function to set them to some reasonable value. For simplicity, let's set the frequency to 87.5 MHz (the minimum possible frequency) and set the volume to five. This is shown here:

```
void FMRadio::init() {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = 87.5;
    volume    = 5;
}
```

It may seem strange that we have to initialize frequency and volume inside of the `init` function. After all, why can't we do something like this?

```
class FMRadio {
public:
    void   init();

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency = 87.5;  // Problem: Not legal C++
    int    volume    = 5;     // Problem: Not legal C++
    double presets[6];
};
```

Unfortunately, this is not legal C++ code. There isn't a particularly good reason why this is the case, and in the next release of C++ this syntax will be supported, but for now we have to manually initialize everything in the `init` function.

Why `init()` is Insufficient

The approach we've outlined above seems to solve all of our problems. Every time that we create an `FMRadio`, we manually invoke the `init` function. This solves our problem, but puts an extra burden on the client. In particular, if a client does not call the `init` function, our object's internals will not be configured properly and any use of the object will almost certainly cause some sort of runtime error.

The problem with `init` is that it does not make logical sense. When you purchase a physical object, most of the time, that object is fully assembled and ready to go. When you buy a stapler, you don't buy the component parts and then assemble it; you buy a finished product. You don't purchase a car and then manually connect the transmission to the rest of the engine; you assume that the car manufacturer has done this for you. In other words, by the time that you begin using an object, you expect it to be assembled. From the standpoint of physical objects, this is because you are buying a logically complete object, not a collection of components. From the standpoint of abstraction, this is because it breaches the wall of abstraction if you are required to set up an object into a well-formed state before you begin using it.

None of the objects we've seen so far have required any function like `init`. The STL `vector` and `map` are initialized to sensible defaults before you begin using them, and `string`s default to holding the empty string without any explicit intervention by the user. But how do they do this? It's through the magic of a special member function called the constructor.

Class Constructors

A *constructor* is a special member function whose job is to initialize the object into a well-formed state before clients start manipulating that object. In this sense, constructors are like the `init` function we wrote earlier. However, constructors have the special property that they are called automatically whenever an object is constructed. That is, if you have a class that defines a constructor, that constructor is guaranteed to execute whenever you create an object of the class type.

Syntactically, a constructor is a member function whose name is the same as the name of the class. For example, the string constructor is a function named string::string, and in our `FMRadio` example, the

constructor is a member function named `FMRadio::FMRadio`. Here is a refined interface for `FMRadio` that includes a class constructor:

```
class FMRadio {
public:
    FMRadio();

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

Notice that the constructor has no return type, not even void. This may seem strange at first, but will make substantially more sense once you see how and where the constructor is invoked.

Syntactically, one implements a constructor just as one would any other member function. The only difference is that the constructor does not have a return type, and so the syntax for implementing a constructor looks like this:

```
FMRadio::FMRadio() {
    /* ... implementation goes here ... */
}
```

Constructors are like any other function, and so we can put whatever code we feel like in the body of the constructor. However, the constructor should ensure that all of the object's data members that need manually initialization are manually initialized. In our case, this means that we might implement the `FMRadio` constructor as follows:

```
FMRadio::FMRadio() {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = 87.5;
    volume    = 5;
}
```

Now, whenever we create an instance of the `FMRadio` type, the object will be set up correctly. That is, when we write code like this:

```
FMRadio myRadio;
if (myRadio.presetExists(1))
    cout << "Preset 1: " << myRadio.getPreset(1) << endl;
else
    cout << "Preset 1 not programmed." << endl;
```

The output will *always* be "Preset 1 not programmed." This is because in the line where we create the `myRadio` object, C++ automatically invokes the constructor, which zeros out all of the presets.

It is illegal to call a class's constructor; C++ will always do this for you. For example, the following code will not compile:

```
FMRadio myRadio;
myRadio.FMRadio(); // Problem: Cannot manually invoke constructor
```

This may seem like an unusual restriction, but is actually quite useful. Because the constructor is invoked when and only when the class is being constructed for the first time, you don't need to worry about unusual conditions where the class is being instantiated but meaningful data is already stored in the class. Additionally, this makes the role of the constructor explicitly clear – its job is to initialize the class to a meaningful state, nothing more. Second, as a consequence, constructors can never return values. The constructor is invoked automatically, not giving you a chance to store a returned value even if one were to exist.

Arguments to Constructors

In the above example, our `FMRadio` constructor takes in no parameters. However, it is possible to create constructors that take in arguments that might be necessary for initialization. For example, our `FMRadio` constructor arbitrarily sets the frequency to 87.5 MHz and the volume to 5 because we need these values to be in certain ranges. There's no particular reason why we should initialize these values this way, but in the absence of information about what the client wants to do with the object we cannot do any better. But what if the client could tell us what she wanted the frequency and volume to be? In that case, we could initialize the frequency and volume to the user's values, in essence creating a radio whose frequency and volume were already set up for the user. To do this, we can create a second `FMRadio` constructor that takes in a frequency and volume, then initializes the radio to those settings.

Syntactically, a constructor of this sort is a member function named `FMRadio` that takes in two parameters. This is shown here:

```
class FMRadio {
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

We could then implement this function as follows:

```
FMRadio::FMRadio(double freq, int vol) {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}
```

Now that we have this constructor, how do we call it? That is, how do we create an object that is initialized using this constructor? The syntax for this is reasonably straightforward and looks like this:

```
FMRadio myRadio(88.5, 5);
```

That is, we write out the type of the object to create, the name of the object to create, and then a parenthesized list of the arguments to pass into the constructor.

You may be wondering why in the case of a zero-argument constructor, we do not need to explicitly spell out that we want to use the default constructor. In other words, why don't we write out code like this:

```
FMRadio myRadio(); // Problem: Legal but incorrect
```

This code is perfectly legal, but it does not do what you'd expect. There is an unfortunate defect in C++ that causes this statement to be interpreted as a function prototype rather than the creation of an object using the default constructor. In fact, C++ will interpret this as "prototype a function called `myRadio` that takes in no arguments and returns an `FMRadio`" rather than "create an `FMRadio` called `myRadio` using the zero-argument constructor." This is sometimes referred to as "C++'s most vexing parse" and causes extremely difficult to understand warnings and error messages. Thus, if you want to invoke the default constructor, omit the parentheses. If you want to invoke a parametrized constructor, parenthesize the arguments.

Another important point to remember when working with multiple constructors is that constructors cannot invoke one another. This is an extension of the rule that you cannot directly call a constructor. If you need to do the same work in multiple constructors, you can either duplicate the code (yuck!) or use a private member function, which we'll discuss later.

Classes Without a Nullary Constructor

A function is called *nullary* if it takes no arguments. For example, the first `FMRadio` constructor we wrote is a nullary constructor, since it takes no arguments. If you define a class and do not provide a constructor, C++ will automatically provide you a default nullary constructor that does absolutely nothing. This is why in the case of `FMRadio`, we needed to provide a nullary constructor to initialize the data members; otherwise they would initialize to arbitrary values. However, if you define a class and provide any constructors, C++ will not automatically generate a nullary constructor for you. This means that it is possible to construct classes that do not have a zero-argument constructor. For example, suppose that we remove the nullary constructor from `FMRadio`; this results in the following class definition:

```
class FMRadio {
public:
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    double frequency;
    int    volume;
    double presets[6];
};
```

Because this class does not have a nullary constructor, we cannot construct instances of it without passing in values for the frequency and volume. That is, the following code is illegal:

```
FMRadio myRadio; // Problem: No default constructor available
```

At first, this may seem like a nuisance. However, this aspect of class design is extremely valuable because it allows you to create types that must be initialized to a meaningful value. For example, suppose that you are designing a class that represents a robot-controlled laser, either for automated welding or delicate surgical procedures. When building such a laser, it is imperative that the laser know how much power to deliver and what points the beam should be directed at. These values absolutely must be initialized to meaningful data, or the laser might deliver megawatts of power at a patient or aim at random points firing the beam. If you wanted to represent the laser as a C++ class, you could force clients to specify this data before using the laser by making a `RobotLaser` class whose only constructor takes in both the laser power and laser coordinates. This means that clients could not create instances of `RobotLaser` without entering coordinates, reducing the possibility of a catastrophic failure.

Private Member Functions

Let's return once again to our `FMRadio` example, this time looking at the implementation of three functions: `setFrequency`, `setPreset`, and `presetExists`. The implementations of these functions are shown here:

```
void FMRadio::setFrequency(double newFreq) {
    assert(newFreq >= 87.5 && newFreq <= 108.0);
    frequency = newFreq;
}

void FMRadio::setPreset(int index, double freq) {
    assert(index >= 1 && index <= 6);
    assert(freq >= 87.5 && freq <= 108.0);
    presets[index - 1] = freq;
}

double FMRadio::presetExists(int index) {
    assert(index >= 1 && index <= 6);
    return presets[index - 1] == 0.0;
}
```

Notice that each highlighted line of code appears in two of the three functions. Normally this isn't too serious a concern, but in this particular case makes the implementation brittle and fragile. In particular, if we ever want to change the number of presets or the maximum frequency range, we'll need to modify multiple parts of the code accordingly or risk inconsistent handling of presets and frequencies. To unify the code, we might consider decomposing this logic into helper functions. However, since the code we're decomposing out is an implementation detail of the `FMRadio` class, class clients shouldn't have access to these helper functions. In other words, we want to create a set of functions that simplify class implementation but which can't be accessed by class clients. For situations like these, we can use a technique called private member functions.

Marking Functions `private`

If you'll recall from earlier, the `private` keyword indicates which parts of a class cannot be accessed by clients. So far we have restricted ourselves to dealing only with private data members, but it is possible to create member functions that are marked private. Like regular member functions, these functions can read and write private class data, and are invoked relative to a receiver object. Unlike public member functions, though, they can only be invoked by the class implementation. Therefore, private member functions are not part of the class's interface and exist solely to simplify the class implementation.

Declaring a private member function is similar to declaring a public member function - we just add the definition to the class's private data. In our `FMRadio` example, we will introduce two helper functions: `checkFrequency`, which asserts that a frequency is in the proper range, and `checkPreset`, which ensures that a preset index is in bounds. The updated class definition for `FMRadio` is shown here:

```
class FMRadio {
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    void   checkFrequency(double freq);
    void   checkPreset(int index);

    double frequency;
    int    volume;
    double presets[6];
};
```

We can then implement these functions as follows:

```
void FMRadio::checkFrequency(double freq) {
    assert(freq >= 87.5 && freq <= 108.8);
}

void FMRadio::checkPreset(int index) {
    assert(index >= 1 && index <= 6);
}
```

Using these functions yields the following implementations of the three aforementioned functions:

```
void FMRadio::setFrequency(double newFreq) {
    checkFrequency(newFreq);
    frequency = newFreq;
}

void FMRadio::setPreset(int index, double freq) {
    checkPreset(index);
    checkFrequency(freq);
    presets[index - 1] = freq;
}

bool FMRadio::presetExists(int index) {
    checkPreset(index);
    return presets[index - 1] == 0.0;
}
```

These functions are significantly cleaner than before, and the class as a whole is much more robust to change.

Simplifying Constructors with Private Functions

Private functions can greatly reduce the implementation complexity of classes with multiple constructors. Recall that our `FMRadio` class has two constructors, one which initializes the `FMRadio` to have a reasonable default frequency and volume, and one which lets class clients specify the initial frequency and volume. The implementation of these two functions is shown here:

```
FMRadio::FMRadio() {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = 87.5;
    volume    = 5;
}

FMRadio::FMRadio(double freq, int vol) {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}
```

These functions are extremely similar in structure, but because C++ does not allow you to manually call a class's constructor. How, then, can the two functions be unified? Simple – we introduce a private member function which does the initialization, then have the two constructors invoke this member function with the proper arguments. This is illustrated below:

```
class FMRadio {
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    void   checkFrequency(double freq);
    void   checkPreset(int index);
    void   initialize(double freq, int vol);

    double frequency;
    int    volume;
    double presets[6];
};

void FMRadio::initialize(double freq, int vol) {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}

FMRadio::FMRadio() {
    initialize(87.5, 5);
}

FMRadio::FMRadio(double freq, int vol) {
    initialize(freq, vol);
}
```

As you can see, private member functions are extremely useful tools. We will continue to use them throughout the remainder of this book, just as you will undoubtedly use them in the course of your programming career.

Partitioning Classes Across Files

One of the motivations behind classes was to provide a means for separating out implementation and interface. We have seen this already through the use of the `public` and `private` access specifiers, which prevent clients from looking at implementation-specific details. However, there is another common means by which implementation is separated from interface, and that is the split between *header files* and *implementation files*. As you've seen before, almost all of the programs you've written begin with a series of `#include` directives which tell the compiler to fetch certain files and include them in your programs. Now that we've reached a critical mass and can begin writing our own classes, we will see how to design your own header files.

At a high level, header files provide a means for exporting a class interface without also exporting unnecessary implementation details. Programmers who wish to use your class in their code can

`#include` the header file containing your class declaration, and the linker will ensure that the class implementation is bundled along with the final program. More concretely, a header file contains the class declaration (including both `public` and `private` members), and the implementation file contains the actual class implementation. For example, suppose that we want to export the `FMRadio` class we've just designed in a header/implementation pair. We'll begin by constructing the header file. Traditionally, header files that contain class declarations have the same name as the class and a .h suffix. In our case, this means that we'll be creating a file called `FMRadio.h` that contains our class definition. This is shown here:

File: `FMRadio.h`

```
#ifndef FMRadio_Included
#define FMRadio_Included

class FMRadio
{
public:
    FMRadio();
    FMRadio(double freq, int vol);

    double getFrequency();
    void   setFrequency(double newFreq);

    int    getVolume();
    void   setVolume(int newVolume);

    double setPreset(int index, double freq);
    bool   presetExists(int index);
    double getPreset(int index);

private:
    void   checkFrequency(double freq);
    void   checkPreset(int index);
    void   initialize(double freq, int vol);

    double frequency;
    int    volume;
    double presets[6];
};

#endif
```

Notice that we've surrounded the header file with an include guard. In case you've forgotten, the include guard is a way to prevent compiler errors in the event that a client `#include`s the same file twice; see the chapter on the preprocessor for more information. Beyond this, though, the header contains just the class interface.

Now that we've built the .h file for our `FMRadio` class, let's see if we can provide a working implementation file. Typically, an implementation file will have the same name as the class, suffixed with the .cpp extension.* Appropriately, we'll name this file `FMRadio.cpp`. Unlike a .h file, which is designed to export the class declaration to clients, the .cpp file just contains the implementation of the class. Here's one possible version of the `FMRadio.cpp` file:

* It is also common to see the .cc extension. Older code might use the .C extension (capital C) or the .c++ extension.

File: `FMRadio.cpp`

```
#include "FMRadio.h"

FMRadio::FMRadio() {
    initialize(87.5, 5);
}

FMRadio::FMRadio(double freq, int vol) {
    initialize(freq, vol);
}

void FMRadio::initialize(double freq, int vol) {
    for(size_t i = 0; i < 6; ++i)
        presets[i] = 0.0;
    frequency = freq;
    volume    = vol;
}

void FMRadio::checkFrequency(double freq) {
    assert(freq >= 87.5 && freq <= 108.8);
}

void FMRadio::checkPreset(int index) {
    assert(index >= 1 && index <= 6);
}

double FMRadio::getFrequency() {
    return frequency;
}

void FMRadio::setFrequency(double newFreq) {
    checkFrequency(newFreq);
    frequency = newFreq;
}

void FMRadio::setPreset(int index, double freq) {
    checkPreset(index);
    checkFrequency(freq);
    presets[index - 1] = freq;
}

bool FMRadio::presetExists(int index) {
    checkPreset(index);
    return presets[index - 1] == 0.0;
}

double FMRadio::getPreset(int index) {
    checkPreset(index);
    return presets[index - 1];
}
```

There are a few important aspects of this .cpp file to note. First, notice that at the top of the file, we `#include`d the .h file containing the class declaration. This is *extremely important* – if we don't include the header file for the class, when the C++ compiler encounters the implementations of the class's member functions, it won't have seen the class declaration and will flag all of the member function implementations as errors. Second, note that when `#include`-ing the .h file, we surrounded the name of the file in "double quotes" instead of <angle brackets.> If you'll recall, this is because the preprocessor

treats include directives using <angle brackets> as instructions to look for standard library files, and since the classes you'll be writing aren't part of the standard library you'll want to use "double quotes" instead.*

Now that we've partitioned the class into a .h/.cpp pair, we can write programs that use the class without having to tediously copy-and-paste the class definition and implementation. For example, here's a short program which manipulates an `FMRadio`. Note that we never actually see the declaration or implementation of the `FMRadio` class; `#include`-ing the header provides the compiler enough information to let us use the `FMRadio`.

```
#include <iostream>
#include "FMRadio.h"
using namespace std;

int main() {
    FMRadio myRadio;
    myRadio.setFrequency(88.5);
    myRadio.setVolume(8);
    /* ... etc. ... */
}
```

Throughout the remainder of this book, whenever we design and build classes, we will assume that the classes are properly partitioned between .h and .cpp files.

Chapter Summary

- Software systems are often on the order of millions of lines of code, far larger than even the most competent programmers can ever keep track of at once.
- A single incorrect value in a software system can cause that entire system to fail.
- The maximum number of possible interactions in a software system grows exponentially in the number of components of that system.
- Abstractions give a way to present a complex object in simpler terms.
- Abstractions partition users into clients and implementers, each with separate tasks. This separation is sometimes referred to as the wall of abstraction.
- Abstractions describe many possible implementations, and encapsulation prevents clients from peeking at that implementation.
- The way in which a client interacts with an object is called that object's interface.
- Abstraction reduces the number of components in a software system, reducing the maximum complexity of that system.
- C++ `struct`s lack encapsulation because their implementation is their interface.
- The C++ class concept is a realization of an interface paired with an implementation.

* If you ever have the honor of getting to write a new standard library class, please contact me... I'd love to offer comments and suggestions!

- The members of a class that are listed public form that class's interface and are accessible to anyone.

- The members of a class that are listed private are part of the class implementation and can only be viewed by member functions of that class.

- Constructors allow implementers to enforce invariants from the moment the class is created.

- Private member functions allow implementers to decompose code without revealing the implementation to clients.

- Class implementations are traditionally partitioned into a .h file containing the class definition and a .cpp file containing the class implementation.

- Design class interfaces before implementations to avoid overspecializing the interface on an implementation artifact.

Practice Problems

1. In our discussion of abstraction, we talked about how interfaces and modularity can exponentially reduce the maximum complexity of a system. Can you think of any examples from the real world where introducing indirection makes a complex system more manageable?

2. What is the motivation behind functions along the lines of `getFrequency` and `setFrequency` over just having a public frequency data member?

3. When is a constructor invoked? Why are constructors useful?

4. What is the difference between a public member function and a private member function?

5. What goes in a class's .h file? In its .cpp file?

6. We've talked at length about the streams library and STL without mentioning much of how those libraries are implemented behind-the-scenes. Explain why abstraction makes it possible to use these libraries without full knowledge of how they work.

7. Suppose that C++ were designed somewhat differently in that data members marked private could only be read but not written to. That is, if a data member called `volume` were marked private, then clients could read the value by writing `myObject.volume`, but could not write to the volume variable directly. This would prohibit clients of a class from modifying the implementation incorrectly, since any operations that could change the object's data members would have to go through the public interface. However, this setup has a serious design flaw that would make class implementations difficult to change. What is this flaw? *(Hint: Think back to the* `volume/attenuation` *example from earlier)*

8. Below is an interface for a class that represents a grocery list:

```
class GroceryList {
public:
    GroceryList();

    void addItem(string quantity, string item);
    void removeItem(string item);

    string itemQuantity(string item);
    bool itemExists(string item);
};
```

   The `GroceryList` constructor sets the grocery list to contain no items. The `addItem` function adds a certain quantity of an item to the grocery list. For example, the call

```
gList.addItem("One Gallon", "Milk");
```

   would add the item "Milk" to the list with quantity "One Gallon." If the item already exists in the list, then `addItem` should replace the original quantity with the new quantity.

   The `removeItem` function should delete the specified item off of the shopping list. `itemExists` returns whether the specified item exists in the shopping list, and `itemQuantity` takes in an item and returns the quantity associated with it in the list. If the item doesn't exist, `itemQuantity` can either raise an assert error or return the empty string.

   Provide an implementation for the `GroceryList` class. You are free to use whatever implementation you feel is best, and can implement the member functions as you see fit. However, you might find it useful to use a `map<string, string>` to represent the items in the list.

9. What is the advantage of making a `GroceryList` class over just using a raw `map<string, string>`?

10. Does the `GroceryList` class need a constructor? Why or why not?

11. Give an example of a parameterized constructor you have encountered in the STL.

12. Why are parameterized constructors useful?

13. Keno is a popular gambling game with similarities to a lottery or bingo. Players place a bet and pick a set of numbers between 1 and 80, inclusive. The number of numbers chosen can be anywhere from one to twenty, with each having a different payoff scale. Once the players have chosen their numbers, twenty random numbers between 1 and 80 are chosen, and players receive a payoff based on how many numbers they picked that matched the chosen numbers. For example, if a player picked five numbers and all five were chosen, she might win around $1,000 for a one- or two-dollar bet. The actual payoffs are based on the probabilities of hitting k numbers out of n chosen, but this is irrelevant for our discussion.

    Suppose that you are interested in writing a program that lets the user play Keno. You are not interested in the payoffs, just letting the user enter numbers and reporting which of the user's numbers came up. To do this, you decide to write a class `KenoGame` with the following interfaces:

```
class KenoGame {
public:
    KenoGame();

    void addNumber(int value);
    size_t numChosen();

    size_t numWinners(vector<int>& values);
};
```

The `KenoGame` constructor initializes the class however you see fit. `addNumber` takes in a number from the user and adds it to the set of numbers the user guessed. The `numChosen` member function returns how many numbers the user has picked so far. Finally, the `numWinners` function takes in a `vector<int>` corresponding to the numbers that were chosen and returns how many of the user's numbers were winners.

Write an implementation of the `KenoGame` class.

14. Refer back to the implementation of *Snake* from the chapter on STL containers. Design and implement a class that represents a snake. What operations will you support in the public interface? How will you implement it?

Chapter 9: Refining Abstractions

In the previous chapter, we explored the class concept and saw how to use classes to model an interface paired with an implementation. You learned how to realize the idealized versions of abstraction and encapsulation using the `public` and `private` keywords, as well as how to use constructors to enforce class invariants. However, our tour of classes has just begun, and there are many nuances of class design we have yet to address. For example, since class clients cannot look at the class implementation, how can they tell which parts of the public interface are designed to read the class's state and which parts will write it? How can you more accurately control how constructors initialize data? And how can you share data across all instances of a class? These questions are all essentially variants on a common theme: how can we refine our abstractions to make them more precise?

This chapter explores some of C++'s language features that allow you as a programmer to more clearly communicate your intentions when designing classes. The tools you will learn in this chapter will follow you through the rest of your programming career, and appreciating exactly where each is applicable will give you a significant advantage when designing software.

Parameterizing Classes with Templates

One of the most important lessons an upstart computer scientist or software engineer can learn is *decomposition* or *factoring* – breaking problems down into smaller and smaller pieces and solving each subproblem separately. At the heart of decomposition is the concept of *generality* – code should avoid overspecializing on a single problem and should be robust enough to adapt to other situations. Take as an example the STL. Rather than specializing the STL container classes on a single type, the authors decided to parameterize the containers over the types they store. This means that the code written for the `vector` class can be used to store almost any type, and the `map` can use arbitrary types as key/value pairs. Similarly, the STL algorithms were designed to operate on all types of iterators rather than on specific container classes, making them flexible and adaptable.

The STL is an excellent example of how versatile, flexible, and powerful C++ templates can be. In C++ a *template* is just that – a code pattern that can be instantiated to produce a type or function that works on an arbitrary type. Up to this point you've primarily been a client of template code, and now it's time to gear up to write your own templates. In this section we'll cover the basics of templates and give a quick tour of how template classes operate under the hood. We will make extensive use of templates later in this text and especially in the extended examples, and hopefully by the time you've finished reading this book you'll have an appreciation for just how versatile templates can be.

Class Templates

In C++, a *class template* is a class that, like the STL `vector` or `map`, is parameterized over some number of types. In a sense, a class template is a class with a hole in it. When a client uses a template class, she fills in these holes to yield a complete type. You have already seen this with the STL containers: you cannot create a variable of type `vector` or `map`, though you *can* create a variable of type `vector<int>` or `map<string, string>`.

Class templates are most commonly used to create types that represent particular data structures. For example, the `vector` class template is an implementation of a linear sequence using a dynamically-allocated array as an implementation. The operations that maintain the dynamic array are more or less

independent of the type of elements in that array. By writing `vector` as a class template rather than a concrete class, the designers of the STL make it possible to use linear sequences of arbitrary C++ types.

Of course, not all classes should be written as class templates. For example, the `FMRadio` class from the previous chapter is an unlikely candidate for a class template because it does not hold a collection of data that could be of arbitrary type. Although `FMRadio` does hold multiple pieces of data (notably the radio's presets), those presets are always radio frequencies, which we've encoded with `double`s. It would not make sense for the `FMRadio`'s presets to be stored as `vector<int>`s, nor as `string`s. As a general rule, most classes don't need to be written as class templates.

Defining a Class Template

Once you've decided that the class you're writing is best parameterized over some arbitrary type, you can indicate to C++ that you're defining a template class by using the `template` keyword and specifying what types the template should be parameterized over. For example, suppose that we want to define our own version of the `pair` struct used by the STL. If we want to call this struct `MyPair` and have it be parameterized over two types, we can write the following:

```
template <typename FirstType, typename SecondType> struct MyPair {
    FirstType first;
    SecondType second;
};
```

Here, the syntax `template <typename FirstType, typename SecondType>` indicates to C++ that what follows is a class template that is parameterized over two types, one called `FirstType` and one called `SecondType`. In many ways, type arguments to a class template are similar to regular arguments to C++ functions. For example, the actual names of the parameters are unimportant as far as clients are concerned, much in the same way that the actual names of parameters to functions are unimportant. The above definition is functionally equivalent to this one below:

```
template <typename One, typename Two> struct MyPair {
    One first;
    Two second;
};
```

Within the body of the class template, we can use the names `One` and `Two` (or `FirstType` and `SecondType`) to refer to the types that the client specifies when she instantiates `MyPair`, much in the same way that parameters inside a function correspond to the values passed into the function by its caller.

In this above example, we used the `typename` keyword to introduce a type argument to a class template. If you work on other C++ code bases, you might see the above class template written as follows:

```
template <class FirstType, class SecondType> struct MyPair {
    FirstType first;
    SecondType second;
};
```

In this instance, `typename` and `class` are completely equivalent to one another. However, I find the use of `class` misleading because it incorrectly implies that the parameter must be a class type. This is not the case – you can still instantiate templates that are parameterized using `class` with primitive types like `int` or `double`. From here on out, we will use `typename` instead of `class`.*

* You can only substitute `class` for `typename` in this instance – it's illegal to declare a regular C++ class using the `typename` keyword.

To create an instance of `MyPair` specialized over some particular types, we specify the name of the class template, followed by the type arguments surrounded by angle brackets. For example:

```
MyPair<int, string> one; // A pair of an int and a string.
one.first = 137;
one.second = "Templates are cool!";
```

This syntax should hopefully be familiar from the STL.

Classes and `struct`s are closely related to one another, so unsurprisingly the syntax for declaring a template class is similar to that for a template `struct`. Let's suppose that we want to convert our `MyPair` `struct` into a class with full encapsulation (i.e. with accessor methods and constructors instead of exposed data members). Then we would begin by declaring `MyPair` as

```
template <typename FirstType, typename SecondType> class MyPair {
public:
    /* ... */

private:
    FirstType first;
    SecondType second;
};
```

Now, what sorts of functions should we define for our `MyPair` class? Ideally, we'd like to have some way of accessing the elements stored in the pair, so we'll define a pair of functions `getFirst` and `setFirst` along with an equivalent `getSecond` and `setSecond`. This is shown here:

```
template <typename FirstType, typename SecondType> class MyPair {
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};
```

Notice that we're using the template arguments `FirstType` and `SecondType` to stand for whatever types the client parameterizes `MyPair` over. We don't need to indicate that `FirstType` and `SecondType` are at all different from other types like `int` or `string`, since the C++ compiler already knows that from the `template` declaration. In fact, with a few minor restrictions, once you've defined a template argument, you can use it anywhere that an actual type could be used and C++ will understand what you mean.

Now that we've declared these functions, we should go about implementing them in the intuitive way. If `MyPair` were not a template class, we could write the following:

```
FirstType MyPair::getFirst() { // Problem: Not legal syntax
    return first;
}
```

The problem with this above code is that `MyPair` is a class *template*, not an actual class. If we don't tell C++ that we're trying to implement a member function for a class template, the compiler won't understand what we mean. Thus the proper way to implement this member function is

```
template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst() {
    return first;
}
```

Here, we've explicitly prefaced the implementation of `getFirst` with a template declaration and we've marked that the member function we're implementing is for `MyPair<FirstType, SecondType>`. The template declaration is necessary for C++ to figure out what `FirstType` and `SecondType` mean here, since without this information the compiler would think that `FirstType` and `SecondType` were actual types instead of placeholders for types. That we've mentioned this function is available inside `MyPair<FirstType, SecondType>` instead of just `MyPair` is also mandatory since there is no real `MyPair` class – after all, `MyPair` is a class *template*, not an actual class.

The other member functions can be implemented similarly. For example, here's an implementation of `setSecond`:

```
template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond(SecondType newValue) {
    second = newValue;
}
```

When implementing member functions for template classes, you do *not* need to repeat the template definition if you define the function inside the body of the template class. Thus the following code is perfectly legal:

```
template <typename FirstType, typename SecondType> class MyPair {
public:
    FirstType getFirst() {
        return first;
    }
    void setFirst(FirstType newValue) {
        first = newValue;
    }

    SecondType getSecond() {
        return second;
    }
    void setSecond(SecondType newValue) {
        second = newValue;
    }

private:
    FirstType first;
    SecondType second;
};
```

The reason for this is that inside of the class template, the compiler already knows that `FirstType` and `SecondType` are templates, and it's not necessary to remind it.

Now, let's suppose that we want to define a member function called `swap` which accepts as input a reference to another `MyPair` class, then swaps the elements in that `MyPair` with the elements in the receiver object. Then we can prototype the function like this:

```
template <typename FirstType, typename SecondType> class MyPair {
public:
    FirstType getFirst() {
        return first;
    }
    void setFirst(FirstType newValue) {
        first = newValue;
    }

    SecondType getSecond() {
        return second;
    }
    void setSecond(SecondType newValue) {
        second = newValue;
    }

    void swap(MyPair& other);

private:
    FirstType first;
    SecondType second;
};
```

Even though `MyPair` is a template class parameterized over two arguments, inside the body of the `MyPair` template class definition we can use the name `MyPair` without mentioning that it's a `MyPair<FirstType, SecondType>`. This is perfectly legal C++ and will come up more when we begin discussing copying behavior in a few chapters. The actual implementation of `swap` is left as an exercise.

.h and .cpp files for template classes

When writing a C++ class, you normally partition the class into two files: a .h file containing the declaration and a .cpp file containing the implementation. The C++ compiler can then compile the code contained in the .cpp file and then link it into the rest of the program when needed. When writing a template class, however, breaking up the definition like this will cause linker errors. The reason is that C++ templates are just that – they're *templates* for C++ code. Whenever you write code that instantiates a template class, C++ generates code for the particular instance of the class by replacing all references to the template parameters with the arguments to the template. For example, with the `MyPair` class defined above, if we create a `MyPair<int, string>`, the compiler will generate code internally that looks like this:

```
class MyPair<int, string> {
public:
    int getFirst();
    void setFirst(int newValue);

    string getSecond();
    void setSecond(string newValue);

private:
   int first;
   string second;
}
```

```
int MyPair<int, string>::getFirst() {
    return first;
}

void MyPair<int, string>::setFirst(int newValue) {
    first = newValue;
}

string MyPair<int, string>::getSecond() {
    return second;
}

void MyPair<int, string>::setSecond(string newValue) {
    second = newValue;
}
```

At this point, compilation continues as usual.

But what would happen if the compiler didn't have access to the implementation of the `MyPair` class? That is, let's suppose that we've created a header file, `my-pair.h`, that contains only the class declaration for `MyPair`, as shown here:

***File**: `my-pair.h`*

```
#ifndef MyPair_Included // Include guard prevents multiple inclusions
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair {
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);

private:
    FirstType first;
    SecondType second;
};

#endif
```

Suppose that we have a file that `#include`s the `my-pair.h` file and then tries to use the `MyPair` class. Since all that the compiler has seen of `MyPair` is the above class definition, the compiler will only generate the following code for `MyPair`:

```
class MyPair<int, string> {
public:
    int getFirst();
    void setFirst(int newValue);

    string getSecond();
    void setSecond(string newValue);
private:
   int first;
   string second;
}
```

Notice that while all the member functions of `MyPair<int, string>` have been *prototyped*, they haven't been *implemented* because the compiler didn't have access to the implementations of each of these member functions. In other words, if a template class is instantiated and the compiler hasn't seen implementations of its member functions, the resulting template class will have no code for its member functions. This means that the program won't link, and our template class is now useless.

When writing a template class for use in multiple files, the entire class definition, including implementations of member functions, must be visible in the header file. One way of doing this is to create a .h file for the template class that contains both the class definition and implementation without creating a matching .cpp file. This is the approach adopted by the C++ standard library; if you open up any of the headers for the STL, you'll find the complete (and cryptic) implementations of all of the functions and classes exported by those headers.

To give a concrete example of this approach, here's what the `my-pair.h` header file might look like if it contained both the class and its implementation:

***File**: `my-pair.h`*

```
/* This method of packaging the .h/.cpp pair puts the entire class definition and
 * implementation into the .h file.  There is no .cpp file for this header.
 */

#ifndef MyPair_Included
#define MyPair_Included

template <typename FirstType, typename SecondType> class MyPair {
public:
    FirstType getFirst();
    void setFirst(FirstType newValue);

    SecondType getSecond();
    void setSecond(SecondType newValue);
private:
    FirstType first;
    SecondType second;
};

template <typename FirstType, typename SecondType>
    FirstType MyPair<FirstType, SecondType>::getFirst() {
    return first;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setFirst(FirstType newValue) {
    first = newValue;
}

template <typename FirstType, typename SecondType>
    SecondType MyPair<FirstType, SecondType>::getSecond() {
    return second;
}

template <typename FirstType, typename SecondType>
    void MyPair<FirstType, SecondType>::setSecond(SecondType newValue) {
    second = newValue;
}

#endif
```

Putting the class and its definition inside the header file is a valid way to prevent linker errors, but it seems to violate the principle of separation of interface and implementation. After all, the reason we have both .h and .cpp files is to hide a class implementation in a file that clients never have to look at. Unfortunately, barring some particularly unsightly and hacky abuses of the preprocessor, you will need to structure template code in this manner.

The Two Meanings of `typename`

One of the more unfortunate quirks of the C++ language is the dual meaning of the `typename` keyword. As mentioned previously, when defining a template class, you can the `typename` keyword to declare type parameters for the template class. However, there is another use of the `typename` keyword that can easily catch you off guard unless you're on the lookout for it. Suppose, for example, that we wish to implement a class akin to the STL `stack` which represents a LIFO container. Because the abstract notion of a stack only concerns the *ordering* of the elements in the container rather than the *type* or *contents* of the elements in the container, we should probably consider implementing the stack as a template class. Here is one possible interface for such a class, which we'll call `Stack` to differentiate it from the STL `stack`:

```
template <typename T> class Stack {
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();
};
```

There are many ways that we could implement this `Stack` class: we could use dynamically-allocated arrays, or the STL `vector` or `deque` containers. Of these three choices, the `vector` and `deque` are certainly simpler than using dynamically-allocated arrays. Moreover, since all of the additions and deletions from a stack occur at the end of the container, the `deque` is probably a more suitable container with which we could implement our `Stack`. We'll therefore implement the `Stack` using a `deque`, as shown here:

```
template <typename T> class Stack {
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();

private:
    deque<T> elems;
};
```

Notice that we've used the template parameter `T` to parameterize the `deque`. This is perfectly valid, and is quite common when implementing template classes.

Given this implementation strategy, we can implement each of the member functions as follows. Make sure that you can read this code; it's fairly template-dense.

```
template <typename T> void Stack<T>::push(T value) {
    elems.push_front(value);
}

template <typename T> T Stack<T>::pop() {
    T result = elems.front();
    elems.pop_front();
    return result;
}

template <typename T> size_t Stack<T>::size() {
    return elems.size();
}

template <typename T> bool Stack<T>::empty() {
    return elems.empty();
}
```

This is a perfectly reasonable implementation of a stack, and in fact the STL `stack` implementation is very similar to this one.

Now, suppose that we're interested in extending the functionality of the `Stack` so that class clients can iterate over the elements of the `Stack` in the order that they will be removed. For example, if we push the elements 1, 2, 3, 4, 5 onto the stack, the iteration would visit the elements in the order 5, 4, 3, 2, 1. This functionality is usually not found on a `Stack`, but is useful for debugging (e.g. printing out the contents of the stack) or modifying the elements of the `Stack` after they've already been inserted. To do this, we'll need to add `begin()` and `end()` functions to the `Stack` class that return iterators over the underlying `deque`. Because the internal `deque` is a `deque<T>`, these iterators have type `deque<T>::iterator`. Consequently, you might think that we would update the interface as follows:

```
template <typename T> class Stack {
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();

    deque<T>::iterator begin(); // Problem: Illegal syntax.
    deque<T>::iterator end();   // Problem: Illegal syntax.

private:
    deque<T> elems;
};
```

This code is perfectly well-intentioned, but unfortunately is not legal C++ code. The problem has to do with the fact that `deque<T>` is a *dependent type*, a type that "depends" on a template parameter. Intuitively, this is because `deque<T>` isn't a concrete type – it's a pattern that says "once you give me the type `T`, I'll give you back a `deque` of `T`'s". Due to a somewhat arcane restriction in the C++ language, if you try to access a type nested inside of a dependent type inside of a template class (for example, trying to use the `iterator` type nested inside `deque<T>`), you must preface that type with the `typename` keyword. The correct version of the `Stack` class is as follows:

```
template <typename T> class Stack {
public:
    void push(T value);
    T pop();

    size_t size();
    bool empty();

    typename deque<T>::iterator begin(); // Now correct
    typename deque<T>::iterator end();   // Now correct

private:
    deque<T> elems;
};
```

This syntactic oddity is one of the truly embarrassing parts of C++. There is no high-level reason why `typename` should be necessary, and its existence is a perpetual source of confusion and frustration among new C++ programmers. I wholeheartedly wish that I could give you a nice clean explanation as to why `typename` is necessary, but the real answer is highly technical and in many ways unsatisfactory. Of course, this doesn't excuse you from having to put the `typename` keyword in when it's necessary, and you'll have to make sure to use it where appropriate. The good news is that `typename` is unnecessary in most circumstances. You only need to use the `typename` keyword when accessing a type nested inside of a dependent type. From a practical standpoint, this means that if you want to look up a type nested inside of a type that's either a template parameter or is parameterized over a template parameter, you must preface the type with the `typename` keyword. In the examples used in the upcoming chapters, this will only occur when looking up `iterator`s inside of STL containers that themselves are parameterized over a template argument, such as a `deque<T>::iterator` or a `vector<T>::iterator`.

To complete the above example, the implementation of the `begin` and `end` functions are shown here:

```
template <typename T> typename deque<T>::iterator Stack<T>::begin() {
    return elems.begin();
}

template <typename T> typename deque<T>::iterator Stack<T>::end() {
    return elems.end();
}
```

These functions might be the densest pieces of code you've encountered so far. The code `template <typename T>` declares that the member function implementation is an implementation of a template class's member function. `typename deque<T>::iterator` is the return type of the function, and `Stack<T>::begin()` is the name of the member function and the (empty) parameter list. When writing template classes, code like this is fairly ubiquitous, but with practice you'll be able to read this code much more easily.

Clarifying Interfaces with `const`

At its core, C++ is a language based on modifying program state. `int`s get incremented in `for` loops; `vector`s have innumerable calls to `clear`, `resize`, and `push_back`; and console I/O overwrites variables with values read directly from the user.

Consider the following function:

```
void LoadFileContents(string filename, vector<string>& out) {
    ifstream input(filename.c_str()); // Open the file
    out.clear();

    string line;
    while (getline(input, line))
        out.push_back(line);
}
```

This function takes in a string containing the name of a file, then reads the contents of the file into a `vector<string>` specified as a reference parameter. Because this function writes the result to an *existing* `vector` rather than creating a new `vector` for output, we say that the function has *side effects*. Side effects are extremely common in C++ code, and in fact without side effects C++ programs would be very difficult to write. However, when working with increasing large software systems, side effects can be dangerous. As mentioned last chapter, a single incorrect bit can take down an entire software system. Consequently, you must be very careful when designing functions with side effects so that the scope of what those side effects can modify is minimized. To see exactly why this is, let's consider the extreme case. Suppose every function in a program is allowed to modify *any* piece of data in the program. That is, whenever a function is called, the values of all variables in all functions might be changed. What would this mean for programming? Certainly, it would be much more difficult to reason about how programs operate. Consider, for example, the following loop:

```
for (size_t k = 0; k < 100; ++k)
    MyFunction();
```

Here, we iterate over the first one hundred integers, calling some function called `MyFunction`. What will this program do? Certainly it depends on the implementation of `MyFunction`, but a reasonable programmer would probably infer that `MyFunction` will be called exactly one hundred times. But this is making the reasonable assumption that because `MyFunction` isn't passed `k` as a parameter, it has no way of modifying the local variable `k` in the calling function. However, we're assuming that functions are allowed to modify *any* data in the program. Given this assumption, there's no reason that the `MyFunction` function couldn't change the value of `k` whenever it's called. It might, for example, set `k` to be 0 on every iteration, meaning that the loop will never terminate (see if you can convince yourself why this is). Similarly, the function might increment `k` by one every time it's called, causing the loop to execute half as many times as it should (since `k` will take on values 0, 2, 4, 8, ... instead of 0, 1, 2, 3, ...). Without looking at the implementation of `MyFunction`, there would be no way to know exactly what will happen to `k`. Trying to infer what the program will do by looking at its complete source code would be substantially more complicated, and building programs more than a few hundred lines of code would quickly become difficult or impossible.

Hopefully the above example has convinced you that allowing functions to make arbitrary changes to program state is not a viable option. Fortunately, C++ is specifically designed to allow programmers to constrain where data can be modified. Many of the programming concepts we've explored so far revolve around this idea. For example:

- **Avoiding global variables**. You have probably been hammered repeatedly with the idea that global variables can be hazardous. Global variables make programs significantly harder to maintain because globals can be modified by any function, at any time, for any reason. This means that if a program encounters an error because a global variable has an incorrect value, it is difficult to track down exactly where in the program that variable received the incorrect value. By using local variables instead of globals, it is easier to track down exactly where errors occur by following which functions have access to those variables.

- **Marking data members `private`.** We initially explored encapsulation from a theoretical perspective as a means for separating implementation from interface. However, encapsulation also helps control where side effects can occur in a program. If a class's data members are marked `private`, then any changes to those data members must result from the class's public interface. Verifying that the class's interface is implemented correctly can therefore increase confidence that data members aren't mercilessly clobbered.

- **Decomposing large functions.** Besides making code cleaner, more maintainable, and easier to follow, decomposition minimizes the amount of code that has access to each local variable. If a task is well-decomposed, then each function will have access only to a small number of variables and thus cannot affect much program state.

Each of these programming patterns ensure that data can only be modified in places where a programmer has explicitly granted particular functions access to that data. However, C++ provides an even stronger mechanism for preventing unexpected side effects – the `const` keyword. You have already seen `const` in the context of global constants, but the `const` keyword has many other uses. This section introduces the mechanics of `const` (for example, where `const` can be used and what it means in these contexts) and how to use it properly in C++ code.

`const` Variables

So far, you've only seen `const` in the context of global constants. For example, given the following global declaration:

```
const int MyConstant = 137;
```

Whenever you refer to the value `MyConstant` in code, the compiler knows that you're talking about the value 137. If later in your program you were to write `MyConstant = 42`, the complier would flag the line as an error because code to this effect modifies a value you explicitly indicated should never be modified. However, `const` is not limited to global constants. You can also declare *local* variables `const` to indicate that their values should never change. Consider the following code snippet:

```
for (set<int>::iterator itr = mySet.lower_bound(42);
     itr != mySet.upper_bound(137); ++itr) {
    /* ... manipulate *itr ... */
}
```

This code iterates over all of the values in an STL `set` whose values are in the range [42, 137].* However, this code is not nearly as efficient as it could be. Because C++ evaluates the looping condition of a `for` loop on each iteration, the program will evaluate the statement `itr != mySet.upper_bound(137)` once per loop iteration, so the program will recompute `mySet.upper_bound(137)` multiple times. Although the STL `set` is highly optimized and the `upper_bound` function is particularly fast (on a `set` with n elements, upper_`bound` runs in time proportional to $\log_2 n$), if there are many elements in the range [42, 137] the overhead of multiple calls to `upper_bound` may be noticeable. To fix this, we might consider computing `mySet.upper_bound` exactly once, storing the value somewhere, and then referencing the precomputed value inside the `for` loop. Here's one possible implementation:

```
set<int>::iterator stop = mySet.upper_bound(137);
for (set<int>::iterator itr = mySet.lower_bound(42); itr != stop; ++itr) {
    /* ... manipulate *itr ... */
}
```

* If you're a bit rusty on the `upper_bound` and `lower_bound` functions, refer back to the chapter on STL associative containers.

This version of the loop will run much faster than its previous incarnation. However, this new version of the loop now depends on the fact that `stop` holds the value of `mySet.upper_bound(137)` throughout the loop. If we accidentally overwrite `stop`, we'll end up iterating the wrong number of times. In other words, the variable `stop` isn't really a variable – it shouldn't *vary* – but instead should be a constant. To indicate to C++ that the value of `stop` shouldn't change, we can mark the `stop` variable `const`. This prevents us from changing the value of `stop`, and will cause a compile-time error if we try to do so. The updated code is shown here:

```
const set<int>::iterator stop = mySet.upper_bound(137);
for (set<int>::iterator itr = mySet.lower_bound(42); itr != stop; ++itr) {
    /* ... manipulate *itr ... */
}
```

This is your first glimpse of a `const` local variable. `const` local variables are similar to global constants: they must be initialized to a value, their values can't change during the course of execution, etc. In fact, the only difference between a `const` local variable and a global constant is *scope*. Global constants are globally visible and persist throughout the course of a program, while `const` local variables are created and destroyed like regular local variables.

`const` Objects

The main idea behind `const` is to let programmers communicate that the values of certain variables should not change during program execution. When working with primitive types, the meaning of "should not change" is fairly clear: an `int` changes if it is incremented, decremented or overwritten; a `bool` changes if it flips from `true` to `false`; etc. However, when working with variables of class type, our notion of "should not change" becomes substantially more nuanced. To give you a sense for why this is, let's consider a `const string`, a C++ `string` whose contents cannot be modified. We can declare a `const string` as we would any other `const` variable. For example:

```
const string myString = "This is a constant string!";
```

Note that, like all `const` variables, we are still allowed to assign the `string` an initial value.

Because the `string` is `const`, we're not allowed to modify its contents, but we can still perform some basic operations on it. For example, here's some code that prints out the contents of a `const string`:

```
const string myString = "This is a constant string!";
for(size_t i = 0; i < myString.length(); ++i)
    cout << myString[i] << endl;
```

To us humans, the above code seems completely fine and indeed it is legal C++ code. But how does the compiler know that the `length` function doesn't modify the contents of the `string`? This question may seem silly – of *course* the `length` function won't change the length of the string – but this is only obvious because we humans have a gut feeling about how a function called `length` should behave. The compiler, on the other hand, knows nothing of natural language, and could care less whether the function were named "`length`" or "`zyzzyzplyx`." This raises a natural question: given an arbitrary class, how can the compiler tell which member functions might modify the receiver object and which ones cannot? To answer this question, let's look at the prototype for the `string` member function `length`:*

* The actual implementation of the `string` class looks very different from this because string is a class template rather than an actual class. For our discussion, though, this simplification is perfectly valid.

```
class string {
public:
    size_t length() const;

    /* ... etc. ... */
};
```

Note that there is a `const` after the member function declaration. This is another use of the `const` keyword that indicates that the member function does not modify any of the class's instance variables. That is, when calling a `const` member function, you're guaranteed that the object's contents cannot change. (This isn't *technically* true, as you'll see later, but it's a perfectly valid way of thinking about `const` functions).

When working with `const` objects, you are only allowed to call member functions on that object that have been explicitly marked `const`. That is, even if you have a function that doesn't modify the object, unless you tell the compiler that the member function is `const`, the compiler will treat it as a non-`const` function. This may seem like a nuisance, but has the advantage that it forces you to decide whether or not a member function should be `const` before you begin implementing it. That is, the `const`ness of a member function is an *interface* design decision, not an *implementation* design decision.

To see how `const` member functions work in practice, let's consider a simple `Point` class that stores a point in two-dimensional space. Using the getter/setter paradigm, we end up with this class definition:

```
class Point {
public:
    Point(double x, double y);

    double getX();
    double getY();

    void setX(double newX);
    void setY(double newY);

private:
    double x, y;
};
```

Let's take a minute to think about which of these functions should be `const` and which should not be. Clearly, the `setX` and `setY` functions should not be `const`, since these operations by their very nature modify the receiver object. But what about `getX` and `getY`? Neither of these functions should modify the receiver object, since they're designed to let clients query the object's internal state. We should therefore mark these functions `const` to indicate that they cannot modify the object. This gives us the following definition of `Point`:

```
class Point {
public:
    Point(double x, double y);

    double getX() const;
    double getY() const;

    void setX(double newX);
    void setY(double newY);

private:
    double x, y;
};
```

There's only one function we've ignored so far – the `Point` constructor. However, in C++ it's illegal to mark a constructor `const`, since the typical operation of a constructor runs contrary to the notion of `const`. Take a minute to think about why this is; you'll be a better C++ coder for it!

Now that we've marked the `getX` and `getY` functions `const`, we can think about how we might go about implementing these functions. You might think that we would implement them just as we would regular member functions, and you would *almost* be right. However, the fact that the function is `const` is part of that function's signature, and so in the implementation of the `getX` and `getY` functions we will need to explicitly indicate that those member functions are `const`. Here is one possible implementation of `getX`; similar code can be written for `getY`.

```
double Point::getX() const {
     return x;
}
```

Forgetting to add this `const` can be a source of much frustration because the C++ treats `getX()` and `getX() const` as two different functions. We will discuss why this is later in this chapter.

In a `const` member function, all the class's instance variables are treated as `const`. You can read their values, but must not modify them. Similarly, inside a `const` member function, you cannot call other non-`const` member functions. The reason for this is straightforward: because non-`const` member functions can modify the receiver object, if a `const` member function could invoke a non-`const` function, then the `const` function might indirectly modify the receiver object. But beyond these restrictions, `const` member functions can do anything that regular member functions can. Suppose, for example, that we wish to update the `Point` class to support a member function called `distanceToOrigin` which returns the distance between the receiver object and the point (0, 0). Because this function shouldn't modify the receiver object, we'll mark it `const`, as shown here:

```
class Point {
public:
    Point(double x, double y);

    double getX() const;
    double getY() const;

    void setX(double newX);
    void setY(double newY);

    double distanceToOrigin() const;

private:
    double x, y;
};
```

Mathematically, the distance between a point and the origin is defined as $\sqrt{x^2+y^2}$. Using the `sqrt` function from the `<cmath>` header file, we can implement the `distanceToOrigin` function as follows:

```
void Point::distanceToOrigin() const {
     double dx = getX();    // Legal!  getX is const.
     double dy = y;         // Legal!  Reading an instance variable.
     dx *= dx;              // Legal!  We're modifying dx, which isn't an
                            //         instance variable.
     dy *= dy;              // Legal!  Same reason as above.
     return sqrt(dx + dy); // Legal!  sqrt is a free function that can't
                            //         modify the current object.
}
```

Although this function is marked `const`, we have substantial leeway with what we can do in the implementation. We can call the `getX` function, since it too is marked `const`. We can also read the value of `y` and store it in another variable because this doesn't change its value. Additionally, we can change the values of the local variables `dx` and `dy`, since doing so doesn't change any of the receiver object's data members. Remember, `const` member functions guarantee that the *receiver object* doesn't change, not that the function doesn't change the values of any variables. Finally, we can call free functions, since those functions don't have access to the class's data members and therefore cannot modify the receiver.

`const` References

Throughout this text we've used pass-by-reference by default when passing heavy objects like `vector`s and `map`s as parameters to functions. This improves program efficiency by avoiding expensive copy operations. Unfortunately, though, using pass-by-reference in this way makes it more difficult to reason about a function's behavior. For example, suppose you see the following function prototype:

```
void DoSomething(vector<int>& vec);
```

You know that this function accepts a `vector<int>` by reference, but it's not clear *why*. Does `DoSomething` modify the contents of the `vector<int>`, or is it just accepting by reference to avoid making a deep copy of the `vector`? Without knowing which of the two meanings of pass-by-reference the function writer intended, you should be wary about passing any important data into this function. Otherwise, you might end up losing important data as the function destructively modifies the parameter.

We are in an interesting situation. If we don't use pass-by-reference on functions that take large objects as parameters, our programs will pay substantial runtime costs unnecessarily. On the other hand, if we do pass large objects by reference, we make it more difficult to reason about exactly what the functions in our

program are trying to do. In other words, we can make a tradeoff between *efficiency* and *clarity*. In many cases, this tradeoff is necessary. Clean, straightforward algorithms are often fast and efficient, but more often than not they are slower than their more intricate counterparts. But in this particular arena, there is an easy way to gain the efficiency of pass-by-reference without the associated ambiguity: *`const` references*.

A `const` reference is, in many ways, like a normal reference. `const` references refer to objects and variables declared elsewhere in the program, and any operations performed on the reference are instead performed on the object being referred to. However, unlike regular references, `const` references treat the object they alias as though it were `const`. In other words, `const` references capture the notion of *looking* at an object without being able to *modify* it.

To see how `const` references work in practice, let's consider an example. Suppose that we want to write a function which prints out the contents of a `vector<int>`. Such a function clearly should not modify the `vector`, and so we can prototype this function as follows:

```
void PrintVector(const vector<int>& vec);
```

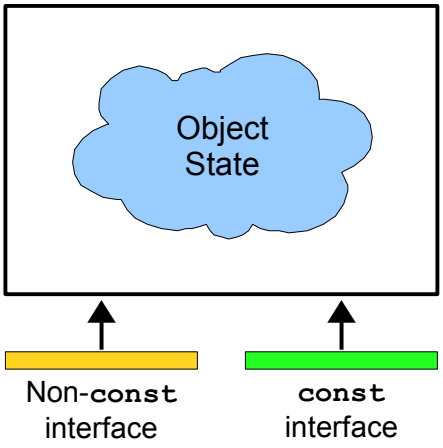
Notice that this function takes in a `const vector<int>&`. This is a `const` reference (also called a *reference-to-`const`*). Inside the `PrintVector` function, the `vec` parameter is treated as though it were `const`, and so we cannot make any changes to it. Thus the following implementation of `PrintVector` is perfectly legal:

```
void PrintVector(const vector<int>& vec) {
    for (size_t k = 0; k < vec.size(); ++k)
        cout << vec[k] << endl;
}
```

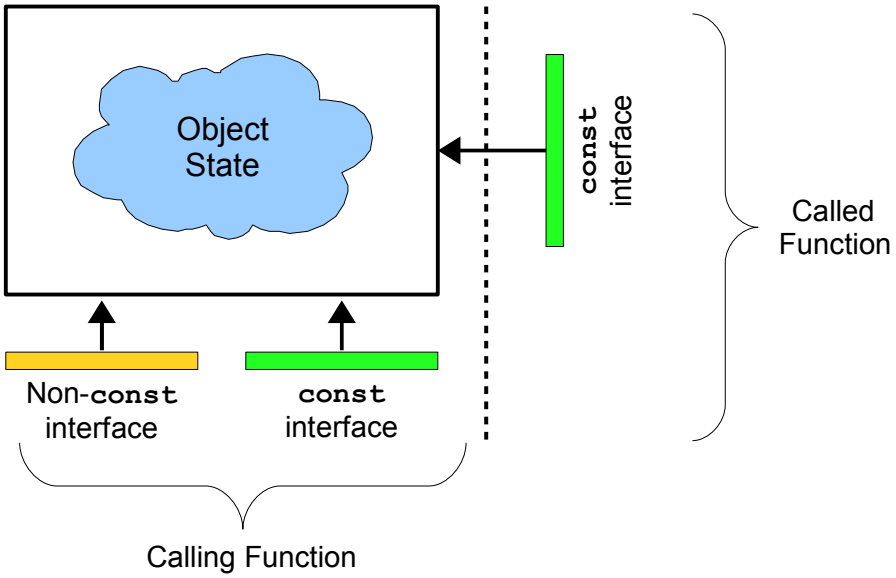
Although the `PrintVector` function takes in a reference to a `const vector<int>`, it is perfectly legal to pass both `const` and non-`const` `vector<int>`s to `PrintVector`. Whether or not the original `vector` is `const`, inside the `PrintVector` function C++ treats the vector as though it were `const`. Thus it's legal (and encouraged) to write code like this:

```
void PrintVector(const vector<int>& vec) {
    for (size_t k = 0; k < vec.size(); ++k)
        cout << vec[k] << endl;
}

int main() {
    vector<int> myVector(NUM_INTS);

    PrintVector(myVector);   // Legal!  myVector treated const in PrintVector

    myVector.push_back(137); // Legal!  myVector isn't const out here.
}
```

You might be a bit uneasy with the idea of passing a non-`const` variable into a function that takes a reference-to-`const`. After all, something of type `Type` isn't the same as something of type `const Type`. We can't assign values to `const` objects, nor can we invoke their non-`const` member functions. However, it is perfectly safe to treat a non-`const` object as though it were `const` because the legal operations on a `const` object are a *subset* of the legal operations on a non-`const` object. That is, every object's public interface can be split into two parts, a `const` interface of non-mutating operations and a non-`const` interface of operations which change the object's state. This is shown below:



In this picture, the object's internal state is represented by the fuzzy cloud, with the `const` and non-`const` interfaces each having access to the internals. When an object is non-`const`, it has both interfaces; when `const` it has only the `const` interface. Using this mental model, let's think about what happens when we pass an object by reference-to-`const` into a function. Because the called function takes in a reference-to-`const`, we can treat the function as though it expects only the `const` interface for an object. Graphically:



What does this idea mean for you as a programmer? In particular, when writing functions that need to be able to look at data but not modify it, you should strongly consider using pass-by-reference-to-`const`. This gives you the benefits of pass-by-reference (higher efficiency) with the added guarantee that the parameter won't be destructively modified. Of course, while it's legal to pass non-`const` objects to functions accepting `const` references, you cannot pass `const` objects into functions accepting non-`const` references. The reason for this is simple: if an object is marked `const`, its value cannot be changed. If a `const` object could be passed into a function by non-`const` reference, that function could modify the original object, subverting `const`ness. You can think of `const` as a universal accepter and of non-`const` as the universal donor – you can convert both `const` and non-`const` data to `const` data, but you can't convert `const` data to non-`const` data. Thinking about this using our two-interface analogy, if you have access to a class's non-`const` interface, you can always ignore it and just use the `const` interface. However, if you only have access to the `const` interface, you can't suddenly give yourself access to the non-`const` interface.

Although `const` references behave for the most part like regular references, there is one particularly important behavioral aspect where they diverge. Suppose we are given the following prototype for a function called `DoSomething`, which takes in a reference to an `int`:

```
void DoSomething(int& x);
```

Given this prototype, each of the following calls to `DoSomething` is illegal:

```
DoSomething(137);      // Problem: Cannot pass literal by reference
DoSomething(2.71828);  // Problem: Cannot pass literal by reference

double myDouble;
DoSomething(myDouble); // Problem: int& cannot bind to double
```

Let's examine exactly why each of these three calls fail. In the first case, we tried to pass the integer literal into the `DoSomething` function. This will cause problems if `DoSomething` tries to modify its parameter. Suppose, for example, that `DoSomething` is implemented as follows:

```
void DoSomething(int& x) {
    x = 0;
}
```

If we pass 137 directly into `DoSomething`, the the line `x = 0` would try to store the value 0 into the integer literal 137. This is clearly nonsensical, and so the compiler disallows it. The second erroneous call to `DoSomething` (where we pass in 2.71828) fails for the same reason. However, what of the third call, `DoSomething(myDouble)`? This fails because `myDouble` is a `double`, not an `int`, and although it's possible to typecast a `double` to an `int` the C++ language explicitly says that this is not acceptable. This may seem harsh, but it allows C++ programs to run extremely efficiently because the compiler can assume that the parameter `x` is bound to an actual `int`, not something implicitly convertible to an `int`.*

However, suppose we change the prototype of `DoSomething` to accept its parameter by `const` reference, as shown here:

```
void DoSomething(const int& x);
```

Then all of the following calls to `DoSomething` are perfectly legal:

```
DoSomething(137);      // Legal
DoSomething(2.71828);  // Legal

double myDouble;
DoSomething(myDouble);  // Legal!
```

Why the difference? Think about why all of the above examples caused problems when mixed with non-`const` references. In the first case, we might accidentally assign a new value to an integer literal; the second case ran into similar problems. In the third case, due to hardware restrictions, we cannot bind an `int&` to a `double` because writing a value to that `int&` would result in incorrect behavior. All of these

* I know that this explanation might seem a bit fuzzy, primarily because the main reason is technical and has to do with how `int`s and `double`s are represented in the machine. If you try to execute the machine code instructions to store an integer value into a variable that's declared as a `double`, the `double` will take on a completely meaningless value that has nothing to do with the integer that we intended to store in it. If you're interested in learning more about why this is, consider taking a compilers course or studying an assembly language (MIPS or x86). If you still don't understand why `int&`s can't be bound to `double`s, send me an email and I can try to explain things in more detail.

cases have to do with the fact that the reference can be used to modify the object it's bound to. But when working with `const` references, none of these problems are possible because the referenced value can't be changed through the reference.

Because normal restrictions on references do not apply to `const` references, you can treat pass-by-reference-to-`const` as a smarter version of pass-by-value. Any value that could be passed by value can be passed by reference-to-`const`, but when using reference-to-`const` objects won't be copied in most cases. We will address this later in this chapter. For now, treating pass-by-reference-to-`const` as a more efficient pass-by-reference will be wise.

`const` and Pointers

The `const` keyword is useful, but has its share of quirks. Perhaps the most persistent source of confusion when working with `const` arises when mixing `const` and pointers. For example, suppose that you want to declare a C string as a global constant. Since to declare a global C++ `string` constant you use the syntax

```
const string kGlobalCppString = "This is a string!";
```

You might assume that to make a global C string constant, the syntax would be:

```
const char* kGlobalStr = "This is a string!"; // Problem: Legal but incorrect
```

This syntax is *partially* correct. If you were ever to write `kGlobalString[0] = 'X'`, rather than getting segmentation faults at runtime (see the C strings chapter for more info), you'd instead get a compiler error that would direct you to the line where you tried to modify the global constant. But unfortunately this variable declaration contains a subtle but crucial mistake. Suppose, for example, that we write the following code:

```
kGlobalString = "Reassigned!";
```

Here, we reassign `kGlobalString` to point to the string literal "Reassigned!" Note that we didn't modify the contents of the character sequence `kGlobalString` points to – instead we changed *what character sequence `kGlobalString` points to*. In other words, we modified the *pointer*, not the *pointee*, and so the above line will compile correctly and other code that references `kGlobalString` will suddenly begin using the string "Reassigned!" instead of "This is a string!" as we would hope.

C++ distinguishes between two similar-sounding entities: a *pointer-to-`const`* and a *`const` pointer*. A pointer-to-`const` is a pointer like `kGlobalString` that points to data that cannot be modified. While you're free to reassign pointers-to-`const`, you cannot change the value of the elements they point to. To declare a pointer-to-`const`, use the syntax **`const Type* myPointer`**, with the `const` on the left of the star. Alternatively, you can declare pointers-to-`const` by writing **`Type const* myPointer`**.

A *`const` pointer*, on the other hand, is a pointer that cannot be assigned to point to a different value. Thus with a `const` pointer, you can modify the *pointee* but not the *pointer*. To declare a `const` pointer, you use the syntax **`Type* const myConstPointer`**, with the `const` on the right side of the star. Here, `myConstPointer` can't be reassigned, but you are free to modify the value it points to.

To illustrate by analogy, a pointer-to-`const` is like a telescope – it can look at other objects, and freely change which objects it looks at, but it cannot apply any changes to those objects. A `const` pointer, on the other hand, is like an industrial laser. The laser can be turned on at high power to cut a sheet of metal, or at low power to get a sense of what the metal looks like, but the beam is always pointed at the same place. You wouldn't try to cut a sheet of metal with a telescope, nor would you try to look at an object at a

distance by blasting a high-energy laser at it. Remembering whether you want a pointer-to-`const` (look but don't touch) or a `const` pointer (touch, but only touch one thing) will be tricky at first, but will become more natural as you mature as a programmer.

Note that the syntax for a pointer-to-`const` is `const Type * ptr` while the syntax for a `const` pointer is `Type * const ptr`. The only difference is where the `const` is in relation to the star. One trick for remembering which is which is to read the variable declaration from right-to-left. For example, reading `const Type * ptr` backwards says that "`ptr` is a pointer to a `Type` that's `const`," while `Type * const ptr` read backwards is "`ptr` is a `const` pointer to a `Type`."

Returning to the C string example, to make `kGlobalString` behave as a true C string constant, we'd need to make the pointer both a `const` pointer and a pointer-to-`const`. This may seem strange, but is perfectly legal C++. The result is a `const` pointer-to-`const`, a pointer that can only refer to one object and that cannot change the value of that object. Syntactically, this looks as follows:

```
const char * const kGlobalString = "This is a string!";
```

Note that there are *two* `const`s here – one before the star and one after it. Here, the first `const` indicates that you are declaring a pointer-to-`const`, while the second means that the pointer itself is `const`. Using the trick of reading the declaration backwards, here we have "`kGlobalString` is a `const` pointer to a `char` that's `const`." This is the correct way to make the C string completely `const`, although it is admittedly a bit clunky.

The following table summarizes what types of pointers you can create with `const`:

Declaration Syntax	Name	Can reassign?	Can modify pointee?
`const Type* myPtr`	Pointer-to-`const`	Yes	**No**
`Type const* myPtr`	Pointer-to-`const`	Yes	**No**
`Type* const myPtr`	`const` pointer	**No**	Yes
`const Type* const myPtr`	`const` pointer-to-`const`	**No**	**No**
`Type const* const myPtr`	`const` pointer-to-`const`	**No**	**No**

As with references and references-to-`const`, it is legal to set a pointer-to-`const` to point to a non-`const` object. This simply means that the object cannot be modified through the pointer-to-`const`.

`const_iterator`

Suppose you have a function that accepts a `vector<string>` by reference-to-`const` and you'd like to print out its contents. You might want to write code that looks like this:

```
void PrintVector(const vector<string>& myVector) {
    for(vector<string>::iterator itr = myVector.begin(); // Problem
        itr != myVector.end(); ++itr)
        cout << *itr << endl;
}
```

Initially, this code seems perfectly fine, but unfortunately the compiler will give you some positively ferocious errors if you try to compile this code. The problem has to do with a subtlety involving STL iterators and `const`. Notice that in the first part of the `for` loop we declare an object of type `vector<string>::iterator`. Because the `vector` is `const`, somehow the compiler has to know that the

iterator you're getting to the `vector` can't modify the `vector`'s contents. Otherwise, we might be able to do something like this:

```
/* Note: This code doesn't compile.  It just shows off what happens if we
 * could get an iterator to a const vector.
 */
void EvilFunction(const vector<string>& myVector) {
    vector<string>::iterator itr = myVector.begin();

    *itr = 42; // Just modified a const object!
}
```

In other words, if we could get an iterator to iterate over a `const vector`, that iterator could be used in fiendish and diabolical ways to modify the contents of the `vector`, something we promised not to do. This raises an interesting issue. Let's reconsider our (currently flawed) implementation of `PrintVector`:

```
void PrintVector(const vector<string>& myVector) {
    for(vector<string>::iterator itr = myVector.begin();  // Problem
        itr != myVector.end(); ++itr)
        cout << *itr << endl;
}
```

This code doesn't compile because the `for` loop tries to get an iterator that traverses the `vector`. As shown above, given an iterator over a `const vector`, it's possible to modify the contents of that vector and subvert `const`ness. But in this function we *don't* modify the contents of the `vector` – we're harmlessly traversing the `vector` elements and printing its contents! So why does the compiler cryptically complain about our code? The reason is that *`const`ness is conservative*. When the C++ compiler checks your code to ensure that you haven't violated the sanctity of `const`, its analysis is imprecise. Rather than determining whether or not your code actually modifies a `const` variable, it checks for syntactic structures which violate `const` – do you assign a a `const` variable? Do you invoke a non-`const` function on a `const` variable? Do you pass a `const` variable into a function which takes an argument by non-`const` reference? Because of this, it is possible to write code that cannot possibly change the value of a `const` variable but which is still rejected by the compiler. For example, consider the following code snippet:

```
void SubtleFunction(const vector<string>& myVector) {
    if (myVector.empty())
        myVector.clear(); // Error!  Calls non-const function.
}
```

This function checks to see if the parameter is the empty vector, and, if so, calls `clear` on that vector. Calling `clear` on the empty `vector` does nothing to that `vector`, and so technically speaking this function never changes the value of its parameter. However, the C++ compiler will still reject this code, because you invoked `clear` (a non-`const` member function) on a `const vector`.

Why does the compiler take this approach? The answer is that it is *provably impossible* to build a compiler that can actually determine whether or not a C++ function will change the value of a particular variable. You read that correctly – no compiler, no matter how sophisticated or clever, can correctly determine in all cases whether a C++ program will read or write a particular variable. Because of this, C++'s rules for `const`ness have a margin of error. Some programs that will never change the value of a certain variable will cause compiler errors, but any program that correctly obeys `const` will ensure that `const` variables are never overwritten. This explains why, in our simple `PrintVector` example, the compiler complained. Although we never actually overwrite the elements of the `vector` using our iterator, the fact that someone with an iterator *could* overwrite the elements of the `vector` is enough to cause the compiler to panic.

Because raw iterators don't play nicely with `const` containers, we're going to need to change our code. One idea you may have had would be to mark the iterator `const` to prevent it from overwriting the elements of the `vector`. While well-intentioned, this approach won't work. A `const` iterator is like a `const` pointer – it can't change what element it iterates over, but it can change the value of the elements it iterates over. This is the reverse of what we want – we want an iterator that can't change the values it looks at but can change which elements it iterates over. For this, we can use *`const_iterator`*s. Each STL container that defines an iterator also defines a `const_iterator` that can read the values from the container but not write them. Using a `const_iterator`, we can rewrite our implementation of `PrintVector` as follows:

```
void PrintVector(const vector<string>& myVector) {
     for(vector<string>::const_iterator itr = myVector.begin(); // Correct!
         itr != myVector.end(); ++itr)
         cout << *itr << endl;
}
```

To maintain `const`ness, you cannot use `const_iterator`s in functions like `insert` or `erase` that modify containers. You can, however, define iterator ranges using `const_iterator`s for algorithms like `binary_search` that don't modify the ranges they apply to.

There is one subtle point we have glossed over in this discussion – how does the `vector` *know* that it should hand back a `const_iterator` when marked `const` and a regular iterator otherwise? That is, how do the `vector`'s `begin` and `end` functions hand back objects of two different types based on whether or not the `vector` is `const`? The answer may surprise you. Here is a (slightly simplified) version of the `vector` interface which showcases the `begin` and `end` functions:

```
template <typename T> class vector {
public:
    iterator begin();
    iterator end();

    const_iterator begin() const;
    const_iterator end() const;

    /* ... etc. ... */
};
```

Notice that there are *two* `begin` functions – one of which is non-`const` and returns a regular `iterator`, and one of which is `const` and returns a `const_iterator`. There are similarly two versions `end` function. This is a technique known as *`const`-overloading* and allows a function to have two different behaviors based on whether or not an object is `const`. When a `const`-overloaded function is invoked, the version of the function is called that matches the `const`ness of the receiver object. For example, if you call `begin()` on a `const vector`, it will invoke the `const` version of `begin()` and return a `const_iterator`. If `begin()` is invoked on a non-`const vector`, then the non-`const` version of `begin()` will be invoked and the function will yield a regular `iterator`. We will see some examples of `const`-overloading in upcoming sections.

Limitations of `const`

Although `const` is a useful programming construct, certain aspects of `const` are counterintuitive and can lead to subtle violations of `const`ness. One common problem arises when using pointers in `const`

member functions. Suppose you have the following implementation of class `Vector`, which acts like a `vector<int>`:

```
class Vector {
public:
    /* ... other members ...*/
    void constFunction() const;

private:
    int* elems;
};
```

Consider the following legal implementation of `constFunction`:

```
Vector::constFunction() const {
    elems[0] = 137;
}
```

Unfortunately, while this code modifies the value of the object pointed to by `elems`, it is perfectly legal C++ code because it doesn't modify the value of `elems` – instead, it modifies the value of the elements *pointed at* by `elems`. In effect, because the member function is declared `const`, `elems` acts as a *`const` pointer* (the point*er* can't change) instead of a *pointer-to-`const`* (the point*ee* can't change). This raises the issue of the distinction between "bitwise `const`ness" and "semantic `const`ness." *Bitwise `const`ness*, which is the type enforced by C++, means that `const` objects are prohibited from making any bitwise changes to themselves. In the above example, since the value of the `elems` pointer didn't change, C++ considers the `constFunction` implementation `const`-correct. However, from the viewpoint of *semantic `const`ness*, `const` classes should be prohibited from modifying anything that would make the object appear somehow different. With regards to the above scenario with `elems`, the class isn't semantically `const` because the object, while `const`, was able to modify its data.

When working with `const` it's important to remember that while C++ will enforce bitwise `const`ness, you must take care to ensure that your program is semantically `const`. From your perspective as a programmer, if you invoke a `const` member function on an object, you would expect the receiver to be unchanged. If the function isn't semantically `const`, however, this won't be the case, and a `const` member function might make significant changes to the object's state.

To demonstrate the difference between bitwise and semantically `const` code, let's consider another member function of the `Vector` class that simply returns the internally stored string:

```
int* Vector::rawElems() const {
    return elems;
}
```

Initially, this code looks correct. Since returning `theString` doesn't modify the receiver object, the function is bitwise `const`. But this code entirely bypasses `const`ness. Consider, for example, this code:

```
void ProcessRawElements(const Vector& v) {
    int* elems = v.rawElems();
    for (size_t i = 1; i < v.size(); ++i) // Problem: Subverts const!
        elems[i - 1] = elems[i];
}
```

Here, we use the pointer obtained from `rawElems` to indirectly move around the elements of the array. Although `v` is marked `const` in this example, we somehow have changed its contents. This entirely defeats

the purpose of `const` and should convey why maintaining semantic `const`ness is a crucial part of good programming practice.

The above implementation of `rawElems` is fatally flawed and allows clients to subvert the `const`ness of the receiver object. How can we modify `rawElems` so that the above code no longer works? One particularly elegant solution is to modify the signature of `rawElems` so that it returns a `const int*` instead of a raw `int*`. For example:

```
const int* Vector::rawElems() const {
    return elems;
}
```

Because the returned array has been marked `const`, clients cannot modify any of the characters in the returned sequence. As a general rule of thumb, avoid returning non-`const` pointers from member functions that are marked `const`. There are exceptions to this rule, of course, but in most cases `const` functions should return pointers-to-`const`.

`mutable`

Because C++ enforces bitwise `const`ness rather than semantic `const`ness, you might find yourself in a situation where a member function changes an object's bitwise representation while still being semantically `const`. At first this might seem unusual – how could we possibly leave the object in the same logical state if we change its binary representation? – but such situations can arise in practice. For example, suppose that we want to write a class that represents a grocery list. The class definition is provided here:

```
class GroceryList {
public:
    GroceryList(const string& filename); // Load from a file.

    /* ... other member functions ... */

    string getItemAt(int index) const;

private:
    vector<string> data;
};
```

The `GroceryList` constructor takes in a filename representing a grocery list (with one element per line), then allows us to look up items in the list using the member function `getItemAt`. Initially, we might want to implement this class as follows:

```
GroceryList::GroceryList(const string& filename) {
    /* Read in the entire contents of the file and store in the vector. */
    ifstream input(filename.c_str());
    data.insert(data.begin(), istream_iterator<string>(input),
                              istream_iterator<string>());
}

/* Returns the element at the position specified by index. */
string GroceryList::getItemAt(int index) const {
    return data[index];
}
```

Here, the `GroceryList` constructor takes in the name of a file and reads the contents of that file into a `vector<string>` called `data`. The `getItemAt` member function then accepts an index and returns the corresponding element from the `vector`. While this implementation works correctly, in many cases it is needlessly inefficient. Consider the case where our grocery list is several million lines long (maybe if

we're literally trying to find enough food to feed an army), but where we only need to look at the first few elements of the list. With the current implementation of `GroceryList`, the `GroceryList` constructor will read in the entire grocery list file, an operation which undoubtedly will take a long time to finish and dwarfs the small time necessary to retrieve the stored elements. How can we resolve this problem?

There are several strategies we could use to eliminate this inefficiency. Perhaps the easiest approach is to have the constructor open the file, and then to only read in data when it's explicitly requested in the `getItemAt` function. That way, we don't read any data unless it's absolutely necessary. Here is one possible implementation:

```
class GroceryList {
public:
    GroceryList(const string& filename);

    /* ... other member functions ... */

    string getItemAt(int index); // Problem: No longer const

private:
    vector<string> data;
    ifstream sourceStream;
};

GroceryList::GroceryList(const string& filename) {
    sourceStream.open(filename.c_str()); // Open the file.
}

string GroceryList::getItemAt(int index) {
    /* Read in enough data to satisfy the request.  If we've already read it
     * in, this loop will not execute and we won't read any data.
     */
    while(index >= data.length()) {
        string line;
        getline(sourceStream, line);

        data.push_back(line);
    }
    return data[index];
}
```

Unlike our previous implementation, the new `GroceryList` constructor opens the file without reading any data. The new `getItemAt` function is slightly more complicated. Because we no longer read all the data in the constructor, when asked for an element, one of two cases will be true. First, we might have already read in the data for that line, in which case we simply hand back the value stored in the `data` object. Second, we may need to read more data from the file. In this case, we loop reading data until there are enough elements in the `data vector` to satisfy the request, then return the appropriate string.

Although this new implementation is more efficient,* the `getItemAt` function can no longer be marked `const` because it modifies both the `data` and `sourceStream` data members. If you'll notice, though, despite the fact that the `getItemAt` function is not bitwise `const`, it is semantically `const`. `GroceryList` is supposed to encapsulate an immutable grocery list, and by shifting the file reading from the constructor to `getItemAt` we have only changed the implementation, not the guarantee that `getItemAt` will not modify the list. We've reached an impasse: the interface for `GroceryList` should not depend on its

* The general technique of deferring computations until they are absolutely required is called *lazy evaluation* and is an excellent way to improve program efficiency.

implementation, and so the `getItemAt` function should be marked `const`. However, we have just produced a perfectly reasonable implementation of `GroceryList` that is not bitwise `const`, meaning that the interface needs to change to accommodate the implementation. Given our two conflicting needs – good interface design and good implementation design – how can we strike a balance?

For situations such as these, where a function is semantically `const` but not bitwise `const`, C++ provides the `mutable` keyword. `mutable` is an attribute that can be applied to data members that indicates that those data members can be modified inside member functions that are marked `const`. Using `mutable`, we can rewrite the `GroceryList` class definition to look like this:

```
class GroceryList {
public:
    GroceryList(const string& filename); // Load from a file.

    /* ... other member functions ... */

    string getItemAt(int index) const; // Now marked const

private:
    /* These data members now mutable. */
    mutable vector<string> data;
    mutable ifstream sourceStream;
};
```

Because `data` and `sourceStream` are both `mutable`, the new implementation of `getItemAt` can now be marked `const`, as shown above.

`mutable` is a special-purpose keyword that should be used sparingly and with caution. Mutable data members are exempt from the type-checking rules normally applied to `const` and consequently are prone to the same errors as non-`const` variables. Also, once data members have been marked `mutable`, *any* member function can modify them, so be sure to double-check your code for correctness. Most importantly, though, do not use `mutable` to silence compiler warnings and errors unless you're absolutely certain that it's the right thing to do. If you do, you run the risk of having functions marked `const` that are neither bitwise nor semantically `const`, entirely defeating the purpose of the `const` keyword.

`const`-Correctness

> *I still sometimes come across programmers who think `const` isn't worth the trouble. "Aw, `const` is a pain to write everywhere," I've heard some complain. "If I use it in one place, I have to use it all the time. And anyway, other people skip it, and their programs work fine. Some of the libraries that I use aren't `const`-correct either. Is `const` worth it?"*
>
> *We could imagine a similar scene, this time at a rifle range: "Aw, this gun's safety is a pain to set all the time. And anyway, some other people don't use it either, and some of them haven't shot their own feet off..."*
>
> *Safety-incorrect riflemen are not long for this world. Nor are `const`-incorrect programmers, carpenters who don't have time for hard-hats, and electricians who don't have time to identify the live wire. There is no excuse for ignoring the safety mechanisms provided with a product, and there is particularly no excuse for programmers too lazy to write `const`-correct code.*
>
> – Herb Sutter, author of *Exceptional C++* and all-around C++ guru. [Sut98]

Now that you're familiar with the mechanics of `const`, we'll explore how to use `const` correctly in real-world C++ code. In the remainder of this section, we will explore *`const`-correctness*, a system for using `const` to indicate the effects of your functions (or lack thereof). From this point forward, *all* of the code in this book will be `const`-correct and you should make a serious effort to `const`-correct your own code.

What is `const`-correctness?

At a high-level, `const`-correct code is code that clearly indicates which variables and functions cannot modify program state. More concretely, `const`-correctness requires that `const` be applied consistently and pervasively. In particular, `const`-correct code tends to use `const` as follows:

- **Objects are never passed by value**. Any object that would be passed by value is instead passed by reference-to-`const` or pointer-to-`const`.
- **Member functions which do not change state are marked `const`.** Similarly, a function that is not marked `const` should mutate state somehow.
- **Variables which are set but never modified are marked `const`.** Again, a variable not marked `const` should have its value changed at some point.

Let us take some time to explore the ramifications of each of these items individually.

Objects are never passed by value

C++ has three parameter-passing mechanisms – pass-by-value, pass-by-reference, and pass-by-pointer. The first of these requires C++ to make a full copy of the parameter being passed in, while the latter two initialize the parameter by copying a pointer to the object instead of the full object.* When passing primitive types (`int`, `double`, `char*`, etc.) as parameters to a function, the cost of a deep copy is usually negligible, but passing a heavy object like a `string`, `vector`, or `map` can at times be as expensive as the body of the function using the copy. Moreover, when passing objects by value to a function, those objects also need to be cleaned up by their destructors once that function returns. The cost of passing an object by value is thus at least the cost of a call to the class's copy constructor (discussed in a later chapter) and a call to the destructor, whereas passing that same object by reference or by pointer simply costs a single pointer copy.

To avoid incurring the overhead of a full object deep-copy, you should avoid passing objects by value into functions and should instead opt to pass either by reference or by pointer. To be `const`-correct, moreover, you should consider passing the object by reference-to-`const` or by pointer-to-`const` if you don't plan on mutating the object inside the function. In fact, you can treat pass-by-reference-to-`const` or pass-by-pointer-to-`const` as the smarter, faster way of passing an object by value. With both pass-by-value and pass-by-reference-to-`const`, the caller is guaranteed that the object will not change value inside the function call.

There is one difference between pass-by-reference-to-`const` and pass-by-value, though, and that's when using pass-by-value the function gets a fresh object that it is free to destructively modify. When using pass-by-reference-to-`const`, the function cannot mutate the parameter. At times this might be a bit vexing. For example, consider the `ConvertToLowerCase` function we wrote in the earlier chapter on STL algorithms:

---

* References are commonly implemented behind-the-scenes in a manner similar to pointers, so passing an object by reference is at least as efficient as passing an object by pointer.

```
string ConvertToLowerCase(string toConvert) {
    transform(toConvert.begin(), toConvert.end(),
              toConvert.begin(),
              ::tolower);
    return toConvert;
}
```

Here, if we simply change the parameter from being passed-by-value to being passed-by-reference-to-`const`, the code won't compile because we modify the `toConvert` variable. In situations like these, it is sometimes preferable to use pass-by-value, but alternatively we can rewrite the function as follows:

```
string ConvertToLowerCase(const string& toConvert) {
    string result = toConvert;
    transform(result.begin(), result.end(), result.begin(), ::tolower);
    return result;
}
```

Here, we simply create a new variable called `result`, initialize it to the parameter `toConvert`, then proceed as in the above function.

Member functions which do not change state are `const`

If you'll recall from our earlier discussion of `const` member functions, when working with `const` instances of a class, C++ only allows you to invoke member functions which are explicitly marked `const`. No matter how innocuous a function is, if it isn't explicitly marked `const`, you cannot invoke it on a `const` instance of an object. This means that when designing classes, you should take great care to mark `const` every member function that does not change the state of the object. Is this a lot of work? Absolutely! Does it pay off? Of course!

As an extreme example of why you should always mark nonmutating member functions `const`, suppose you try to pass a CS106B/X `Vector` to a function by reference-to-`const`. Since the `Vector` is marked as `const`, you can only call `Vector` member functions that themselves are `const`. Unfortunately, *none* of the `Vector`'s member functions are `const`, so you can't call *any* member functions of a `const Vector`. A `const` CS106B/X `Vector` is effectively a digital brick. As fun as bricks are, from a functional standpoint they're pretty much useless, so do make sure to `const`ify your member functions.

If you take care to `const` correct all member functions that don't modify state, then your code will have an additional, stronger property: member functions which are *not* marked `const` are guaranteed to make some sort of change to the receiver's internal state. From an interface perspective this is wonderful – if you want to call a particular function that isn't marked `const`, you can almost guarantee that it's going to make some form of modification to the receiver object. Thus when you're getting accustomed to a new code base, you can quickly determine what operations on an object modify that object and which just return some sort of internal state.

Variables which are set but never changed are `const`

Variables vary. That's why they're called variables. Constants, on the other hand, do not. Semantically, there is a huge difference between the sorts of operations you can perform on constants and the operations you can perform on variables, and using one where you meant to use the other can cause all sorts of debugging headaches. Using `const`, we can make explicit the distinction between constant values and true variables, which can make debugging and code maintenance much easier. If a variable is `const`, you cannot inadvertently pass it by reference or by pointer to a function which subtly modifies it, nor can you accidentally overwrite it with `=` when you meant to check for equality with `==`. Many years after

you've marked a variable `const`, programmers trying to decipher your code will let out a sigh of relief as they realize that they don't need to watch out for subtle operations which overwrite or change its value.

Without getting carried away, you should try to mark as many local variables `const` as possible. The additional compile-time safety checks and readability will more than compensate for the extra time you spent typing those extra five characters.

Example: CS106B/X Map

As an example of what `const`-correctness looks like in practice, we'll consider how to take a variant of the CS106B/X `Map` class and modify it so that it is `const`-correct. The initial interface looks like this:

```
template <typename ValueType> class Map {
public:
    Map(int sizeHint = 101);
    ~Map();

    int size();
    bool isEmpty();

    void put(string key, ValueType value);
    void remove(string key);
    bool containsKey(string key);

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(string key);
    ValueType& operator[](string key);

    void clear();

    void mapAll(void fn(string key, ValueType val));

    template <typename ClientDataType>
    void mapAll(void fn(string key, ValueType val, ClientDataType& data),
                ClientDataType& data);

    Iterator iterator();

private:
    /* ... Implementation specific ... */
};
```

The `operator[]` function shown here is what's called an *overloaded operator* and is the function that lets us write code to the effect of `myMap["Key"] = value` and `value = myMap["Key"]`. We will cover overloaded operators in a later chapter, but for now you can think of it simply as a function that is called whenever the `Map` has the element-selection brackets applied to it.

The first set of changes we should make to the `Map` is to mark all of the public member functions which don't modify state `const`. This results in the following interface:

```
/* Note: Still more changes to make.  Do not use this code as a reference! */
template <typename ValueType> class Map {
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(string key, ValueType value);
    void remove(string key);
    bool containsKey(string key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(string key) const;
    ValueType& operator[](string key);

    void clear();

    void mapAll(void fn(string key, ValueType val)) const;
    template <typename ClientDataType>
    void mapAll(void fn(string key, ValueType val, ClientDataType& data),
                ClientDataType& data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};
```

The `size`, `isEmpty`, and `containsKey` functions are all `const` because they simply query object properties without changing the `Map`. `get` is also `const` since accessing a key/value pair in the `Map` does not actually modify the underlying state, but `operator[]` should definitely *not* be marked `const` because it may update the container if the specified key does not exist.

The trickier functions to `const`-correct are `mapAll` and `iterator`. Unlike the STL iterators, CS106B/X iterators are read-only and can't modify the underlying container. Handing back an iterator to the `Map` contents therefore cannot change the `Map`'s contents, so we have marked `iterator const`. In addition, since `mapAll` passes its arguments to the callback function by value, there is no way for the callback function to modify the underlying container. It should therefore be marked `const`.

Now that the interface has its member functions `const`-ified, we should make a second pass over the `Map` and replace all instances of pass-by-value with pass-by-reference-to-`const`. In general, objects should never be passed by value and should always be passed either by pointer or reference with the appropriate `const`ness. This eliminates unnecessary copying and can make programs perform asymptotically better. The resulting class looks like this:

```
/* Note: Still more changes to make.  Do not use this code as a reference! */
template <typename ValueType> class Map {
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(const string& key, const ValueType& value);
    void remove(const string& key);
    bool containsKey(const string& key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    ValueType get(const string& key) const;
    ValueType& operator[](const string& key);

    void clear();

    void mapAll(void (fn)(const string& key, const ValueType& val)) const;
    template <typename ClientDataType>
    void mapAll(void (fn)(const string& key, const ValueType& val,
                          ClientDataType& data),
                ClientDataType &data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};
```

The parameters to `put`, `remove`, `containsKey`, `get`, and `operator[]` have all been updated to use pass-by-reference-to-`const` instead of pass-by-value. The trickier functions to modify are the `mapAll` functions. These functions themselves accept function pointers which initially took their values by value. We have updated them appropriately so that the function pointers accept their arguments by reference-to-`const`, since we assume that the class client will also be `const`-correct. Note that we did *not* mark the `ClientDataType&` parameter to `mapAll const`, since the `Map` client may actually want to modify that parameter.

There is one last change to make, and it concerns the `get` function, which currently returns a copy of the value associated with a given key. At a high-level, there is nothing intuitively wrong with returning a copy of the stored value, but from an efficiency standpoint we may end up paying a steep runtime cost by returning the object by value. After all, this requires a full object deep copy, plus a call to the object's destructor once the returned object goes out of scope. Instead, we'll modify the interface such that this function returns the object by reference-to-`const`. This allows the `Map` client to look at the value and, if they choose, copy it, but prevents clients from intrusively modifying the `Map` internals through a `const` function. The final, correct interface for `Map` looks like this:

```
/* const-corrected version of the CS106B/X Map. */
template <typename ValueType> class Map {
public:
    Map(int sizeHint = 101);
    ~Map();

    int size() const;
    bool isEmpty() const;

    void put(const string& key, const ValueType& value);
    void remove(const string& key);
    bool containsKey(const string& key) const;

    /* get causes an Error if the key does not exist.  operator[] (the
     * function which is called when you use the map["key"] syntax) creates
     * an element with the specified key if the key does not already exist.
     */
    const ValueType& get(const string& key) const;
    ValueType& operator[](const string& key);

    void clear();

    void mapAll(void (fn)(const string& key, const ValueType& val)) const;
    template <typename ClientDataType>
    void mapAll(void (fn)(const string& key, const ValueType& val,
                          ClientDataType& data),
                ClientDataType &data) const;

    Iterator iterator() const;

private:
    /* ... Implementation specific ... */
};
```

As an interesting intellectual exercise, compare this code to the original version of the `Map`. The interface declaration is considerably longer than before because of the additional `const`s, but ultimately is more pleasing. Someone unfamiliar with the interface can understand, for example, that the `Map`'s `Iterator` type cannot modify the underlying container (since otherwise the `iterator()` function wouldn't be `const`), and can also note that `mapAll` allows only a read-only map operation over the `Map`. This makes the code more self-documenting, a great boon to programmers responsible for maintaining this code base in the long run.

Why be `const`-correct?

As you can see from the example with the CS106B/X `Map`, making code `const`-correct can be tricky and time-consuming. Indeed, typing out all the requisite `const`s and `&`s can become tedious after a while. So why should you want to be `const`-correct in the first place?

There are multiple reasons why code is better off `const`-correct than non-`const`-correct. Here are a few:

- **Code correctness**. If nothing else, marking code `const` whenever possible reduces the possibility for lurking bugs in your code. Because the compiler can check which regions of the code are and are not mutable, your code is less likely to contain logic errors stemming either from a misuse of an interface or from a buggy implementation of a member function.

- **Code documentation**. `const`-correct code is self-documenting and clearly indicates to other programmers what it is and is not capable of doing. If you are presented an interface for an entirely foreign class, you may still be able to figure out which methods are safe to call with important data by noting which member functions are `const` or accept parameters by reference-to-`const`.

- **Library integration**. The C++ standard libraries and most third-party libraries are fully `const`-correct and expect that any classes or functions that interface with them to be `const`-correct as well. Writing code that is not `const`-correct can prevent you from fully harnessing the full power of some of these libraries.

Optimizing Construction with Member Initializer Lists

We've just concluded a whirlwind tour of `const`, and now it's time to change gears and talk about an entirely different aspect of class design: the member initializer list.

Normally, when you create a class, you'll initialize all of its data members in the body constructor. However, in some cases you'll need to initialize instance variables before the constructor begins running. Perhaps you'll have a `const` instance variable that you cannot assign a value, or maybe you have an object as an instance variable where you do not want to use the default constructor. For situations like these, C++ has a construct called the *member initializer list* that you can use to fine-tune the way your data members are set up. This section discusses initializer list syntax, situations where initializer lists are appropriate, and some of the subtleties of initializer lists.

How C++ Constructs Objects

To fully understand why initializer lists exist in the first place, you'll need to understand the way that C++ creates and initializes new objects.

Let's suppose you have the following class:

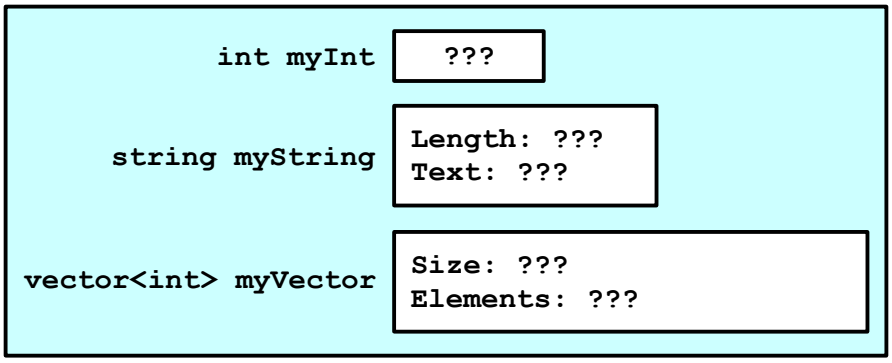
```
class SimpleClass {
public:
    SimpleClass();

private:
    int myInt;
    string myString;
    vector<int> myVector;
};
```

Let's define the `SimpleClass` constructor as follows:

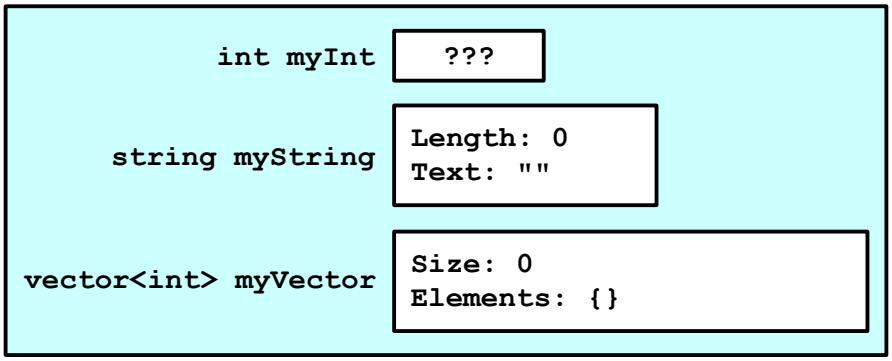
```
SimpleClass::SimpleClass() {
    myInt = 5;
    myString = "C++!";
    myVector.resize(10);
}
```

What happens when you create a new instance of the class `MyClass`? It turns out that the simple line of code `MyClass mc` actually causes a cascade of events that goes on behind the scenes. Let's take a look at what happens, step-by-step.

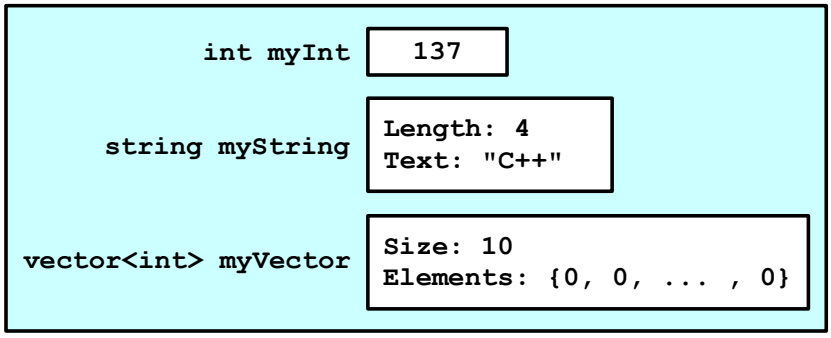
The first step in constructing a C++ object is simply to get enough space to hold all of the object's data members. The memory is not initialized to any particular value, so initially all of your object's data members hold garbage values. In memory, this looks something like this:



As you can see, none of the instance variables have been initialized, so they all contain junk. At this point, C++ calls the default constructor of each instance variable. For primitive types, this leaves the variables unchanged. After this step, our object looks something like this:



Finally, C++ will invoke the object's constructor so you can perform any additional initialization code. Using the constructor defined above, the final version of the new object will look like this:



At this point, our object is fully-constructed and ready to use.

However, there's one thing to consider here. Before we reached the `SimpleClass` constructor, C++ called the default constructor on both `myString` and `myVector`. `myString` was therefore initialized to the empty string, and `myVector` was constructed to hold no elements. However, in the `SimpleClass` constructor, we immediately assigned `myString` to hold "C++!" and resized `myVector` to hold ten elements. This means that we effectively initialized `myString` and `myVector` *twice* – once with their default constructors and once in the `SimpleClass` constructor.*

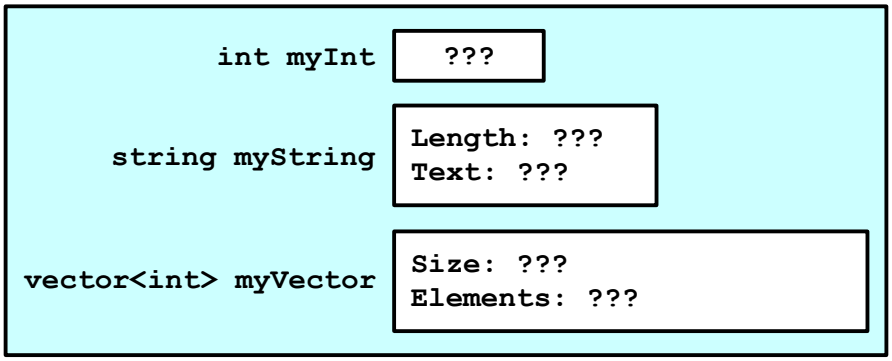
To improve efficiency and resolve certain other problems which we'll explore later, C++ has a feature called an *initializer list*. An initializer list is simply a series of values that C++ will use instead of the default values to initialize instance variables. For example, in the above example, you can use an initializer list to specify that the variable `myString` should be set to "C++!" before the constructor even begins running.

To use an initializer list, you add a colon after the constructor and then list which values to initialize which variables with. For example, here's a modified version of the `SimpleClass` constructor that initializes all the instance variables in an initializer list instead of in the constructor:

```
SimpleClass::SimpleClass() : myInt(5), myString("C++!"), myVector(10) {
    // Note: Empty constructor
}
```

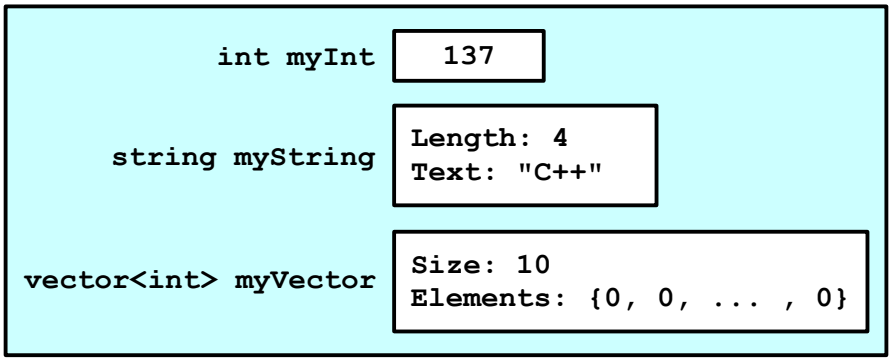
Here, we're telling C++ to initialize the variables `myInt` and `myString` to 5 and "C++!," respectively, before the class constructor is even called. Also, by writing `myVector(10)`, we're telling C++ to invoke the parametrized constructor of `myVector` passing in the value 10, which creates a `vector` with ten elements. This time, when we create a new object of type `myVector`, the creation steps will look like this:

First, as in the previous case, the object is allocated somewhere in memory and all variables have garbage values:



Next, C++ invokes all of the constructors for the object's data members using the values specified in the initializer list. The object now looks like this:

* Technically speaking, the objects are only initialized once, but the runtime efficiency is as though the objects were initialized multiple times. We'll talk about the differences between initialization and assignment in a later chapter.



Finally, C++ invokes the `MyClass` constructor, which does nothing. The final version of the class thus is identical to the above version.

As you can see, the values of the instance variables `myInt`, `myString`, and `myVector` are correctly set before the `SimpleClass` constructor is invoked. This is considerably more efficient than the previous version and will run much faster.

Note that while in this example we used initializer lists to initialize all of the object's instance variables, there is no requirement that you do so. However, in practice it's usually a good idea to set up all variables in an initializer list to make clear what values you want for each of your data members.

Parameters in Initializer Lists

In the above example, the initializer list we used specified constant values for each of the data members. However, it's both legal and useful to initialize data members with expressions instead of literal constants. For example, consider the following class, which encapsulates a rational number:

```
class RationalNumber
{
public:
    RationalNumber(int numerator = 0, int denominator = 1);

    /* ... */
private:
    int numerator, denominator;
};
```

The following is a perfectly legal constructor that initializes the data members to the values specified as parameters to the function:

```
RationalNumber::RationalNumber(int numerator, int denominator) :
    numerator(numerator), denominator(denominator)
{
    // Empty constructor
}
```

C++ is smart enough to realize that the syntax `numerator(numerator)` means to initialize the `numerator` data member to the value held by the `numerator` parameter, rather than causing a compile-time error or

initializing the `numerator` data member to itself. Code of this form might indicate that you need to rename the parameters to the constructor, but is perfectly legal.

On an unrelated note, notice that in the `RationalNumber` class declaration we specified that the `numerator` and `denominator` parameters to `RationalNumber` were equal to zero and one, respectively. These are default arguments to the constructor and allow us to call the constructor with fewer than two parameters. If we don't specify the parameters, C++ will use these values instead. For example:

```
RationalNumber fiveHalves(5, 2);
RationalNumber three(3); // Calls constructor with arguments (3, 1)
RationalNumber zero; // Calls constructor with arguments (0, 1)
```

You can use default arguments in any function, provided that if a single parameter has a default argument every parameter after it also has a default. Thus the following code is illegal:

```
void DoSomething(int x = 5, int y); // Problem: y needs a default
```

While the following is legal:

```
void DoSomething(int x, int y = 5); // Legal
```

When writing functions that take default arguments, you should *only* specify the default arguments in the function prototype, not the function definition. If you don't prototype the function, however, you should specify the defaults in the definition. C++ is very strict about this and even if you specify the same defaults in both the prototype and definition the compiler will complain.

When Initializer Lists are Mandatory

Initializer lists are useful from an efficiency standpoint. However, there are times where initializer lists are the only syntactically legal way to set up your instance variables.

Suppose we'd like to make an object called `Counter` that supports two functions, `increment` and `decrement`, that adjust an internal counter. However, we'd like to add the restriction that the `Counter` can't drop below 0 or exceed a user-defined limit. Thus we'll use a parametrized constructor that accepts an `int` representing the maximum value for the `Counter` and stores it as an instance variable. Since the value of the upper limit will never change, we'll mark it `const` so that we can't accidentally modify it in our code. The class definition for `Counter` thus looks something like this:

```
class Counter {
public:
    Counter(int maxValue);

    void increment();
    void decrement();
    int getValue() const;

private:
    int value;
    const int maximum;
};
```

Then we'd *like* the constructor to look like this:

```
Counter::Counter(int maxValue) {
    value = 0;
    maximum = maxValue; // Problem: Writing to a const value!
}
```

Unfortunately, the above code isn't valid because in the second line we're assigning a value to a variable marked `const`. Even though we're in the constructor, we still cannot violate the sanctity of `const`ness. To fix this, we'll initialize the value of `maximum` in the initializer list, so that `maximum` will be *initialized* to the value of `maxValue`, rather than *assigned* the value `maxValue`. This is a subtle distinction, so make sure to think about it before proceeding.

The correct version of the constructor is thus

```
Counter::Counter(int maxValue) : value(0), maximum(maxValue) {
    // Empty constructor
}
```

Note that we initialized `maximum` based on the constructor parameter `maxValue`. Interestingly, if we had forgotten to initialize `maximum` in the initializer list, the compiler would have reported an error. In C++, it is *mandatory* to initialize all `const` primitive-type instance variables in an initializer list. Otherwise, you'd have constants whose values were total garbage.

Another case where initializer lists are mandatory arises when a class contains objects with no legal or meaningful default constructor. Suppose, for example, that you have an object that stores a CS106B/X `Set` of a custom type `customT` with comparison callback `MyCallback`. Since the `Set` requires you to specify the callback function in the constructor, and since you're always going to use `MyCallback` as that parameter, you might think that the syntax looks like this:

```
class SetWrapperClass {
public:
    SetWrapperClass();

private:
    Set<customT> mySet(MyCallback); // Problem: Need a comparison function
};
```

Unfortunately, this isn't legal C++ syntax. However, you can fix this by rewriting the class as

```
class SetWrapperClass {
public:
    SetWrapperClass();

private:
    Set<customT> mySet; // Note: no parameters specified
};
```

And then initializing `mySet` in the initializer list as

```
SetWrapperClass::SetWrapperClass() : mySet(MyCallback) {
    // Yet another empty constructor!
}
```

Now, when the object is created, `mySet` will have `MyCallback` passed to its constructor and everything will work out correctly.

Multiple Constructors

If you write a class with multiple constructors (which, after we discuss of copy constructors, will be most of your classes), you'll need to make initializer lists for each of your constructors. That is, an initializer list for one constructor won't invoke if a different constructor is called.

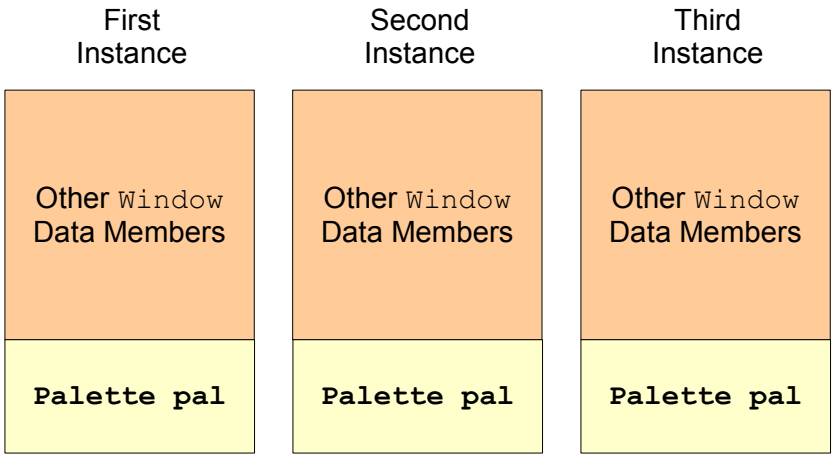
Sharing Information With `static`

Suppose that we're developing a windowed operating system and want to write the code that draws windows on the screen. We decide to create a class `Window` that exports a `drawWindow` function. In order to display the window correctly, `drawWindow` needs access to a `Palette` object that performs primitive rendering operations like drawing lines, arcs, and filled polygons. Assume that we know that the window will always be drawn with the same `Palette`. Given this description, we might initially design `Window` so that it has a `Palette` as a data member, as shown here:

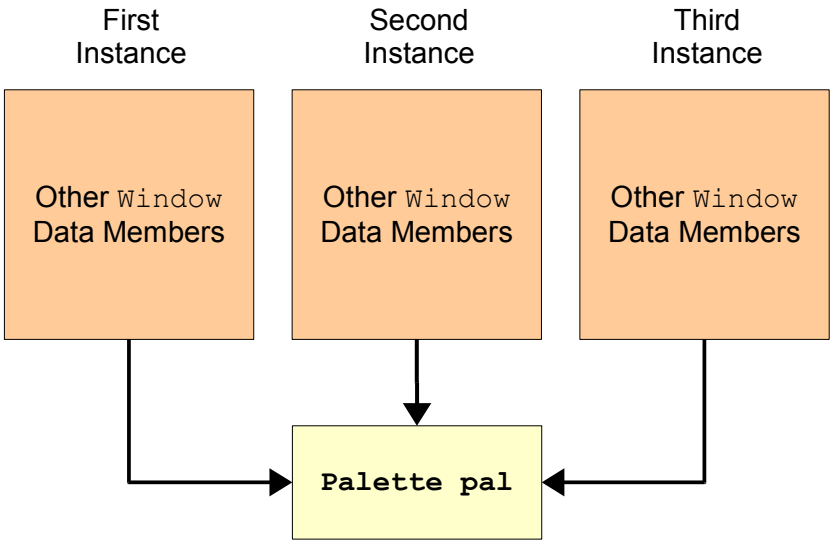
```
class Window {
public:
    /* ... constructors, destructors, etc. ...*/

    /* All windows can draw themselves. */
    void drawWindow();
private:
    /* ... other data members ... */
    Palette pal;
};
```

Now, every window has its own palette and can draw itself appropriately.

There's nothing fundamentally wrong with this setup, but it contains a small flaw. Let's suppose that we have three different window objects. In memory, those objects would look like this:



Since `pal` is a data member of `Window`, every `Window` has its own `Palette`. There might be an arbitrary number of windows on screen at any time, but there's only one screen and it doesn't make sense for every window to have its own palette. After all, each window is likely to use a similar set of colors as those used by every other window, and it seems more reasonable for every window to share a single palette, as shown here:



How can we model this in code? Using the techniques so far, we have few options. First, we could create a global `Palette` object, then have each `Window` use this global `Palette`. This is a particularly bad choice for two reasons:

- **It uses global variables**. Independently of any other strengths and weaknesses of this approach, global variables are a big programming no-no. Globals can be accessed and modified anywhere in the program, making debugging difficult should problems arise. It is also possible to inadvertently reference global variables inside of unrelated functions, leading to subtle bugs that can take down the entire program.

- **It lacks encapsulation**. Because the `Palette` is a global variable, other parts of the program can modify the `Window Palette` without going through the `Window` class. This leads to the same sorts of problems possible with public data members: class invariants breaking unexpectedly, code written with one version of `Window` breaking when the `Window` is updated, etc.

Second, we could have each `Window` object contain a *pointer* to a `Palette` object, then pass a shared `Palette` as a parameter to each instance of `Window`. For example, we could design the class like this:

```
class Window {
public:
    Window(Palette* p, /* ... */);
    /* ... other constructors, destructors, etc. ...*/

    /* All windows can draw themselves. */
    void drawWindow();

private:
    /* ... other data members ... */
    Palette* pal;
};
```

This allows us to share a single `Palette` across multiple `Window`s and looks remarkably like the above diagram. However, this approach has its weaknesses:

- **It complicates `Window` creation**. Let's think about how we would go about creating `Window`s with this setup. Before creating our first `Window`, we'd need to create a `Palette` to associate with it, as shown here:

```
Palette* p = new Palette;
Window*  w = new Window(p, /* ... */);
```

  If later we want to create more `Window`s, we'd need to keep track of the original `Palette` we used so that we can provide it as a parameter to the `Window` constructor. This means that any part of the program responsible for `Window` management needs to know about the shared `Palette`.

- **It violates encapsulation**. Clients of `Window` shouldn't have to know how `Window`s are implemented, and by requiring `Window` users to explicitly manage the shared `Palette` we're exposing too much detail about the `Window` class. This approach also locks us in to a fixed implementation. For example, what if we want to switch from `Palette` to a `TurboPalette` that draws twice as quickly? With the current approach all `Window` clients would need to upgrade their code to match the new implementation.

- **It complicates resource management**. Who is responsible for cleaning up the `Window` `Palette` at the end of the program? `Window` clients shouldn't have to, since the `Palette` really belongs to the `Window` class. But no particular `Window` owns the `Palette`, since each instance of `Window` shares a single `Palette`. There are systems we could use to make cleanup work correctly (see the later extended example on smart pointers for one possibility), but they increase program complexity.

Both of these approaches have their individual strengths, but have drawbacks that outweigh their benefits. Let's review exactly what we're trying to do. We'd like to have a single `Palette` that's shared across multiple different `Window`s. Moreover, we'd like this `Palette` to obey all of the rules normally applicable to class design: it should be encapsulated and it should be managed by the class rather than clients. Using the techniques we've covered so far it is difficult to construct a solution with these properties. For a clean solution, we'll need to introduce a new language feature: *static data members*.

Static Data Members

Static data members are data members associated with a class as a whole rather than a particular instance of that class. In the above example with `Window` and `Palette`, the window `Palette` is associated with *`Window`s in general* rather than any one specific `Window` object and is an ideal candidate for a static data member.

In many ways static data members behave similarly to regular data members. For example, if a class has a private static data member, only member functions of the class can access that variable. However, static data members behave differently from other data members because there is only one copy of each static data member. Each instance of a class containing a static data member shares the same version of that data member. That is, if a single class instance changes a static data member, the change affects all instances of that class.

The syntax for declaring static data members is slightly more complicated than for declaring nonstatic data members. There are two steps: *declaration* and *definition*. For example, if we want to create a static `Palette` object inside of `Window`, we could *declare* the variable as shown here:

```
class Window {
public:
    /* ... constructors, destructors, etc. ...*/

    /* All windows can draw themselves. */
    void drawWindow();

private:
    /* ... other data members ... */
    static Palette sharedPal;
};
```

Here, `sharedPal` is declared as a static data member using the `static` keyword. But while we've *declared* `sharedPal` as a static data member, we haven't *defined* `sharedPal` yet. Much in the same way that functions are separated into prototypes (declarations) and implementations (definitions), static data members have to be both declared inside the class in which they reside and defined inside the .cpp file associated with that class. For the above example, inside the .cpp file for the `Window` class, we would write

```
Palette Window::sharedPal;
```

There are two important points to note here. First, when defining the `static` variable, we must use the fully-qualified name (`Window::sharedPal`) instead of just its local name (`sharedPal`). Second, we do *not* repeat the `static` keyword during the variable declaration – otherwise, the compiler will think we're doing something completely different (see the "More to Explore" section). You may have noticed that even though `Window::sharedPal` is private we're still allowed to use it outside the class. This is only legal during definition, and outside of this one context it is illegal to use `Window::sharedPal` outside of the `Window` class.

In some circumstances you may want to create a class containing a static data member where the data member needs to take on an initial value. For example, if we want to create a class containing an `int` as a static data member, we would probably want to initialize the `int` to a particular value. Given the following class declaration:

```
class MyClass {
public:
    void doSomething();

private:
    static int myStaticData;
};
```

It is perfectly legal to initialize `myStaticData` as follows:

```
int MyClass::myStaticData = 137;
```

As you'd expect, this means that `myStaticData` initially holds the value 137.

Although the syntax for creating a static data member can be intimidating, once initialized static data members look just like regular data members. For example, consider the following member function:

```
void MyClass::doSomething() {
    ++myStaticData; // Modifies myStaticData for all classes
}
```

Nothing here seems all that out-of-the-ordinary and this code will work just fine. Note, however, that modifications to `myStaticData` are visible to all other instances of `MyClass`.

Let's consider another example where static data members can be useful. Suppose that you're debugging the windowing code from before and you're pretty sure that you've forgotten to `delete` all instances of `Window` that you've allocated with `new`. Since C++ won't give you any warnings about this, you'll need to do the instance counting yourself. The number of active instances of a class is class-specific information that doesn't pertain to any specific instance of the object, and this is the perfect spot to use static data members. To handle our instance counting, we'll modify the `Window` definition as follows:

```
class Window {
public:
    /* ... constructors, destructors, etc. ...*/

    void drawWindow();

private:
    /* ... other data members ... */
    static Palette sharedPal;
    static int numInstances;
};
```

We'll also define the variable outside the class as

```
int Window::numInstances = 0;
```

We know that whenever we create a new instance of a class, the class's constructor will be called. This means that if we increment `numInstances` inside the `Window` constructor, we'll correctly track the number of times the a `Window` has been created. Thus, we'll rewrite the `Window` constructor as follows:

```
Window::Window(/* ... */) {
    /* ... All older initialization code ... */
    ++numInstances;
}
```

Similarly, we'll decrement `numInstances` in the `Window` destructor. We'll also have the destructor print out a message if this is the last remaining instance so we can see how many instances are left:

```
Window::~Window() {
    /* ... All older cleanup code ... */
    --numInstances;
    if(numInstances == 0)
        cout << "No more Windows!*" << endl;
}
```

Static Member Functions

Inside of member functions, a special variable called `this` acts as a pointer to the current object. Whenever you access a class's instance variables inside a member function, you're really accessing the instance variables of the `this` pointer. For example, given the following `Point` class:

---

* This is not meant as a slight to Microsoft.

```
class Point {
public:
    Point(int xLoc, int yLoc);
    int getX() const;
    int getY() const;

private:
    int x, y;
};
```

If we implement the `Point` constructor as follows:

```
Point::Point(int xLoc, int yLoc) {
    x = xLoc;
    y = yLoc;
}
```

This code is equivalent to

```
Point::Point(int xLoc, int yLoc) {
    this->x = xLoc;
    this->y = yLoc;
}
```

How does C++ know what value `this` refers to? The answer is subtle but important. Suppose that we have a `Point` object called `pt` and that we write the following code:

```
int x = pt.getX();
```

The C++ compiler converts this into code along the lines of

```
int x = Point::getX(&pt);
```

Where `Point::getX` is prototyped as

```
int Point::getX(Point *const this);
```

This is not legal C++ code, but illustrates what's going on behind the scenes whenever you call a member function.

The mechanism behind member function calls should rarely be of interest to you as a programmer. However, the fact that an *N*-argument member function is really an (*N*+1)-argument free function can cause problems in a few places. For example, suppose that we want to provide a comparison function for `Point`s that looks like this:

```
class Point {
public:
    Point(int xLoc, int yLoc);
    int getX() const;
    int getY() const;

    bool compareTwoPoints(const Point& one, const Point& two) const;

private:
    int x, y;
};

bool Point::compareTwoPoints(const Point& one, const Point& two) const {
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}
```

If you have a `vector<Point>` that you'd like to pass to the STL `sort` algorithm, you'll run into trouble if you try to use this syntax:

```
sort(myVector.begin(), myVector.end(), &Point::compareTwoPoints); // Problem
```

The problem is that `sort` expects a comparison function that takes two parameters and returns a `bool`. However, `Point::compareTwoPoints` takes *three* parameters: two points to compare and an invisible "`this`" pointer. Thus the above code will generate an error.

If you want to define a comparison or predicate function inside of a class, you'll want that member function to not have an invisible `this`. What does this mean from a practical standpoint? A member function without a `this` pointer does not have a receiver object, and thus can only operate on its parameters and any static data members of the class it's declared in (since that data is particular to the class rather than any particular instance). Functions of this sort are called *static member functions* and can be created using the `static` keyword. In the above example with `Point`, we could create the `Point` comparison function as a `static` member function using the following syntax:

```
class Point {
public:
    Point(int xLoc, int yLoc);

    int getX() const;
    int getY() const;

    static bool compareTwoPoints(const Point& one, const Point& two);

private:
    int x, y;
};

bool Point::compareTwoPoints(const Point& one, const Point& two) const
{
    if(one.x != two.x)
        return one.x < two.x;
    return one.y < two.y;
}
```

Now, the above call to `sort` will work since `compareTwoPoints` would no longer have a `this` pointer.

Unlike static data members, when writing static member functions you do not need to separate the code out into a separate declaration and definition. You may want to do so anyway, though.

Let's return to our earlier example about tracking the number of `Window` instances currently in use. While it's nice that the destructor prints out a message when the last instance has been cleaned up, we'd prefer a more robust model where we can check how many more copies of the class exist. This function is not specific to a particular class instance, so we'll make this function static. We'll call this function `getRemainingInstances` and implement it as shown here:

```
class Window {
public:
    /* ... constructors, destructors, etc. ...*/
    void drawWindow();

    static int getRemainingInstances();

private:
    /* ... other data members ... */
    static Palette sharedPal;
    static int numInstances;
};

Palette Window::sharedPal;
int Window::numInstances = 0;

int Window::getRemainingInstances()
{
    return numInstances;
}
```

As with static data, note that when defining static member functions, you omit the `static` keyword. Only put `static` inside the class declaration.

You can invoke static member functions either using the familiar `object.method` syntax, or you can use the fully qualified name of the function. For example, with the above example, we could check how many remaining instances there were of the `MyClass` class by calling `getRemainingInstances` as follows:

```
cout << Window::getRemainingInstances() << endl;
```

`const` and `static`

Unfortunately, the `const` and `static` keywords do not always interact intuitively. One of the biggest issues to be aware of is that `const` member functions can modify `static` data. For example, consider the following class:

```
class ConstStaticClass {
public:
    void constFn() const;

private:
    static int staticData;
};

int ConstStaticClass::staticData = 0;
```

Then the following implementation of `constFn` is completely valid:

```
void ConstStaticClass::constFn() const {
    ++staticData;
}
```

Although the above implementation of `constFn` increments a static data member, the above code will compile and run without any problems. The reason is that the code doesn't modify the receiver object. Static data members are not associated with a particular class instance, so modifications to static data members do not change the state of any one instance of the class.

Additionally, since `static` member functions don't have a `this` pointer, they cannot be declared `const`. In the case of `getNumInstances`, this means that although the function doesn't modify any class data, we still cannot mark it `const`.

Integral Class Constants

There is one other topic concerning the interaction of `const` and `static`: class constants. Suppose we want to make a constant variable accessible only in the context of a class. What we want is a variable that's `const`, so it's immutable, and `static`, so that all copies of the class share the data. It's legal to declare these variables like this:

```
class ClassConstantExample {
public:
    /* Omitted. */

private:
    static const int MyConstant;
};

const int ClassConstantExample::MyConstant = 137;
```

Note the `const` in the definition of `ClassConstantExample::MyConstant`.

However, since the double declaration/definition can be a bit tedious, C++ has a built-in shorthand you can use when declaring class constants of integral types. That is, if you have a `static const int` or a `static const char`, you can condense the definition and declaration into a single statement by writing;

```
class ClassConstantExample {
public:
    /* Omitted. */

private:
    static const int MyConstant = 137; // Condense into a single line
};
```

This shorthand is common in professional code. Be careful when using the shorthand, though, because some older compilers won't correctly interpret it. Also, be aware that this only works with *integral types*, so you cannot initialize a `static const double` or `static const float` this way.

Integrating Seamlessly with Conversion Constructors

When designing classes, you might find that certain data types can logically be converted into objects of the type you're creating. For example, when writing the aforementioned rational number class, you might note that raw `int`s could have a defined conversion to `RationalNumber` objects. In these situations, it

may be useful to define *implicit conversions* between the two types. To define implicit conversions, C++ uses *conversion constructors*, constructors that accept a single parameter and initialize an object to be a copy of that parameter.

While useful, conversion constructors have several major idiosyncrasies, especially when C++ interprets normal constructors as conversion constructors. This section explores implicit type conversions, conversion constructors, and how to prevent coding errors stemming from inadvertent conversion constructors.

Implicit Conversions

In C++, an *implicit conversion* is a conversion from one type to another that doesn't require an explicit typecast. Perhaps the simplest example is the following conversion from an `int` to a `double`:

```
double myDouble = 137 + 2.71828;
```

Here, even though 137 is an `int` while 2.71828 is a `double`, C++ will implicitly convert it to a `double` so the operation can proceed smoothly.

When C++ performs implicit conversions, it does not "magically" figure out how to transform one data type into another. Rather, it creates a temporary object of the correct type that's initialized to the value of the implicitly converted object. Thus the above code is functionally equivalent to

```
double temp = double(myInt);
double myDouble = temp + 2.71828;
```

As seen here, the compiler created a temporary variable of type `double`, then initialized it by converting the integer `myInt` to a `double`. When C++ performs these conversions, it uses a special function called a *conversion constructor* to initialize the new object. Conversion constructors are simply class constructors that accept a single parameter and initialize the new object to a copy of the parameter. In the `double` example, the newly-created `double` had the same value as the `int` parameter.

Conversion constructors are surprisingly easy to write, and in fact our `RationalNumber` class already has a conversion constructor. I've reprinted this class below:

```
class RationalNumber
{
public:
    RationalNumber(int numerator = 0, int denominator = 1);

    /* ... */
private:
    int numerator, denominator;
};
```

Given just this class, we can write code like the following:

```
RationalNumber myNumber = 137;
```

How is this possible? **137** is an **int**, not a **RationalNumber**. The reason is that C++ interprets this code as a call to the **RationalNumber** constructor, passing in the integer 137. That is, the code is equivalent to

```
RationalNumber myNumber(137);
```

Which itself is equivalent to the more verbose

```
RationalNumber myNumber(137, 1);
```

In general, if you define a class that has a constructor that can be called with one argument, C++ will treat this constructor as a conversion constructor, and will translate code of the form

```
Type variable = value;
```

Into code of the format

```
Type variable(value);
```

While conversion constructors are quite useful in a wide number of circumstances, the fact that C++ automatically treats all single-parameter constructors as conversion constructors can lead to convoluted or nonsensical code. One of my favorite examples of "conversion-constructors-gone-wrong" comes from an older version of the CS106B/X ADT class libraries. Originally, the CS106B/X `Vector` was defined as

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 10); // Hint about the size of the Vector

    /* ... */
};
```

Nothing seems all that out-of-the-ordinary here – we have a `Vector` template class that lets you give the class a hint about the number of elements you will be storing in it. However, because the constructor accepts a single parameter, C++ will interpret it as a conversion constructor and thus will let us implicitly convert from `int`s to `Vector`s. This can lead to some very strange behavior. For example, given the above class definition, consider the following code:

```
Vector<int> myVector = 137;
```

This code, while nonsensical, is legal and equivalent to `Vector<int> myVector(137)`. Fortunately, this probably won't cause any problems at runtime – it just doesn't make sense in code.

However, suppose we have the following code:

```
void DoSomething(Vector<int>& myVector) {
    myVector = NULL;
}
```

This code is totally legal even though it makes no logical sense. Since `NULL` is `#define`d to be 0, The above code will create a new `Vector<int>` initialized with the parameter 0 and then assign it to `myVector`. In other words, the above code is equivalent to

```
void DoSomething(Vector<int>& myVector) {
    Vector<int> tempVector(0);
    myVector = tempVector;
}
```

`tempVector` is empty when it's created, so when we assign `tempVector` to `myVector`, we'll set `myVector` to the empty vector. Thus the nonsensical line `myVector = 0` is effectively an obfuscated call to `myVector.clear()`.

This is a quintessential example of why conversion constructors can be dangerous. When writing single-argument constructors, you run the risk of letting C++ interpret your constructor as a conversion constructor.

`explicit`

To prevent problems like the one described above, C++ provides the `explicit` keyword to indicate that a constructor must not be interpreted as a conversion constructor. If a constructor is marked `explicit`, it indicates that the constructor should not be considered for the purposes of implicit conversions. For example, let's look at the current version of the CS106B/X `Vector`, which has its constructor marked `explicit`:

```
template <typename ElemType> class Vector
{
public:
    explicit Vector(int sizeHint = 10); // Hint the size of the Vector

    /* ... */
};
```

Now, if we write code like

```
Vector<int> myVector = 10;
```

We'll get a compile-time error since there's no implicit conversion from `int` to `Vector<int>`. However, we can still write

```
Vector<int> myVector(10);
```

Which is what we were trying to accomplish in the first place. Similarly, we eliminate the `myVector = 0` error, and a whole host of other nasty problems.

When designing classes, if you have a single-argument constructor that is not intended as a conversion function, you *must* mark it `explicit` to avoid running into the "implicit conversion" trap. While indeed this is more work for you as an implementer, it will make your code safer and more stable.

Chapter Summary

- Templates can be used to define a family of abstractions that depend on an arbitrary type.
- The `typename` keyword is used to declare parameters to a template class.
- A template class's interface and implementation should be put into the .h file and no .cpp file should be created for the class.
- The `typename` keyword is also used in front of types nested inside of dependent types.
- Marking a variable `const` prevents its value from being changed after the variable is initialized.
- A `const` member function cannot modify any of the class's data members.
- `const` member functions clarify interfaces by indicating which member functions read values and which member functions write values.

- `const` can have different meanings when applied to pointers based on where the `const` occurs.
- C++ enforces bitwise `const`ness; it is up to you to ensure that your classes are semantically `const`.
- The `mutable` keyword allows you to write semantically `const` functions which are not bitwise `const`.
- Member initializer lists initialize data members to particular values before the constructor begins running.
- The `static` keyword allows you to indicate that certain data is shared across all instances of a class.
- `static` data members must be declared in the .h file and defined in the .cpp file.
- `static` member functions are functions associated with a class as a whole, rather than a particular instance of a class.
- `static` member functions are invoked by writing `ClassName::functionName()`.
- Integral class constants can be initialized in the body of the class and do not need to be separately defined.
- Conversion constructors allow classes to be initialized to values of a different type.
- The `explicit` keyword prevents accidental implicit conversions from occurring.

Practice Problems

1. How do you declare a class template?
2. How do you implement member functions for a class template?
3. Is there a difference between the `typename` and `class` keywords when declaring template arguments?
4. When is it necessary to preface a type with the `typename` keyword in a class template?
5. The following line of code declares a member function inside a class:

   ```
   const char * const MyFunction(const string& input) const;
   ```

   Explain what each `const` in this statement means.
6. What is `const`-overloading?
7. What is the difference between semantic `const`ness and bitwise `const`ness?
8. What is the difference between a `const` pointer and a pointer-to-`const`?
9. How are `const` references different from regular references?

10. What does the `mutable` keyword do?

11. What are the steps involved in class construction? In what order do they execute?

12. How do you declare an initializer list?

13. What is `static` data and how does it differ from regular member data?

14. What are the two steps required to add `static` data to a class?

15. What is a static member function? How do you call a static member function?

16. What is a conversion constructor?

17. Explain what the `explicit` keyword does.

18. The STL `map`'s bracket operator accepts a key and returns a reference to the value associated with that key. If the key is not found, the `map` will insert a new key/value pair so that the returned reference is valid. Is this function bitwise `const`? Semantically `const`?

19. When working with pointers to pointers, `const` can become considerably trickier to read. For example, a `const int * const * const` is a `const` pointer to a `const` pointer to a `const int`, so neither the pointer, its pointee, or its pointee's pointee can be changed. What is an `int * const *`? How about an `int ** const **`?

20. The CS106B/X `Vector` has the following interface:

```
template <typename ElemType> class Vector {
public:
    Vector(int sizeHint = 0);

    int size();
    bool isEmpty();

    ElemType getAt(int index);
    void setAt(int index, ElemType value);

    ElemType& operator[](int index);

    void add(ElemType elem);
    void insertAt(int index, ElemType elem);
    void removeAt(int index);

    void clear();

    void mapAll(void fn(ElemType elem));
    template <typename ClientDataType>
        void mapAll(void fn(ElemType elem, ClientDataType & data),
                    ClientDataType & data);

    Iterator iterator();
};
```

Modify this interface so that it is `const`-correct. *(Hint: You may need to `const`-overload some of these functions)*

21. Modify the Snake simulation code from the earlier extended example so that it is `const`-correct.

22. Explain each of the steps involved in object construction. Why do they occur in the order they do? Why are each of them necessary?

23. Why must a function with a single parameter with default value must have default values specified for each parameter afterwards?

24. NASA is currently working on Project Constellation, which aims to resume the lunar landings and ultimately to land astronauts on Mars. The spacecraft under development consists of two parts – an orbital module called Orion and a landing vehicle called Altair. During a lunar mission, the Orion vehicle will orbit the Moon while the Altair vehicle descends to the surface. The Orion vehicle is designed such that it does not necessarily have to have an Altair landing module and consequently can be used for low Earth orbit missions in addition to lunar journeys. You have been hired to develop the systems software for the spacecraft. Because software correctness and safety are critically important, you want to design the system such that the compiler will alert you to as many potential software problems as possible.

    Suppose that we have two classes, one called `OrionModule` and one called `AltairModule`. Since every Altair landing vehicle is associated with a single `OrionModule`, you want to define the `AltairModule` class such that it stores a pointer to its `OrionModule`. The `AltairModule` class should be allowed to modify the `OrionModule` it points to (since it needs to be able to dock/undock and possibly to borrow CPU power for critical landing maneuvers), but it should under no circumstance be allowed to change which `OrionModule` it's associated with. Here is a skeleton implementation of the `AltairModule` class:

```
class AltairModule
public:
    /* Constructor accepts an OrionModule representing the Orion
     * spacecraft this Altair is associated with, then sets up
     * parentModule to point to that OrionModule.
     */
    AltairModule(OrionModule* owner);

    /* ... */
private:
    OrionModule* parentModule;
};
```

    Given the above description about what the `AltairModule` should be able to do with its owner `OrionModule`, appropriately insert `const` into the definition of the `parentModule` member variable. Then, implement the constructor `AltairModule` such that the `parentModule` variable is initialized to point to the `owner` parameter.

25. Explain why `static` member functions cannot be marked `const`.

26. Write a class `UniquelyIdentified` such that each instance of the class has a unique ID number determined by taking the ID number of the previous instance and adding one. The first instance should have ID number 1. Thus the third instance of the class will have ID 3, the ninety-sixth instance 96, etc. Also write a `const`-correct member function `getUniqueID` that returns the class's unique ID. Don't worry about reusing older IDs if their objects go out of scope.

27. The C header file `<cstdlib>` exports two functions for random number generation – `srand`, which seeds the randomizer, and `rand`, which generates a pseudorandom `int` between 0 and the constant `RAND_MAX`. To make the pseudorandom values of `rand` appear truly random, you can seed the randomizer using the value returned by the `time` function exported from `<ctime>`. The syntax is `srand((unsigned int)time(NULL))`. Write a class `RandomGenerator` that exports a function `next` that returns a random `double` in the range [0, 1). When created, the `RandomGenerator` class should seed the randomizer with `srand` only if a previous instance of `RandomGenerator` hasn't already seeded it.

28. Does it make sense to initialize static data members in a member initializer list? Explain why or why not.

29. Should you ever mark `static` data members `mutable`? Why or why not?

These practice problems concern a `RationalNumber` class that encapsulates a rational number (that is, a number expressible as the quotient of two integers). `RationalNumber` is declared as follows:

```
class RationalNumber
{
public:
    RationalNumber(int num = 0, int denom = 1) :
        numerator(num), denominator(denom) {}

    double getValue() const {
        return static_cast<double>(numerator) / denominator;
    }

    void setNumerator(int value) {
        numerator = value;
    }
    void setDenominator(int value) {
        denominator = value;
    }

private:
    int numerator, denominator;
};
```

The constructor to `RationalNumber` accepts two parameters that have default values. This means that if you omit one or more of the parameters to `RationalNumber`, they'll be filled in using the defaults. Thus all three of the following lines of code are legal:

```
RationalNumber zero; // Value is 0 / 1 = 0
RationalNumber five(5); // Value is 5 / 1 = 5
RationalNumber piApprox(355, 113); // Value is 355/113 = 3.1415929203...
```

30. Explain why the `RationalNumber` constructor is a conversion constructor.

31. Write a `RealNumber` class that encapsulates a real number (any number on the number line). It should have a conversion constructor that accepts a `double` and a default constructor that sets the value to zero. *(Note: You only need to write one constructor. Use `RationalNumber` as an example)*

32. Write a conversion constructor that converts `RationalNumbers` into `RealNumbers`.

33. If a constructor has two or more arguments and no default values, can it be a conversion constructor?

34. C++ will apply at most one implicit type conversion at a time. That is, if you define three types `A`, `B`, and `C` such that `A` is implicitly convertible to `B` and `B` is implicitly convertible to `C`, C++ will not automatically convert objects of type `A` to objects of type `C`. Give an reason for why this might be. *(Hint: Add another implicit conversion between these types)*

Chapter 10: Operator Overloading

Consider the following C++ code snippet:

```
vector<string> myVector(kNumStrings);
for(vector<string>::iterator itr = myVector.begin();
    itr != myVector.end(); ++itr)
    *itr += "Now longer!";
```

Here, we create a `vector<string>` of a certain size, then iterate over it concatenating "Now longer!" to each of the strings. Code like this is ubiquitous in C++, and initially does not appear all that exciting. However, let's take a closer look at how this code is structured. First, let's look at exactly what operations we're performing on the iterator:

```
vector<string> myVector(kNumStrings);
for(vector<string>::iterator itr = myVector.begin();
    itr != myVector.end(); ++itr)
    *itr += "Now longer!";
```

In this simple piece of code, we're comparing the iterator against `myVector.end()` using the `!=` operator, incrementing the iterator with the `++` operator, and dereferencing the iterator with the `*` operator. At a high level, this doesn't seem all that out of the ordinary, since STL iterators are designed to look like regular pointers and these operators are all well-defined on pointers. But the key thing to notice is that STL iterators *aren't* pointers, they're *objects*, and `!=`, `*`, and `++` aren't normally defined on objects. We can't write code like `++myVector` or `*myMap = 137`, so why can these operations be applied to `vector<string>::iterator`?

Similarly, notice how we're concatenating the string "Now longer!" onto the end of the string:

```
vector<string> myVector(kNumStrings);
for(vector<string>::iterator itr = myVector.begin();
    itr != myVector.end(); ++itr)
    *itr += "Now longer!";
```

Despite the fact that `string` is an object, somehow C++ "knows" what it means to apply `+=` to `string`s.

All of the above examples are instances of *operator overloading*, the ability to specify how operators normally applicable to primitive types can interact with custom classes. Operator overloading is ubiquitous in professional C++ code and, used correctly, can make your programs more concise, more readable, and more template-friendly.

There are two overarching purposes of operator overloading. First, operator overloading enables your custom classes to act like primitive types. That is, if you have a class like `vector` that mimics a standard C++ array, you can allow clients to use array notation to access individual elements. Similarly, when designing a class encapsulating a mathematical entity (for example, a complex number), you can let clients apply mathematical operators like `+`, `-`, and `*` to your type as though it were built into the language. Second, operator overloading enables your code to interact correctly with template and library code. For example, you can overload the `<<` operator to make a class compatible with the streams library, or the `<` operator to interface with STL containers.

This chapter discusses general topics in operator overloading, demonstrating how to overload some of the more common operators. It also includes tricks and pitfalls to be aware of when overloading certain operators.

A Word of Warning

I would be remiss to discuss operator overloading without first prefacing it with a warning: operator overloading is a double-edged sword. When used correctly, operator overloading can lead to intuitive, template-friendly code that elegantly performs complex operations behind the scenes. However, incorrectly overloaded operators can lead to incredibly subtle bugs. Seemingly innocuous code along the lines of `*myItr = 5` can cause serious problems if implemented incorrectly, and without a solid understanding of how to overload operators you may find yourself in serious trouble.

There is a pearl of design wisdom that is particularly applicable to operator overloading:

> ***The Principle of Least Astonishment**:* A function's name should communicate its behavior and should be consistent with other naming conventions and idioms.

The principle of least astonishment should be fairly obvious – you should design functions so that clients can understand what those functions do simply by looking at the functions' names; that is, clients of your code should not be "astonished" that a function with one name does something entirely different. For example, a function named `DoSomething` violates the principle of least astonishment because it doesn't communicate what it does, and a function called `ComputePrimes` that reads a grocery list from a file violates the principle because the name of the function is completely different from the operation it performs. However, other violations of the principle of least astonishment are not as blatant. For example, a custom container class whose `empty` member function erases the contents of the container violates the principle of least astonishment because C++ programmers expect `empty` to mean "is the container empty?" rather than "empty the container." Similarly, a class that is bitwise `const` but not semantically `const` violates the principle, since programmers assume that objects can't be modified by `const` member functions.

When working with operator overloading, it is crucial to adhere to the principle of least astonishment. C++ lets you redefine almost all of the built-in operators however you choose, meaning that it's possible to create code that does something completely different from what C++ programmers might expect. For example, suppose that you are working on a group project and that one of your teammates writes a class `CustomVector` that acts like the STL `vector` but which performs some additional operations behind the scenes. Your program contains a small bug, so you look over your teammate's code and find the following code at one point:

```
CustomVector one = /* ... */, two = /* ... */;
one %= two;
```

What does the line `one %= two` do? Syntactically, this says "take the remainder when dividing `one` by `two`, then store the result back in `one`." But this makes no sense – how can you divide one `CustomVector` by another? You ask your teammate about this, and he informs you that the `%=` operator means "remove all elements from `one` that are also in `two`." This is an egregious violation of the principle of least astonishment. The code neither communicates what it does nor adheres to existing convention, since the semantics of the `%=` operator are meaningless when applied to linear data structures. This is not to say, of course, that having the ability to remove all elements from one `CustomVector` that are contained in another is a bad idea – in fact, it can be quite useful – but this functionality should be provided by a properly-named member function rather than a cryptically-overloaded operator. For example, consider the following code:

```
CustomVector one = /* ... */, two = /* ... */;
one.removeAllIn(two);
```

This code accomplishes the same task as the above code, but does so by explicitly indicating what operation is being performed. This code is much less likely to confuse readers and is far more descriptive than before.

As another example, suppose that your teammate also implements a class called `CustomString` that acts like the standard `string` type, but provides additional functionality. You write the following code:

```
CustomString one = "Hello", two = " World";
one += two;
cout << one + "!" << endl;
```

Intuitively, this should create two strings, then concatenate the second onto the end of the first. Next, the code prints out `one` followed by an exclamation point, yielding "Hello World!" Unfortunately, when you compile this code, you get an unusual error message – for some reason the code `one += two` compiles correctly, but the compiler rejects the code `one + "!"`. In other words, your teammate has made it legal to concatenate two strings using `+=`, but not by using `+`. Again, think back to the principle of least astonishment. Programmers tacitly expect that objects that can be added with `+` can be added with `+=` and vice-versa, and providing only half of this functionality is likely to confuse code clients.

The moral of this story is simple: when overloading operators, make sure that you adhere to existing conventions. If you don't, you will end up with code that is either incorrect, confusing, or both.

Hopefully this grim introduction has not discouraged you from using operator overloading. Operator overloading is extraordinarily useful and you will not be disappointed with the possibilities that are about to open up to you. With that said, let's begin discussing the mechanics behind this powerful technique.

Defining Overloaded Operators

We introduced operator overloading as a mechanism for redefining how the built-in operators apply to custom classes. Syntactically, how do we communicate to the C++ compiler that we want to redefine these operators? The answer is somewhat odd. Here's the original motivating example we had at the beginning of the chapter:

```
vector<string> myVector(kNumStrings);
for(vector<string>::iterator itr = myVector.begin();
    itr != myVector.end(); ++itr)
    *itr += "Now longer!";
```

When you provide this code to the C++ compiler, it interprets it as follows:

```
vector<string> myVector(kNumStrings);
for(vector<string>::iterator itr = myVector.begin();
    operator!= (itr, myVector.end());
    itr.operator++())
    (itr.operator*()).operator +=("Now longer!");
```

Notice that everywhere we used the built-in operator in conjunction with an object, the compiler reinterpreted the operator as a call to a specially-named function called `operator` **`op`**, where **`op`** is the particular operator we used. Thus `itr != myVector.end()` translated into `operator!= (itr, myVector.end())`, `++itr` was interpreted as `itr.operator++()`, etc. Although these functions may

have cryptic names, they're just regular functions. Operator overloading is simply *syntax sugar*, a way of rewriting one operation (in this case, function calls) using a different syntax (here, the built-in operators). Overloaded operators are not somehow "more efficient" than other functions simply because the function calls aren't explicit, nor are they treated any different from regular functions. They are special only in that they can are invoked using the built-in operators rather than through an explicit function calls.

If you'll notice, some of the operators used above were translated into member function calls (particularly `++` and `*`), while others (`!=`) were translated into calls to free functions. With a few exceptions, any operator can be overloaded as either a free function or a member function. Determining whether to use free functions or member functions for overloaded operators is a bit tricky, and we'll discuss it more as we continue our tour of overloaded operators.

Each of C++'s built-in operators has a certain number of *operands*. For example, the plus operator (`a + b`) has two operands corresponding to the values on the left- and right-hand sides of the operator. The pointer dereference operator (`*itr`), on the other hand, takes in only one operand: the pointer to dereference. When defining a function that is an overloaded operator, you will need to ensure that your function has one parameter for each of the operator's operands. For example, suppose that we want to define a type `RationalNumber` which encapsulates a rational number (a ratio of two integers). Because it's mathematically sound to add two rational numbers, we might want to consider overloading the `+` operator as applied to `RationalNumber` so that we can add `RationalNumber`s using an intuitive syntax. What would such a function look like? If we implement the `+` operator as a member function of `RationalNumber`, the syntax would be as follows:

```
class RationalNumber {
public:
    const RationalNumber operator+ (const RationalNumber& rhs) const;

    /* ... etc. ... */
};
```

(You might be wondering why the return type is `const RationalNumber`. For now, you can ignore that... we'll pick this up in the next section)

With `operator+` defined this way, then addition of `RationalNumber`s will be translated into calls to the member function `operator+` on `RationalNumber`. For example, the following code:

```
RationalNumber one, two;
RationalNumber three = one + two;
```

will be interpreted by the compiler as

```
RationalNumber one, two;
RationalNumber three = one.operator +(two);
```

Notice that the code `one + two` was interpreted as `one.operator+ (two)`. That is, the receiver object of the `operator+` function is the left-hand operand, while the argument is the right-hand argument. This is not a coincidence, and in fact C++ will guarantee that the relative ordering of the operands is preserved. `one + two` will never be interpreted as `two.operator+ (one)` under any circumstance, and our implementation of `operator+` can take this into account.

Alternatively, we could consider implementing `operator+` as a free function taking in two arguments. If we chose this approach, then the interface for `RationalNumber` would be as follows:

```
class RationalNumber {
public:
     /* ... etc. ... */
};

const RationalNumber operator+ (const RationalNumber& lhs,
                                const RationalNumber& rhs);
```

In this case, the code

```
RationalNumber one, two;
RationalNumber three = one + two;
```

would be interpreted by the compiler as

```
RationalNumber one, two;
RationalNumber three = operator+ (one, two);
```

Again, the relative ordering of the parameters is guaranteed to be stable, and so you can assume that the first parameter to `operator+` will always be on the left-hand side of the operator.

In some cases, two operators are syntactically identical but have different meanings. For example, the - operator can refer either to the binary subtraction operator (`a - b`) or the unary negation operator (`-a`). If overload the - operator, how does the compiler know whether your overloaded operator is the unary or binary version of -? The answer is rather straightforward: if the function operates on two pieces of data, the compiler treats `operator-` as the binary subtraction operator, and if the function uses just one piece of data it's considered to be the unary negation operator. Let's make this discussion a bit more concrete. Suppose that we want to implement subtraction on the `RationalNumber` class. Because the binary subtraction operator has two operands, we would provide subtraction as an overloaded operator either as a free function:

```
const RationalNumber operator- (const RationalNumber& lhs,
                                const RationalNumber& rhs);
```

or, alternatively, as a member function:

```
class  RationalNumber {
public:
    const RationalNumber operator- (const RationalNumber& rhs) const;

    /* ... etc. ... */
};
```

Notice that both of these functions operate on two pieces of data. In the first case, the function takes in two parameters, and in the second, the receiver object is one piece of data and the parameter is the other. If we now want to provide an implementation of `operator-` which represents the unary negation operator, we could implement it as a free function with the following signature:

```
const RationalNumber operator- (const RationalNumber& arg);
```

Or as a member function of this form:

```
class  RationalNumber {
public:
    const RationalNumber operator- () const;

    /* ... etc. ... */
};
```

Again, don't worry about why the return type is a `const RationalNumber`. We'll address this shortly.

What Operator Overloading Cannot Do

When overloading operators, you cannot define brand-new operators like `#` or `@`. After all, C++ wouldn't know the associativity or proper syntax for the operator (is `one # two + three` interpreted as `(one # two) + three` or `one # (two + three)`?) Additionally, you cannot overload any of the following operators:

Operator	Syntax	Name
`::`	`MyClass::value`	Scope resolution
`.`	`one.value`	Member selection
`?:`	`a > b ? -1 : 1`	Ternary conditional
`.*`	`a.*myClassPtr;`	Pointer-to-member selection (beyond the scope of this text)
`sizeof`	`sizeof(MyClass)`	Size of object
`typeid`	`typeid(MyClass)`	Runtime type information operator

`(T)`
`static_cast`
`dynamic_cast`
`reinterpret_cast`
`const_cast`
	`(int) myClass;`	Typecast

Note that operator overloading only lets you define what built-in operators mean when applied to *objects*. You cannot use operator overloading to redefine what addition means as applied to `int`s, nor can you change how pointers are dereferenced. Then again, by the principle of least astonishment, you wouldn't want to do this anyway.

Lvalues and Rvalues

Before we begin exploring some of the implementation issues associated with overloaded operators, we need to take a quick detour to explore two concepts from programming language theory called *lvalues* and *rvalues*. Lvalues and rvalues stand for "left values" and "right values" are are so-named because of where they can appear in an assignment statement. In particular, an lvalue is a value that can be on the left-hand side of an assignment, and an rvalue is a value that can only be on the right-hand side of an assignment. For example, in the statement `x = 5`, `x` is an lvalue and `5` is an rvalue. Similarly, in `*itr = 137`, `*itr` is the lvalue and `137` is the rvalue.

It is illegal to put an rvalue on the left-hand side of an assignment statement. For example, `137 = 42` is illegal because `137` is an rvalue, and `GetInteger() = x` is illegal because the return value of `GetInteger()` is an rvalue. However, it *is* legal to put an lvalue on the right-hand side of an assignment, as in `x = y` or `x = *itr`.

At this point the distinction between lvalues and rvalues seems purely academic. "Okay," you might say, "some things can be put on the left-hand side of an assignment statement and some things can't. So what?" When writing overloaded operators, the lvalue/rvalue distinction is extremely important. Because operator overloading lets us define what the built-in operators mean when applied to objects of class type, we have to be very careful that overloaded operators return lvalues and rvalues appropriately. For example, by default the + operator returns an rvalue; that is, it makes no sense to write

```
(x + y) = 5;
```

Since this would assign the value 5 to the result of adding `x` and `y`. However, if we're not careful when overloading the + operator, we might accidentally make the above statement legal and pave the way for nonsensical but legal code. Similarly, it *is* legal to write

```
myArray[5] = 137;
```

So the element-selection operator (the brackets operator) should be sure to return an lvalue instead of an rvalue. Failure to do so will make the above code illegal when applied to custom classes.

Recall that an overloaded operator is a specially-named function invoked when the operator is applied to an object of a custom class type. Thus the code

```
(x + y) = 5;
```

is equivalent to

```
operator+ (x, y) = 5;
```

if either `x` or `y` is an object. Similarly, if `myArray` is an object, the code

```
myArray[5] = 137;
```

is equivalent to

```
myArray.operator[](5) = 137;
```

To ensure that these functions have the correct semantics, we need to make sure that `operator+` returns an rvalue and that `operator[]` returns an lvalue. How can we enforce this restriction? The answer has to do with the return type of the two functions. To make a function that returns an lvalue, have that function return a non-`const` reference. For example, the following function returns an lvalue:

```
string& LValueFunction();
```

Because a reference is just another name for a variable or memory location, this function hands back a reference to an lvalue and its return value can be treated as such. To have a function return an rvalue, have that function return a `const` object by value. Thus the function

```
const string RValueFunction();
```

returns an rvalue.* The reason that this trick works is that if we have a function that returns a `const` object, then code like

```
RValueFunction() = 137;
```

is illegal because the return value of `RValueFunction` is marked `const`.

Lvalues and rvalues are difficult to understand in the abstract, but as we begin to actually overload particular operators the difference should become clearer.

Overloading the Element Selection Operator

Let's begin our descent into the realm of operator overloading by discussing the overloaded element selection operator (the `[ ]` operator, used to select elements from arrays). You've been using the overloaded element selection operator ever since you encountered the `string` and `vector` classes. For example, the following code uses the `vector`'s overloaded element selection operator:

```
for(int i = 0; i < myVector.size(); ++i)
    myVector[i] = 137;
```

In the above example, while it looks like we're treating the `vector` as a primitive array, we are instead calling the a function named `operator []`, passing `i` as a parameter. Thus the above code is equivalent to

```
for(int i = 0; i < myVector.size(); ++i)
    myVector.operator [](i) = 137;
```

To write a custom element selection operator, you write a member function called `operator []` that accepts as its parameter the value that goes inside the brackets. Note that while this parameter can be of any type (think of the STL `map`), you can only have a single value inside the brackets. This may seem like an arbitrary restriction, but makes sense in the context of the principle of least astonishment: you can't put multiple values inside the brackets when working with raw C++ arrays, so you shouldn't do so when working with custom objects.

When writing `operator []`, as with all overloaded operators, you're free to return objects of whatever type you'd like. However, remember that when overloading operators, it's essential to maintain the same functionality you'd expect from the naturally-occurring uses of the operator. In the case of the element selection operator, this means that the return value should be an lvalue, and in particular a reference to some internal class data determined by the index. For example, here's one possible prototype of the C++ `string`'s element selection operator:

```
class string {
public:
    /* ... */

    char& operator [] (size_t position);
};
```

* Technically speaking any non-reference value returned from a function is an rvalue. However, when returning objects from a function, the rvalue/lvalue distinction is blurred because the assignment operator and other operators are member functions that can be invoked regardless of whether the receiver is an rvalue or lvalue. The additional `const` closes this loophole.

Here, `operator[]` takes in an `int` representing the index and returns a reference to the character at that position in the `string`. If `string` is implemented as a wrapper for a raw C string, then one possible implementation for `operator[]` might be

```
char& string::operator[] (size_t index) {
    return theString[index]; // Assuming theString is an array of characters
}
```

Because `operator[]` returns a reference to an element, it is common to find `operator[]` paired with a `const`-overload that returns a `const` reference to an element in the case where the receiver object is immutable. There are exceptions to this rule, such as the STL `map`, but in the case of `string` we should provide a `const` overload, as shown here:

```
class string {
public:
    /* ... */

          char& operator [] (size_t position);
    const char& operator [] (size_t position) const;
};
```

The implementation of the `const operator[]` function is identical to the non-`const` version.

When writing the element selection operator, it's completely legal to modify the receiver object in response to a request. For example, with the STL `map`, `operator[]` will silently create a new object and return a reference to it if the key isn't already in the `map`. This is part of the beauty of overloaded operators – you're allowed to perform any necessary steps to ensure that the operator makes sense.

Unfortunately, if your class encapsulates a multidimensional object, such as a matrix or hierarchical key-value system, you cannot "overload the `[][]` operator." A class is only allowed to overload one level of the bracket syntax; it's not legal to design objects that doubly-overload `[]`.*

Overloading Compound Assignment Operators

The compound assignment operators are operators of the form ***op***= (for example, `+=` and `*=`) that update an object's value but do not overwrite it. Compound assignment operators are often declared as member functions with the following basic prototype:

```
MyClass& operator += (const ParameterType& param)
```

For example, suppose we have the following class, which represents a vector in three-dimensional space:

```
class Vector3D {
public:
    /* ... */
private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

* There is a technique called *proxy objects* that can make code along the lines of `myObject[x][y]` legal. The trick is to define an `operator[]` function for the class that returns another object that itself overloads `operator[]`. We'll see this trick used in the upcoming chapter on a custom `grid` class.

It is legal to add two mathematical vectors to one another; the result is the vector whose components are the pairwise sum of each of the components of the source vectors. If we wanted to define a `+=` operator for `Vector3D` to let us perform this addition, we would modify the interface of `Vector3D` as follows:

```
class Vector3D {
public:
    /* ... */
    Vector3D& operator+= (const Vector3D& other);

private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

This could then be implemented as

```
Vector3D& Vector3D::operator +=(const Vector3D& other) {
    for(int i = 0; i < NUM_COORDINATES; ++i)
       coordinates[i] += other.coordinates[i];
    return *this;
}
```

If you'll notice, `operator+=` returns `*this`, a reference to the receiver object. Recall that when overloading operators, you should make sure to define your operators such that they work identically to the C++ built-in operators. It turns out that the `+=` operator yields an lvalue, so the code below, though the quintessence of abysmal style, is perfectly legal:

```
int one, two, three, four;
(one += two) += (three += four);
```

Since overloaded operators let custom types act like primitives, the following code should also compile:

```
Vector3D one, two, three, four;
(one += two) += (three += four);
```

If we expand out the calls to the overloaded `+=` operator, we find that this is equivalent to

```
Vector3D one, two, three, four;
one.operator+=(two).operator +=(three.operator +=(four));
```

Note that the reference returned by `one.operator+=(two)` then has its own `+=` operator invoked. Since `operator +=` is not marked `const`, had the `+=` operator returned a `const` reference, this code would have been illegal. Make sure to have any (compound) assignment operator return `*this` as a non-`const` reference.

Unlike the regular assignment operator, with the compound assignment operator it's commonly meaningful to accept objects of different types as parameters. For example, we might want to make expressions like `myVector *= 137` for `Vector3D`s meaningful as a scaling operation. In this case, we can simply define an `operator *=` that accepts a `double` as its parameter. For example:

```
class Vector3D {
public:
    /* ... */
    Vector3D& operator += (const Vector3D& other);
    Vector3D& operator *= (double scaleFactor);

private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

Despite the fact that the receiver and parameter have different types, this is perfectly legal C++. Here's one possible implementation:

```
Vector3D& Vector3D::operator*= (double scaleFactor) {
    for(int k = 0; k < NUM_COORDINATES; ++k)
        coordinates[k] *= scaleFactor;
    return *this;
}
```

Although we have implemented `operator+=` and `operator*=` for the `Vector3D` class, C++ will not automatically provide us an implementation of `operator-=` and `operator/=`, despite the fact that those functions can easily be implemented as wrapped calls to the operators we've already implemented. This might seem counterintuitive, but prevents errors from cases where seemingly symmetric operations are undefined. For example, it is legal to multiply a vector and a matrix, though the division is undefined. For completeness' sake, we'll prototype `operator-=` and `operator/=` as shown here:

```
class Vector3D {
public:
    /* ... */
    Vector3D& operator += (const Vector3D& other);
    Vector3D& operator -= (const Vector3D& other);

    Vector3D& operator *= (double scaleFactor);
    Vector3D& operator /= (double scaleFactor);

private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

Now, how might we go about implementing these operators? `operator/=` is the simplest of the two and can be implemented as follows:

```
Vector3D& Vector3D::operator /= (double scaleFactor) {
    *this *= 1.0 / scaleFactor;
    return *this;
}
```

This implementation, though cryptic, is actually quite elegant. The first line, `*this *= 1.0 / scaleFactor`, says that we should multiply the receiver object (`*this`) by the reciprocal of `scaleFactor`. The `*=` operator is the compound multiplication assignment operator that we wrote above, so this code invokes `operator*=` on the receiver. In fact, this code is equivalent to

```
Vector3D& Vector3D::operator /= (double scaleFactor) {
    operator*= (1.0 / scaleFactor);
    return *this;
}
```

Depending on your taste, you might find this particular syntax more readable than the first version. Feel free to use either version.

Now, how would we implement `operator-=`, which performs a componentwise subtraction of two `Vector3D`s? At a high level, subtracting one vector from another is equal to adding the inverse of the second vector to the first, so we might want to write code like this:

```
Vector3D& Vector3D::operator -= (const Vector3D& other) {
    *this += -other;
    return *this;
}
```

That is, we add `-other` to the receiver object. But this code is illegal because we haven't defined the unary minus operator as applied to `Vector3D`s. Not to worry – we can overload this operator as well. The syntax for this function is as follows:

```
class Vector3D {
public:
    /* ... */
    Vector3D& operator += (const Vector3D& other);
    Vector3D& operator -= (const Vector3D& other);

    Vector3D& operator *= (double scaleFactor);
    Vector3D& operator /= (double scaleFactor);

    const Vector3D operator- () const;

private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

There are four pieces of information about this function that deserve attention:

- The name of the unary minus function is `operator -`.
- The function takes no parameters. This lets C++ know that the function is the *unary* minus function (I.e. `-myVector`) rather than the *binary* minus function (`myVector - myOtherVector`).
- The function returns a `const Vector3D`. The unary minus function returns an *rvalue* rather than an *lvalue*, since code like `-x = 137` is illegal. As mentioned above, this means that the return value of this function should be a `const Vector3D`.
- The function is marked `const`. Applying the unary minus to an object doesn't change its value, and to enforce this restriction we'll mark `operator - const`.

One possible implementation of `operator-` is as follows:

```
const Vector3D Vector3D::operator- () const {
    Vector3D result;
    for(int k = 0; k < NUM_COORDINATES; ++k)
        result.coordinates[k] = -coordinates[k];
    return result;
}
```

Note that the return type of this function is `const Vector3D` while the type of `result` inside the function is `Vector3D`. This isn't a problem, since returning an object from a function yields a new temporary object and it's legal to initialize a `const Vector3D` using a `Vector3D`.

When writing compound assignment operators, as when writing regular assignment operators, you must be careful that self-assignment works correctly. In the above example with `Vector3D`'s compound assignment operators we didn't need to worry about this because the code was structured correctly. However, when working with the C++ `string`'s `+=` operator, since the `string` needs to allocate a new buffer capable of holding the current `string` appended to itself, it would need to handle the self-assignment case, either by explicitly checking for self-assignment or through some other means.

Overloading Mathematical Operators

In the previous section, we provided overloaded versions of the `+=` family of operators. Thus, we can now write classes for which expressions of the form `one += two` are valid. However, the seemingly equivalent expression `one = one + two` will still not compile, since we haven't provided an implementation of the lone `+` operator. C++ will not automatically provide implementations of related operators given a single overloaded operator, since in some cases this could result in nonsensical or meaningless behavior.

The built-in mathematical operators yield rvalues, so code like `(x + y) = 137` will not compile. Consequently, when overloading the mathematical operators, make sure they return rvalues as well by having them return `const` objects.

Let's consider an implementation of `operator +` for our `Vector3D` class. Because the operator yields an rvalue, we're supposed to return a `const Vector3D`, and based on our knowledge of parameter passing, we know that we should accept a `const Vector3D &` as a parameter. There's one more bit we're forgetting, though, and that's to mark the `operator +` function `const`, since `operator +` creates a new object and doesn't modify either of the values used in the arithmetic statement. This results in the following code

```
class Vector3D {
public:
    /* ... */
    Vector3D& operator += (const Vector3D& other);
    const Vector3D operator+ (const Vector3D& other);

    Vector3D& operator -= (const Vector3D& other);

    Vector3D& operator *= (double scaleFactor);
    Vector3D& operator /= (double scaleFactor);

    const Vector3D operator- () const;

private:
    static const int NUM_COORDINATES = 3;
    double coordinates[NUM_COORDINATES];
};
```

How are we to implement this function? We could just do a component-by-component addition, but it's actually much easier to just write the function in terms of our `operator +=`. The full version of this code is shown below:

```
const Vector3D Vector3D::operator +(const Vector3D& other) const {
    Vector3d result = *this; // Make a deep copy of this Vector3D.
    result += other;         // Use existing addition code.
    return result;
}
```

Now, all of the code for `operator +` is unified, which helps cut down on coding errors.

There is an interesting and common case we haven't addressed yet – what if one of the operands isn't of the same type as the class? For example, if you have a `Matrix` class that encapsulates a 3x3 matrix, as shown here:

```
class Matrix {
public:
    /* ... */

    Matrix& operator *= (double scalar); // Scale all entries

private:
    static const int MATRIX_SIZE = 3;
    double entries[MATRIX_SIZE][MATRIX_SIZE];
};
```

Note that there is a defined `*=` operator that scales all elements in the matrix by a `double` factor. Thus code like `myMatrix *= 2.71828` is well-defined. However, since there's no defined `operator *`, currently we cannot write `myMatrix = myMatrix * 2.71828`.

Initially, you might think that we could define `operator *` just as we did `operator +` in the previous example. While this will work in most cases, it will lead to some problems we'll need to address later. For now, however, let's add the member function `operator *` to `Matrix`, which is defined as

```
const Matrix Matrix::operator *(double scalar) const {
    MyMatrix result = *this;
    result *= scalar;
    return result;
}
```

Now, we can write expressions like `myMatrix = myMatrix * 2.71828`. However, what happens if we write code like `myMatrix = 2.71828 * myMatrix`? This is a semantically meaningful expression, but unfortunately it won't compile. When interpreting overloaded operators, C++ will always preserve the order of values in an expression.* Thus `2.71828 * myMatrix` is *not* the same as `myMatrix * 2.71828`. Remember that the reason that `myMatrix * 2.71828` is legal is because it's equivalent to `myMatrix.operator *(2.71828)`. The expression `2.71828 * myMatrix`, on the other hand, is illegal because C++ will try to expand it into `(2.71828).operator *(myMatrix)`, which makes no sense.

* One major reason for this is that sometimes the arithmetic operators won't be commutative. For example, given matrices **A** and **B**, **AB** is not necessarily the same as **BA**, and if C++ were to arbitrarily flip parameters it could result in some extremely difficult-to-track bugs.

To fix this, we can make `operator *` a free function that accepts two parameters, a `double` and a `Matrix`, and returns a `const Matrix`. Thus code like `2.71828 * myMatrix` will expand into calls to `operator *(2.71828, myMatrix)`. The new version of `operator *` is defined below:

```
const Matrix operator * (double scalar, const Matrix& matrix) {
    Matrix result = *matrix;
    matrix *= scalar;
    return result;
}
```

But here we run into the same problem as before if we write `myMatrix * 2.71828`, since we haven't defined a function accepting a `Matrix` as its first parameter and an `double` as its second. To fix this, we'll define a *second* free function `operator *` with the parameters reversed that's implemented as a call to the other version:

```
const Matrix operator *(const Matrix& matrix, double scalar) {
    return scalar * matrix; // Calls operator* (scalar, matrix)
}
```

As a general rule, mathematical operators like `+` should always be implemented as free functions. This prevents problems like those described here.

One important point to notice about overloading the mathematical operators versus the compound assignment operators is that it's considerably faster to use the compound assignment operators over the standalone mathematical operators. Not only do the compound assignment operators work in-place (that is, they modify existing objects), but they also return references instead of full objects. From a performance standpoint, this means that given these three `string`s:

```
string one = "This ";
string two = "is a ";
string three = "string!";
```

Consider these two code snippets to concatenate all three strings:

```
/* Using += */
string myString = one;
myString += two;
myString += three;

/* Using + */
string myString = one + two + three
```

Oddly, the second version of this code is considerably slower than the first because the `+` operator generates temporary objects. Remember that `one + two + three` is equivalent to

```
operator +(one, operator +(two, three))
```

Each call to `operator +` returns a new `string` formed by concatenating the parameters, so the code `one + two + three` creates two temporary `string` objects. The first version, on the other hand, generates no temporary objects since the `+=` operator works in-place. Thus while the first version is less sightly, it is significantly faster than the second.

Overloading ++ and --

Overloading the increment and decrement operators can be a bit tricky because there are two versions of each operator: *prefix* and *postfix*. Recall that `x++` and `++x` are different operations – the first will evaluate to the value of `x`, then increment `x`, while the second will increment `x` and then evaluate to the updated value of `x`. You can see this below:

```
int x = 0
cout << x++ << endl; // Prints: 0
cout << x << endl;   // Prints: 1

x = 0;
cout << ++x << endl; // Prints: 1
cout << x << endl;   // Prints: 1
```

Although this distinction is subtle, it's tremendously important for efficiency reasons. In the postfix version of `++`, since we have to return the value of the variable was before it was incremented, we'll need to make a full copy of the old version and then return it. With the prefix `++`, since we're returning the current value of the variable, we can simply return a reference to it. Thus the postfix `++` can be noticeably slower than the prefix version; this is the reason that when advancing an STL iterator it's faster to use the prefix increment operator.

The next question we need to address is how we can legally use `++` and `--` in regular code. Unfortunately, it can get a bit complicated. For example, the following code is totally legal:

```
int x = 0;
++++++++++++++x; // Increments x seven times.
```

This is legal because it's equivalent to

```
++(++(++(++(++(++(++x))))));
```

The prefix `++` operator returns the variable being incremented as an *lvalue*, so this statement means "increment x, then increment x again, etc."

However, if we use the postfix version of `++`, as seen here:

```
x++++++++++++++; // Error
```

We get a compile-time error because `x++` returns the original value of x as an *rvalue*, which can't be incremented because that would require putting the rvalue on the left side of an assignment (in particular, x = x + 1).

Now, let's actually get into some code. Unfortunately, we can't just sit down and write `operator ++`, since it's unclear *which* `operator ++` we'd be overloading. C++ uses a hack to differentiate between the prefix and postfix versions of the increment operator: when overloading the prefix version of `++` or `--`, you write `operator ++` as a function that takes no parameters. To overload the postfix version, you'll overload `operator ++`, but the overloaded operator will accept as a parameter the integer value 0. In code, these two declarations look like

```
class MyClass {
public:
    /* ... */

    MyClass& operator ++(); // Prefix
    const MyClass operator ++(int dummy); // Postfix

private:
    /* ... */
};
```

Note that the prefix version returns a `MyClass&` as an lvalue and the postfix version a `const MyClass` as an rvalue.

We're allowed to implement `++` and `--` in any way we see fit. However, one of the more common tricks is to write the `++` implementation as a wrapped call to `operator +=`. Assuming you've provided this function, we can then write the prefix `operator ++` as

```
MyClass& MyClass::operator ++() {
    *this += 1;
    return *this;
}
```

And the postfix `operator ++` as

```
const MyClass MyClass::operator ++(int dummy) {
    MyClass oldValue = *this; // Store the current value of the object.
    ++*this;
    return oldValue;
}
```

Notice that the postfix `operator++` is implemented in terms of the prefix `operator++`. This is a fairly standard technique and cuts down on the amount of code that you will need to write for the functions.

In your future C++ career, you may encounter versions of `operator++` that look like this:

```
const MyClass MyClass::operator ++(int) {
    MyClass oldValue = *this; // Store the current value of the object.
    ++*this;
    return oldValue;
}
```

Although this function takes in an `int` parameter, that parameter does not have a name. It turns out that it is perfectly legal C++ code to write functions that accept parameters but do not give names to those parameters. In this case, the function acts just like a regular function that accepts a parameter, except that the parameter cannot be used inside of the function. In the case of `operator++`, this helps give a cue to readers that the integer parameter is not meaningful and exists solely to differentiate the prefix and postfix versions of the function.

Overloading Relational Operators

Perhaps the most commonly overloaded operators (other than `operator =`) are the relational operators; for example, `<` and `==`. Unlike the assignment operator, by default C++ does not provide relational operators for your objects. This means that you must explicitly overload the `==` and related operators to

use them in code. The prototype for the relational operators looks like this (written for <, but can be for any of the relational operators):

```
class MyClass {
public:
    /* ... */
    bool operator < (const MyClass& other) const;

private:
    /* ... */
};
```

You're free to choose any means for defining what it means for one object to be "less than" another. However, when doing so, you must take great care to ensure that your less-than operator defines a *total ordering* on objects of your type. This means that the following must be true about the behavior of the less-than operator:

- **Trichotomy**: For any *a* and *b*, exactly one of $a < b$, $a = b$, and $b < a$ is true.
- **Transitivity**: If $a < b$ and $b < c$, then $a < c$.

These properties of < are important because they allow the notion of *sorted order* to make sense. If either of these conditions does not hold, then it is possible to encounter strange situations in which a collection of elements cannot be put into ascending order. For example, suppose that we have the following class, which represents a point in two-dimensional space:

```
class Point {
public:
    Point(double x, double y);

    double getX() const;
    void setX(double value);

    double getY() const;
    void setY(double value);
}
```

Now consider the following implementation of a less-than operator for comparing `Point`s:

```
bool operator< (const Point& one, const Point& two) {
    return one.getX() < two.getX() && one.getY() < two.getY();
}
```

Intuitively, this may seem like a reasonable definition of the < operator: point *a* is less than point *b* if both coordinates of *a* are less than the corresponding coordinates of *b*. However, this implementation of < is bound to cause problems. In particular, consider the following code:

```
Point one(1, 0), two(0, 1);
cout << (one < two) << endl;
cout << (two < one) << endl;
```

Here, we create two points called `one` and `two` and compare them using the < operator. What will the first line print? Using the above definition of `operator<`, the comparison `one < two` will evaluate to false because the *x* coordinate of `one` is greater than the *x* coordinate of `two`. But what about `two < one`? In this case, `two`'s *x* coordinate is less than `one`'s, but its *y* coordinate is greater than `one`'s. Consequently, we have that `two < one` also evaluates to false. We have reached a precarious situation. We have found two

values, `one` and `two`, such that `one` and `two` do not have equal values, but neither is less than the other. This means that we could not possibly sort a list of elements containing both `one` and `two`, since neither one precedes the other.

The problem with the above implementation of `operator<` is that it violates trichotomy. Recall from the above definition of a total ordering that trichotomy means that exactly one of $a < b$, $a = b$, $a > b$ must hold for any a and b. Our definition of `operator<` does not have this property, as illustrated above. Consequently, we have a legal implementation of `operator<` that is wholly incorrect. We'll need to redefine how `operator<` works in order to ensure that trichotomy holds.

One common strategy for implementing `operator<` is to use what's called a *lexicographical ordering*. To illustrate a lexicographically ordering, consider the words **about** and **above** and think about how you would compare them alphabetically. You'd begin by noting that the first letter of each word was the same, as was the second and the third. However, the fourth letter of the words disagree, and in particular the letter **u** from **about** precedes the letter **v** from **above**. Consequently, we would say that **about** comes lexicographically before **above**. Interestingly, though, the last letter of **about** (**t**) comes after the last letter of **above** (**e**). We don't care, though, because we stopped comparing letters as soon as we found the first mismatch in the words.

This strategy has an elegant analog for arbitrary types. Given a type, one way to implement `operator<` is as follows. Given two objects a and b of that type, check whether the first field of a and b are not the same. If so, say that a is smaller if its first field is smaller than b's first field. Otherwise, look at the second field. If the fields are not the same, then return a if a's second field is smaller than b's and b otherwise. If not, then look at the third field, etc. To give you a concrete example of how this works, consider the following revision to the `Point`'s `operator<` function:

```
bool operator< (const Point& one, const Point& two) {
    if (one.getX() != two.getX()) return one.getX() < two.getX();
    return one.getY() < two.getY();
}
```

Here, we first check whether the points disagree in their x coordinate. If so, we say that `one` is less than `two` only if it has a smaller x coordinate. Otherwise, if the points agree in their x coordinate, then whichever has the lower y coordinate is said to have the smaller value. Amazingly, this implementation strategy results in an ordering that is both trichotic and transitive, exactly the properties we want out of the < operator.

Of course, this strategy works on classes that have more than two fields, provided that you compare each field one at a time. It is an interesting exercise to convince yourself that a lexicographical ordering on any type obeys trichotomy, and that such an ordering obeys transitivity as well.

Once you have a working implementation of `operator<`, it is possible to define all five other relational operators solely in terms of the `operator<`. This is due to the following set of relations:

```
 A <  B  □  A < B
A <= B  □  !(B < A)
A == B  □  !(A < B || B < A)
A != B  □  A < B || B < A
A >= B  □  !(A < B)
 A >  B  □  B < A
```

For example, we could implement `operator>` for the `Point` class as

```
bool operator> (const Point& one, const Point& two) {
    return two < one;
}
```

We could similarly implement `operator<=` for `Point`s as

```
bool operator<= (const Point& one, const Point& two) {
    return !(one < two);
}
```

This is a fairly standard technique, and it's well worth the effort to remember it.

Storing Objects in STL `map`s

Up to this point we've avoided storing objects as keys in STL `map`s. Now that we've covered operator overloading, though, you have the necessary knowledge to store objects in the STL `map` and `set` containers.

Internally, the STL `map` and `set` are layered on binary trees that use the relational operators to compare elements. Due to some clever design decisions, STL containers and algorithms only require the `<` operator to compare two objects. Thus, to store a custom class inside a `map` or `set`, you simply need to overload the `<` operator and the STL will handle the rest. For example, here's some code to store a `Point struct` in a `map`:

```
struct pointT {
    int x, y;

    bool operator < (const pointT& other) const {
        if(x != other.x)
            return x < other.x;
        return y < other.y;
    }
};
map<pointT, int> myMap; // Now works using the default < operator.
```

You can use a similar trick to store objects as values in a `set`.

`friend`

Normally, when you mark a class's data members private, only instances of that class are allowed to access them. However, in some cases you might want to allow specific other classes or functions to modify private data. For example, if you were implementing the STL `map` and wanted to provide an iterator class to traverse it, you'd want that iterator to have access to the `map`'s underlying binary tree. There's a slight problem here, though. Although the iterator is an integral component of the `map`, like all other classes, the iterator cannot access private data and thus cannot traverse the tree.

How are we to resolve this problem? Your initial thought might be to make some public accessor methods that would let the iterator modify the object's internal data representation. Unfortunately, this won't work particularly well, since then *any* class would be allowed to use those functions, something that violates the principle of encapsulation. Instead, to solve this problem, we can use the C++ `friend` keyword to grant the iterator class access to the `map` or `set`'s internals. Inside the `map` declaration, we can write the following:

```
template <typename KeyType, typename ValueType> class map {
public:
    /* ... */

    friend class iterator;
    class iterator {
        /* ... iterator implementation here ... */
    };
};
```

Now, since `iterator` is a `friend` of `map`, it can read and modify the `map`'s private data members.

Just as we can grant other classes `friend` access to a class, we can give `friend` access to global functions. For example, if we had a free function `ModifyMyClass` that accepted a `MyClass` object as a reference parameter, we could let `ModifyMyClass` modify the internal data of `MyClass` if inside the `MyClass` declaration we added the line

```
class MyClass {
public:
    /* ... */
    friend void ModifyMyClass(MyClass& param);

};
```

The syntax for `friend` can be misleading. Even though we're prototyping `ModifyMyClass` inside the `MyClass` function, because `ModifyMyClass` is a `friend` of `MyClass` it is **not** a member function of `MyClass`. After all, the purpose of the `friend` declaration is to give outside classes and functions access to the `MyClass` internals.

When using `friend`, there are two key points to be aware of. First, the `friend` declaration must precede the actual implementation of the `friend` class or function. Since C++ compilers only make a single pass over the source file, if they haven't seen a `friend` declaration for a function or class, when the function or class tries to modify your object's internals, the compiler will generate an error. Second, note that while `friend` is quite useful in some circumstances, it can quickly lead to code that entirely defeats the purpose of encapsulation. Before you grant `friend` access to a piece of code, make sure that the code has a legitimate reason to be modifying your object. That is, don't make code a `friend` simply because it's easier to write that way. Think of `friend` as a way of extending a class definition to include other pieces of code. The class, together with all its `friend` code, should comprise a logical unit of encapsulation.

When overloading an operator as a free function, you might want to consider giving that function `friend` access to your class. That way, the functions can efficiently read your object's private data without having to go through getters and setters.

Unfortunately, `friend` does not interact particularly intuitively with template classes. Suppose we want to provide a `friend` function `PQueueFriend` for a template version of the CS106B/X `PQueue`. If `PQueueFriend` is declared like this:

```
template <typename T> void PQueueFriend(const PQueue<T>& pq) {
   /* ... */
}
```

You'll notice that `PQueueFriend` itself is a template function. This means that when declaring `PQueueFriend` a `friend` of the template `PQueue`, we'll need to make the `friend` declaration templatized, as shown here:

```
template <typename T> class PQueue {
public:
    /* ... */
    template <typename T> friend PQueueFriend(const PQueue<T>& pq);
};
```

If you forget the `template` declaration, then your code will compile correctly but will generate a linker error. While this can be a bit of nuisance, it's important to remember since it arises frequently when overloading the stream operators, as you'll see below.

Overloading the Stream Insertion Operator

Have you ever wondered why `cout << "Hello, world!" << endl` is syntactically legal? It's through the overloaded << operator in conjunction with `ostream`s.* In fact, the entire streams library can be thought of as a gigantic library of massively overloaded << and >> operators.

The C++ streams library is designed to give you maximum flexibility with your input and output routines and even lets you define your own stream insertion and extraction operators. This means that you are allowed to define the << and >> operators so that expressions like `cout << myClass << endl` and `cin >> myClass` are well-defined. However, when writing stream insertion and extraction operators, there are huge number of considerations to keep in mind, many of which are beyond the scope of this text. This next section will discuss basic strategies for overloading the << operator, along with some limitations of the simple approach.

As with all overloaded operators, we need to consider what the parameters and return type should be for our overloaded << operator. Before considering parameters, let's think of the return type. We know that it should be legal to chain stream insertions together – that is, code like `cout << 1 << 2 << endl` should compile correctly. The << operator associates to the left, so the above code is equal to

```
(((cout << 1) << 2) << endl);
```

Thus, we need the << operator to return an `ostream`. Now, we don't want this stream to be `const`, since then we couldn't write code like this:

```
cout << "This is a string!" << setw(10) << endl;
```

Since if `cout << "This is a string!"` evaluated to a `const` object, we couldn't set the width of the next operation to 10. Also, we cannot return the stream by value, since stream classes have their copy functions marked private. Putting these two things together, we see that the stream operators should return a non-`const` reference to whatever stream they're referencing.

Now let's consider what parameters we need. We need to know what stream we want to write to or read from, so initially you might think that we'd define overloaded stream operators as member functions that look like this:

```
class MyClass {
public:
   ostream& operator << (ostream& input) const; // Problem: Legal but incorrect
};
```

* As a reminder, the `ostream` class is the base class for output streams. This has to do with inheritance, which we'll cover in a later chapter, but for now just realize that it means that both `stringstream` and `ofstream` are specializations of the more generic `ostream` class.

Unfortunately, this isn't correct. Consider the following two code snippets:

```
cout << myClass;
myClass << cout;
```

The first of these two versions makes sense, while the second is backwards. Unfortunately, with the above definition of `operator <<`, we've accidentally made the second version syntactically legal. The reason is that these two lines expand into calls to

```
cout.operator <<(myClass);
myClass.operator <<(cout);
```

The first of these two isn't defined, since `cout` doesn't have a member function capable of writing our object (if it did, we wouldn't need to write a stream operator in the first place!). However, based on our previous definition, the second version, while semantically incorrect, is syntactically legal. Somehow we need to change how we define the stream operator so that we are allowed to write `cout << myClass`. To fix this, we'll make the overloaded stream operator a free function that takes two parameters – an `ostream` to write to and a `myClass` object to write. The code for this is:

```
ostream& operator << (ostream& stream, const MyClass& mc) {
    /* ... implementation ... */
    return stream;
}
```

While this code will work correctly, because `operator <<` is a free function, it doesn't have access to any of the private data members of `MyClass`. This can be a nuisance, since we'd like to directly write the contents of `MyClass` out to the stream without having to go through the (possibly inefficient) getters and setters. Thus, we'll declare `operator <<` a `friend` inside the `MyClass` declaration, as shown here:

```
class MyClass {
public:
    /* More functions. */
    friend ostream& operator <<(ostream& stream, const MyClass& mc);

};
```

Now, we're all set to do reading and writing inside the body of the insertion operator. It's not particularly difficult to write the stream insertion operator – all that we need to do is print out all of the meaningful class information with some formatting information. So, for example, given a `Point` class representing a point in 2-D space, we could write the insertion operator as

```
ostream& operator <<(ostream& stream, const Point& pt) {
    stream << '(' << pt.x << ", " << pt.y << ')';
    return stream;
}
```

While this code will work in most cases, there are a few spots where it just won't work correctly. For example, suppose we write the following code:

```
cout << "01234567890123456789" << endl; // To see the number of characters.
cout << setw(20) << myPoint << endl;
```

Looking at this code, you'd expect that it would cause `myPoint` to be printed out and padded with space characters until it is at least twenty characters wide. Unfortunately, this isn't what happens. Since `operator <<` writes the object one piece at a time, the output will look something like this:

```
01234567890123456789
                   (0, 4)
```

That's nineteen spaces, followed by the actual `Point` data. The problem is that when we invoke `operator <<`, the function writes a single `(` character to `stream`. It's this operation, *not the `Point` as a whole*, that will get aligned to 20 characters. There are many ways to circumvent this problem, but perhaps the simplest is to buffer the output into a `stringstream` and then write the contents of the `stringstream` to the destination in a single operation. This can get a bit complicated, especially since you'll need to copy the stream formatting information over.

Writing a correct stream extraction operator (`operator >>`) is complicated. For more information on writing stream extraction operators, consult a reference.

Overloading * and ->

Consider the following code snippet:

```
for(set<string>::iterator itr = mySet.begin(); itr != mySet.end(); ++itr)
    cout << *itr << " has length " << itr->length() << endl;
```

Here, we traverse a `set<string>` using iterators, printing out each string and its length. Interestingly, even though `set` iterators are not raw pointers (they're objects capable of traversing binary trees), thanks to operator overloading, they can respond to the `*` and `->` operators as though they were regular C++ pointers.

If you create a custom class that acts like a C++ pointer (perhaps a custom iterator or "smart pointer," a topic we'll return to later), you can provide implementations of the pointer dereference and member selection operators `*` and `->` by overloading their respective operator functions. The simpler of these two functions is the pointer dereference operator. To make an object that can be dereferenced to yield an object of type `T`, the syntax for its `*` operator is

```
class PointerClass {
public:
    T& operator *() const;
    /* ... */
};
```

You can invoke the `operator *` function by "dereferencing" the custom pointer object. For example, the following code:

```
*myCustomPointer = 137;
```

is completely equivalent to

```
myCustomPointer.operator *() = 137;
```

Because we can assign a value to the result of `operator *`, the `operator *` function should return an lvalue (a non-`const` reference).

There are two other points worth noting here. First, how can C++ distinguish this `operator *` for pointer dereference from the `operator *` used for multiplication? The answer has to do with the number of parameters to the function. Since a pointer dereference is a unary operator, the function prototype for the pointer-dereferencing `operator *` takes no parameters. Had we wanted to write `operator *` for

multiplication, we would have written a function `operator *` that accepts a parameter (or a free function accepting two parameters). Second, why is `operator *` marked `const`? This has to do with the difference between `const` pointers and pointers-to-`const`. Suppose that we have a `const` instance of a custom pointer class. Since the pointer *object* is `const`, it acts as though it is a `const` pointer rather than a pointer-to-`const`. Consequently, we should be able to dereference the object and modify its stored pointer without affecting its `const`ness.

The arrow operator `operator ->` is slightly more complicated than `operator *`. Initially, you might think that `operator ->` would be a binary operator, since you use the arrow operator in statements like `myClassPtr->myElement`. However, C++ has a rather clever mechanism for `operator ->` that makes it a unary operator. A class's `operator ->` function should return a pointer to the object that the arrow operator should actually be applied to. This may be a bit confusing, so an example is in order. Suppose we have a class `CustomStringPointer` that acts as though it's a pointer to a C++ `string` object. Then if we have the following code:

```
CustomStringPointer myCustomPointer;
cout << myCustomPointer->length() << endl;
```

This code is equivalent to

```
CustomStringPointer myCustomPointer;
cout << (myCustomPointer.operator ->())->length() << endl;
```

In the first version of the code, we treated the `myCustomPointer` object as though it was a real pointer by using the arrow operator to select the `length` function. This code expands out into two smaller steps:

1. The `CustomStringPointer`'s `operator ->` function is called to determine which pointer the arrow should be applied to.

2. The returned pointer then has the `->` operator applied to select the `length` function.

Thus when writing the `operator ->` function, you simply need to return the pointer that the arrow operator should be applied to. If you're writing a custom iterator class, for example, this is probably the element being iterator over.

We'll explore one example of overloading these operators in a later chapter.

List of Overloadable Operators

The following table lists the most commonly-used operators you're legally allowed to overload in C++, along with any restrictions about how you should define the operator.

Operator	Yields	Usage
`=`	Lvalue	`MyClass& operator =(const MyClass& other);` See the the earlier chapter for details.
`+= -= *= /= %=` (etc.)	Lvalue	`MyClass& operator +=(const MyClass& other);` When writing compound assignment operators, make sure that you correctly handle "self-compound-assignment."

`+ - * / %`
 	
(etc.)
	Rvalue	`const MyClass operator + (const MyClass& one,`
		`  const MyClass& two);`
		
		These operator should be defined as a free functions.

`< <= ==`
`> >= !=`
	Rvalue	`bool operator < (const MyClass& other) const;`
		`bool operator < (const MyClass& one,`
		`  const MyClass& two);`
		
		If you're planning to use relational operators only for the STL container classes, you just need to overload the < operator. Otherwise, you should overload all six so that users aren't surprised that `one != two` is illegal while `!(one == two)` is defined.

`[]`	Lvalue	`  ElemType& operator [](const KeyType& key);`
		`const ElemType& operator [](const KeyType& key) const;`
		
		Most of the time you'll need a `const`-overloaded version of the bracket operator. Forgetting to provide one can lead to a real headache!

`++ --`	Prefix: Lvalue Postfix: Rvalue	Prefix version: `MyClass& operator ++();` Postfix version: `const MyClass operator ++(int dummy);`
`-`	Rvalue	`const MyClass operator -() const;`
`*`	Lvalue	`ElemType& operator *() const;` With this function, you're allowing your class to act as though it's a pointer. The return type should be a reference to the object it's "pointing" at. This is how the STL iterators and smart pointers work. Note that this is the unary `*` operator and is not the same as the `*` multiplicative operator.
`->`	Lvalue	`ElemType* operator ->() const;` If the `->` is overloaded for a class, whenever you write `myClass->myMember`, it's equivalent to `myClass.operator ->()->myMember`. Note that the function should be `const` even though the object returned can still modify data. This has to do with how pointers can legally be used in C++. For more information, refer to the chapter on `const`.

`<< >>`	Lvalue	`friend ostream& operator << (ostream& out,`
		`  const MyClass& mc);`
		`friend istream& operator >> (istream& in,`
		`  MyClass& mc);`

`()`	Varies	See the chapter on functors.

Extended Example: `grid`

The STL encompasses a wide selection of associative and sequence containers. However, one useful data type that did not find its way into the STL is a multidimensional array class akin to the CS106B/X `Grid`. In this extended example, we will implement an STL-friendly version of the CS106B/X `Grid` class, which we'll call `grid`, that will support STL-compatible iterators, intuitive element-access syntax, and relational operators. Once we're done, we'll have an industrial-strength container class we will use later in this book to implement more complex examples.

Implementing a fully-functional `grid` may seem daunting at first, but fortunately it's easy to break the work up into several smaller steps that culminate in a working class.

Step 0: Implement the Basic `grid` Class.

Before diving into some of the `grid`'s more advanced features, we'll begin by implementing the `grid` basics. Below is a partial specification of the `grid` class that provides core functionality:

Figure 0: Basic (incomplete) interface for the `grid` class

```
template <typename T> class grid {
public:
    /* Constructors, destructors. */
    grid();                                 // Create empty grid
    grid(size_t rows, size_t cols);         // Construct to specified size

    /* Resizing operations. */
    void clear();                           // Empty the grid
    void resize(size_t rows, size_t cols); // Resize the grid

    /* Query operations. */
    size_t numRows() const;                // Returns number of rows in the grid
    size_t numCols() const;                // Returns number of columns in the grid
    bool empty() const;                    // Returns whether the grid is empty
    size_t size() const;                   // Returns the number of elements

    /* Element access. */
          T& getAt(size_t row, int col);    // Access individual elements
    const T& getAt(int row, int col) const; // Same, but const
};
```

These functions are defined in greater detail here:

`grid();`	Constructs a new, empty `grid`.
`grid(size_t rows, size_t cols);`	Constructs a new `grid` with the specified number of rows and columns. Each element in the `grid` is initialized to its default value.
`void clear();`	Resizes the `grid` to 0x0.
`void resize(size_t rows, size_t cols);`	Discards the current contents of the `grid` and resizes the `grid` to the specified size. Each element in the `grid` is initialized to its default value.
`size_t numRows() const;` `size_t numCols() const;`	Returns the number of rows and columns in the `grid`.
`bool empty() const;`	Returns whether the `grid` contains no elements. This is true if either the number of rows or columns is zero.
`size_t size() const;`	Returns the number of total elements in the `grid`.
`T& getAt(size_t row, size_t col);` `const T& getAt(size_t row, size_t col) const;`	Returns a reference to the element at the specified position. This function is `const`-overloaded. We won't worry about the case where the indices are out of bounds.

Because `grid`s can be dynamically resized, we will need to back `grid` with some sort of dynamic memory management. Because the `grid` represents a two-dimensional entity, you might think that we need to use

a dynamically-allocated multidimensional array to store `grid` elements. However, working with dynamically-allocated multidimensional arrays is tricky and greatly complicates the implementation. Fortunately, we can sidestep this problem by implementing the two-dimensional `grid` object using a single-dimensional array. To see how this works, consider the following 3x3 grid:

0	1	2
3	4	5
6	7	8

We can represent all of the elements in this grid using a one-dimensional array by laying out all of the elements sequentially, as seen here:

0	1	2	3	4	5	6	7	8

If you'll notice, in this ordering, the three elements of the first row appear in order as the first three elements, then the three elements of the second row in order, and finally the three elements of the final row in order. Because this one-dimensional representation of a two-dimensional object preserves the ordering of individual rows, it is sometimes referred to as *row-major order*.

To represent a grid in row-major order, we need to be able to convert between grid coordinates and array indices. Given a coordinate (*row*, *col*) in a grid of dimensions (*nrows*, *ncols*), the corresponding position in the row-major order representation of that grid is given by *index* = *col* + *row* * *ncols*. The intuition behind this formula is that because the ordering within any row is preserved, each horizontal step in the grid translates into a single step forward or backward in the row-major order representation of the grid. However, each vertical step in the grid requires us to advance forward to the next row in the linearized grid, skipping over *ncols* elements.

Using row-major order, we can back the `grid` class with a regular STL `vector`, as shown here:

```
template <typename T> class grid {
public:
    grid();
    grid(size_t rows, size_t cols);

    void clear();
    void resize(size_t rows, size_t cols);

    size_t numRows() const;
    size_t numCols() const;
    bool empty() const;
    size_t size() const;

          T& getAt(size_t row, size_t col);
    const T& getAt(size_t row, size_t col) const;

private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};
```

Serendipitously, implementing the `grid` with a `vector` allows us to use C++'s automatically-generated copy constructor and assignment operator for `grid`. Since `vector` already manages its own memory, we don't need to handle it manually.

Note that we explicitly keep track of the number of rows and columns in the `grid` even though the `vector` stores the total number of elements. This is necessary so that we can compute indices in the row-major ordering for points in two-dimensional space.

The above functions have relatively straightforward implementations that are given below:

```
template <typename T> grid<T>::grid() : rows(0), cols(0) {

}

template <typename T>
grid<T>::grid(size_t rows, size_t cols)
  : elems(rows * cols), rows(rows), cols(cols) {
}

template <typename T> void grid<T>::clear() {
    resize(0, 0);
}

template <typename T> void grid<T>::resize(size_t rows, size_t cols) {
    /* See below for assign */
    elems.assign(rows * cols, ElemType());

    /* Explicit this-> required since parameters have same name as members. */
    this->rows = rows;
    this->cols = cols;
}

template <typename T> size_t grid<T>::numRows() const {
    return rows;
}

template <typename T> size_t grid<T>::numCols() const {
    return cols;
}

template <typename T> bool grid<T>::empty() const {
    return size() == 0;
}

template <typename T> size_t grid<T>::size() const {
    return numRows() * numCols();
}

/* Use row-major ordering to access the proper element of the vector. */
template <typename T> T& grid<T>::getAt(size_t row, size_t col) {
    return elems[col + row * cols];
}
template <typename T> const T& grid<T>::getAt(size_t row,size_t col) const {
    return elems[col + row * cols];
}
```

Most of these functions are one-liners and are explained in the comments. The only function that you may find interesting is `resize`, which uses the `vector`'s `assign` member function. `assign` is similar to `resize` in that it changes the size of the `vector`, but unlike `resize` `assign` discards all of the current `vector` contents and replaces them with the specified number of copies of the specified element. The use of `ElemType()` as the second parameter to `assign` means that we will fill the `vector` with copies of the default value of the type being stored (since `ElemType()` uses the temporary object syntax to create a new `ElemType`).

Step 1: Add Support for Iterators

Now that we have the basics of a `grid` class, it's time to add iterator support. This will allow us to plug the `grid` directly into the STL algorithms and will be invaluable in a later chapter.

Like the `map` and `set`, the `grid` does not naturally lend itself to a linear traversal – after all, `grid` is two-dimensional – and so we must arbitrarily choose an order in which to visit elements. Since we've implemented the `grid` in row-major order, we'll have `grid` iterators traverse the grid row-by-row, top to bottom, from left to right. Thus, given a 3x4 `grid`, the order of the traversal would be

0	1	2
3	4	5
6	7	8
9	10	11

This order of iteration maps naturally onto the row-major ordering we've chosen for the `grid`. If we consider how the above grid would be laid out in row-major order, the resulting array would look like this:

0	1	2	3	4	5	6	7	8	9	10	11

Thus this iteration scheme maps to a simple linear traversal of the underlying representation of the `grid`. Because we've chosen to represent the elements of the `grid` using a `vector`, we can iterate over the elements of the `grid` using `vector` iterators. We thus add the following definitions to the `grid` class:

```
template <typename T> class grid {
public:
    grid();
    grid(size_t rows, size_t cols);

    void clear();
    void resize(size_t rows, size_t cols);

    size_t numRows() const;
    size_t numCols() const;
    bool empty() const;
    size_t size() const;

    T& getAt(size_t row, size_t col);
    const T& getAt(size_t row, size_t col) const;

    typedef typename vector<T>::iterator iterator;
    typedef typename vector<T>::const_iterator const_iterator;

private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};
```

Now, clients of `grid` can create `grid<int>::iterator`s rather than `vector<int>::iterator`s. This makes the interface more intuitive and increases encapsulation; since `iterator` is a `typedef`, if we later decide to replace the underlying representation with a dynamically-allocated array, we can change the `typedef`s to

```
typedef ElemType* iterator;
typedef const ElemType* const_iterator;
```

And clients of the `grid` will not notice any difference.

Notice that in the above `typedef`s we had to use the `typename` keyword to name the type `vector<ElemType>::iterator`. This is the pesky edge case mentioned in the chapter on templates and somehow manages to creep into more than its fair share of code. Since `iterator` is a nested type inside the template type `vector<ElemType>`, we have to use the `typename` keyword to indicate that `iterator` is the name of a type rather than a class constant.

We've now defined an `iterator` type for our `grid`, so what functions should we export to the `grid` clients? We'll at least want to provide support for `begin` and `end`, as shown here:

```
template <typename T> class grid {
public:
    grid();
    grid(size_t rows, size_t cols);

    void clear();
    void resize(size_t rows, size_t cols);

    size_t numRows() const;
    size_t numCols() const;
    bool empty() const;
    size_t size() const;

    T& getAt(size_t row, size_t col);
    const T& getAt(size_t row, size_t col) const;

    typedef typename vector<T>::iterator iterator;
    typedef typename vector<T>::const_iterator const_iterator;

          iterator begin();
    const_iterator begin() const;
          iterator end();
    const_iterator end() const;
private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};
```

We've provided two versions of each function so that clients of a `const grid` can still use iterators. These functions are easily implemented by returning the value of the underlying `vector`'s `begin` and `end` functions, as shown here:

```
template <typename T> typename grid<T>::iterator grid<T>::begin() {
    return elems.begin();
}
```

Notice that the return type of this function is `typename grid<ElemType>::iterator` rather than just `iterator`. Because `iterator` is a nested type inside `grid`, we need to use `grid<ElemType>::iterator` to specify which iterator we want, and since `grid` is a template type we have to use the `typename` keyword to indicate that `iterator` is a nested type. Otherwise, this function should be straightforward.

The rest of the functions are implemented here:

```
template <typename T> typename grid<T>::const_iterator grid<T>::begin() const {
    return elems.begin();
}

template <typename T> typename grid<T>::iterator grid<T>::end() {
    return elems.end();
}

template <typename T> typename grid<T>::const_iterator grid<T>::end() const {
    return elems.end();
}
```

Because the `grid` is implemented in row-major order, elements of a single row occupy consecutive locations in the `vector`. It's therefore possible to return iterators delineating the start and end of each row in the `grid`. This is useful functionality, so we'll provide it to clients of the `grid` through a pair of member functions `row_begin` and `row_end` (plus `const` overloads). These functions are declared here:

```
template <typename T> class grid {
public:
    grid();
    grid(size_t rows, size_t cols);

    void clear();
    void resize(size_t rows, size_t cols);

    size_t numRows() const;
    size_t numCols() const;
    bool empty() const;
    size_t size() const;

    T& getAt(size_t row, size_t col);
    const T& getAt(size_t row, size_t col) const;

    typedef typename vector<ElemType>::iterator iterator;
    typedef typename vector<ElemType>::const_iterator const_iterator;

          iterator begin();
    const_iterator begin() const;
          iterator end();
    const_iterator end() const;

          iterator row_begin(size_t row);
    const_iterator row_begin(size_t row) const;
          iterator row_end(size_t row);
    const_iterator row_end(size_t row) const;

private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};
```

Before implementing these functions, let's take a minute to figure out exactly where the iterations we return should point to. Recall that the element at position (*row*, 0) in a grid of size (*rows*, *cols*) can be found at position *row* * *cols*. We should therefore have `row_begin(row)` return an iterator to the `row` * *cols* element of the `vector`. Since there are *cols* elements in each row and `row_end` should return an iterator to one position past the end of the row, this function should return an iterator to the position *cols* past the location returned by `row_begin`. Given this information, we can implement these functions as shown here:

```
template <typename T> typename grid<T>::iterator grid<T>::row_begin(int row) {
    return begin() + numCols() * row;
}

template <typename T>
    typename grid<T>::const_iterator grid<T>::row_begin(int row) const {
    return begin() + numCols() * row;
}
```

```
template <typename T> typename grid<T>::iterator grid<T>::row_end(int row) {
    return row_begin(row) + numCols();
}

template <typename T>
    typename grid<T>::const_iterator grid<T>::row_end(int row) const {
    return row_begin(row) + numCols();
}
```

We now have an elegant `iterator` interface for the `grid` class. We can iterate over the entire container as a whole, just one row at a time, or some combination thereof. This enables us to interface the `grid` with the STL algorithms. For example, to zero out a `grid<int>`, we can use the `fill` algorithm, as shown here:

```
fill(myGrid.begin(), myGrid.end(), 0);
```

We can also sort the elements of a row using `sort`:

```
sort(myGrid.row_begin(0), myGrid.row_end(0));
```

With only a handful of functions we're now capable of plugging directly into the full power of the algorithms. This is part of the beauty of the STL – had the algorithms been designed to work on containers rather than iterator ranges, this would not have been possible.

Step 2: Add Support for the Element Selection Operator

When using regular C++ multidimensional arrays, we can write code that looks like this:

```
int myArray[137][42];
myArray[2][4] = 271828;
myArray[9][0] = 314159;
```

However, with the current specification of the `grid` class, the above code would be illegal if we replaced the multidimensional array with a `grid<int>`, since we haven't provided an implementation of `operator []`.

Adding support for element selection to linear classes like the `vector` is simple – we simply have the brackets operator return a reference to the proper element. Unfortunately, it is much trickier to design `grid` such that the bracket syntax works correctly. The reason is that if we write code that looks like this:

```
grid<int> myGrid(137, 42);
int value = myGrid[2][4];
```

By replacing the bracket syntax with calls to `operator []`, we see that this code expands out to

```
grid<int> myGrid(137, 42);
int value = (myGrid.operator[] (2)).operator[] (4);
```

Here, there are *two* calls to `operator []`, one invoked on `myGrid` and the other on the value returned by `myGrid.operator[](2)`. To make the above code compile, the object returned by the `grid`'s `operator[]` must *itself* define an `operator []` function. It is this returned object, rather than the `grid` itself, which is responsible for retrieving the requested element from the `grid`. Since this temporary object is used to perform a task normally reserved for the `grid`, it is sometimes known as a *proxy object*.

How can we implement the `grid`'s `operator []` so that it works as described above? First, we will need to define a new class representing the object returned by the `grid`'s `operator []`. In this discussion, we'll call it `MutableReference`, since it represents an object that can call back into the `grid` and mutate it. For simplicity and to maximize encapsulation, we'll define `MutableReference` inside of `grid`. This results in the following interface for `grid`:

```
template <typename T> class grid {
public:
    /* ... previously-defined functions ... */

    class MutableReference {
    public:
        friend class grid;
        T& operator[] (size_t col);

    private:
        MutableReference(grid* owner, size_t row);

        grid* const owner;
        const size_t row;
    };

private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};
```

The `MutableReference` object stores some a pointer to the `grid` that created it, along with the index passed in to the `grid`'s `operator []` function when the `MutableReference` was created. That way, when we invoke the `MutableReference`'s `operator []` function specifying the *col* coordinate of the grid, we can pair it with the stored *row* coordinate, then query the `grid` for the element at (*row*, *col*). We have also made `grid` a `friend` of `MutableReference` so that the `grid` can call the private constructor necessary to initialize a `MutableReference`.

We can implement `MutableReference` as follows:

```
template <typename T>
    grid<T>::MutableReference::MutableReference(grid* owner, int row) :
       owner(owner), row(row) {
}

template <typename T>
    T& grid<T>::MutableReference::operator[] (int col) {
    return owner->getAt(row, col);
}
```

Notice that because `MutableReference` is a nested class inside `grid`, the implementation of the `MutableReference` functions is prefaced with `grid<ElemType>::MutableReference` instead of just `MutableReference`. However, in this particular case the pesky `typename` keyword is not necessary because we are prototyping a function inside `MutableReference` rather than using the type `MutableReference` in an expression.

Now that we've implemented `MutableReference`, we'll define an `operator []` function for the `grid` class that constructs and returns a properly-initialized `MutableReference`. This function accepts an *row*

coordinate, and returns a `MutableReference` storing that row number and a pointer back to the `grid`. That way, if we write

```
int value = myGrid[1][2];
```

The following sequences of actions occurs:

1. `myGrid.operator[]` is invoked with the parameter 1.
2. `myGrid.operator[]` creates a `MutableReference` storing the *row* coordinate 1 and a means for communicating back with the `myGrid` object.
3. `myGrid.operator[]` returns this `MutableReference`.
4. The returned `MutableReference` then has its `operator[]` function called with parameter 2.
5. The returned `MutableReference` then calls back to the `myGrid` object and asks for the element at position (1, 2).

This sequence of actions is admittedly complex, but is transparent to the client of the `grid` class and runs efficiently.

`operator[]` is defined and implemented as follows:

```
template <typename T> class grid {
public:
    /* ... previously-defined functions ... */

    class MutableReference {
    public:
        friend class grid;
        T& operator[] (size_t col);

    private:
        MutableReference(grid* owner, size_t row);

        grid* const owner;
        const size_t row;
    };
    MutableReference operator[] (int row);

private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};

template <typename T>
    typename grid<T>::MutableReference grid<T>::operator[] (int row) {
    return MutableReference(this, row);
}
```

Notice that we've only provided an implementation of the non-`const` version of `operator[]`. But what if we want to use `operator[]` on a `const grid`? We would similarly need to return a proxy object, but that object would need to guarantee that `grid` clients could not write code like this:

```
const grid<int> myGrid(137, 42);
myGrid[0][0] = 2718; // Ooops!  Modified const object!
```

To prevent this sort of problem, we'll have the `const` version of `operator[]` return a proxy object of a different type, called `ImmutableReference` which behaves similarly to `MutableReference` but which returns `const` references to the elements in the `grid`. This results in the following interface for `grid`:

```
template <typename T> class grid {
public:
    /* ... previously-defined functions ... */

    class MutableReference {
    public:
        friend class grid;
        T& operator[] (size_t col);

    private:
        MutableReference(grid* owner, size_t row);

        grid* const owner;
        const size_t row;
    };
    MutableReference operator[] (int row);

    class ImmutableReference {
    public:
        friend class grid;
        const T& operator[] (size_t col) const;

    private:
        MutableReference(const grid* owner, size_t row);

        const grid* const owner;
        const size_t row;
    };
    ImmutableReference operator[] (size_t row) const;

private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};
```

`ImmutableReference` and the `const` version of `operator[]` are similar to `MutableReference` and the non-`const` version of `operator[]`, and to save space we won't write it here. The complete listing of the `grid` class at the end of this chapter contains the implementation if you're interested.

Step 3: Define Relational Operators

Now that our `grid` has full support for iterators and a nice bracket syntax that lets us access individual elements, it's time to put on the finishing touches. As a final step in the project, we'll provide implementations of the relational operators for our `grid` class. We begin by updating the `grid` interface to include the following functions:

```
template <typename T> class grid {
public:
    /* ... previously-defined functions ... */

    bool operator <  (const grid& other) const;
    bool operator <= (const grid& other) const;
    bool operator == (const grid& other) const;
    bool operator != (const grid& other) const;
    bool operator >= (const grid& other) const;
    bool operator >  (const grid& other) const;

private:
    vector<T> elems;
    size_t rows;
    size_t cols;
};
```

Note that of the six operators listed above, only the `==` and `!=` operators have intuitive meanings when applied to `grid`s. However, it also makes sense to define a `<` operator over `grid`s so that we can store them in STL `map` and `set` containers, and to ensure consistency, we should define the other three operators as well.

Because there is no natural interpretation for what it means for one `grid` to be "less than" another, we are free to implement these functions in any way that we see fit, provided that we obey transitivity and trichotomy. As mentioned earlier it is possible to implement all six of the relational operators in terms of the less-than operator. One strategy for implementing the relational operators is thus to implement just the less-than operator and then to define the other five as wrapped calls to `operator <`. But what is the best way to determine whether one `grid` compares less than another? One general approach is to define a *lexicographical ordering* over `grid`s. We will compare each field one at a time, checking to see if the fields are equal. If so, we move on to the next field. Otherwise, we immediately return that one `grid` is less than another without looking at the remaining fields. If we go through every field and find that the `grid`s are equal, then we can return that neither grid is less than the other. This is similar to the way that we might order words alphabetically – we find the first mismatched character, then return which word compares first. We can begin by implementing `operator <` as follows:

```
template <typename T> bool grid<T>::operator < (const grid& other) const {
    /* Compare the number of rows and return if there's a mismatch. */
    if(rows != other.rows)
        return rows < other.rows;

    /* Next compare the number of columns the same way. */
    if(cols != other.cols)
        return cols < other.cols;

    /* ... */
}
```

Here, we compare the `rows` fields of the two objects and immediately return if they aren't equal. We can then check the `cols` fields in the same way. Finally, if the two `grid`s have the same number of rows and columns, we need to check how the elements of the `grid`s compare. Fortunately, this is straightforward thanks to the STL `lexicographical_compare` algorithm. `lexicographical_compare` accepts four iterators delineating two ranges, then lexicographically compares the elements in those ranges and returns if the first range compares lexicographically less than the second. Using `lexicographical_compare`, we can finish our implementation of `operator <` as follows:

```
template <typename T> bool grid<T>::operator < (const grid& other) const {
    /* Compare the number of rows and return if there's a mismatch. */
    if(rows != other.rows)
        return rows < other.rows;

    /* Next compare the number of columns the same way. */
    if(cols != other.cols)
        return cols < other.cols;

    return lexicographical_compare(begin(), end(), other.begin(), other.end());
}
```

All that's left to do now is to implement the other five relational operators in terms of `operator <`. This is done below:

```
template <typename T> bool grid<T>::operator >=(const grid& other) const {
    return !(*this < other);
}

template <typename T> bool grid<T>::operator ==(const grid& other) const {
    return !(*this < other) && !(other < *this);
}

template <typename T> bool grid<T>::operator !=(const grid& other) const {
    return (*this < other) || (other < *this);
}

template <typename T> bool grid<T>::operator > (const grid& other) const {
    return other < *this;
}

template <typename T> bool grid<T>::operator <=(const grid& other) const {
    return !(other < *this);
}
```

At this point we're done! We now have a complete working implementation of the `grid` class that supports iteration, element access, and the relational operators. To boot, it's implemented on top of the `vector`, meaning that it's slick and efficient. This class should be your one-stop solution for applications that require a two-dimensional array.

More to Explore

Operator overloading is an enormous topic in C++ and there's simply not enough space to cover it all in this chapter. If you're interested in some more advanced topics, consider reading into the following:

1. **Overloaded `new` and `delete`**: You are allowed to overload the `new` and `delete` operators, in case you want to change how memory is allocated for your class. Note that the overloaded `new` and `delete` operators simply change how memory is allocated, not what it means to write `new MyClass`. Overloading `new` and `delete` is a complicated task and requires a solid understanding of how C++ memory management works, so be sure to consult a reference for details.

2. **Conversion functions**: In an earlier chapter, we covered how to write conversion constructors, functions that convert objects of other types into instances of your new class. However, it's possible to use operator overloading to define an implicit conversion from objects of your class into objects of other types. The syntax is `operator Type()`, where `Type` is the data type to convert your object to. Many professional programmers advise against conversion functions, so make sure that they're really the best option before proceeding.

Practice Problems

Operator overloading is quite difficult because your functions must act as though they're the built-in operators. Here are some practice problems to get you used to overloading operators:

1. What is an overloaded operator?

2. What is an lvalue? An rvalue? Does the `+` operator yield an lvalue or an rvalue? How about the pointer dereference operator?

3. Are overloaded operators inherently more efficient than regular functions?

4. Explain how to implement `operator +` in terms of `operator +=`.

5. What is the signature of an overloaded operator for subtraction? For unary negation?

6. How do you differentiate between the prefix and postfix versions of the `++` operator?

7. What does the `->` operator return?

8. What is a friend function? How do you declare one?

9. How do you declare an overloaded stream insertion operator?

10. What is trichotomy and why is it important to C++ programmers?

11. What is transitivity and why is it important to C++ programmers?

12. In Python, it is legal to use negative array indices to mean "the element that many positions from the end of the array." For example, `myArray[-1]` would be the last element of an array, `myArray[-2]` the penultimate element, etc. Using operator overloading, it's possible to implement this functionality for a custom array class. Do you think it's a good idea to do so? Why or why not? Think about the principle of least astonishment when answering.

13. Why is it better to implement + in terms of += instead of += in terms of +? *(Hint: Think about the number of objects created using += and using +.)*

14. Consider the following definition of a `Span` struct:

```
struct Span {
    int start, stop;
};
```

    The `Span` struct allows us to define the range of elements from [`start`, `stop`) as a single variable. Given this definition of `Span` and assuming `start` and `stop` are both non-negative, provide another bracket operator for our `Vector` class that selects a range of elements.

15. Consider the following interface for a class that iterates over a container of `ElemType`s:

```
class iterator {
public:
    bool operator== (const iterator& other);
    bool operator!= (const iterator& other);

    iterator operator++ ();

    ElemType* operator* () const;
    ElemType* operator-> () const;
};
```

    There are several mistakes in the definition of this iterator. What are they? How would you fix them?

16. The implementation of the `grid` class's `operator==` and `operator!=` functions implemented those operators in terms of the less-than operator. This is somewhat inefficient, since it's more direct to simply check if the two `grid`s are equal or unequal rather than to use the comparison operators. Rewrite the `grid`'s `operator==` function to directly check whether the `grid`s are identical. Then rewrite `operator!=` in terms of `operator==`.

Chapter 11: Resource Management

This chapter is about two things – putting away your toys when you're done with them, and bringing enough of your toys for everyone to share. These are lessons you (hopefully!) learned in kindergarten which happen to pop up just about everywhere in life. We're supposed to clean up our own messes so that they don't accumulate and start to interfere with others, and try to avoid hogging things so that others don't hurt us by trying to take those nice things away from us.

The focus of this chapter is how to play nice with others when it comes to *resources*. In particular, we will explore two of C++'s most misunderstood language features, the *copy constructor* and the *assignment operator*. By the time you're done reading this chapter, you should have a much better understanding of how to manage resources in ways that will keep your programs running and your fellow programmers happy. The material in this chapter is somewhat dense, but fear not! We'll make sure to go over all of the important points neatly and methodically.

Consider the STL `vector`. Internally, `vector` is backed by a dynamically-allocated array whose size grows when additional space is needed for more elements. For example, a ten-element `vector` might store those elements in an array of size sixteen, increasing the array size if we call `push_back` seven more times. Given this description, consider the following code:

```
vector<int> one(kNumInts);
for(size_t k = 0; k < one.size(); ++k)
    one.push_back(int(k));

vector<int> two = one;
```

In the first three lines, we fill `one` with the first `kNumInts` integers, and in the last line we create a new `vector` called `two` that's a copy of `one`. How does C++ know how to correctly copy the data from `one` into `two`? It can't simply copy the pointer to the dynamically-allocated array from `one` into `two`, since that would cause `one` and `two` to share the same data and changes to one `vector` would show up in the other. Somehow C++ is aware that to copy a `vector` it needs to dynamically allocate a new array of elements, then copy the elements from the source to the destination. This is not done by magic, but by two special functions called the *copy constructor* and the *assignment operator*, which control how to copy instances of a particular class.

Before discussing the particulars of the copy constructor and assignment operator, we first need to dissect exactly how an object can be copied. In order to copy an object, we first have to answer an important question – where do we put the copy? Do we store it in a new object, or do we reuse an existing object? These two options are fundamentally different from one another and C++ makes an explicit distinction between them. The first option – putting the copy into a new location – creates the copy by *initializing* the new object to the value of the object to copy. The second – storing the copy in an existing variable – creates the copy by *assigning* the existing object the value of the object to copy. What do these two copy mechanisms look like in C++? That is, when is an object initialized to a value, and when is it assigned a new value?

In C++, initialization can occur in three different places:

1. *A variable is created as a copy of an existing value.* For example, suppose we write the following code:

```
MyClass one;
MyClass two = one;
```

Here, since `two` is told to hold the value of `one`, C++ will *initialize* `two` as a copy of `one`. Although it looks like we're assigning a value to `two` using the = operator, since it is a newly-created object, the = indicates initialization, not assignment. In fact, the above code is equivalent to the more explicit initialization code below:

```
MyClass one;
MyClass two(one);    // Identical to above.
```

This syntax makes more clear that `two` is being created as a copy of `one`, indicating initialization rather than assignment.

2. *An object is passed by value to a function.* Consider the following function:

```
void MyFunction(MyClass arg) {
     /* ... */
}
```

If we write

```
MyClass mc;
MyFunction(mc);
```

Then the function `MyFunction` somehow has to set up the value of `arg` inside the function to have the same value as `mc` outside the function. Since `arg` is a new variable, C++ will *initialize* it as a copy of `mc`.

3. *An object is returned from a function by value.* Suppose we have the following function:

```
MyClass MyFunction() {
    MyClass mc;
    return mc;
}
```

When the statement `return mc` executes, C++ needs to return the `mc` object from the function. However, `mc` is a local variable inside the `MyFunction` function, and to communicate its value to the `MyFunction` caller C++ needs to create a copy of `mc` before it is lost. This is done by creating a temporary `MyClass` object for the return value, then *initializing* it to be a copy of `mc`.

Notice that in all three cases, initialization took place because some new object was created as a copy of an existing object. In the first case this was a new local variable, in the second a function parameter, and in the third a temporary object.

Assignment in C++ is much simpler than initialization and only occurs if an existing object is explicitly assigned a new value. For example, the following code will *assign* `two` the value of `one`, rather than *initializing* `two` to `one`:

```
MyClass one, two;
two = one;
```

It can be tricky to differentiate between initialization and assignment because in some cases the syntax is almost identical. For example, if we rewrite the above code as

```
MyClass one;
MyClass two = one;
```

`two` is now initialized to `one` because it is declared as a new variable. Always remember that the assignment only occurs when giving an existing object a new value.

Why is it important to differentiate between assignment and initialization? After all, they're quite similar; in both cases we end up with a new copy of an existing object. However, assignment and initialization are fundamentally different operations. When *initializing* a new object as a copy of an existing object, we simply need to copy the existing object into the new object. When *assigning* an existing object a new value, the existing object's value ceases to be and we must make sure to clean up any resources the object may have allocated before setting it to the new value. In other words, initialization is a straight copy, while assignment is cleanup followed by a copy. This distinction will become manifest in the code we will write for the copy functions later in this chapter.

Copy Functions: Copy Constructors and Assignment Operators

Because initialization and assignment are separate tasks, C++ handles them through two different functions called the *copy constructor* and the *assignment operator*. The copy constructor is a special constructor responsible for initializing new class instances as copies of existing instances of the class. The assignment operator is a special function called an *overloaded operator* (see the chapter on operator overloading for more details) responsible for assigning the receiver object the value of some other instance of the object. Thus the code

```
MyClass one;
MyClass two = one;
```

will initialize `two` to `one` using the copy constructor, while the code

```
MyClass one, two;
two = one;
```

will assign `one` to `two` using the assignment operator.

Syntactically, the copy constructor is written as a one-argument constructor whose parameter is another instance of the class accepted by reference-to-`const`. For example, given the following class:

```
class MyClass {
public:
    MyClass();
    ~MyClass();

    /* ... */
};
```

The copy constructor would be declared as follows:

```
class MyClass {
public:
    MyClass();
    ~MyClass();

    MyClass(const MyClass& other); // Copy constructor

    /* ... */
};
```

The syntax for the assignment operator is substantially more complex than that of the copy constructor because it is an overloaded operator; in particular, `operator =`. For reasons that will become clearer later in the chapter, the assignment operator should accept as a parameter another instance of the class by reference-to-`const` and should return a non-`const` reference to an object of the class type. For a concrete example, here's the assignment operator for `MyClass`:

```
class MyClass {
public:
    MyClass();
    ~MyClass();

    MyClass(const MyClass& other); // Copy constructor
    MyClass& operator = (const MyClass& other); // Assignment operator
    /* ... */
};
```

We'll defer discussing exactly why this syntax is correct until later, so for now you should take it on faith.

What C++ Does For You

Unless you specify otherwise, C++ will automatically provide any class you write with a basic copy constructor and assignment operator that invoke the copy constructors and assignment operators of all the class's data members. In many cases, this is exactly what you want. For example, consider the following class:

```
class DefaultClass {
public:
    /* ... */

private:
    int myInt;
    string myString;
};
```

Suppose you have the following code:

```
DefaultClass one;
DefaultClass two = one;
```

The line `DefaultClass two = one` will invoke the copy constructor for `DefaultClass`. Since we haven't explicitly provided our own copy constructor, C++ will initialize `two.myInt` to the value of `one.myInt` and `two.myString` to `one.myString`. Since `int` is a primitive and `string` has a well-defined copy constructor, this code is totally fine.

However, in many cases this is not the behavior you want. Let's consider the example of a class `Vector` that acts as a wrapper for a dynamic array. Suppose we define `Vector` as shown here:

```
class Vector {
public:
    Vector();
    ~Vector();
    /* Note: No copy constructor or assignment operator */

    /* ... */

private:
    int* elems;
    /* ... */
};
```

Here, if we rely on C++'s default copy constructor or assignment operator, we'll run into trouble. For example, consider the following code:

```
Vector one;
Vector two = one;
```

Because we haven't provided a copy constructor, C++ will initialize `two.elems` to `one.elems`. Since `elems` is an `int*`, instead of getting a deep copy of the elements, we'll end up with two pointers to the same array. Thus changes to `one` will show up in `two` and vice-versa. This is dangerous, especially when the destructors for both `one` and `two` try to deallocate the memory for `elems`. In situations like these, you'll need to override C++'s default behavior by providing your own copy constructors and assignment operators.

There are a few circumstances where C++ does not automatically provide default copy constructors and assignment operators. If your class contains a reference or `const` variable as a data member, your class will not automatically get an assignment operator. Similarly, if your class has a data member that doesn't have a copy constructor or assignment operator (for example, an `ifstream`), your class won't be copyable. There is one other case involving inheritance where C++ won't automatically create the copy functions for you, and in the chapter on inheritance we'll see how to exploit this to disable copying.

The Rule of Three

There's a well-established C++ principle called the "rule of three" that identifies most spots where you'll need to write your own copy constructor and assignment operator. If this were a math textbook, you'd probably see the rule of three written out like this:

Theorem *(The Rule of Three)*: If a class has any of the following three member functions:

- Destructor
- Copy Constructor
- Assignment Operator

Then that class should have all three of those functions.

Corollary: If a class has a destructor, it should also have a copy constructor and assignment operator.

The rule of three holds because in almost all situations where you have any of the above functions, C++'s default behavior won't correctly manage your objects. In the above example with `Vector`, this is the case because copying the `elems*` pointer doesn't actually duplicate the elements array. Similarly, if you have a

class holding an open file handle, making a shallow copy of the object might cause crashes further down the line as the destructor of one class closed the file handle, corrupting the internal state of all "copies" of that object.

Both C++ libraries and fellow C++ coders will expect that, barring special circumstances, all objects will correctly implement the three above functions, either by falling back on C++'s default versions or by explicitly providing correct implementations. Consequently, you *must* keep the rule of three in mind when designing classes or you will end up with insidious or seemingly untraceable bugs as your classes start to destructively interfere with each other.

Writing Copy Constructors

For the rest of this chapter, we'll discuss copy constructors and assignment operators through a case study of a `Vector` class, a generalization of the above `Vector` which behaves similarly to the STL `vector`. The class definition for `Vector` looks like this:

```
template <typename T> class Vector {
public:
    Vector();
    Vector(const Vector& other);              // Copy constructor
    Vector& operator =(const Vector& other); // Assignment operator
    ~Vector();

    typedef T* iterator;
    typedef const T* const_iterator;

    iterator begin();
    iterator end();
    const_iterator begin() const;
    const_iterator end() const;

    /* ... other member functions ... */
private:
    T* array;
    size_t allocatedLength;
    size_t logicalLength;
    static const size kStartSize = 16;
};
```

Internally, `Vector` is implemented as a dynamically-allocated array of elements. Two data members, `allocatedLength` and `logicalLength`, track the allocated size of the array and the number of elements stored in it, respectively. `Vector` also has a class constant `kStartSize` that represents the default size of the allocated array.

The `Vector` constructor is defined as

```
template <typename T> Vector<T>::Vector() {
    allocatedLength = kStartSize;
    logicalLength = 0;
    array = new T[allocatedLength];
}
```

Similarly, the `Vector` destructor is

```
template <typename T> Vector<T>::~Vector() {
    delete [] array;
}
```

Now, let's write the copy constructor. We know that we need to match the prototype given in the class definition, so we'll write that part first:

```
template <typename T> Vector<T>::Vector(const Vector& other) {
    /* ... */
}
```

Inside the copy constructor, we need to initialize the object so that we're holding a deep copy of the other `Vector`. This necessitates making a full deep-copy of the other `Vector`'s array, as well as copying over information about the size and capacity of the other `Vector`. This second step is relatively straightforward, and can be done as follows:

```
template <typename T> Vector<T>::Vector(const DebugVector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    /* ... */
}
```

Note that this implementation of the copy constructor sets `logicalLength` to `other.logicalLength` and `allocatedLength` to `other.allocatedLength`, even though `other.logicalLength` and `other.allocatedLength` explicitly reference private data members of the `other` object. This is legal because `other` is an object of type `Vector` and the copy constructor is a member function of `Vector`. A class can access both its private fields and private fields of other objects of the same type. This is called *sibling access* and is true of any member function, not just the copy constructor. If the copy constructor were not a member of `Vector` or if `other` were not a `Vector`, this code would not be legal.

Now, we'll make a deep copy of the other `Vector`'s elements by allocating a new array that's the same size as `other`'s and then copying the elements over. The code looks something like this:

```
template <typename T> Vector<T>::Vector(const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    for(size_t i = 0; i < logicalLength; ++i)
        array[i] = other.array[i];
}
```

Interestingly, since `Vector` is a template, it's unclear what the line `array[i] = other.array[i]` will actually do. If we're storing primitive types, then the line will simply copy the values over, but if we're storing objects, the line invokes the class's assignment operator. Notice that in both cases the object will be correctly copied over. This is one of driving forces behind defining copy constructors and assignment operators, since template code can assume that expressions like `object1 = object2` will be meaningful.

An alternative means for copying data over from the other object uses the STL `copy` algorithm. Recall that `copy` takes three parameters – two delineating an input range of iterators and one denoting the beginning of an output range – then copies the specified iterator range to the destination. Although designed to work on iterators, it is possible to apply STL algorithms directly to ranges defined by raw C++ pointers. Thus we could rewrite the copy constructor as follows:

```
template <typename T> Vector<T>::Vector(const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

Here, the range spanned by `other.begin()` and `other.end()` is the entire contents of the other `Vector`, and `array` is the beginning of the newly-allocated data we've reserved for this `Vector`. I personally find this syntax preferable to the explicit `for` loop, since it increases readability.

At this point we have a complete and correct implementation of the copy constructor. The code for this constructor is not particularly dense, and it's remarkably straightforward. In some cases, however, it can be a bit trickier to write a copy constructor. We'll see some of these cases later in the chapter.

Writing Assignment Operators

We've now successfully written a copy constructor for our `Vector` class. Unfortunately, writing an assignment operator is significantly more involved than writing a copy constructor. C++ is designed to give you maximum flexibility when designing an assignment operator, and thus won't alert you if you've written a syntactically legal assignment operator that is completely incorrect. For example, consider this legal but incorrect assignment operator for an object of type `MyClass`:

```
void MyClass::operator =(const MyClass& other) {
    cout << "I'm sorry, Dave.  I'm afraid I can't copy that object." << endl;
}
```

Here, if we write code like this:

```
MyClass one, two;
two = one;
```

Instead of making `two` a deep copy of `one`, instead we'll get a message printed to the screen and `two` will remain unchanged. This is one of the dangers of a poorly-written assignment operator – code that looks like it does one thing can instead do something totally different. This section discusses how to correctly implement an assignment operator by starting with invalid code and progressing towards a correct, final version.

Let's start off with a simple but incorrect version of the assignment operator for `Vector`. Intuitively, since both the copy constructor and the assignment operator make a copy of another object, we might consider implementing the assignment operator by naively copying the code from the copy constructor into the assignment operator. This results in the following (incorrect!) version of the assignment operator:

```
/* Many major mistakes here.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

This code is based off the copy constructor, which we used to initialize the object as a copy of an existing object. Unfortunately, this code contains a substantial number of mistakes that we'll need to correct

before we end up with the final version of the function. Perhaps the most serious error here is the line `array = new T[allocatedLength]`. When the assignment operator is invoked, this `Vector` already holds its own array of elements. This line therefore orphans the old array and leaks memory. To fix this, before we make this object a copy of the one specified by the parameter, we'll take care of the necessary deallocations. This is shown here:

If you'll notice, we've already written the necessary cleanup code in the `DebugVector` destructor. Rather than rewriting this code, we'll decompose out the generic cleanup code into a `clear` function, as shown here:

```
/* Many major mistakes here.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    delete [] array;

    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

At this point, we can make a particularly useful observation. If you'll notice, the cleanup code to free the existing array is identical to the code for the destructor, which has the same task. This is no coincidence. In general, when writing an assignment operator, the assignment operator will need to free all of the resources acquired by the object, much in the same way that the destructor must. To avoid unnecessary code duplication, we can factor out the code to free the `Vector`'s resources into a helper function called `clear()`, which is shown here:

```
template <typename T> void Vector<T>::clear() {
    delete [] array;
}
```

We can then rewrite the destructor as

```
template <typename T> Vector<T>::~Vector() {
    clear();
}
```

And we can insert this call to `clear` into our assignment operator as follows:

```
/* This code still has errors.  Do not use it as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    clear();

    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

Along the same lines, you might have noticed that all of the code after the call to `clear` is exactly the same code we wrote inside the body of the copy constructor. This isn't a coincidence – in fact, in most cases you'll have a good deal of overlap between the assignment operator and copy constructor. Since we can't

invoke our own copy constructor directly (or *any* constructor, for that matter), instead we'll decompose the copying code into a member function called `copyOther` as follows:

```
template <typename T> void Vector<T>::copyOther(const Vector& other) {
    logicalLength = other.logicalLength;
    allocatedLength = other.allocatedLength;

    array = new T[allocatedLength];
    copy(other.begin(), other.end(), array);
}
```

Now we can rewrite the copy constructor as

```
template <typename T> Vector<T>::Vector(const Vector& other) {
    copyOther(other);
}
```

And the assignment operator as

```
/* Not quite perfect yet.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    clear();
    copyOther(other);
}
```

This simplifies the copy constructor and assignment operator and highlights the general pattern of what the two functions should do. With a copy constructor, you'll simply copy the contents of the other object. With an assignment operator, you'll clear out the receiver object, then copy over the data from another object.

However, we're still not done yet. There are two more issues we need to fix with our current implementation of the assignment operator. The first one has to do with *self-assignment*. Consider, for example, the following code:

```
MyClass one;
one = one;
```

While this code might seem a bit silly, cases like this come up frequently when accessing elements indirectly through pointers or references. Unfortunately, with our current `DebugVector` assignment operator, this code will lead to unusual runtime behavior, possibly including a crash. To see why, let's trace out the state of our object when its assignment operator is invoked on itself.

At the start of the assignment operator, we call `clear` to clean out the object for the copy. During this call to `clear`, we deallocate the memory associated with the object. We then invoke the `copyOther` function to set the current object to be a copy of the receiver object. Unfortunately, things don't go quite as expected. Because we're assigning the object to itself, the parameter to the assignment operator is the receiver object itself. This means that when we called `clear` trying to clean up the resources associated with the receiver object, we also cleaned up all the resources associated with the parameter to the assignment operator. In other words, `clear` destroyed both the data we wanted to clean up and the data we were meaning to copy. The call to `copyOther` will therefore copy garbage data into the receiver object, since the resources it means to copy have already been cleaned up. This is extremely bad, and will almost certainly cause a program crash.

When writing assignment operators, you *must* ensure that your code correctly handles self-assignment. While there are many ways we can do this, perhaps the simplest is to simply check to make sure that the object to copy isn't the same object pointed at by the `this` pointer. The code for this logic looks like this:

```
/* Not quite perfect yet.  Do not use this code as a reference! */
template <typename T> void Vector<T>::operator= (const Vector& other) {
    if(this != &other) {
        clear();
        copyOther(other);
    }
}
```

Note that we check `if(this != &other)`. That is, we compare *the addresses* of the current object and the parameter. This will determine whether or not the object we're copying is exactly the same object as the one we're working with. In the practice problems for this chapter, you'll explore what would happen if you were to write `if(*this != other)`. One detail worth mentioning is that the self-assignment check is not necessary in the copy constructor, since an object can't be a parameter to its own constructor.

There's one final bug we need to sort out, and it has to do with how we're legally allowed to use the `=` operator. Consider, for example, the following code:

```
MyClass one, two, three;
three = two = one;
```

This code is equivalent to `three = (two = one)`. Since our current assignment operator does not return a value, `(two = one)` does not have a value, so the above statement is meaningless and the code will not compile. We thus need to change our assignment operator so that performing an assignment like `two = one` yields a value that can then be assigned to other values. The final version of our assignment operator is thus

```
/* The correct version of the assignment operator. */
template <typename T> Vector<T>& Vector<T>::operator= (const Vector& other) {
    if(this != &other) {
        clear();
        copyOther(other);
    }
    return *this;
}
```

One General Pattern

Although each class is different, in many cases the default constructor, copy constructor, assignment operator, and destructor will share a general pattern. Here is one possible skeleton you can fill in to get your copy constructor and assignment operator working.

```
MyClass::MyClass() : /* Fill in initializer list. */ {
    /* Default initialization here. */
}

MyClass::MyClass(const MyClass& other) {
    copyOther(other);
}
```

```
MyClass& MyClass::operator =(const MyClass& other) {
    if(this != &other) {
        clear();
        // Note: When we cover inheritance, there's one more step here.
        copyOther(other);
    }
    return *this;
}

MyClass::~MyClass() {
    clear();
}
```

Semantic Equivalence and `copyOther` Strategies

Consider the following code snippet:

```
    Vector<int> one;
    Vector<int> two = one;
```

Here, we know that `two` is a copy of `one`, so the two objects should behave identically to one another. For example, if we access an element of `one`, we should get the same value as if we had accessed the corresponding element of `two` and vice-versa. However, while `one` and `two` are indistinguishable from each other in terms of functionality, their memory representations are not identical because `one` and `two` point to two different dynamically-allocated arrays. This raises the distinction between *semantic equivalence* and *bitwise equivalence*. Two objects are said to be *bitwise equivalent* if they have identical representations in memory. For example, any two `int`s with the value 137 are bitwise equivalent, and if we define a `pointT` struct as a pair of `int`s, any two `pointT`s holding the same values will be bitwise equivalent. Two objects are *semantically equivalent* if, like `one` and `two`, any operations performed on the objects will yield identical results. When writing a copy constructor and assignment operator, you attempt to convert an object into a semantically equivalent copy of another object. Consequently, you are free to pick any copying strategy that creates a semantically equivalent copy of the source object.

In the preceding section, we outlined one possible implementation strategy for a copy constructor and assignment operator that uses a shared function called `copyOther`. While in the case of the `DebugVector` it was relatively easy to come up with a working `copyOther` implementation, when working with more complicated objects, it can be difficult to devise a working `copyOther`. For example, consider the following class, which represents a mathematical set implemented as a linked list:

```
template <typename T> class ListSet {
public:
    ListSet();
    ListSet(const ListSet& other);
    ListSet& operator =(const ListSet& other);
    ~ListSet();

    void insert(const T& toAdd);
    bool contains(const T& toFind) const;

private:
    struct cellT {
        T data;
        cellT* next;
    };
    cellT* list;

    void copyOther(const ListSet& other);
    void clear();
};
```

This `ListSet` class exports two functions, `insert` and `contains`, that insert an element into the list and determine whether the list contains an element, respectively. This class represents a mathematical set, an *unordered* collection of elements, so the underlying linked list need not be in any particular order. For example, the lists `{0, 1, 2, 3, 4}` and `{4, 3, 2, 1, 0}` are semantically equivalent because checking whether a number is an element of the first list yields the same result as checking whether the number is in the second. In fact, any two lists containing the same elements are semantically equivalent to one another. This means that there are multiple ways in which we could implement `copyOther`. Consider these two:

```
/* Version 1: Duplicate the list as it exists in the original ListSet. */
template <typename T> void ListSet<T>::copyOther(const ListSet& other) {
    /* Keep track of what the current linked list cell is. */
    cellT** current = &list;

    /* Iterate over the source list. */
    for(cellT* source = other.list; source != NULL; source = source->next) {
        /* Duplicate the cell. */
        *current = new cellT;
        (*current)->data = source->data;
        (*current)->next = NULL;

        /* Advance to next element. */
        current = &((*current)->next);
    }
}

/* Version 2: Duplicate list in reverse order of original ListSet */
template <typename T> void ListSet<T>::copyOther(const ListSet& other) {
    for(cellT* source = other.list; source != NULL; source = source->next) {
        cellT* newNode = new cellT;
        newNode->data = source->data;
        newNode->next = list;
        list = newNode;
    }
}
```

As you can see, the second version of this function is much, *much* cleaner than the first. There are no address-of operators floating around, so everything is expressed in terms of simpler pointer operations. But while the second version is cleaner than the first, it duplicates the list in reverse order. This may initially seem problematic but is actually perfectly safe. As the original object and the duplicate object contain the same elements in *some* order, they will be semantically equivalent, and from the class interface we would be unable to distinguish the original object and its copy.

There is one implementation of `copyOther` that is considerably more elegant than either of the two versions listed above:

```
/* Version 3: Duplicate list using the insert function */
template <typename T> void ListSet<T>::copyOther(const ListSet& other) {
    for(cellT* source = other.list; source != NULL; source = source->next)
        insert(source->data);
}
```

Notice that this implementation uses the `ListSet`'s public interface to insert the elements from the source `ListSet` into the receiver object. This version of `copyOther` is unquestionably the cleanest. If you'll notice, it doesn't matter exactly how `insert` adds elements into the list (indeed, `insert` could insert the elements at random positions), but we're guaranteed that at the end of the `copyOther` call, the receiver object will be semantically equivalent to the parameter.

Conversion Assignment Operators

When working with copy constructors, we needed to define an additional function, the assignment operator, to handle all the cases in which an object can be copied or assigned. However, in the chapter on conversion constructors, we provided a conversion constructor without a matching "conversion assignment operator." It turns out that this is not a problem because of how the assignment operator is invoked. Suppose that we have a `CString` class that has a defined copy constructor, assignment operator, and conversion constructor that converts raw C++ `char *` pointers into `CString` objects. Now, suppose we write the following code:

```
CString myCString;
myCString = "This is a C string!";
```

Here, in the second line, we assign an existing `CString` a new value equal to a raw C string. Despite the fact that we haven't defined a special assignment operator to handle this case, the above is perfectly legal code. When we write the line

```
myCString = "This is a C string!";
```

C++ converts it into the equivalent code

```
myCString.operator= ("This is a C string!");
```

This syntax may look entirely foreign, but is simply a direct call to the assignment operator. Recall that the assignment operator is a function named `operator =`, so this code passes the C string `"This is a C string!"` as a parameter to `operator =`. Because `operator =` accepts a `CString` object rather than a raw C string, C++ will invoke the `CString` conversion constructor to initialize the parameter to `operator =`. Thus this code is equivalent to

```
myCString.operator =(CString("This is a C string!"));
```

In other words, the conversion constructor converts the raw C string into a `CString` object, then the assignment operator sets the receiver object equal to this temporary `CString`.

In general, you need not provide a "conversion assignment operator" to pair with a conversion constructor. As long as you've provided well-defined copy behavior, C++ will link the conversion constructor and assignment operator together to perform the assignment.

Disabling Copying

In CS106B/X we provide you the `DISALLOW_COPYING` macro, which causes a compile error if you try to assign or copy objects of the specified type. `DISALLOW_COPYING`, however, is not a standard C++ feature. Without using the CS106B/X library, how can we replicate the functionality? We can't prevent object copying by simply not defining a copy constructor and assignment operator. All this will do is have C++ provide its own default version of these two functions, which is not at all what we want. To solve this problem, instead we'll provide an assignment operator and copy constructor, but declare them private so that class clients can't access them. For example:

```
class CannotBeCopied {
public:
    CannotBeCopied();
    /* Other member functions. */

private:
    CannotBeCopied(const CannotBeCopied& other);
    CannotBeCopied& operator = (const CannotBeCopied& other);
};
```

Now, if we write code like this:

```
CannotBeCopied one;
CannotBeCopied two = one;
```

We'll get a compile-time error on the second line because we're trying to invoke the copy constructor, which has been declared private. We'll get similar behavior when trying to use the assignment operator.

This trick is almost one hundred percent correct, but does have one edge case: what if we try to invoke the copy constructor or assignment operator inside a member function of the class? The copy functions might be private, but that doesn't mean that they don't exist, and if we call them inside a member function might accidentally create a copy of an otherwise uncopyable object. To prevent this from happening, we'll use a cute trick. Although we'll *prototype* the copy functions inside the private section of the class, we won't *implement* them. This means that if we accidentally do manage to call either function, we will get a linker error because the compiler can't find code for either function. This is admittedly a bit hackish, so in C++0x, the next revision of C++, there will be a way to explicitly indicate that a class is uncopyable. In the meantime, though, the above approach is perhaps your best option. We'll see another way to do this later when we cover inheritance.

Extended Example: `SmartPointer`

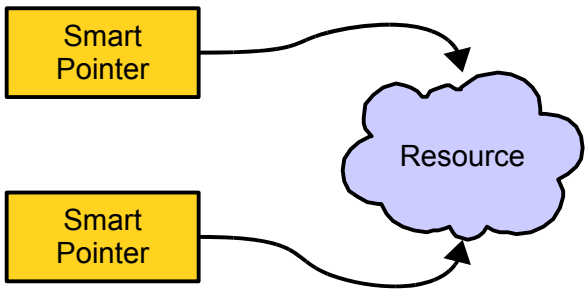
In C++ parlance, a raw pointer like an `int*` or a `char*` is sometimes called a *dumb pointer* because the pointer has no "knowledge" of the resource it owns. If an `int*` goes out of scope, it doesn't inform the object it's pointing at and makes no attempt whatsoever to clean it up. The `int*` doesn't own its resource, and assigning one `int*` to another doesn't make a deep copy of the resource or inform the other `int*` that another pointer now references its pointee.

Because raw pointers are so problematic, many C++ programmers prefer to use *smart pointers*, objects that mimic raw pointers but which perform functions beyond merely pointing at a resource. For example, the C++ standard library class `auto_ptr`, which we'll cover in the chapter on exception handling, acts like a regular pointer except that it automatically calls `delete` on the resource it owns when it goes out of scope. Other smart pointers are custom-tuned for specific applications and might perform functions like logging access, synchronizing multithreaded applications, or preventing accidental null pointer dereferences. Thanks to operator overloading, smart pointers can be built to look very similar to regular C++ pointers. We can provide an implementation of `operator *` to support dereferences like `*myPtr`, and can define `operator ->` to let clients write code to the effect of `myPtr->clear()`. Similarly, we can write copy constructors and assignment operators for smart pointers that do more than just transfer a resource.

Reference Counting

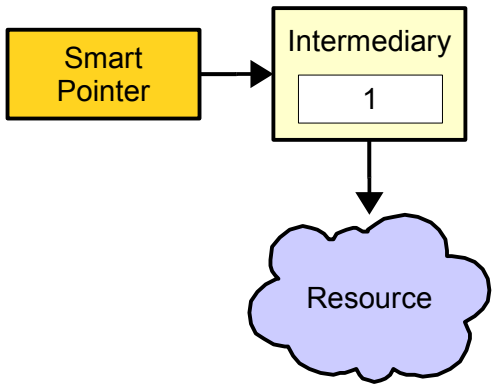
Memory management in C++ is tricky. You must be careful to balance every `new` with exactly one `delete`, and must make sure that no other pointers to the resource exist after `delete`-ing it to ensure that later on you don't access invalid memory. If you `delete` memory too many times you run into undefined behavior, and if you `delete` it too few you have a memory leak. Is there a better way to manage memory? In many cases, yes, and in this extended example we'll see one way to accomplish this using a technique called *reference counting*. In particular, we'll design a smart pointer class called `SmartPointer` which acts like a regular C++ pointer, except that it uses reference counting to prevent resource leaks.

To motivate reference counting, let's suppose that we have a smart pointer class that stores a pointer to a resource. The destructor for this smart pointer class can then `delete` the resource automatically, so clients of the smart pointer never need to explicitly clean up any resources. This system is fine in restricted circumstances, but runs into trouble as soon as we have several smart pointers pointing to the same resource. Consider the scenario below:



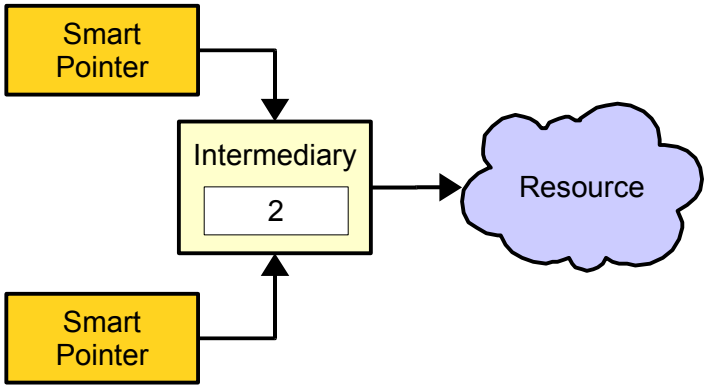
Both of these pointers can access the stored resource, but unfortunately neither smart pointer knows of the other's existence. Here we hit a snag. If one smart pointer cleans up the resource while the other still points to it, then the other smart pointer will point to invalid memory. If both of the pointers try to reclaim the dynamically-allocated memory, we will encounter a runtime error from double-`delete`-ing a resource. Finally, if neither pointer tries to clean up the memory, we'll get a memory leak.

To resolve this problem, we'll use a system called *reference counting* where we will explicitly keep track of the number of pointers to a dynamically-allocated resource. While there are several ways to make such a system work, perhaps the simplest is to use an intermediary object. This can be seen visually:



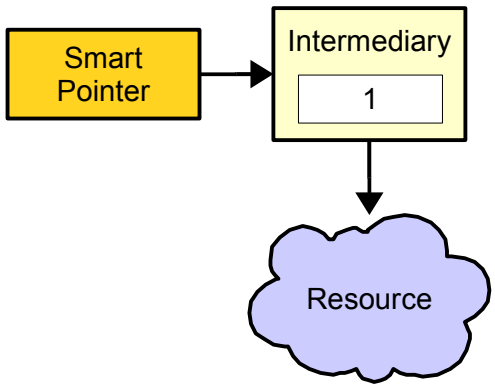
Now, the smart pointer stores a pointer to an intermediary object rather than a pointer directly to the resource. This intermediary object has a counter (called a *reference counter*) that tracks the number of smart pointers accessing the resource, as well as a pointer to the managed resource. This intermediary object lets the smart pointers tell whether or not they are the only pointer to the stored resource; if the reference count is anything other than one, some other pointer shares the resource. Provided that we accurately track the reference count, each pointer can tell if it's the last pointer that knows about the resource and can determine whether to deallocate it.

To see how reference counting works, let's walk through an example. Given the above system, suppose that we want to share the resource with another smart pointer. We simply make this new smart pointer point to the same intermediary object as our original pointer, then update the reference count. The resulting scenario looks like this:

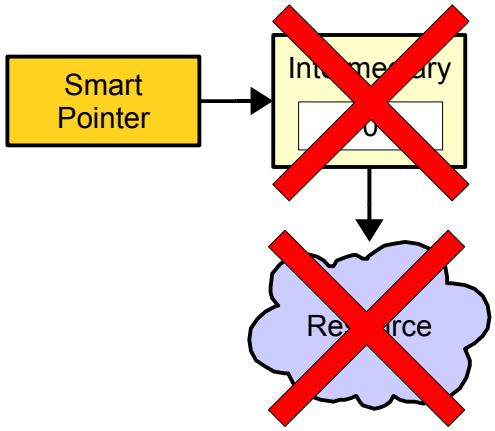


Although in this diagram we only have two objects pointing to the intermediary, the reference-counting system allows for any number of smart pointers to share a single resource.

Now, suppose one of these smart pointers needs to stop pointing to the resource – maybe it's being assigned to a different resource, or perhaps it's going out of scope. That pointer decrements the reference count of the intermediary variable and notices that the reference count is nonzero. This means that at least one smart pointer still references the resource, so the smart pointer simply leaves the resource as it is. Memory now looks like this:



Finally, suppose this last smart pointer needs to stop pointing to this resource. It decrements the reference count, but this time notices that the reference count is zero. This means that no other smart pointers reference this resource, and the smart pointer knows that it needs to deallocate the resource and the intermediary object, as shown here:



The resource has now been deallocated and no other pointers reference the memory. We've safely and effectively cleaned up our resources. Moreover, this process is completely automatic – the user never needs to explicitly deallocate any memory.

The following summarizes the reference-counting scheme described above:

- When creating a smart pointer to manage newly-allocated memory, first create an intermediary object and make the intermediary point to the resource. Then, attach the smart pointer to the intermediary and set the reference count to one.

- To make a new smart pointer point to the same resource as an existing one, make the new smart pointer point to the old smart pointer's intermediary object and increment the intermediary's reference count.

- To remove a smart pointer from a resource (either because the pointer goes out of scope or because it's being reassigned), decrement the intermediary object's reference count. If the count reaches zero, deallocate the resource and the intermediary object.

While reference counting is an excellent system for managing memory automatically, it does have its limitations. In particular, reference counting can sometimes fail to clean up memory in "reference cycles," situations where multiple reference-counted pointers hold references to one another. If this happens, none of the reference counters can ever drop to zero, since the cyclically-linked elements always refer to one another. But barring this sort of setup, reference counting is an excellent way to automatically manage memory. In this extended example, we'll see how to implement a reference-counted pointer, which we'll

call `SmartPointer`, and will explore how the correct cocktail of C++ constructs can make the resulting class slick and efficient.

Designing `SmartPointer`

The above section details the *implementation* the `SmartPointer` class, but we have not talked about its *interface*. What functions should we provide? We'll try to make `SmartPointer` resemble a raw C++ pointer as closely as possible, meaning that it should support `operator *` and `operator ->` so that the client can dereference the `SmartPointer`. Here is one possible interface for the `SmartPointer` class:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *  () const;
    T* operator -> () const;
};
```

Here is a breakdown of what each of these functions should do:

`explicit SmartPointer(T* memory);`

Constructs a new `SmartPointer` that manages the resource specified as the parameter. The reference count is initially set to one. We will assume that the provided pointer came from a call to `new`. This function is marked `explicit` so that we cannot accidentally convert a regular C++ pointer to a `SmartPointer`. At first this might seem like a strange design decision, but it prevents a wide range of subtle bugs. For example, suppose that this constructor is not `explicit` and consider the following function:

```
void PrintString(const SmartPointer<string>& ptr) {
    cout << *ptr << endl;
}
```

This function accepts a `SmartPointer` by reference-to-`const`, then prints out the stored string. Now, what happens if we write the following code?

```
string* ptr = new string("Yay!");
PrintString(ptr);
delete ptr;
```

The first line dynamically-allocates a `string`, passes it to `PrintString`, and finally deallocates it. Unfortunately, this code will almost certainly cause a runtime crash. The problem is that `PrintString` expects a `SmartPointer<string>` as a parameter, but we've provided a `string*`. C++ notices that the `SmartPointer<string>` has a conversion constructor that accepts a `string*`, and makes a temporary `SmartPointer<string>` using the pointer we passed as a parameter. This new `SmartPointer` starts tracking the pointer with a reference count of one. After the function returns, the parameter is cleaned up and its destructor invokes. This decrements the reference count to zero, and then deallocates the pointer stored in the `SmartPointer`. The above code then tries to `delete ptr` a second time, causing a runtime crash. To prevent this problem, we'll mark the constructor `explicit`, which makes the implicit conversion illegal and prevents this buggy code from compiling.

`SmartPointer(const SmartPointer& other);` Constructs a new `SmartPointer` that shares the resource contained in another `SmartPointer`, updating the reference count appropriately.
`SmartPointer& operator=(const SmartPointer& other);` Causes this `SmartPointer` to stop pointing to the resource it's currently managing and to share the resource held by another `SmartPointer`. If the smart pointer was the last pointer to its resource, it `delete`s it.
`~SmartPointer();` Detaches the `SmartPointer` from the resource it's sharing, freeing the associated memory if necessary.
`T& operator* () const;` "Dereferences" the pointer and returns a reference to the object being pointed at. Note that `operator*` is `const`; see the last chapter for more information why.
`T* operator-> () const;` Returns the object that the arrow operator should really be applied to if the arrow is used on the `SmartPointer`. Again, see the last chapter for more information on this.

Given this public interface for `SmartPointer`, we can now begin implementing the class. We first need to decide on how we should represent the reference-counting information. One simple method is to define a private `struct` inside `SmartPointer` that represents the reference-counting intermediary. This looks as follows:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *  () const;
    T* operator -> () const;

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;
};
```

Here, the `resource` field of the `Intermediary` is the actual pointer to the stored resource and `refCount` is the reference count. Notice that we did not declare the reference count as a direct data member of the `SmartPointer`, but rather in the `Intermediary` object. This is because the reference count of a resource is not owned by any one `SmartPointer`, but rather is shared across all `SmartPointer`s that point to a particular resource. This way, any changes to the reference count by one `SmartPointer` will become visible in all of the other `SmartPointer`s referencing the resource. You might ask – could we have made the `refCount` a `static` data member? This would indeed make the reference count visible across

multiple `SmartPointer`s, but unfortunately it won't work out correctly. In particular, if we use `SmartPointer` to manage multiple resources, each one needs to have its own `refCount` or changes to the `refCount` for a particular *resource* will show up in the `refCount` for other resources.

Given this setup, we can implement the `SmartPointer` constructor by creating a new `Intermediary` that points to the specified resource and has an initial reference count of one:

```
template <typename T> SmartPointer<T>::SmartPointer(T* res) {
    data = new Intermediary;
    data->resource = res;
    data->refCount = 1;
}
```

It's very important that we allocate the `Intermediary` object on the heap rather than as a data member. That way, when the `SmartPointer` is cleaned up (either by going out of scope or by an explicit call to delete), if it isn't the last pointer to the shared resource, the intermediary object isn't cleaned up.

We can similarly implement the destructor by decrementing the reference count, then cleaning up memory if appropriate. Note that if the reference count hits zero, we need to delete both the resource *and* the intermediary. Forgetting to deallocate either of these leads to memory leaks, the exact problem we wanted to avoid. The code for this is shown here:

```
template <typename T> SmartPointer<T>::~SmartPointer() {
    --data->refCount;
    if(data->refCount == 0) {
        delete data->resource;
        delete data;
    }
}
```

This is an interesting destructor in that it isn't guaranteed to actually clean up any memory. Of course, this is exactly the behavior we want, since the memory might be shared among multiple `SmartPointer`s.

Implementing `operator *` and `operator ->` simply requires us to access the pointer stored inside the `SmartPointer`. These two functions can be implemented as follows:*

```
template <typename T> T& SmartPointer<T>::operator * () const {
    return *data->resource;
}
template <typename T> T* SmartPointer<T>::operator -> () const {
    return data->resource;
}
```

Now, we need to implement the copy behavior for this `SmartPointer`. One way to do this is to write helper functions clear and `copyOther` which perform `deallocation` and copying. We will use a similar

---

* It is common to see `operator ->` implemented as

```
RetType* MyClass::operator -> () const
{
    return &**this;
}
```

`&**this` is interpreted by the compiler as `&(*(*this))`, which means "dereference the `this` pointer to get the receiver object, then dereference the receiver. Finally, return the address of the referenced object." At times this may be the best way to implement `operator ->`, but I advise against it in general because it's fairly cryptic.

approach here, except using functions named `detach` and `attach` to make explicit the operations we're performing. This leads to the following definition of `SmartPointer`:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *  () const;
    T* operator -> () const;

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};
```

Now, what should these functions do? The first of these, `detach`, should detach the `SmartPointer` from the shared intermediary and clean up the memory if it was the last pointer to the shared resource. In case this sounds familiar, it's because this is exactly the behavior of the `SmartPointer` destructor. To avoid code duplication, we'll move the code from the destructor into detach as shown here:

```
template <typename T> void SmartPointer<T>::detach() {
    --data->refCount;
    if(data->refCount == 0) {
        delete data->resource;
        delete data;
    }
}
```

We can then implement the destructor as a wrapped call to detach, as seen here:

```
template <typename T> SmartPointer<T>::~SmartPointer() {
    detach();
}
```

The attach function, on the other hand, makes this SmartPointer begin pointing to the specified Intermediary and increments the reference count. Here's one possible implementation of attach:

```
template <typename T> void SmartPointer<T>::attach(Intermediary* to) {
    data = to;
    ++data->refCount;
}
```

Given these two functions, we can implement the copy constructor and assignment operator for `SmartPointer` as follows:

```
template <typename T> SmartPointer<T>::SmartPointer(const SmartPointer& other){
    attach(other.data);
}

template <typename T>
SmartPointer<T>& SmartPointer<T>::operator= (const SmartPointer& other) {
    if(this != &other) {
        detach();
        attach(other.data);
    }
    return *this;
}
```

It is crucial that we check for self-assignment inside the `operator=` function, since otherwise we might destroy the data that we're trying to keep track of!

At this point we have a rather slick `SmartPointer` class. Here's some code demonstrating how a client might use `SmartPointer`:

```
SmartPointer<string> myPtr(new string);
*myPtr = "This is a string!";
cout << *myPtr << endl;

SmartPointer<string> other = myPtr;
cout << *other << endl;
cout << other->length() << endl;
```

The beauty of this code is that client code using a `SmartPointer<string>` looks almost identical to code using a regular C++ pointer. Isn't operator overloading wonderful?

Extending `SmartPointer`

The `SmartPointer` defined above is useful but lacks some important functionality. For example, suppose that we have the following function:

```
void DoSomething(string* ptr);
```

Suppose that we have a `SmartPointer<string>` managing a resource and that we want to pass the stored string as a parameter to `DoSomething`. Despite the fact that `SmartPointer<string>` mimics a `string*`, it technically is not a `string*` and C++ won't allow us to pass the `SmartPointer` into `DoSomething`. Somehow we need a way to have the `SmartPointer` hand back the resource it manages.

Notice that the only `SmartPointer` member functions that give back a pointer or reference to the actual resource are `operator*` and `operator->`. Technically speaking, we *could* use these functions to pass the stored `string` into `DoSomething`, but the syntax would be messy (in the case of `operator*`) or nightmarish (for `operator ->`). For example:

```
SmartPointer<string> myPtr(new string);

/* To use operator* to get the stored resource, we have to first dereference
 * the SmartPointer, then use the address-of operator to convert the returned
 * reference into a pointer.
 */
DoSomething(&*myPtr);

/* To use operator-> to get the stored resource, we have to explicitly call the
 * operator-> function.  Yikes!
 */
DoSomething(myPtr.operator-> ());
```

Something is clearly amiss and we cannot reasonably expect clients to write code like this routinely. We'll need to extend the `SmartPointer` class to provide a way to return the stored pointer directly. This necessitates the creation of a new member function, which we'll call `get`, to do just that. Given a function like this, we could then invoke `DoSomething` as follows:

```
DoSomething(myPtr.get());
```

The updated interface for `SmartPointer` looks like this:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *  () const;
    T* operator -> () const;

    T* get() const;

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};
```

The implementation of `get` is fairly straightforward and is shown here:

```
template <typename T> T* SmartPointer<T>::get() const {
    return data->resource;
}
```

Further Extensions

There are several more extensions to the `SmartPointer` class that we might want to consider, of which this section explores two. The first is rather straightforward. At times, we might want to know exactly how many `SmartPointer`s share a resource. This might enable us to perform some optimizations, in

particular a technique called *copy-on-write*. We will not explore this technique here, though you are encouraged to do so on your own.

Using the same logic as above, we'll define another member function called `getShareCount` which returns the number of `SmartPointer`s pointing to the managed resource (including the receiver object). This results in the following class definition:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *  () const;
    T* operator -> () const;

    T*  get() const;
    size_t getShareCount() const;

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};
```

And the following implementation:

```
template <typename T> size_t SmartPointer<T>::getShareCount() const {
    return data->refCount;
}
```

The last piece of functionality we'll consider is the ability to "reset" the `SmartPointer` to point to a different resource. When working with a `SmartPointer`, at times we may just want to drop whatever resource we're holding and begin managing a new one. As you might have suspected, we'll add yet another member function called `reset` which resets the `SmartPointer` to point to a new resource. The final interface and code for `reset` is shown here:

```
template <typename T> class SmartPointer {
public:
    explicit SmartPointer(T* memory);
    SmartPointer(const SmartPointer& other);
    SmartPointer& operator =(const SmartPointer& other);
    ~SmartPointer();

    T& operator *  () const;
    T* operator -> () const;

    T*     get() const;
    size_t getShareCount() const;
    void   reset(T* newRes);

private:
    struct Intermediary {
        T* resource;
        size_t refCount;
    };
    Intermediary* data;

    void detach();
    void attach(Intermediary* other);
};

template <typename T> void SmartPointer<T>::reset(T* newRes) {
    /* We're no longer associated with our current resource, so drop it. */
    detach();

    /* Attach to a new intermediary object. */
    data = new Intermediary;
    data->resource = newRes;
    data->refCount = 1
}
```

Practice Problems

The only way to learn copy constructors and assignment operators is to play around with them to gain experience. Here are some practice problems and thought questions to get you started:

1. When is the copy constructor invoked?
2. When is the assignment operator invoked?
3. What is the signature of the copy constructor?
4. What is the signature of the assignment operator?
5. What is the rule of three? What are the "three" it refers to?
6. What is the behavior of the default-generated copy constructor and assignment operator?
7. Why does the assignment operator have to check for self-assignment but the copy constructor not need to check for "self-initialization?"
8. What is bitwise equivalence? What is semantic equivalence? Which of the two properties should be guaranteed by the two copy functions?

9. What is a smart pointer?

10. What is reference-counting?

11. Realizing that the copy constructor and assignment operator for most classes have several commonalities, you decide to implement a class's copy constructor using the class's assignment operator. For example, you try implementing the `Vector`'s copy constructor as

```
template <typename T> Vector<T>::Vector(const Vector& other) {
    *this = other;
}
```

(Since `this` is a pointer to the receiver object, `*this` is the receiver object, so `*this = other` means to assign the receiver object the value of the parameter `other`)

This idea, while well-intentioned, has a serious flaw that causes the copy constructor to almost always cause a crash. Why is this? *(Hint: Were any of the `Vector` data members initialized before calling the assignment operator? Walk through the assignment operator and see what happens if the receiver object's data members haven't been initialized.)*

12. It is illegal to write a copy constructor that accepts its parameter by value. Why is this? However, it's perfectly acceptable to have an assignment operator that accepts its parameter by value. Why is this legal? Why the difference?

13. An alternative implementation of the assignment operator uses a technique called *copy-and-swap*. The copy-and-swap approach is broken down into two steps. First, we write a member function that accepts a reference to another instance of the class, then exchanges the data members of the receiver object and the parameter. For example, when working with the `DebugVector`, we might write a function called `swapWith` as follows:

```
template <typename ElemType> void Vector<ElemType>::swapWith(Vector& other)
{
    swap(array, other.array);
    swap(logicalLength, other.logicalLength);
    swap(allocatedLength, other.allocatedLength);
}
```

Here, we use the STL `swap` algorithm to exchange data members. Notice that we never actually make a deep-copy of any of the elements in the array – we simply swap pointers with the other `DebugVector`. We can then implement the assignment operator as follows:

```
template <typename T> Vector<T>& Vector<T>::operator= (const Vector& other)
{
    DebugVector temp(other);
    swapWith(temp);
    return *this;
}
```

Trace through this implementation of the assignment operator and explain how it sets the receiver object to be a deep-copy of the parameter. What function actually deep-copies the data? What function is responsible for cleaning up the old data members?

14. When writing an assignment operator using the pattern covered earlier in the chapter, we had to explicitly check for self-assignment in the body of the assignment operator. Explain why this is no longer necessary using the copy-and-swap approach, but why it still might be a good idea to insert the self-assignment check anyway.

15. A singleton class is a class that can have at most one instance. Typically, a singleton class has its default constructor and destructor marked private so that clients cannot instantiate the class directly, and exports a `static` member function called `getInstance()` that returns a reference to the only instance of the class. That one instance is typically a private static data member of the class. For example:

```
class Singleton {
public:
    static Singleton& getInstance();

private:
    Singleton();  // Clients cannot call this function; it's private
    ~Singleton(); // ... nor can they call this one

    static Singleton instance; // ... but they can be used here because
                               // instance is part of the class.
};

Singleton Singleton::instance;
```

Does it make sense for a singleton class to have a copy constructor or assignment operator? If so, implement them. If not, modify the `Singleton` interface so that they are disabled.

16. Given this chapter's description about how to disable copying in a class, implement a macro `DISALLOW_COPYING` that accepts as a parameter the name of the current class such that if `DISALLOW_COPYING` is placed into the private section of a class, that class is uncopyable. Note that it is legal to create macros that span multiple lines by ending each line with the `\` character. For example, the following is all one macro:

```
#define CREATE_PRINTER(str) void Print##str() {\
    cout << #str << endl;\
}
```

17. Consider the following alternative mechanism for disabling copying in a class: instead of marking those functions private, instead we implement those functions, but have them call `abort` (a function from `<cstdlib>` that immediately terminates the program) after printing out an error message. For example:

```
class PseudoUncopyable {
 public:
    PseudoUncopyable(const PseudoUncopyable& other) {
        abort();
    }
    PseudoUncopyable& operator= (const PseudoUncopyable& other) {
        abort();
        return *this; // Never reached; suppresses compiler warnings
    }
};
```

Why is this approach a bad idea?

18. Should you copy `static` data members in a copy constructor or assignment operator? Why or why not?

19. In the canonical implementation of the assignment operator we saw earlier in this chapter, we used the check `if (this != &other)` to avoid problems with self-assignment. In this exercise, we'll see what happens if we replace this check with `if (*this != other)`.

    1. What is the meaning of `if (*this != other)`? Will this code compile for any class, or does that class have to have a special property?

    2. Will the check `if (*this != other)` correctly detect whether an object is being assigned to itself? Will it detect anything else?

    3. Assume that the `Vector` has an implementation of `operator!=` that checks whether the operands have exactly the same size and elements. What is the asymptotic (big-O) complexity of the check `if(*this != other)`? How about `if (this != &other)`? Does this give you a better sense why the latter is preferable to the former?

20. In a sense, our implementation of the `Vector` assignment operator is wasteful. It works by completely discarding the internal array, then constructing a new array to hold the other `Vector`'s elements. An alternative implementation would work as follows. If the other `Vector`'s elements can fit in the space currently allocated by the `Vector`, then the elements from the other `Vector` are copied directly into the existing space. Otherwise, new space is allocated as before. Rewrite the `Vector`'s `operator=` function using this optimization. Why won't this technique work for the copy constructor?

Chapter 12: Error Handling

> *Forty years ago, `goto`-laden code was considered perfectly good practice. Now we strive to write structured control flows. Twenty years ago, globally accessible data was considered perfectly good practice. Now we strive to encapsulate data. Ten years ago, writing functions without thinking about the impact of exceptions was considered good practice. Now we strive to write exception-safe code.*
>
> *Time goes on. We live. We learn.*
>
> – Scott Meyers, author of *Effective C++* and one of the leading experts on C++. [Mey05]

In an ideal world, network connections would never fail, files would always exist and be properly formatted, users would never type in malformed input, and computers would never run out of memory. Realistically, though, all of the above can and will occur and your programs will have to be able to respond to them gracefully. In these scenarios, the normal function-call-and-return mechanism is not robust enough to signal and report errors and you will have to rely on *exception handling*, a C++ language feature that redirects program control in case of emergencies.

Exception handling is a complex topic and will have far-reaching effects on your C++ code. This chapter introduces the motivation underlying exception handling, basic exception-handling syntax, and some advanced techniques that can keep your code operating smoothly in an exception-filled environment.

A Simple Problem

Up to this point, all of the programs you've written have proceeded linearly – they begin inside a special function called `main`, then proceed through a chain of function calls and returns until (hopefully) hitting the end of `main`. While this is perfectly acceptable, it rests on the fact that each function, given its parameters, can perform a meaningful task and return a meaningful value. However, in some cases this simply isn't possible.

Suppose, for example, that we'd like to write our own version of the CS106B/X `StringToInteger` function, which converts a `string` representation of an number into an `int` equivalent. One possible (partial) implementation of `StringToInteger` might look like this:*

```
int StringToInteger(const string &input) {
    stringstream converter(input);
    int result; // Try reading an int, fail if we're unable to do so.

    converter >> result;
    if (converter.fail())
        // What should we do here?

    char leftover; // See if anything's left over.  If so, fail.
    converter >> leftover;
    if (!converter.fail())
       return result;
    else
       // What should we do here?
}
```

* This is based off of the `GetInteger` function we covered in the chapter on streams. Instead of looping and reprompting the user for input at each step, however, it simply reports errors on failure.

If the parameter `input` is a `string` with a valid integer representation, then this function simply needs to perform the conversion. But what should our function do if the parameter doesn't represent an integer? One possible option, and the one used by the CS106B/X implementation of `StringToInteger`, is to call a function like `Error` that prints an error and terminates the program. This response seems a bit drastic and is a decidedly suboptimal solution for several reasons. First, calling `Error` doesn't give the program a chance to recover from the problem. `StringToInteger` is a simple utility function, not a critical infrastructure component, and if it fails chances are that there's an elegant way to deal with the problem. For example, if we're using `StringToInteger` to convert user input in a text box into an integer for further processing, it makes far more sense to reprompt the user than to terminate the program. Second, in a very large or complicated software system, it seems silly to terminate the program over a simple string error. For example, if this `StringToInteger` function were used in an email client to convert a string representation of a time to an integer format (parsing the hours and minutes separately), it would be disastrous if the program crashed whenever receiving malformed emails. In essence, while using a function like `Error` will prevent the program from continuing with garbage values, it is simply too drastic a move to use in serious code.

This approach suggests a second option, one common in pure C – *sentinel values*. The idea is to have functions return special values meaning "this value indicates that the function failed to execute correctly." In our case, we might want to have `StringToInteger` return -1 to indicate an error, for example. Compared with the "drop everything" approach of `Error` this may seem like a good option – it reports the error and gives the calling function a chance to respond. However, there are several major problems with this method. First, in many cases it is not possible to set aside a value to indicate failure. For example, suppose that we choose to reserve -1 as an error code for `StringToInteger`. In this case, we'd make all of our calls to `StringToInteger` as

```
if (StringToInteger(myParam) == -1) {
    /* ... handle error ... */
}
```

But what happens if the input to `StringToInteger` is the string `"-1"`? In this case, whether or not the `StringToInteger` function completes successfully, it will still return -1 and our code might confuse it with an error case.

Another serious problem with this approach is that if each function that might possibly return an error has to reserve sentinel values for errors, we might accidentally check the return value of one function against the error code of another function. Imagine if there were several constants floating around named `ERROR`, `STATUS_ERROR`, `INVALID_RESULT`, etc., and whenever you called a function you needed to check the return value against the correct one of these choices. If you chose incorrectly, even with the best of intentions your error-checking would be invalid.

Yet another shortcoming of this approach is that in some cases it will be impossible to reserve a value for use as a sentinel. For example, suppose that a function returns a `vector<double>`. What special `vector<double>` should we choose to use as a sentinel?

However, the most serious problem with the above approach is that you as a programmer can ignore the return value without encountering any warnings. Even if `StringToInteger` returns a sentinel value indicating an error, there are no compile-time or runtime warnings if you choose not to check for a return value. In the case of `StringToInteger` this may not be that much of a problem – after all, holding a sentinel value instead of a meaningful value will not immediately crash the program – but this can lead to problems down the line that can snowball into fully-fledged crashes. Worse, since the crash will probably be caused by errors from far earlier in the code, these sorts of problems can be nightmarish to debug.

Surprisingly, experience shows that many programmers – either out of negligence or laziness – forget to check return values for error codes and snowball effects are rather common.

We seem to have reached an unsolvable problem. We'd like an error-handling system that, like `Error`, prevents the program from continuing normally when an error occurs. At the same time, however, we'd like the elegance of sentinel values so that we can appropriately process an error. How can we combine the strengths of both of these approaches into a single system?

Exception Handling

The reason the above example is such a problem is that the normal C++ function-call-and-return system simply isn't robust enough to communicate errors back to the calling function. To resolve this problem, C++ provides language support for an error-messaging system called *exception handling* that completely bypasses function-call-and-return. If an error occurs inside a function, rather than returning a value, you can report the problem to the exception handling system to jump to the proper error-handling code.

The C++ exception handling system is broken into three parts – `try` blocks, `catch` blocks, and `throw` statements. `try` blocks are simply regions of code designated as areas that runtime errors might occur. To declare a `try` block, you simply write the keyword *`try`*, then surround the appropriate code in curly braces. For example, the following code shows off a `try` block:

```
try {
    cout << "I'm in a try block!" << endl;
}
```

Inside of a `try` block, code executes as normal and jumps to the code directly following the `try` block once finished. However, at some point inside a `try` block your program might run into a situation from which it cannot normally recover – for example, a call to `StringToInteger` with an invalid argument. When this occurs, you can report the error by using the *`throw`* keyword to "throw" the exception into the nearest matching `catch` clause. Like `return`, `throw` accepts a single parameter that indicates an object to throw so that when handling the exception your code has access to extra information about the error. For example, here are three statements that each throw objects of different types:

```
throw 0;                        // Throw an int
throw new vector<double>;       // Throw a vector<double> *
throw 3.14159;                  // Throw a double
```

When you throw an exception, it can be caught by a `catch` clause specialized to catch that error. `catch` clauses are defined like this:

```
catch(ParameterType param) {
    /* Error-handling code */
}
```

Here, `ParameterType` represents the type of variable this `catch` clause is capable of catching. `catch` blocks must directly follow `try` blocks, and it's illegal to declare one without the other. Since catch clauses are specialized for a single type, it's perfectly legal to have cascading catch clauses, each designed to pick up a different type of exception. For example, here's code that catches exceptions of type `int`, `vector<int>`, and `string`:

```
    try {
       // Do something
    }
    catch(int myInt) {
       // If the code throws an int, execution continues here.
    }
    catch(const vector<int>& myVector) {
       // Otherwise, if the code throws a vector<int>, execution resumes here.
    }
    catch(const string& myString) {
       // Same for string
    }
```

Now, if the code inside the `try` block throws an exception, control will pass to the correct `catch` block. You can visualize exception handling as a room of people and a ball. The code inside the `try` block begins with the ball and continues talking as long as possible. If an error occurs, the `try` block throws the ball to the appropriate `catch` handler, which begins executing.

Let's return to our earlier example with `StringToInteger`. We want to signal an error in case the user enters an invalid parameter, and to do so we'd like to use exception handling. The question, though, is what type of object we should throw. While we can choose whatever type of object we'd like, C++ provides a header file, `<stdexcept>`, that defines several classes that let us specify what error triggered the exception. One of these, `invalid_argument`, is ideal for the situation. `invalid_argument` accepts in its constructor a `string` parameter containing a message representing what type of error occurred, and has a member function called `what` that returns what the error was.* We can thus rewrite the code for `StringToInteger` as

```
int StringToInteger(const string& input) {
   stringstream converter(input);
   int result; // Try reading an int, fail if we're unable to do so.

   converter >> result;
   if (converter.fail())
      throw invalid_argument("Cannot parse " + input + " as an integer.");

   char leftover; // See if anything's left over.  If so, fail.
   converter >> leftover;
   if (!converter.fail())
      return result;
   else
      throw invalid_argument(string("Unexpected character: ") + leftover);
}
```

Notice that while the function itself does not contain a `try`/`catch` pair, it nonetheless has a `throw` statement. If this statement is executed, then C++ will step backwards through all calling functions until it finds an appropriate `catch` statement. If it doesn't find one, then the program will halt with a runtime error. Now, we can write code using `StringToInteger` that looks like this:

* `what` is a poor choice of a name for a member function. Please make sure to use more descriptive names in your code!

```
try {
    int result = StringToInteger(myString);
    cout << "The result was: " << result;
}
catch(const invalid_argument& problem) {
    cout << problem.what() << endl; // Prints out the error message.
}
cout << "Yay!  We're done." << endl;
```

Here, if `StringToInteger` encounters an error and throws an exception, control will jump out of the `try` block into the `catch` clause specialized to catch objects of type `invalid_argument`. Otherwise, code continues as normal in the `try` block, then skips over the `catch` clause to print "Yay! We're done."

There are several things to note here. First, if `StringToInteger` throws an exception, control *immediately* breaks out of the `try` block and jumps to the `catch` clause. Unlike the problems we had with our earlier approach to error handling, here, if there is a problem in the `try` block, we're guaranteed that the rest of the code in the `try` block will not execute, preventing runtime errors stemming from malformed objects. Second, if there is an exception and control resumes in the `catch` clause, once the `catch` block finishes running, control does **not** resume back inside the `try` block. Instead, control resumes directly following the `try`/`catch` pair, so the program above will print out "Yay! We're done." once the `catch` block finishes executing. While this might seem unusual, remember that the reason for exception handling in the first place is to halt code execution in spots where no meaningful operation can be defined. Thus if control leaves a `try` block, chances are that the rest of the code in the `try` could not complete without errors, so C++ does not provide a mechanism for resuming program control. Third, note that we caught the `invalid_argument` exception by reference (`const invalid_argument&` instead of `invalid_argument`). As with parameter-passing, exception-catching can take values either by value or by reference, and by accepting the parameter by reference you can avoid making an unnecessary copy of the thrown object.

A Word on Scope

Exception handling is an essential part of the C++ programming language because it provides a system for recovering from serious errors. As its name implies, exception handling should be used only for *exceptional* circumstances – errors out of the ordinary that necessitate a major change in the flow of control. While you can use exception handling as a fancy form of function call and return, it is highly recommended that you avoid doing so. Throwing an exception is *much* slower than returning a value because of the extra bookkeeping required, so be sure that you're only using exception handling for serious program errors.

Also, the exception handling system will only respond when manually triggered. Unless a code snippet explicitly `throw`s a value, a `catch` block cannot respond to it. This means that you cannot use exception handling to prevent your program from crashing from segmentation faults or other pointer-based errors, since pointer errors result in operating-system level process termination, not C++-level exception handling.*

Programming with Exception Handling

While exception handling is a robust and elegant system, it has several sweeping implications for C++ code. Most notably, when using exception handling, unless you are absolutely certain that the classes and functions you use never throw exceptions, you must treat your code as though it might throw an exception

* If you use Microsoft's Visual Studio development environment, you might notice that various errors like null-pointer dereferences and stack overflows result in errors that mention "unhandled exception" in their description. This is a Microsoft-specific feature and is different from C++'s exception-handling system.

at any point. In other words, you can never assume that an entire code block will be completed on its own, and should be prepared to handle cases where control breaks out of your functions at inopportune times. For example, consider the following function:

```
void SimpleFunction() {
    int* myArray = new int[128];
    DoSomething(myArray);
    delete [] myArray;
}
```

Here, we allocate space for a raw array, pass it to a function, then deallocate the memory. While this code seems totally safe, when you introduce exceptions into the mix, this code can be very dangerous. What happens, for example, if `DoSomething` throws an exception? In this case, control would jump to the nearest `catch` block and the line `delete [] myArray` would never execute. As a result, our program will leak the array. If this program runs over a sufficiently long period of time, eventually we will run out of memory and our program will crash.

There are three main ways that we can avoid these problems. First, it's completely acceptable to just avoid exception-handling all together. This approach might seem like a cop-out, but it is a completely valid option that many C++ developers choose. Several major software projects written in C++ do not use exception handling (including the Mozilla Firefox web browser), partially because of the extra difficulties encountered when using exceptions. However, this approach results in code that runs into the same problems discussed earlier in this chapter with `StringToInteger` – functions can only communicate errors through return values and programmers must be extra vigilant to avoid ignoring return values.

The second approach to writing exception-safe code uses a technique called "catch-and-rethrow." Let's return to the above code example with a dynamically-allocated character buffer. We'd like to guarantee that the array we've allocated gets deallocated, but as our code is currently written, it's difficult to do so because the `DoSomething` function might throw an exception and interrupt our code flow. If there is an exception, what if we were able to somehow intercept that exception, clean up the buffer, and then propagate the exception outside of the `SimpleFunction` function? From an outside perspective, it would look as if the exception had come from inside the `DoSomething` function, but in reality it would have taken a quick stop inside `SimpleFunction` before proceeding outwards.

The reason this method works is that *it is legal to throw an exception from inside a `catch` block*. Although `catch` blocks are usually reserved for error handling, there is nothing preventing us from throwing the exception we catch. For example, this code is completely legal:

```
try{
    try {
        DoSomething();
    }
    catch(const invalid_argument& error) {
       cout << "Inner block: Error: " << error.what() << endl;
       throw error; // Propagate the error outward
    }
}
catch(const invalid_argument& error) {
    cout << "Outer block: Error:  " << error.what() << endl;
}
```

Here, if the `DoSomething` function throws an exception, it will first be caught by the innermost `try` block, which prints it to the screen. This `catch` handler then throws `error` again, and this time it is caught by the outermost `catch` block.

With this technique, we can almost rewrite our `SimpleFunction` function to look something like this:

```
void SimpleFunction() {
    int myArray = new int[128];

    /* Try to DoSomething.  If it fails, catch the exception and rethrow it. */
    try {
        DoSomething(myCString);
    }
    catch (/* What to catch? */) {
        delete [] myArray;
        throw /* What to throw? */;
    }

    /* Note that if there is no exception, we still need to clean things up. */
    delete [] myArray;
}
```

There's a bit of a problem here – what sort of exceptions should we catch? Suppose that we know every sort of exception `DoSomething` might throw. Would it be a good idea to write a `catch` block for each one of these types? At first this may seem like a good idea, but it can actually cause more problems than it solves. First, in each of the `catch` blocks, we'd need to write the same `delete []` statement. If we were to make changes to the `SimpleFunction` function that necessitated more cleanup code, we'd need to make progressively more changes to the `SimpleFunction catch` cascade, increasing the potential for errors. Also, if we forget to catch a specific type of error, or if `DoSomething` later changes to throw more types of errors, then we might miss an opportunity to catch the thrown exception and will leak resources. Plus, if we don't know what sorts of exceptions `DoSomething` might throw, this entire approach will not work.

The problem is that in this case, we want to tell C++ to catch *anything* that's thrown as an exception. We don't care about what the type of the exception is, and need to intercept the exception simply to ensure that our resource gets cleaned up. Fortunately, C++ provides a mechanism specifically for this purpose. To catch an exception of any type, you can use the special syntax `catch(...)`, which catches any exception. Thus we'll have the `catch` clause inside `DoSomething` be a `catch(...)` clause, so that we can catch any type of exception that `DoSomething` might throw. But this causes another problem: we'd like to rethrow the exception, but since we've used a `catch(...)` clause, we don't have a name for the specific exception that's been caught. Fortunately, C++ has a special use of the `throw` statement that lets you throw the current exception that's being processed. The syntax is

```
throw;
```

That is, a lone `throw` statement with no parameters. Be careful when using `throw;`, however, since if you're not inside of a `catch` block the program will crash!

The final version of `SimpleFunction` thus looks like this:

```
void SimpleFunction() {
        int myArray = new int[128];

        /* Try to DoSomething.  If it fails, catch the exception and rethrow it. */
        try {
            DoSomething(myCString);
        }
        catch (...) {
            delete [] myArray;
            throw;
        }

        /* Note that if there is no exception, we still need to clean things up. */
        delete [] myArray;
    }
```

As you can tell, the "catch-and-rethrow" approach to exception handling results in code that can be rather complicated. While in some circumstances catch-and-rethrow is the best option, in many cases there's a much better alternative that results in concise, readable, and thoroughly exception-safe code – object memory management.

Object Memory Management and RAII

C++'s memory model is best described as "dangerously efficient." Unlike other languages like Java, C++ does not have a garbage collector and consequently you must manually allocate and deallocate memory. At first, this might seem like a simple task – just `delete` anything you allocate with `new`, and make sure not to `delete` something twice. However, it can be quite difficult to keep track of all of the memory you've allocated in a program. After all, you probably won't notice any symptoms of memory leaks unless you run your programs for hours on end, and in all likelihood will have to use a special tool to check memory usage. You can also run into trouble where two objects each point to a shared object. If one of the objects isn't careful and accidentally `delete`s the memory while the other one is still accessing it, you can get some particularly nasty runtime errors where seemingly valid data has been corrupted. The situation gets all the more complicated when you introduce exception-handling into the mix, where the code to `delete` allocated memory might not be reached because of an exception.

In some cases having a high degree of control over memory management can be quite a boon to your programming, but much of the time it's simply a hassle. What if we could somehow get C++ to manage our memory for us? While building a fully-functional garbage collection system in C++ would be just short of impossible, using only basic C++ concepts it's possible to construct an excellent approximation of automatic memory management. The trick is to build *smart pointers*, objects that acquire a resource when created and that clean up the resource when destroyed. That is, when the objects are constructed, they wrap a newly-allocated pointer inside an object shell that cleans up the mess when the object goes out of scope. Combined with features like operator overloading, it's possible to create slick smart pointers that look almost exactly like true C++ pointers, but that know when to free unused memory.

The C++ header file `<memory>` exports the `auto_ptr` type, a smart pointer that accepts in its constructor a pointer to dynamically-allocated memory and whose constructor calls `delete` on the resource.* `auto_ptr` is a template class whose template parameter indicates what type of object the `auto_ptr` will "point" at. For example, an `auto_ptr<string>` is a smart pointer that points to a `string`. Be careful – if you write `auto_ptr<string *>`, you'll end up with an `auto_ptr` that points to a `string *`, which is similar to a `string **`. Through the magic of operator overloading, you can use the regular dereference and arrow operators on an `auto_ptr` as though it were a regular pointer. For example, here's some code

* Note that `auto_ptr` calls `delete`, not `delete []`, so you cannot store dynamically-allocated arrays in `auto_ptr`. If you want the functionality of an array with automatic memory management, use a `vector`.

that dynamically allocates a `vector<int>`, stores it in an `auto_ptr`, and then adds an element into the `vector`:

```
/* Have the auto_ptr point to a newly-allocated vector<int>.  The constructor
 * is explicit, so we must use parentheses.
 */
auto_ptr<vector<int> > managedVector(new vector<int>);
managedVector->push_back(137); // Add 137 to the end of the vector.
(*managedVector)[0] = 42; // Set element 0 by dereferencing the pointer.
```

While in many aspects `auto_ptr` acts like a regular pointer with automatic deallocation, `auto_ptr` is fundamentally different from regular pointers in assignment and initialization. Unlike objects you've encountered up to this point, assigning or initializing an `auto_ptr` to hold the contents of another *destructively modifies* the source `auto_ptr`. Consider the following code snippet:

```
auto_ptr<int> one(new int);
auto_ptr<int> two;
two = one;
```

After the final line executes, `two` will hold the resource originally owned by `one`, and `one` will be empty. During the assignment, `one` relinquished ownership of the resource and cleared out its state. Consequently, if you use `one` from this point forward, you'll run into trouble because it's not actually holding a pointer to anything. While this is highly counterintuitive, it has several advantages. First, it ensures that there can be at most one `auto_ptr` to a resource, which means that you don't have to worry about the contents of an `auto_ptr` being cleaned up out from underneath you by another `auto_ptr` to that resource. Second, it means that it's safe to return `auto_ptr`s from functions without the resource getting cleaned up. When returning an `auto_ptr` from a function, the original copy of the `auto_ptr` will transfer ownership to the new `auto_ptr` during return-value initialization, and the resource will be transferred safely.* Finally, because each `auto_ptr` can assume that it has sole ownership of the resource, `auto_ptr` can be implemented extremely efficiently and has almost zero overhead.

As a consequence of the "`auto_ptr` assignment is transference" policy, you must be careful when passing an `auto_ptr` by value to a function. Since the parameter will be initialized to the original object, it will empty the original `auto_ptr`. Similarly, you should not store `auto_ptr`s in STL containers, since when the containers reallocate or balance themselves behind the scenes they might assign `auto_ptr`s around in a way that will trigger the object destructors.

For reference, here's a list of the member functions of the `auto_ptr` template class:

---

* For those of you interested in programming language design, C++ uses what's known as *copy semantics* for most of its operations, where assigning objects to one another creates copies of the original objects. `auto_ptr` seems strange because it uses *move semantics*, where assigning `auto_ptr`s to one another transfers ownership of some resource. Move semantics are not easily expressed in C++ and the code to correctly implement `auto_ptr` is surprisingly complex and requires an intricate understanding of the C++ language. The next revision of C++, C++0x, will add several new features to the language to formalize and simply move semantics and will replace `auto_ptr` with `unique_ptr`, which formalizes the move semantics.

`explicit auto_ptr (Type* resource)`	`auto_ptr<int> ptr(new int);` Constructs a new `auto_ptr` wrapping the specified pointer, which must be from dynamically-allocated memory.

`auto_ptr(auto_ptr& other)`	`auto_ptr<int> one(new int);`
	`auto_ptr<int> two = one;`
	
	Constructs a new `auto_ptr` that acquires resource ownership from the `auto_ptr` used in the initialization. Afterwards, the old `auto_ptr` will not encapsulate any dynamically-allocated memory.

`T& operator *() const`	`*myAutoPtr = 137;` Dereferences the stored pointer and returns a reference to the memory it's pointing at.
`T* operator-> () const`	`myStringAutoPtr->append("C++!");` References member functions of the stored pointer.
`T* release()`	`int *regularPtr = myPtr.release();` Relinquishes control of the stored resource and returns it so it can be stored in another location. The `auto_ptr` will then contain a `NULL` pointer and will not manage the memory any more.

`void reset(T* ptr = NULL)`	`myPtr.reset();`
	`myPtr.reset(new int);`
	
	Releases any stored resources and optionally stores a new resource inside the `auto_ptr`.

`T* get() const`	`SomeFunction(myPtr.get()); // Retrieve stored resource` Returns the stored pointer. Useful for passing the managed resource to other functions.

Of course, dynamically-allocated memory isn't the only C++ resource that can benefit from object memory management. For example, when working with OS-specific libraries like Microsoft's Win32 library, you will commonly have to manually manage handles to system resources. In spots like these, writing wrapper classes that act like `auto_ptr` but that do cleanup using methods other than a plain `delete` can be quite beneficial. In fact, the system of having objects manage resources through their constructors and destructors is commonly referred to as *resource acquisition is initialization*, or simply RAII.

Exceptions and Smart Pointers

Up to this point, smart pointers might seem like a curiosity, or perhaps a useful construct in a limited number of circumstances. However, when you introduce exception handling to the mix, smart pointers will be invaluable. In fact, in professional code where exceptions can be thrown at almost any point, smart pointers are almost as ubiquitous as regular C++ pointers.

Let's suppose you're given the following linked list cell struct:

```
struct nodeT {
    int data;
    nodeT *next;
};
```

Now, consider this function:

```
nodeT* GetNewCell() {
    nodeT* newCell = new nodeT;
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell;
}
```

This function allocates a new `nodeT` cell, then tells it to hold on to the value returned by `SomeComplicatedFunction`. If we ignore exception handling, this code is totally fine, provided of course that the calling function correctly holds on to the `nodeT *` pointer we return. However, when we add exception handling to the mix, this function is a recipe for disaster. What happens if `SomeComplicatedFunction` throws an exception? Since `GetNewCell` doesn't have an associated `try` block, the program will abort `GetNewCell` and search for the nearest `catch` clause. Once the `catch` finishes executing, we have a problem – we allocated a `nodeT` object, but we didn't clean it up. Worse, since `GetNewCell` is no longer running, we've lost track of the `nodeT` entirely, and the memory is orphaned.

Enter `auto_ptr` to save the day. Suppose we change the declaration `nodeT* newCell` to `auto_ptr<nodeT> newCell`. Now, if `SomeComplicatedFunction` throws an exception, we won't leak any memory since when the `auto_ptr` goes out of scope, it will reclaim the memory for us. Wonderful! Of course, we also need to change the last line from `return newCell` to `return newCell.release()`, since we promised to return a `nodeT *`, not an `auto_ptr<nodeT>`. The new code is printed below:

```
nodeT* GetNewCell() {
    auto_ptr<nodeT> newCell(new nodeT);
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell.release(); // Tell the auto_ptr to stop managing memory.
}
```

This function is now wonderfully exception-safe thanks to `auto_ptr`. Even if we prematurely exit the function from an exception in `SomeComplicatedFunction`, the `auto_ptr` destructor will ensure that our resources are cleaned up. However, we can make this code even safer by using the `auto_ptr` in yet another spot. What happens if we call `GetNewCell` but don't store the return value anywhere? For example, suppose we have a function like this:

```
void SillyFunction() {
    GetNewCell(); // Oh dear, there goes the return value.
}
```

When we wrote `GetNewCell`, we tacitly assumed that the calling function would hold on to the return value and clean the memory up at some later point. However, it's totally legal to write code like `SillyFunction` that calls `GetNewCell` and entirely discards the return value. This leads to memory leaks, the very problem we were trying to solve earlier. Fortunately, through some creative use of `auto_ptr`, we can eliminate this problem. Consider this modified version of `GetNewCell`:

```
auto_ptr<nodeT> GetNewCell() {
    auto_ptr<nodeT> newCell(new nodeT);
    newCell->next = NULL;
    newCell->data = SomeComplicatedFunction();
    return newCell; // See below
}
```

Here, the function returns an `auto_ptr`, which means that the returned value is itself managed. Now, if we call `SillyFunction`, even though we didn't grab the return value of `GetNewCell`, because `GetNewCell` returns an `auto_ptr`, the memory will still get cleaned up.

Documenting Invariants with `assert`

The exception-handling techniques we've covered so far are excellent ways of handling and recovering from errors that can only be detected at compile-time. If a network connection fails to open, or your graphics card fails to initialize correctly, you can use exceptions to report the error so that your program can detect and recover from the problem.

However, there is an entirely different class of problems that your programs might encounter at runtime – logic errors. As much as we'd all like to think that we can write perfect software on the first try, we all make mistakes when designing programs. We pass `NULL` pointers into functions that expect them to be non-`NULL`. We make accidental changes to linked lists while iterating over them. We pass in values by reference that we meant to pass in by value. These are normal errors in the programming process, and while time and experience can reduce their frequency, they can never entirely be eliminated. The question then arises – given that you are going to make mistakes during development, how can you design your software to make it easier to detect and correct these errors?

When designing software, at various points in the program you will expect certain conditions to hold true. You might expect that a certain integer is even, or that a pointer is non-`NULL`, etc. If these conditions don't hold, it's often a sign that your program contains a bug.

One trick you can use to make it easier to detect and diagnose bugs is to have the program check that these invariants hold at runtime. If they do, then everything is going according to plan, but if for some reason the invariants do not hold it could signal the presence of a bug. If the program can then report that an invariant failed to hold, it will make it significantly easier to debug. For this purpose, C++ provides the `assert` macro. `assert`, exported by the header `<cassert>`, checks to see that some condition holds true. If so, the macro has no effect. Otherwise, it prints out the statement that did not evaluate to true, along with the file and line number in which it was written, then terminates the program. For example, consider the following code:

```
void MyFunction(int *myPtr) {
    assert(myPtr != NULL);
    *myPtr = 137;
}
```

If a caller passes a null pointer into `MyFunction`, the assert statement will halt the program and print out a message that might look something like this:

```
Assertion Failed: 'myPtr != NULL': File: main.cpp, Line: 42
```

Because assert abruptly terminates the program without giving the rest of the application a chance to respond, you should not use `assert` as a general-purpose error-handling routine. In practical software development, `assert` is usually used to express programmer assumptions about the state of execution that can only be broken if the software is written incorrectly. If an `assert` fails, it means that the programmer made a mistake, not that something unusual occurred at runtime. For errors that might arise during normal execution, such as missing files or malformed user input, user exception handling. For errors that represent a bug in the original code, `assert` is a much better choice.

Let's consider a concrete example. Assume we have some enumerated type `Color`, which might look like this:

```
enum Color {Red, Green, Blue, Magenta, Cyan, Yellow, Black, White};
```

Now, suppose that we want to write a function called `IsPrimaryColor` that takes in a `Color` and reports whether that color is a primary color (red, green, or blue). Here's one implementation:

```
bool IsPrimaryColor(Color c) {
    switch(c) {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            return false;
    }
}
```

Here, if the color is `Red`, `Green`, or `Blue`, we return that the color is indeed a primary color. Otherwise, we return that it is not a primary color. However, what happens if the parameter is not a valid `Color`, perhaps if the call is `IsPrimaryColor(Color(-1))`? In this function, since we assume that that the parameter is indeed a color, we might want to indicate that to the program by explicitly putting in an `assert` test. Here's a modified version of the function, using assert and assuming the existence of a function `IsColor`:

```
bool IsPrimaryColor(Color c) {
    assert(IsColor(c)); // We assume that this is really a color.
    switch (c) {
        case Red:
        case Green:
        case Blue:
            return true;
        default:
            /* Otherwise, must not be a primary color. */
            return false;
    }
}
```

Now, if the caller passes in an invalid `Color`, the program will halt with an assertion error pointing us to the line that caused the problem. If we have a good debugger, we should be able to figure out which caller erroneously passed in an invalid Color and can better remedy the problem. Were we to ignore this case entirely, we might have considerably more trouble debugging the error, since we would have no indication of where the problem originated.

While assert can be used to catch a good number of programmer errors during development, it has the unfortunate side-effect of slowing a program down at runtime because of the overhead of the extra checking involved. Consequently, most major compilers disable the `assert` macro in release or optimized builds. This may seem dangerous, since it eliminates checks for inconsistent state, but is actually not a problem because, in theory, you shouldn't be compiling a release build of your program if assert statements fail during execution.* Because `assert` is entirely disabled in optimized builds, you should use `assert` only to check that specific relations hold true, never to check the return value of a function. If an `assert` contains a call to a function, when assert is disabled in release builds, the function won't be called,

* In practice, this isn't always the case. But it's still a nice theory!

leading to different behavior in debug and release builds. This is a persistent source of debugging headaches.

More to Explore

Exception-handling and RAII are complex topics that have impressive ramifications for the way that your write C++ code. However, we simply don't have time to cover every facet of exception handling. In case you're interested in exploring more advanced topics in exception handling and RAII, consider looking into the following:

1. **The Standard Exception Classes**: In this chapter we discussed `invalid_argument`, one of the many exception classes available in the C++ standard library. However, there are several more exception classes that form an elaborate hierarchy. Consider reading into some of the other classes – some of them even show up in the STL!

2. **Exception Specifications**. Because functions can throw exceptions at any time, it can be difficult to determine which pieces of code can and cannot throw exceptions. Fortunately, C++ has a feature called an *exception specification* which indicates what sorts of exceptions a function is allowed to throw. When an exception leaves a function with an exception specification, the program will abort unless the type of the exception is one of the types mentioned in the specification.

3. **Function `try` Blocks**. There is a variant of a regular try block that lets you put the entire contents of a function into a try/catch handler pair. However, it is a relatively new feature in C++ and is not supported by several popular compilers. Check a reference for more information.

4. **`new` and Exceptions**. If your program runs out of available memory, the `new` operator will indicate a failure by throwing an exception of type `bad_alloc`. When designing custom container classes, it might be worth checking against this case and acting accordingly.

5. **The Boost Smart Pointers**: While `auto_ptr` is useful in a wide variety of circumstances, in many aspects it is limited. Only one `auto_ptr` can point to a resource at a time, and `auto_ptr`s cannot be stored inside of STL containers. The Boost C++ libraries consequently provide a huge number of smart pointers, many of which employ considerably more complicated resource-management systems than `auto_ptr`. Since many of these smart pointers are likely to be included in the next revision of the C++ standard, you should be sure to read into them.

Bjarne Stroustrup (the inventor of C++) wrote an excellent introduction to exception safety, focusing mostly on implementations of the C++ Standard Library. If you want to read into exception-safe code, you can read it online at http://www.research.att.com/~bs/3rd_safe.pdf. Additionally, there is a most excellent reference on `auto_ptr` available at http://www.gotw.ca/publications/using_auto_ptr_effectively.htm that is a great resource on the subject.

Practice Problems

1. Explain why the `auto_ptr` constructor is marked `explicit`. *(Hint: Give an example of an error you can make if the constructor is not marked `explicit`).*

2. The `SimpleFunction` function from earlier in this chapter ran into difficulty with exception-safety because it relied on a manually-managed C string. Explain why this would not be a problem if it instead used a C++ `string`.

3. Consider the following C++ function:

```
void ManipulateStack(stack<string>& myStack) {
    if (myStack.empty())
        throw invalid_argument("Empty stack!");

    string topElem = myStack.top();
    myStack.pop();

    /* This might throw an exception! */
    DoSomething(myStack);

    myStack.push(topElem);
}
```

This function accepts as input a C++ `stack<string>`, pops off the top element, calls the `DoSomething` function, then pushes the element back on top. Provided that the `DoSomething` function doesn't throw an exception, this code will guarantee that the top element of the `stack` does not change before and after the function executes. Suppose, however, that we wanted to absolutely guarantee that the top element of the stack never changes, even if the function throws an exception. Using the catch-and-rethrow strategy, explain how to make this the case.

5. Write a class called `AutomaticStackManager` whose constructor accepts a `stack<string>` and pops off the top element (if one exists) and whose destructor pushes the element back onto the `stack`. Using this class, rewrite the code in Problem 4 so that it's exception safe. How does this version of the code compare to the approach using catch-and-rethrow?

Part Three

Generic Programming

Chapter 13: Functors

Consider a simple task. Suppose you have a `vector<string>` and you'd like to count the number of `string`s that have length less than five. You stumble upon the STL `count_if` algorithm, which accepts a range of iterators and a predicate function, then returns the number of elements in the range for which the function returns true. For example, you could use `count_if` as follows to count the number of even integers in a `vector`:

```
bool IsEven(int val) {
    return val % 2 == 0;
}

vector<int> myVector = /* ... */
int numEvens = count_if(myVector.begin(), myVector.end(), IsEven);
```

In our case, since we want to count the number of strings with length less than five, we could write a function like this one:

```
bool LengthIsLessThanFive(const string& str) {
    return str.length() < 5;
}
```

And then call `count_if(myVector.begin(), myVector.end(), LengthIsLessThanFive)` to get the number of short strings in the `vector`. Similarly, if we want to count the number of strings with length less than ten, we could write a `LengthIsLessThanTen` function like this one:

```
bool LengthIsLessThanTen(const string& str) {
    return str.length() < 10;
}
```

and then call `count_if(myVector.begin(), myVector.end(), LengthIsLessThanTen)`. In general, if we know in advance what length we want to compare the string lengths against, we can write a function that returns whether a particular string's length is less than that value, then pass it into `count_if` to get our result. This approach is legal C++, but is not particularly elegant. Every time we want to compare the string length against a particular value, we have to write an entirely new function to perform the comparison. Good programming practice suggests that we should instead just write *one* function that looks like this:

```
bool LengthIsLessThan(const string& str, size_t length) {
    return str.length() < length;
}
```

This more generic function takes in a string and a length, then returns whether the string's length is less than the requested length. This way, we can specify the maximum length as the second parameter rather than writing multiple instances of similar functions.

While this new function is more generic than the previous version, unfortunately we can't use it in conjunction with `count_if`. `count_if` requires a *unary* function (a function taking only one argument) as its final parameter, and the new `LengthIsLessThan` is a *binary* function. Our new `LengthIsLessThan` function, while more generic than the original version, is actually *less useful* in this context. There must be some way to compromise between the two approaches. We need a way to construct a function that takes

in only one parameter (the string to test), but which can be customized to accept an arbitrary maximum length. How can we do this?

This problem boils down to a question of data flow. To construct this hybrid function, we need to somehow communicate the upper bound into the function so that it can perform the comparison. So how can we give this data to the function? Recall that a function has access the following information:

- Its local variables.
- Its parameters.
- Global variables.

Is there some way that we can store the maximum length of the string in one of these locations? We can't store it in a local variable, since local variables don't persist between function calls and aren't accessible to callers. As mentioned above, we also can't store it in a parameter, since `count_if` is hardcoded to accept a unary function. That leaves global variables. We *could* solve this problem using global variables: we would store the maximum length in a global variable, then compare the string parameter length against the global. For example:

```
size_t gMaxLength; // Value to compare against

bool LengthIsLessThan(const string& str) {
    return str.length() < gMaxLength;
}
```

This approach works: if our `vector<string>` is called `v`, then we can count the number of elements less than some value by writing

```
gMaxLength = /* ... some value ... */
int numShort = count_if(v.begin(), v.end(), LengthIsLessThan);
```

But just because this approach works does not mean that it is optimal. This approach is deeply flawed for several reasons, a handful of which are listed here:

- **It is error-prone**. Before we use `LengthIsLessThan`, we must take care to set `gMaxLength` to the maximum desired length. If we forget to do so, then `LengthIsLessThan` will use the wrong value in the comparison and we will get the wrong answer. Moreover, because there is no formal relationship between the `gMaxLength` variable and the `LengthIsLessThan` function, the compiler can't verify that we correctly set `gMaxLength` before calling `LengthIsLessThan`, putting an extra burden on the programmer

- **It is not scalable**. If every time we encounter a problem like this one we create a new global variable, programs we write will begin to fill up with global variables that are used only in the context of a single function. This leads to *namespace pollution*, where too many variables are in scope and it is easy to accidentally use one when another is expected.

- **It uses global variables**. Any use of global variables should send a shiver running down your spine. Global variables should be avoided at all costs, and the fact that we're using them here suggests that something is wrong with this setup.

None of the options we've considered are feasible or attractive. There has to be a better way to solve this, but how?

Functors to the Rescue

The fundamental issue at heart here is that a unary function does not have access to enough information to answer the question we're asking. Essentially, we want a unary function to act like a binary function without taking an extra parameter. Using only the tools we've seen so far, this simply isn't possible. To solve this problem, we'll turn to a more powerful C++ entity: a *functor*. A functor (or *function object*) is an C++ class that acts like a function. Functors can be called using the familiar function call syntax, and can yield values and accept parameters just like regular functions. For example, suppose we create a functor class called `MyClass` imitating a function accepting an `int` and returning a `double`. Then we could "call" an object of type `MyClass` as follows:

```
MyClass myFunctor;
cout << myFunctor(137) << endl; // "Call" myFunctor with parameter 137
```

Although `myFunctor` is an object, in the second line of code we treat it as though it were a function by invoking it with the parameter 137.

At this point, functors might seem utterly baffling: why would you ever want to create an object that behaves like a function? Don't worry, we'll answer that question in a short while. In the meantime, we'll discuss the syntax for functors and give a few motivating examples.

To create a functor, we create an object that overloads the function call operator, `operator ()`. The name of this function is a bit misleading – it is a function called `operator ()`, not a function called `operator` that takes no parameters. Despite the fact that the name looks like "operator parentheses," we're not redefining what it means to parenthesize the object. Instead, we're defining a function that gets called if we invoke the object like a function. Thus in the above code,

```
cout << myFunctor(137) << endl;
```

is equivalent to

```
cout << myFunctor.operator()(137) << endl;
```

Unlike other operators we've seen so far, when overloading the function call operator, you're free to return an object of any type (or even `void`) and can accept any number of parameters. Remember that the point of operator overloading is to allow objects to act like built-in types, and since a regular function can have arbitrarily many parameters and any return type, functors are allowed the same freedom. For example, here's a sample functor that overloads the function call operator to print out a string:

```
class MyFunctor {
public:
    void operator() (const string& str) const {
        cout << str << endl;
    }
};
```

Note that in the function definition there are two sets of parentheses. The first group is for the function name – `operator ()` – and the second for the parameters to `operator ()`. If we separated the implementation of `operator ()` from the class definition, it would look like this:

```
class MyFunctor {
public:
    void operator() (const string& str) const;
};

void MyFunctor::operator() (const string& str) const {
    cout << str << endl;
}
```

Now that we've written `MyFunctor`, we can use it as follows:

```
MyFunctor functor;
functor("Functor power!");
```

This code calls the functor and prints out "Functor power!"

At this point functors might seem like little more than a curiosity. "Sure," you might say, "I can make an object that can be called like a function. But what does it buy me?" A lot more than you might initially suspect, it turns out. The key difference between a function and a functor is that a functor's function call operator is a *member function* whereas a raw C++ function is a *free function*. This means that a functor can access the following information when being called:

- Its local variables.
- Its parameters.
- Global variables.
- **Class data members**.

This last point is extremely important and is the key difference between a regular function and a functor. If a functor's `operator()` member function requires access to data beyond what can be communicated by its parameters, we can store that information as a data member inside the functor class. Since `operator()` is a member of the functor class, it can then access that data freely. For example, consider the following functor class:

```
class StringAppender {
public:
    /* Constructor takes and stores a string. */
    explicit StringAppender(const string& str) : toAppend(str) {}

    /* operator() prints out a string, plus the stored suffix. */
    void operator() (const string& str) const {
        cout << str << ' ' << toAppend << endl;
    }

private:
    const string toAppend;
};
```

This functor's constructor takes in a string and stores it for later use. Its `operator ()` function accepts a string, then prints that string suffixed with the string stored by the constructor. We can then use the `StringAppender` functor like this:

```
StringAppender myFunctor("is awesome");
myFunctor("C++");
```

This code will print out "C++ is awesome," since the constructor stored the string "is awesome" and we passed "C++" as a parameter to the function. If you'll notice, though, in the actual function call we only passed in one piece of information – the string "C++." This is precisely why functors are so useful. Like regular functions, functors are invoked with a fixed number of parameters. Unlike raw functions, however, functors can be constructed to store as much information is necessary to solve the task at hand.

Let's return to the above example with `count_if`. Somehow we need to provide a unary function that can return whether a string is less than an arbitrary length. To solve this problem, instead of writing a unary function, we'll create a unary *functor* whose constructor stores the maximum length and whose `operator ()` accepts a string and returns whether it's of the correct length. Here's one possible implementation:

```
class ShorterThan {
public:
    /* Accept and store an int parameter */
    explicit ShorterThan(size_t maxLength) : length(maxLength) {}

    /* Return whether the string length is less than the stored int. */
    bool operator() (const string& str) const {
        return str.length() < length;
    }

private:
    const size_t length;
};
```

In this code, the constructor accepts a single `size_t`, then stores it as the `length` data member. From that point forward, whenever the functor is invoked on a particular string, the functor's `operator()` function can compare the length of that string against `length` data member. This is exactly what we want – a unary function that knows what value to compare the parameter's length against. To tie everything together, here's the code we'd use to count the number of strings in the `vector` that are shorter than the specified value:

```
ShorterThan st(length);
count_if(myVector.begin(), myVector.end(), st);
```

Functors are incredible when combined with STL algorithms for this very reason – they look and act like regular functions, but have access to extra information. This is just our first taste of functors, as we continue our exploration of C++ you will recognize exactly how much they will influence your program design.

Look back to the above code with `count_if`. If you'll notice, we created a new `ShorterThan` object, then fed it to `count_if`. After the call to `count_if`, odds are we'll never use that particular `ShorterThan` object again. This is an excellent spot to use temporary objects, since we need a new `ShorterThan` for the function call but don't plan on using it afterwards. Thus, we can convert this code:

```
ShorterThan st(length)
count_if(myVector.begin(), myVector.end(), st);
```

Into this code:

```
count_if(myVector.begin(), myVector.end(), ShorterThan(length));
```

Here, `ShorterThan(length)` constructs a temporary `ShorterThan` functor with parameter `length`, then passes it to the `count_if` algorithm. Don't get tripped up by the syntax – `ShorterThan(length)` does *not* call the `ShorterThan`'s `operator ()` function. Instead, it invokes the `ShorterThan` constructor with

the parameter `length` to create a temporary object. Even if we had written the `operator()` function to take in an `int`, C++ would realize that the parentheses here means "construct an object" instead of "invoke `operator()`" from context.

Writing Functor-Compatible Code

In previous chapters, you've seen how to write code that accepts a function pointer as a parameter. For example, the following code accepts a function that takes and returns a `double`, then prints a table of some values of that function:

```
const double kLowerBound = 0.0;
const double kUpperBound = 1.0;
const int    kNumSteps   = 25;
const double kStepSize   = (kUpperBound - kLowerBound) / kNumSteps;

void TabulateFunctionValues(double function(double)) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

For any function accepting and returning a `double`, we can call `TabulateFunctionValues` with that function as an argument. But what about functors? Can we pass them to `TabulateFunctionValues` as well? As an example, consider the following implementation of a `Reciprocal` functor, whose `operator()` takes in a `double` and returns the reciprocal of that `double`:

```
class Reciprocal {
public:
    double operator() (double val) const {
        return 1.0 / val;
    }
};
```

Given this class implementation, is the following code legal?

```
TabulateFunctionValues(Reciprocal());
```

(Recall that `Reciprocal()` constructs a temporary `Reciprocal` object for use as the parameter to `TabulateFunctionValues`.)

At a high level, this code seems perfectly fine. After all, `Reciprocal` objects can be called as though they were unary functions taking and returning `double`s, so it seems perfectly reasonable to pass a `Reciprocal` into `TabulateFunctionValues`. But despite the similarities, `Reciprocal` is *not* a function – it's a funct*or* – and so the above code will not compile. The problem is that C++'s static type system prevents function pointers from pointing to functors, even if the functor has the same parameter and return type as the function pointer. This is not without reason – the machine code for calling a function is very different from machine code for calling a functor, and if C++ were to conflate the two it would result either in slower function calls or undefined runtime behavior.

Given that this code doesn't compile, how can we fix it? Let's begin with some observations, then generalize to the optimal solution. The above code does not compile because we're trying to provide a `Reciprocal` object to a function expecting a function pointer. This suggests one option – could we rewrite the `TabulateFunctionValues` function such that it accepts a `Reciprocal` as a parameter instead of a function pointer? For example, we could write the following:

```
void TabulateFunctionValues(Reciprocal function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Now, if we call the function as

```
TabulateFunctionValues(Reciprocal());
```

The code is perfectly legal because the argument has type `Reciprocal` and the `TabulateFunctionValues` function is specifically written to take in objects of type `Reciprocal`. But what if we have another functor we want to use in `TabulateFunctionValues`? For example, we might write a functor called `Arccos` that computes the inverse cosine of its parameter, as seen here:

```
class Arccos {
public:
    double operator() (double val) const {
        return acos(val); // Using the acos function from <cmath>
    }
};
```

Unfortunately, if we try to call `TabulateFunctionValues` passing in an `Arccos` object, as shown here:

```
TabulateFunctionValues(Arccos());
```

we'll get yet *another* compile-time error, this time because the `TabulateFunctionValues` function is hardcoded to accept a `Reciprocal`, but we've tried to provide it an object of type `Arccos`. Again, if we rewrite `TabulateFunctionValues` to only accept objects of type `Arccos`, we could alleviate this problem. Of course, in doing so, we would break all code that accepted objects of type `Reciprocal`. How can we resolve this problem? Fortunately, the answer is yes, thanks to a particularly ingenious trick. Below are three versions of `TabulateFunctionValues`, each of which take in a parameter of a different type:

```
void TabulateFunctionValues(double function(double)) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
void TabulateFunctionValues(Reciprocal function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
void TabulateFunctionValues(Arccos function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Notice that the only difference between the three implementations of `TabulateFunctionValues` is the type of the parameter to the function. The rest of the code is identical. This suggests a rather elegant solution using templates. Instead of providing multiple different versions of `TabulateFunctionValues`, each specialized for a particular type of function or functors, we'll write a single *template* version of `TabulateFunctionValues` parameterized over the type of the argument. This is shown here:

```
template <typename UnaryFunction>
    void TabulateFunctionValues(UnaryFunction function) {
    for(double i = kLowerBound; i <= kUpperBound; i += kStepSize)
        cout << "f(" << i << ") = " << function(i) << endl;
}
```

Now, we can pass any type of object to `TabulateFunctionValues` that we want, provided that the argument can be called with a single `double` as a parameter to produce a value. This means that we can pass in raw functions, `Reciprocal` objects, `Arccos` objects, and any other functor classes that happen to mimic functions from `double`s to `double`s. This hearkens back to our discussion of concepts in the previous chapter. By writing `TabulateFunctionValues` as a template function parameterized over an arbitrary type, we let clients provide objects of whatever type they see fit, as long as it can be called as a function taking a `double` and returning a `double`.

When writing functions that require a user-specified callback, you may want to consider parameterizing the function over the type of the callback instead of using function pointers. The resulting code will be more flexible and future generations of programmers will be much the better for your extra effort.

STL Algorithms Revisited

Now that you're armed with the full power of C++ functors, let's revisit some of the STL algorithms we've covered and discuss how to maximize their firepower.

The very first algorithm we covered was `accumulate`, defined in the `<numeric>` header. If you'll recall, `accumulate` sums up the elements in a range and returns the result. For example, given a `vector<int>`, the following code returns the sum of all of the `vector`'s elements:

```
accumulate(myVector.begin(), myVector.end(), 0);
```

The first two parameters should be self-explanatory, and the third parameter (zero) represents the initial value of the sum.

However, this view of `accumulate` is limited, and to treat `accumulate` as simply a way to sum container elements would be an error. Rather, *`accumulate` is a general-purpose function for transforming a collection of elements into a single value.*

There is a second version of the `accumulate` algorithm that takes a binary function as a fourth parameter. This version of `accumulate` is implemented like this:

```
template <typename InputIterator, typename Type, typename BinaryFn>
inline Type accumulate(InputIterator start,
                       InputIterator stop,
                       Type accumulator,
                       BinaryFn fn) {
   while(start != stop) {
      accumulator = fn(accumulator, *start);
      ++start;
   }
   return initial;
}
```

This `accumulate` iterates over the elements of a container, calling the binary function on the accumulator and the current element of the container and storing the result back in the accumulator. In other words, `accumulate` continuously updates the value of the accumulator based on its initial value and the values

contained in the input range. Finally, `accumulate` returns the accumulator. Note that the version of `accumulate` we encountered earlier is actually a special case of the above version where the provided callback function computes the sum of its parameters.

To see `accumulate` in action, let's consider an example. Recall that the STL algorithm `lower_bound` returns an iterator to the first element in a range that compares greater than or equal to some value. However, `lower_bound` requires the elements in the iterator range to be in sorted order, so if you have an unsorted `vector`, you cannot use `lower_bound`. Let's write a function `UnsortedLowerBound` that accepts a range of iterators and a lower bound, then returns the *value* of the least element in the range greater than or equal to the lower bound. For simplicity, let's assume we're working with a `vector<int>` so that we don't get bogged down in template syntax, though this approach can easily be generalized.

Although this function can be implemented using loops, we can leverage off of `accumulate` to come up with a considerably more concise solution. Thus, we'll define a functor class to pass to `accumulate`, then write `UnsortedLowerBound` as a wrapper call to `accumulate` with the proper parameters.
Consider the following functor:

```
class LowerBoundHelper {
public:
    explicit LowerBoundHelper(int lower) : lowestValue(lower) {}
    int operator() (int bestSoFar, int current) {
        return current >= lowestValue && current < bestSoFar?
               current : bestSoFar;
    }

private:
    const int lowestValue;
};
```

This functor's constructor accepts the value that we want to lower-bound. Its `operator ()` function accepts two `int`s, the first representing the lowest known value greater than `lowestValue` and the second the current value. If the value of the current element is greater than or equal to the lower bound and also less than the best value so far, `operator ()` returns the value of the current element. Otherwise, it simply returns the best value we've found so far. Thus if we call this functor on every element in the `vector` and keep track of the return value, we should end up with the lowest value in the `vector` greater than or equal to the lower bound. We can now write the `UnsortedLowerBound` function like this:

```
int UnsortedLowerBound(const vector<int>& input, int lowerBound) {
    return accumulate(input.begin(), input.end(),
                      numeric_limits<int>::max(),
                      LowerBoundHelper(lowerBound));
}
```

Our entire function is simply a wrapped call to `accumulate`, passing a specially-constructed `LowerBoundHelper` object as a parameter. Note that we've used the value `numeric_limits<int>::max()` as the initial value for the accumulator. `numeric_limits`, defined in the `<limits>` header, is a traits class that exports useful information about the bounds and behavior of numeric types, and its `max` static member function returns the maximum possible value for an element of the specified type. We use this value as the initial value for the accumulator since any integer is less than it, so if the range contains no elements greater than the lower bound we will get `numeric_limits<int>::max()` back as a sentinel.

If you need to transform a range of values into a single result (of any type you wish), use `accumulate`. To transform a range of values into another range of values, use `transform`. We discussed transform briefly

in the chapter on STL algorithms in the context of `ConvertToUpperCase` and `ConvertToLowerCase`, but such examples are just the tip of the iceberg. `transform` is nothing short of a miracle function, and it arises a whole host of circumstances.*

Higher-Order Programming

This discussion of functors was initially motivated by counting the number of short `string`s inside of an STL `vector`. We demonstrated that by using `count_if` with a custom functor as the final parameter, we were able to write code that counted the number of elements in a `vector<string>` whose length was less than a certain value. But while this code solved the problem efficiently, we ended up writing so much code that any potential benefits of the STL algorithms were dwarfed by the time spent writing the functor. For reference, here was the code we used:

```
class ShorterThan {
public:
    explicit ShorterThan(size_t maxLength) : length(maxLength) {}
    bool operator() (const string& str) const {
        return str.length() < length;
    }

private:
    size_t length;
};

const size_t myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), ShorterThan(myValue));
```

Consider the following code which also solves the problem, but by using a simple `for` loop:

```
const int myValue = GetInteger();
int total = 0;
for(int i = 0; i < myVector.size(); ++i)
    if(myVector[i].length() < myValue) ++total;
```

This code is considerably more readable than the functor version and is approximately a third as long. By almost any metric, this code is superior to the earlier version.

If you'll recall, we were motivated to write this `ShorterThan` functor because we were unable to use `count_if` in conjunction with a traditional C++ function. Because `count_if` accepts as a parameter a unary function, we could not write a C++ function that could accept both the current container element and the value to compare its length against. However, we did note that were `count_if` to accept a binary function and extra client data, then we could have written a simple C++ function like this one:

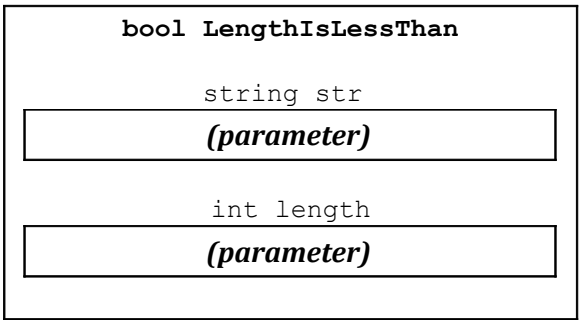
```
bool LengthIsLessThan(const string& str, int threshold) {
    return str.length() < threshold;
}
```

And then passed it in, along with the cutoff length, to the `count_if` function.

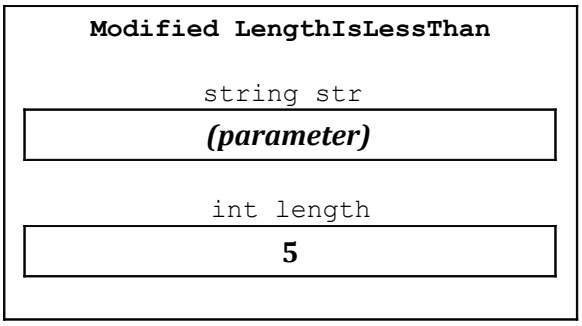
The fundamental problem is that the STL `count_if` algorithm requires a single-parameter function, but the function we want to use requires two pieces of data. We want the STL algorithms to use our two-parameter function `LengthIsLessThan`, but with the second parameter always having the same

* Those of you familiar with functional programming might recognize `accumulate` and `transform` as the classic higher-order functions Map and Reduce.

value. What if somehow we could modify `LengthIsLessThan` by "locking in" the second parameter? In other words, we'd like to take a function that looks like this:



And transform it into another function that looks like this:



Now, if we call this special version of `LengthIsLessThan` with a single parameter (call it `str`), it would be as though we had called the initial version of `LengthIsLessThan`, passing as parameters the value of `str` and the stored value 5. This then returns whether the length of the `str` string is less than 5. Essentially, by binding the second parameter of the two-parameter `LengthIsLessThan` function, we end up with a one-parameter function that describes exactly the predicate function we want to provide to `count_if`. Thus, at a high level, the code we want to be able to write should look like this:

```
count_if(v.begin(), v.end(),
         the function formed by locking 5 as the second parameter of LengthIsLessThan);
```

This sort of programming, where functions can be created and modified just like regular objects, is known as *higher-order programming*. While by default C++ does not support higher-order programming, using functors and the STL functional programming libraries, in many cases it is possible to write higher-order code in C++. In the remainder of this chapter, we'll explore the STL functional programming libraries and see how to use higher-order programming to supercharge STL algorithms.

Adaptable Functions

To provide higher-order programming support, standard C++ provides the `<functional>` library. `<functional>` exports several useful functions that can transform and modify functions on-the-fly to yield new functions more suitable to the task at hand. However, because of several language limitations, the `<functional>` library can only modify specially constructed functions called "adaptable functions," funct*ors* (not regular C++ functions) that export information about their parameter and return types. Fortunately, any one- or two-parameter function can easily be converted into an equivalent adaptable function. For example, suppose you want to make an adaptable function called `MyFunction` that takes a `string` by reference-to-`const` as a parameter and returns a `bool`, as shown below:

```
class MyFunction {
public:
    bool operator() (const string& str) const {
        /* Function that manipulates a string */
    }
};
```

Now, to make this function an adaptable function, we need to specify some additional information about the parameter and return types of this functor's `operator ()` function. To assist in this process, the functional library defines a helper template class called `unary_function`, which is prototyped below:

```
template <typename ParameterType, typename ReturnType>
   class unary_function;
```

The first template argument represents the type of the parameter to the function; the second, the function's return type.

Unlike the other classes you have seen before, the `unary_function` class contains no data members and no member functions. Instead, it performs some behind-the-scenes magic with the `typedef` keyword to export the information expressed in the template types to the rest of the functional programming library. Since we want our above functor to also export this information, we'll inheritance to import all of the information from `unary_function` into our `MyFunction` functor. Because `MyFunction` accepts as a parameter an object of type `string` and returns a variable of type `bool`, we will have `MyFunction` inherit from the type `unary_function<string, bool>`. The syntax to accomplish this is shown below:

```
class MyFunction : public unary_function<string, bool> {
public:
    bool operator() (const string& str) const {
        /* Function that manipulates a string */
    }
};
```

We'll explore inheritance in more detail later, but for now just think of it as a way for importing information from class into another. Note that although the function accepts as its parameter a `const string&`, we chose to use a `unary_function` specialized for the type `string`. The reason is somewhat technical and has to do with how `unary_function` interacts with other functional library components, so for now just remember that you should not specify reference-to-`const` types inside the `unary_function` template parametrization.

The syntax for converting a binary functor into an adaptable binary function works similarly to the above code for unary functions. Suppose that we'd like to make an adaptable binary function that accepts a `string` and an `int` and returns a `bool`. We begin by writing the basic functor code, as shown here:

```
class MyOtherFunction {
public:
    bool operator() (const string& str, int val) const {
        /* Do something, return a bool. */
    }
};
```

To convert this functor into an adaptable function, we'll have it inherit from `binary_function`. Like `unary_function`, `binary_function` is a template class that's defined as

```
template <typename Param1Type, typename Param2Type, typename ResultType>
    class binary_function;
```

Thus the adaptable version of `MyOtherFunction` would be

```
class MyOtherFunction: public binary_function<string, int, bool> {
public:
    bool operator() (const string& str, int val) const {
        /* Do something, return a bool. */
    }
};
```

While the above approach for generating adaptable functions is perfectly legal, it's a bit clunky and we still have a high ratio of boilerplate code to actual logic. Fortunately, the STL functional library provides the powerful but cryptically named `ptr_fun`* function that transforms a regular C++ function into an adaptable function. `ptr_fun` can convert both unary and binary C++ functions into adaptable functions with the correct parameter types, meaning that you can skip the hassle of the above code by simply writing normal functions and then using `ptr_fun` to transform them into adaptable functions. For example, given the following C++ function:

```
bool LengthIsLessThan(string myStr, int threshold) {
    return myStr.length() < threshold;
}
```

If we need to get an adaptable version of that function, we can write `ptr_fun(LengthIsLessThan)` in the spot where the adaptable function is needed.

`ptr_fun` is a useful but imperfect tool. Most notably, you cannot use `ptr_fun` on functions that accept parameters as reference-to-`const`. `ptr_fun` returns a `unary_function` object, and as mentioned above, you cannot specify reference-to-`const` as template arguments to `unary_function`. Also, because of the way that the C++ compiler generates code for functors, code that uses `ptr_fun` can be a bit slower than code using functors.

For situations where you'd like to convert a member function into an adaptable function, you can use the `mem_fun` or `mem_fun_ref` functions. These functions convert member functions into unary functions that accept as input a receiver object, then invoke that member function on the receiver. The difference between `mem_fun` and `mem_fun_ref` is how they accept their parameters – `mem_fun` accepts a *pointer* to the receiver object, while `mem_fun_ref` accepts a *reference* to the receiver. For example, given a `vector<string>`, the following code will print out the lengths of all of the strings in the `vector`:

```
transform(myVector.begin(), myVector.end(), ostream_iterator<int>(cout, "\n"),
          mem_fun_ref(&string::length));
```

Let's dissect this call to `transform`, since there's a lot going on. The first two parameters delineate the input range, in this case the full contents of `myVector`. The third parameter specifies where to put the output, and since here it's an `ostream_iterator` the output will be printed directly to the console instead of stored in some other location. The final parameter is `mem_fun_ref(&string::length)`, a function that accepts as input a `string` and then returns the value of the `length` member function called on that `string`.

`mem_fun_ref` can also be used to convert unary (one-parameter) member functions into adaptable binary functions that take as a first parameter the object to apply the function to and as a second parameter the

* `ptr_fun` is short for "pointer function", not "fun with pointers."

parameter to the function. When we cover binders in the next section, you should get a better feel for exactly how useful this is.

Binding Parameters

Now that we've covered how the STL functional library handles adaptable functions, let's consider how we can use them in practice.

At the beginning of this chapter, we introduced the notion of *parameter binding*, converting a two-parameter function into a one-parameter function by locking in the value of one of its parameters. To allow you to bind parameters to functions, the STL functional programming library exports two functions, `bind1st` and `bind2nd`, which accept as parameters an adaptable function and a value to bind and return new functions that are equal to the old functions with the specified values bound in place. For example, given the following implementation of `LengthIsLessThan`:

```
bool LengthIsLessThan(string str, int threshold) {
    return str.length() < threshold;
}
```

We could use the following syntax to construct a function that's `LengthIsLessThan` with the value five bound to the second parameter:

```
bind2nd(ptr_fun(LengthIsLessThan), 5)
```

The line `bind2nd(ptr_fun(LengthIsLessThan), 5)` first uses `ptr_fun` to generate an adaptable version of the `LengthIsLessThan` function, then uses `bind2nd` to lock the parameter 5 in place. The result is a new unary function that accepts a `string` parameter and returns if that string's length is less than 5, the value we bound to the second parameter. Since `bind2nd` is a function that accepts a function as a parameter and returns a function as a result, `bind2nd` is a function that is sometimes referred to as a *higher-order function*.

Because the result of the above call to `bind2nd` is a unary function that determines if a string has length less than five, we can use the `count_if` algorithm to count the number of values less than five by using the following code:

```
count_if(container.begin(), container.end(),
         bind2nd(ptr_fun(LengthIsLessThan), 5));
```

Compare this code to the functor-based approach illustrated at the start of this chapter. This version of the code is much, *much* shorter than the previous version. If you aren't beginning to appreciate exactly how much power and flexibility the `<functional>` library provides, skip ahead and take a look at the practice problems.

The `bind1st` function acts similarly to `bind2nd`, except that (as its name suggests) it binds the first parameter of a function. Returning to the above example, given a `vector<int>`, we could count the number of elements in that `vector` smaller than the length of string "C++!" by writing

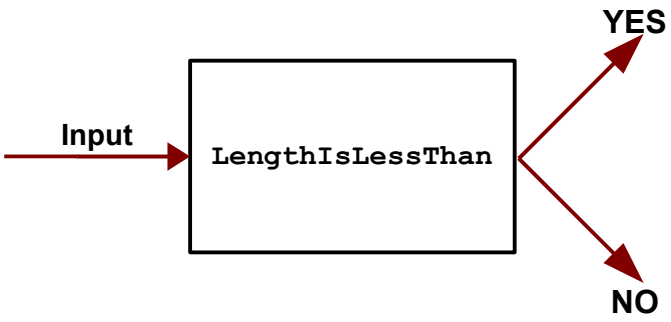
```
count_if(myVector.begin(), myVector.end(),
         bind1st(ptr_fun(LengthIsLessThan), "C++!"));
```

(Admittedly, this isn't the most practical use case for `bind1st`, but it does get the point across).

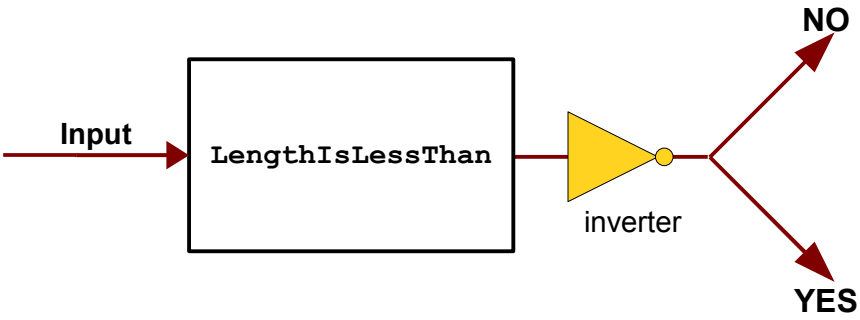
In the STL functional programming library, parameter binding is restricted only to binary functions. Thus you cannot bind a parameter in a three-parameter function to yield a new binary function, nor can you bind the parameter of a unary function to yield a zero-parameter ("nullary") function. For these operations, you'll need to create your own custom functors, as shown in the practice problems at the end of this chapter.

Negating Results

Suppose that given a function `LengthIsLessThan`, we want to find the number of strings in a container that are *not* less than a certain length. While we could simply write another function `LengthIsNotLessThan`, it would be much more convenient if we could somehow tell C++ to take whatever value `LengthIsLessThan` returns and to use the opposite result. That is, given a function that looks like this:



We'd like to change it into a function that looks like this:



This operation is *negation* – constructing a new function whose return value has the opposite value of the input function. There are two STL negator functions – `not1` and `not2` – that return the negated result of a unary or binary predicate function, respectively. Thus, the above function that's a negation of `LengthIsLessThan` could be written as `not2(ptr_fun(LengthIsLessThan))`. Since `not2` returns an adaptable function, we can then pass the result of this function to `bind2nd` to generate a unary function that returns whether a string's length is at least a certain threshold value. For example, here's code that returns the number of `string`s in a container with length at least 5:

```
count_if(container.begin(), container.end(),
         bind2nd(not2(ptr_fun(LengthIsLessThan)), 5));
```

While this line is dense, it elegantly solves the problem at hand by combining and modifying existing code to create entirely different functions. Such is the beauty and simplicity of higher-order programming – why rewrite code from scratch when you already have all the pieces individually assembled?

Operator Functions

Let's suppose that you have a container of `int`s and you'd like to add 137 to each of them. Recall that you can use the STL `transform` algorithm to apply a function to each element in a container and then store the result. Because we're adding 137 to each element, we might consider writing a function like this one:

```
int Add137(int param) {
    return param + 137;
}
```

And then writing

```
transform(container.begin(), container.end(), container.begin(), Add137);
```

While this code works correctly, this approach is not particularly robust. What if later on we needed to increment all elements in a container by 42, or perhaps by an arbitrary value? Thus, we might want to consider replacing `Add137` by a function like this one:

```
int AddTwoInts(int one, int two) {
    return one + two;
}
```

And then using binders to lock the second parameter in place. For example, here's code that's equivalent to what we've written above:

```
transform(container.begin(), container.end(), container.begin(),
          bind2nd(ptr_fun(AddTwoInts), 137));
```

At this point, our code is correct, but it can get a bit annoying to have to write a function `AddTwoInts` that simply adds two integers. Moreover, if we then need code to increment all `double`s in a container by 1.37, we would need to write another function `AddTwoDoubles` to avoid problems from typecasts and truncations. Fortunately, the designers of the STL functional library recognized how tedious it is to write out this sort of code, and so the STL functional library provides a large number of template adaptable function classes that simply apply the basic C++ operators to two values. For example, in the above code, we can use the adaptable function class *`plus<int>`* instead of our `AddTwoInts` function, resulting in code that looks like this:

```
transform(container.begin(), container.end(), container.begin(),
          bind2nd(plus<int>(), 137));
```

Note that we need to write `plus<int>()` instead of simply `plus<int>`, since we're using the temporary object syntax to construct a `plus<int>` object for `bind2nd`. Forgetting the parentheses can cause a major compiler error headache that can take a while to track down. Also notice that we don't need to use `ptr_fun` here, since `plus<int>` is already an adaptable function.

For reference, here's a list of the "operator functions" exported by `<functional>`:

`plus`	`minus`	`multiplies`	`divides`	`modulus`	`negate`
`equal_to`	`not_equal_to`	`greater`	`less`	`greater_equal`	`less_equal`
`logical_and`	`logical_or`	`logical_not`			

To see an example that combines the techniques from the previous few sections, let's consider a function that accepts a `vector<double>` and converts each element in the `vector` to its reciprocal (one divided by

the value). Because we want to convert each element with value x to the value $^1/_x$, we can use a combination of binders and operator functions to solve this problem by binding the value 1.0 to the first parameter of the `divides<double>` functor. The result is a unary function that accepts a parameter of type `double` and returns the element's reciprocal. The resulting code looks like this:

```
transform(v.begin(), v.end(), v.begin(), bind1st(divides<double>(), 1.0));
```

This code is concise and elegant, solving the problem in a small space and making explicit what operations are being performed on the data.

Unifying Functions and Functors

There are a huge number of ways to define a function or function-like object in C++, each of which has slightly different syntax and behavior. For example, suppose that we want to write a function that accepts as input a function that can accept an `int` and return a `double`. While of course we could accept a `double (*) (int)` – a pointer to a function accepting an `int` and returning a `double` – this is overly restrictive. For example, all of the following functions can be used as though they were functions taking in an `int` and returning a `double`:

```
double Fn1(const int&);    /* Accept by reference-to-const, yield double. */
int    Fn2(int);           /* Accept parameter as a int, return int. */
int    Fn3(const int&);    /* Accept reference-to-const int, return int. */
```

In addition, if we just accept a `double (*) (int)`, we also can't accept functors as input, meaning that neither of these objects below – both of which can accept an `int` and return a `double` – could be used:

```
/* Functor accepting an int and returning a double. */
class MyFunctor {
public:
    double operator() (int);
};

/* Adaptable function accepting double and returning a double. */
bind2nd(multiplies<int>(), 137);
```

Earlier in this chapter, we saw how we can write functions that accept any of the above functions using templates, as shown here:

```
template <typename UnaryFunction> void DoSomething(UnaryFunction fn) {
    /* ... */
}
```

If we want to write a *function* that accepts a function as input we can rely on templates, but what if we want to write a *class* that needs to store a function of any arbitrary type? To give a concrete example, suppose that we're designing a class representing a graphical window and we want the client to be able to control the window's size and position. The window object, which we'll assume is of type `Window`, thus allows the user to provide a function that will be invoked whenever the window is about to change size. The user's function then takes in an `int` representing the potential new width of the window and returns an `int` representing what the user wants the new window size to be. For example, if we want to create a window that can't be more than 100 pixels wide, we could pass in this function:

```
int ClampTo100Pixels(int size) {
    return min(size, 100);
}
```

Alternatively, if we want the window size to always be 100 pixels, we could pass in this function:

```
int Always100Pixels(int) {
    return 100; // Ignore parameter
}
```

Given that we need to store a function of an arbitrary type inside the `Window` class, how might we design `Window`? Using the approach outlined above, we could parameterize the `Window` class over the type of the function it stores, as shown here:

```
template <typename WidthFunction> class Window {
public:
   Window(WidthFunction fn, /* ... */);

    /* ... */

private:
    WidthFunction width;

    /* ... */
};
```

Given this implementation of `Window`, we could then specify that a window should be no more than 100 pixels wide by writing

```
Window<int (*)(int)> myWindow(ClampTo100Pixels);
```

This Window class lets us use any reasonable function to determine the window size, but has several serious drawbacks. First, it requires the `Window` client to explicitly parameterize Window over the type of callback being stored. When working with function pointers this results in long and convoluted variable declarations (look above for an example), and when working with library functors such as those in the STL `<functional>` library (e.g. `bind2nd (ptr_fun(MyFunction), 137)`)*, we could end up with a `Window` of such a complicated type that it would be infeasible to use with without the aid of `typedef`. But a more serious problem is that this approach causes two `Window`s that don't use the same type of function to compute width to have completely different types. That is, a `Window` using a raw C++ function to compute its size would have a different type from a `Window` that computed its size with a functor. Consequently, we couldn't make a `vector<Window>`, but instead would have to make a `vector<Window<int (*)(int)> >` or a `vector<Window<MyFunctorType> >`. Similarly, if we want to write a function that accepts a `Window`, we can't just write the following:

```
void DoSomething(const Window& w) { // Error - Window is a template, not a type
    /* ... */
}
```

We instead would have to write

```
template <typename WindowType>
    void DoSomething(const WindowType& w) { // Legal but awkward
    /* ... */
}
```

---

* As an FYI, the type of `bind2nd(ptr_fun(MyFunction), 137)` is

```
binder2nd<pointer_to_binary_function<Arg1, Arg2, Ret> >
```

where `Arg1`, `Arg2`, and `Ret` are the argument and return types of the `MyFunction` function.

It should be clear that templatizing the `Window` class over the type of the callback function does not work well. How can we resolve this problem? In the remainder of this chapter, see a beautiful solution to this problem that will unify our treatment of functors, inheritance, templates, operator overloading, copy functions, and conversion constructors. The result is amazingly elegant and hopefully will impress upon you exactly how powerful functors are as a technique.

Inheritance to the Rescue

Let's take a few minutes to think about the problem we're facing. We have a collection of different objects that each have similar functionality (they can be called as functions), but we don't know exactly which object the user will provide. This sounds exactly like the sort of problem we can solve using inheritance and virtual functions. But we have a problem – inheritance only applies to *objects*, but some of the values we might want to store are simple function pointers, which are primitives. Fortunately, we can apply a technique called the *Fundamental Theorem of Software Engineering* (or FTSE) to solve this problem:

> **Theorem *(*The Fundamental Theorem of Software Engineering*)*:** Any problem can be solved by adding enough layers of indirection.

This is a very useful programming concept that will prove relevant time and time again – make sure you remember it!

In this particular application, the FTSE says that we need to distance ourselves by one level from raw function pointers and functor classes. This leads to the following observation: while we might not be able to treat functors and function pointers polymorphically, we certainly can create a new class hierarchy and then treat that class polymorphically. The idea goes something like this – rather than having the user provide us a functor or function pointer which could be of any type, instead we'll define an abstract class exporting the callback function, then will have the user provide a subclass which implements the callback.

One possible base class in this hierarchy is shown below:

```
class IntFunction {
public:
    /* Polymorphic classes need virtual destructors. */
    virtual ~IntFunction() {}

    /* execute() actually calls the proper function and returns the value. */
    virtual int execute(int value) const = 0;
};
```

`IntFunction` exports a single function called `execute` which accepts an `int` and returns an `int`. This function is marked purely virtual since it's unclear exactly what this function should do. After all, we're trying to store an arbitrary function, so there's no clearly-defined default behavior for `execute`.

We can now modify the implementation of `Window` to hold a pointer to an `IntFunction` instead of being templatized over the type of the function, as shown here:

```
class Window {
public:
    Window(IntFunction* sizeFunction, /* ... */);

    /* ... */
private:
    IntFunction* fn;
};
```

Now, if we wanted to clamp the window to 100 pixels, we can do the following:

```
class ClampTo100PixelsFunction: public IntFunction {
public:
    virtual int execute(int size) const {
        return min(size, 100);
    }
};

Window myWindow(new ClampTo100PixelsFunction, /* ... */);
```

Similarly, if we want to have a window that's always 100 pixels wide, we could write

```
class FixedSizeFunction: public IntFunction {
public:
    virtual int execute(int size) const {
        return 100;
    }
};

Window myWindow(new FixedSizeFunction, /* ... */);
```

It seems as though we've solved the problem – we now have a `Window` class that allows us to fully customize its resizing behavior – what more could we possibly want?

The main problem with our solution is the sheer amount of boilerplate code clients of `Window` have to write if they want to change the window's resizing behavior. Our initial goal was to let class clients pass raw C++ functions and functors to the `Window` class, but now clients have to subclass `IntFunction` to get the job done. Both of the above subclasses are lengthy even though they only export a single function. Is there a simpler way to do this?

The answer, of course, is yes. Suppose we have a pure C++ function that accepts an `int` by value and returns an `int` that we want to use for our resizing function in the `Window` class; perhaps it's the `ClampTo100Pixels` function we defined earlier, or perhaps it's `Always100Pixels`. Rather than defining a new subclass of `IntFunction` for every single one of these functions, instead we'll build a single class that's designed to wrap up a function pointer in a package that's compatible with the `IntFunction` interface. That is, we can define a subclass of `IntFunction` whose constructor accepts a function pointer and whose `execute` calls this function. This is the Fundamental Theorem of Software Engineering in action – we couldn't directly treat the raw C++ function polymorphically, but by abstracting by a level we can directly apply inheritance.

Here's one possible implementation of the subclass:

```
class ActualFunction: public IntFunction {
public:
    explicit ActualFunction(int (*fn)(int)) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    int (*function)(int);
};
```

Now, if we want to use `ClampTo100Pixels` inside of `Window`, we can write:

```
Window myWindow(new ActualFunction(ClampTo100Pixels), /* ... */);
```

There is a bit of extra code for creating the `ActualFunction` object, but this is a one-time cost. We can now use `ActualFunction` to wrap any raw C++ function accepting an `int` and returning an `int` and will save a lot of time typing out new subclasses of `IntFunction` for every callback.

Now, suppose that we have a functor class, which we'll call `MyFunctor`, that we want to use inside the `Window` class. Then we could define a subclass that looks like this:

```
class MyFunctorFunction: public IntFunction {
public:
    explicit MyFunctorFunction(MyFunctor fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    MyFunctor function;
};
```

And could then use it like this:

```
Window myWindow(new MyFunctorFunction(MyFunctor(137)), /* ... */);
```

Where we assume for simplicity that the `MyFunctor` class has a unary constructor.

We're getting much closer to an ideal solution. Hang in there; the next step is pretty exciting.

Templates to the Rescue

At this point we again could just call it quits – we've solved the problem we set out to solve and using the above pattern our `Window` class can use any C++ function or functor we want. However, we are close to an observation that will greatly simplify the implementation of `Window` and will yield a much more general solution.

Let's reprint the two subclasses of `IntFunction` we just defined above which wrap function pointers and functors:

```
class ActualFunction: public IntFunction {
public:
    explicit ActualFunction(int (*fn)(int)) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    int (*const function)(int);
};

class MyFunctorFunction: public IntFunction {
public:
    explicit MyFunctorFunction(MyFunctor fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    MyFunctor function;
};
```

If you'll notice, besides the name of the classes, the only difference between these two classes is what type of object is being stored. This similarity is no coincidence – any callable function or functor would require a subclass with exactly this structure. Rather than requiring the client of `Window` to reimplement this subclass from scratch each time, we can instead create a *template class* that's a subclass of `IntFunction`. It's rare in practice to see templates and inheritance mixed this way, but here it works out beautifully. Here's one implementation:

```
template <typename UnaryFunction> class SpecificFunction: public IntFunction {
public:
    explicit SpecificFunction(UnaryFunction fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

private:
    UnaryFunction function;
};
```

We now can use the `Window` class as follows:

```
Window myWindow(new SpecificFunction<int(*)(int)>(ClampTo100Pixels), /*...*/);
Window myWindow(new SpecificFunction<MyFunctor>(MyFunctor(137)), /*...*/);
```

The syntax here might be a bit tricky, but we've greatly reduced the complexity associated with the `Window` class since clients no longer have to implement their own subclasses of `IntFunction`.

One More Abstraction

This design process has consisted primarily of adding more and more abstractions on top of the system we're designing, and it's time for one final leap. Let's think about what we've constructed so far. We've built a class hierarchy with a single base class and a template for creating as many subclasses as we need. However, everything we've written has been hardcoded with the assumption that the `Window` class is the only class that might want this sort of functionality. Having the ability to store and call a function of any conceivable type is enormously useful, and if we can somehow encapsulate all of the necessary machinery to get this working into a single class, we will be able to reuse what we've just built time and time again. In this next section, that's exactly what we'll begin doing.

We'll begin by moving the code from `Window` that manages the stored function into a dedicated class called `Function`. The basic interface for `Function` is shown below:

```
class Function {
public:
    /* Constructor and destructor.  We'll implement copying in a bit. */
    Function(IntFunction* fn);
    ~Function();

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    IntFunction* function;
};
```

We'll leave the `Function` constructor left as an implicit conversion constructor, since that way we can implicitly convert between a callable `IntFunction` pointer and a `Function` functor. We can then implement the above functions as follows:

```
/* Constructor accepts an IntFunction and stores it. */
Function::Function(IntFunction* fn) : function(fn) {
    // Handled in initializer list
}

/* Destructor deallocates the stored function. */
Function::~Function() {
    delete function;
}

/* Function call just calls through to the pointer and returns the result. */
int Function::operator() (int value) const {
    return function->execute(value);
}
```

Nothing here should be too out-of-the-ordinary – after all, this is pretty much the same code that we had inside the `Window` class.

Given this version of `Function`, we can write code that looks like this:

```
Function myFunction = new SpecificFunction<int (*)(int)>(ClampTo100Pixels);
cout << myFunction(137) << endl; // Prints 100
cout << myFunction(42) << endl; // Prints 42
```

If you're a bit worried that the syntax `new SpecificFunction<int (*)(int)>(ClampTo100Pixels)` is unnecessarily bulky, that's absolutely correct. Don't worry, in a bit we'll see how to eliminate it. In the meantime, however, let's implement the copy behavior for the `Function` class. After all, there's no reason that we shouldn't be able to copy `Function` objects, and defining copy behavior like this will lead to some very impressive results in a bit.

We'll begin by defining the proper functions inside the Function class, as seen here:

```
class Function {
public:
    /* Constructor and destructor. */
    Function(IntFunction* fn);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    IntFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

Now, since the `Function` class contains only a single data member (the `IntFunction` pointer), to make a deep-copy of a `Function` we simply need to make a deep copy of its requisite `IntFunction`. But here we run into a problem. `IntFunction` is an abstract class and we can't tell at compile-time what type of object is actually being pointed at by the function pointer. How, then, can we make a deep-copy of the `IntFunction`? The answer is surprisingly straightforward – we'll just introduce a new virtual function to the `IntFunction` class that returns a deep copy of the receiver object. Since this function duplicates an existing object, we'll call it `clone`. The interface for `IntFunction` now looks like this:

```
class IntFunction {
public:
    /* Polymorphic classes should have virtual destructors. */
    virtual ~IntFunction() { }

    /* execute() actually calls the proper function and returns the value. */
    virtual int execute(int value) const = 0;

    /* clone() returns a deep-copy of the receiver object. */
    virtual IntFunction* clone() const = 0;
};
```

We can then update the template class `SpecificFunction` to implement `clone` as follows:

```
template <typename UnaryFunction> class SpecificFunction: public IntFunction {
public:
    explicit SpecificFunction(UnaryFunction fn) : function(fn) {}

    virtual int execute(int value) const {
        return function(value);
    }

    virtual IntFunction* clone() const {
        return new SpecificFunction(*this);
    }

private:
    UnaryFunction function;
};
```

Here, the implementation of `clone` returns a new `SpecificFunction` initialized via the copy constructor as a copy of the receiver object. Note that we haven't explicitly defined a copy constructor for `SpecificFunction` and are relying here on C++'s automatically-generated copy function to do the trick for us. This assumes, of course, that the `UnaryFunction` type correctly supports deep-copying, but this isn't a problem since raw function pointers can trivially be deep-copied as can all primitive types and it's rare to find functor classes with no copy support.

We can then implement the copy constructor, assignment operator, destructor, and helper functions for Function as follows:

```
Function::~Function() {
    clear();
}

Function::Function(const Function& other) {
    copyOther(other);
}

Function& Function::operator= (const Function& other) {
    if(this != &other) {
        clear();
        copyOther(other);
    }
    return *this;
}

void Function::clear() {
    delete function;
}

void Function::copyOther(const Function& other) {
    /* Have the stored function tell us how to copy itself. */
    function = other.function->clone();
}
```

Our `Function` class is now starting to take shape!

Hiding `SpecificFunction`

Right now our `Function` class has full deep-copy support and using `SpecificFunction<T>` can store any type of callable function. However, clients of `Function` have to explicitly wrap any function they want to

store inside `Function` in the `SpecificFunction` class. This has several problems. First and foremost, this breaks encapsulation. `SpecificFunction` is only used internally to the `Function` class, never externally, so requiring clients of `Function` to have explicit knowledge of its existence violates encapsulation. Second, it requires the user to know the type of every function they want to store inside the Function class. In the case of `ClampTo100Pixels` this is rather simple, but suppose we want to store `bind2nd(multiplies<int>(), 137)` inside of `Function`. What is the type of the object returned by `bind2nd(multiplies<int>(), 137)`? For reference, it's `binder2nd<multiplies<int> >`, so if we wanted to store this in a Function we'd have to write

```
Function myFunction =
    new SpecificFunction<binder2nd<multiplies<int> > >(bind2nd(multiplies<int>(),137));
```

This is a syntactic nightmare and makes the `Function` class terribly unattractive.

Fortunately, however, this problem has a quick fix – we can rewrite the `Function` constructor as a template function parameterized over the type of argument passed into it, then construct the relevant `SpecificFunction` for the `Function` client. Since C++ automatically infers the parameter types of template functions, this means that clients of Function never need to know the type of what they're storing – the compiler will do the work for them. Excellent!

If we do end up making the Function constructor a template, we should also move the `IntFunction` and `SpecificFunction` classes so that they're inner classes of `Function`. After all, they're specific to the implementation of Function and the outside world has no business using them.

The updated interface for the `Function` class is shown here:

```
class Function {
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction fn);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    int operator() (int value) const;

private:
    class IntFunction { /* ... */ };
    template <typename UnaryFunction> class SpecificFunction { /* ... */ };

    IntFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

We can then implement the constructor as follows:

```
template <typename UnaryFunction> Function::Function(UnaryFunction fn) {
    function = new SpecificFunction<UnaryFunction>(fn);
}
```

Since we've left the `Function` constructor not marked explicit, this template constructor is a conversion constructor. Coupled with the assignment operator, this means that we can use `Function` as follows:

```
Function fn = ClampTo100Pixels;
cout << fn(137) << endl; // Prints 100
cout << fn(42) << endl; // Prints 42

fn = bind2nd(multiplies<int>(), 2);
cout << fn(137) << endl; // Prints 274
cout << fn(42) << endl; // Prints 84
```

This is exactly what we're looking for – a class that can store any callable function that takes in an `int` and returns an `int`. If this doesn't strike you as a particularly elegant piece of code, take some time to look over it again.

There's one final step we should take, and that's to relax the restriction that `Function` always acts as a function from `int`s to `int`s. There's nothing special about int, and by giving `Function` clients the ability to specify their own parameter and return types we'll increase the scope of what `Function` is capable of handling. We'll thus templatize `Function` as `Function<ArgType, ReturnType>`. We also need to make some minor edits to `IntFunction` (which we'll rename to `ArbitraryFunction` since `IntFunction` is no longer applicable), but in the interest of brevity we won't reprint them here.

The final interface for `Function` thus looks like this:

```
template <typename ArgType, typename ReturnType> class Function {
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgType value) const;

private:
    class ArbitraryFunction { /* ... */ };
    template <typename UnaryFunction> class SpecificFunction { /* ... */ }

    ArbitraryFunction* function;

    void clear();
    void copyOther(const Function& other);
};
```

To conclude our discussion of `Window`, using the new `Function` type we could rewrite the `Window` class using `Function` as follows:

```
class Window {
public:
    Window(const Function<int, int>& widthFn, /* ... */
    /* ... other member functions ... */

private:
    Function<int, int> widthFunction;
};
```

Now, clients can pass any unary function (or functor) that maps from `int`s to `int`s as a parameter to `Window` and the code will compile correctly.

External Polymorphism

The `Function` type we've just developed is subtle in its cleverness. Because we can convert any callable unary function into a `Function`, when writing code that needs to work with some sort of unary function, we can have that code use `Function` instead of any specific function type. This technique of abstracting away from the particular types that provide a behavior into an object representing that behavior is sometimes known as *external polymorphism*. As opposed to *internal polymorphism*, where we explicitly define a set of classes containing virtual functions, external polymorphism "grafts" a set of virtual functions onto any type that supports the requisite behavior.

Virtual functions can be slightly more expensive than regular functions because of the virtual function table lookup required. External polymorphism is implemented using inheritance and thus also incurs an overhead, but the overhead is slightly greater than regular inheritance. Think for a minute how the `Function` class we just implemented is designed. Calling `Function::operator()` requires the following:

1. Following the `ArbitraryFunction` pointer in the `Function` class to its virtual function table.

2. Calling the function indicated by the virtual function table, which corresponds to the particular `SpecificFunction` being pointed at.

3. Calling the actual function object stored inside the `SpecificFunction`.

This is slightly more complex than a regular virtual function call, and illustrates the cost associated with external polymorphism. That said, in some cases (such as the `Function` case outlined here) the cost is overwhelming offset by the flexibility afforded by external polymorphism.

Implementing the `<functional>` Library

Now what we've seen how the `<functional>` library works from a client perspective, let's discuss how the library is put together. What's so special about adaptable functions? How does `ptr_fun` convert a regular function into an adaptable one? How do functions like `bind2nd` and `not1` work? This discussion will be highly technical and will push the limits of your knowledge of templates, but by the time you're done you should have an excellent grasp of how template libraries are put together. Moreover, the techniques used here are applicable beyond just the `<functional>` library and will almost certainly come in handy later in your programming career.

Let's begin by looking at exactly what an adaptable function is. Recall that adaptable functions are functors that inherit from either `unary_function` or `binary_function`. Neither of these template classes are particularly complicated; here's the complete definition of `unary_function`:*

```
template <typename ArgType, typename RetType> class unary_function {
public:
    typedef ArgType argument_type;
    typedef RetType result_type;
};
```

This class contains no data members and no member functions. Instead, it exports two `typedef`s – one renaming `ArgType` to `argument_type` and one renaming `RetType` to `result_type`. When you create an adaptable function that inherits from `unary_function`, your class acquires these `typedef`s. For example, if we write the following adaptable function:

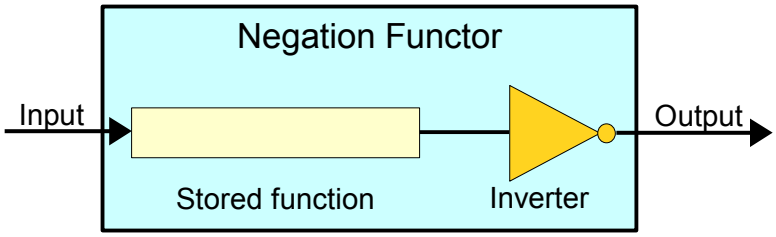
```
class IsPositive: public unary_function<double, bool> {
public:
    bool operator() (double value) const {
        return value > 0.0;
    }
};
```

The statement `public unary_function<double, double>` imports two `typedef`s into `IsPositive`: `argument_type` and `return_type`, equal to `double` and `bool`, respectively. Right now it might not be apparent how these types are useful, but as we begin implementing the other pieces of the `<functional>` library it will become more apparent.

Implementing `not1`

To begin our behind-the-scenes tour of the `<functional>` library, let's see how to implement the `not1` function. Recall that `not1` accepts as a parameter a unary adaptable predicate function, then returns a new adaptable function that yields opposite values as the original function. For example, `not1(IsPositive())` would return a function that returns whether a value is *not* positive.

Implementing `not1` requires two steps. First, we'll create a template functor class parameterized over the type of the adaptable function to negate. This functor's constructor will take as a parameter an adaptable function of the proper type and store it for later use. We'll then implement its `operator()` function such that it calls the stored function and returns the negation of the result. Graphically, this is shown here:



Once we have designed this functor, we'll have `not1` accept an adaptable function, wrap it in our negating functor, then return the resulting object to the caller. This means that the return value of `not1` is an adaptable unary predicate function that returns the opposite value of its parameter, which is exactly what we want.

---

* Technically speaking `unary_function` and `binary_function` are `struct`s, but this is irrelevant here.

Let's begin by writing the template functor class, which we'll call `unary_negate` (this is the name of the functor class generated by the `<functional>` library's `not1` function). We know that this functor should be parameterized over the type of the adaptable function it negates, so we can begin by writing the following:

```
template <typename UnaryPredicate> class unary_negate {
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    /* ... */
private:
    UnaryPredicate p;
};
```

Here, the constructor accepts an object of type `UnaryPredicate`, then stores it in the data member `p`.

Now, let's implement the `operator()` function, which, as you'll recall, should take in a parameter, feed it into the stored function `p`, then return the inverse result. The code for this function looks like this:

```
template <typename UnaryPredicate> class unary_negate {
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool operator() (const /* what goes here? */& param) const {
        return !p(param); // Call function and return the opposite result.
    }
private:
    UnaryPredicate p;
};
```

We've almost finished writing our `unary_negate` class, but we have a slight problem – what is the type of the parameter to `operator()`? This is where adaptable functions come in. Because `UnaryPredicate` is adaptable, it must export a type called `argument_type` corresponding to the type of its argument. We can thus define our `operator()` function to accept a parameter of type `typename UnaryPredicate::argume nt_type` to guarantee that it has the same parameter type as the `UnaryPredicate` class.* The updated code for `unary_negate` looks like this:

```
template <typename UnaryPredicate> class unary_negate {
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool
    operator() (const typename UnaryPredicate::argument_type& param) const {
        return !p(param); // Call stored function and return opposite result.
    }
private:
    UnaryPredicate p;
};
```

That's quite a mouthful, but it's exactly the solution we're looking for. If it weren't for the fact that `UnaryPredicate` is an adaptable function, we would not have been able to determine the parameter type for the `operator()` member function, and code like this would not have been possible.

---

* Remember that the type is `typename UnaryPredicate::argument_type`, not `UnaryPredicate::argument_type`. `argument_type` is nested inside `UnaryPredicate`, and since `UnaryPredicate` is a template argument we have to explicitly use `typename` to indicate that `argument_type` is a type.

There's one step left to finalize this functor class, and that's to make the functor into an adaptable function by having it inherit from the proper `unary_function`. Since the functor's argument type is `typename UnaryPredicate::argument_type` and its return type is `bool`, we'll inherit from `unary_function <typename UnaryPredicate::argument_type, bool>`. The final code for `unary_negate` is shown here:

```
template <typename UnaryPredicate>
class unary_negate:
    public unary_function<typename UnaryPredicate::argument_type, bool>
{
public:
    explicit unary_negate(const UnaryPredicate& pred) : p(pred) {}

    bool
    operator() (const typename UnaryPredicate::argument_type& param) const {
        return !p(param); // Call stored function and return opposite result.
    }
private:
    UnaryPredicate p;
};
```

We've now finished writing our functor class to perform the negation, and all that's left to do is write `not1`. `not1` is much simpler than `unary_negate`, since it simply has to take in a parameter and wrap it in a `unary_negate` functor. This is shown here:

```
template <typename UnaryPredicate>
    unary_negate<UnaryPredicate> not1(const UnaryPredicate& pred) {
    return unary_negate<UnaryPredicate>(pred);
}
```

That's all there is to it – we've successfully implemented `not1`!

You might be wondering why there are two steps involved in writing `not1`. After all, once we have the functor that performs negation, why do we need to write an additional function to create it? The answer is *simplicity*. We don't need `not1`, but having it available reduces complexity. For example, using the `IsPositive` adaptable function from above, let's suppose that we want to write code to find the first nonpositive element in a `vector`. Using the `find_if` algorithm and `not1`, we'd write this as follows:

```
vector<double>::iterator itr =
    find_if(v.begin(), v.end(), not1(IsPositive()));
```

If instead of using `not1` we were to explicitly create a `unary_negate` object, the code would look like this:

```
vector<double>::iterator itr =
    find_if(v.begin(), v.end(), unary_negate<IsPositive>(IsPositive()));
```

That's quite a mouthful. When calling the template function `not1`, the compiler automatically infers the type of the argument and constructs an appropriately parameterized `unary_negate` object. If we directly use `unary_negate`, C++ will not perform type inference and we'll have to spell out the template arguments ourselves. The pattern illustrated here – having a template class and a template function to create it – is common in library code because it lets library clients use complex classes without ever having to know how they're implemented behind-the-scenes.

Implementing `ptr_fun`

Now that we've seen how `not1` works, let's see if we can construct the `ptr_fun` function. At a high level `ptr_fun` and `not1` work the same way – they each accept a parameter, construct a special functor class based on the parameter, then return it to the caller. The difference between `not1` and `ptr_fun`, however, is that there are two different versions of `ptr_fun` – one for unary functions and one for binary functions. The two versions work almost identically and we'll see how to implement them both, but for simplicity we'll begin with the unary case.

To convert a raw C++ unary function into an adaptable unary function, we need to wrap it in a functor that inherits from the proper `unary_function` base class. We'll make this functor's `operator()` function simply call the stored function and return its value. To be consistent with the naming convention of the `<functional>` library, we'll call the functor `pointer_to_unary_function` and will parameterize it over the argument and return types of the function. This is shown here:

```
template <typename ArgType, typename RetType>
class pointer_to_unary_function: public unary_function<ArgType, RetType>
{
public:
    explicit pointer_to_unary_function(ArgType fn(RetType)) : function(fn) {}

    RetType operator() (const ArgType& param) const {
        return function(param);
    }
private:
    ArgType (*function)(RetType);
};
```

There isn't that much code here, but it's fairly dense. Notice that we inherit from `unary_function<ArgType, RetType>` so that the resulting functor is adaptable. Also note that the argument and return types of `operator()` are considerably easier to determine than in the `unary_negate` case because they're specified as template arguments.

Now, how can we implement `ptr_fun` to return a correctly-constructed `pointer_to_unary_function`? Simple – we just write a template function parameterized over argument and return types, accept a function pointer of the appropriate type, then wrap it in a `pointer_to_unary_function` object. This is shown here:

```
template <typename ArgType, typename RetType>
    pointer_to_unary_function<ArgType, RetType>
    ptr_fun(RetType function(ArgType)) {
        return pointer_to_unary_function<ArgType, RetType>(function);
    }
```

This code is fairly dense, but gets the job done.

The implementation of `ptr_fun` for binary functions is similar to the implementation for unary functions. We'll create a template functor called `pointer_to_binary_function` parameterized over its argument and return types, then provide an implementation of `ptr_fun` that constructs and returns an object of this type. This is shown here:

```
template <typename Arg1, typename Arg2, typename Ret>
class pointer_to_binary_function: public binary_function<Arg1, Arg2, Ret> {
public:
    explicit pointer_to_binary_function(Ret fn(Arg1, Arg2)) : function(fn) {}

    Ret operator() (const Arg1& arg1, const Arg2& arg2) const {
        return function(arg1, arg2);
    }
private:
    Ret (*function)(Arg1, Arg2);
};

template <typename Arg1, typename Arg2, typename Ret>
pointer_to_binary_function<Arg1, Arg2, Ret> ptr_fun(Ret function(Arg1, Arg2)) {
    return pointer_to_binary_function<Arg1, Arg2, Ret>(function);
}
```

Note that we now have *two* versions of `ptr_fun` – one that takes in a unary function and one that takes in a binary function. Fortunately, C++ overloading rules allow for the two functions to coexist, since they have different signatures.

Implementing `bind1st`

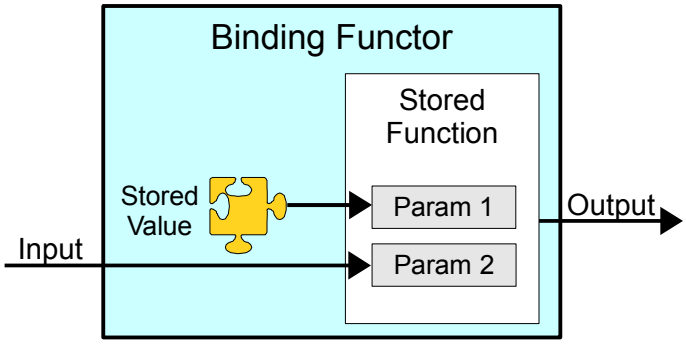
To wrap up our tour of the `<functional>` library, let's see how to implement `bind1st`. If you'll recall, `bind1st` takes in a binary adaptable function and a value, then returns a new unary function equal to the input function with the first parameter locked in place. We'll follow the pattern of `not1` and `ptr_fun` by writing a template functor class called `binder1st` that actually does the binding, then having `bind1st` construct and return an object of the proper type.

Before proceeding with our implementation of `binder1st`, we need to take a quick detour into the inner workings of the `binary_function` class. Like `unary_function`, `binary_function` exports `typedef`s so that other parts of the `<functional>` library can recover the argument and return types of adaptable functions. However, since a binary function has two arguments, the names of the exported types are slightly different. `binary_function` provides the following three `typedef`s:

- **`first_argument_type`**, the type of the first argument,
- **`second_argument_type`**, the type of the second argument, and
- **`result_type`**, the function's return type.

We will need to reference each of these type names when writing `bind1st`.

Now, how do we implement the `binder1st` functor? Here is one possible implementation. The `binder1st` constructor will accept and store an adaptable binary function and the value for its first argument. `binder1st` then provides an implementation of `operator()` that takes a single parameter, then invokes the stored function passing in the function parameter and the saved value. This is shown here:



Let's begin implementing `binder1st`. The functor has to be a template, since we'll be storing an arbitrary adaptable function and value. However, we only need to parameterize the functor over the type of the binary adaptable function, since we can determine the type of the first argument from the adaptable function's `first_argument_type`. We'll thus begin with the following implementation:

```
template <typename BinaryFunction> class binder1st {
    /* ... */
};
```

Now, let's implement the constructor. It should take in two parameters – one representing the binary function and the other the value to lock into place. The first will have type `BinaryFunction`; the second, `typename BinaryFunction::first_argument_type`. This is shown here:

```
template <typename BinaryFunction> class binder1st {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    /* ... */

private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

Phew! That's quite a mouthful, but is the reality of much library template code. Look at the declaration of the `first` data member. Though it may seem strange, this is the correct way to declare a data member whose type is a type nested inside a template argument.

We now have the constructor written and all that's left to take care of is `operator()`. Conceptually, this function isn't very difficult, and if we ignore the parameter and return types have the following implementation:

```
template <typename BinaryFunction> class binder1st {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    /* ret */ operator() (const /* arg */& param) const {
        return function(first, param);
    }

private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

What are the argument and return types for this function? Well, the function returns whatever object is produced by the stored function, which has type `typename BinaryFunction::result_type`. The function accepts a value for use as the second parameter to the stored function, so it must have type `typename BinaryFunction::second_argument_type`. This results in the following code:

```
template <typename BinaryFunction> class binder1st {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    typename BinaryFunction::result_type
    operator()  (const typename BinaryFunction::second_argument_type& param) const {
        return function(first, param);
    }
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

We're almost finished, and all that's left for `binder1st` is to make it adaptable. Using the logic from above, we'll have it inherit from the proper instantiation of `unary_function`, as shown here:

```
template <typename BinaryFunction> class binder1st :
   public unary_function<typename BinaryFunction::second_argument_type,
                         typename BinaryFunction::result_type> {
public:
    binder1st(const BinaryFunction& fn,
              const typename BinaryFunction::first_argument_type& arg) :
                  function(fn), first(arg) {}

    typename BinaryFunction::result_type
    operator()  (const typename BinaryFunction::second_argument_type& param) const {
        return function(first, param);
    }
private:
    BinaryFunction function;
    typename BinaryFunction::first_argument_type first;
};
```

That's it for the `binder1st` class. As you can see, the code is dense and does a lot of magic with `typename` and nested types. Without adaptable functions, code like this would not be possible.

To finish up our discussion, let's implement `bind1st`. This function isn't particularly tricky, though we do need to do a bit of work to extract the type of the value to lock in place:

```
template <typename BinaryFunction>
binder1st<BinaryFunction>
   bind1st(const BinaryFunction& fn,
           const typename BinaryFunction::first_argument_type& arg) {
    return binder1st<BinaryFunction>(fn, arg);
}
```

We now have a complete working implementation of `bind1st`. If you actually open up the `<functional>` header and peek around inside, the code you'll find will probably bear a strong resemblance to what we've written here.

Limitations of the Functional Library

While the STL functional library is useful in a wide number of cases, the library is unfortunately quite limited. `<functional>` only provides support for adaptable unary and binary functions, but commonly you'll encounter situations where you will need to bind and negate functions with more than two parameters. In these cases, one of your only options is to construct functor classes that accept the extra parameters in their constructors. Similarly, there is no support for function composition, so we could not create a function that computes $2x + 1$ by calling the appropriate combination of the plus and multiplies functors. However, the next version of C++, nicknamed "C++0x," promises to have more support for functional programming of this sort. For example, it will provide a general function called `bind` that lets you bind as many values as you'd like to a function of arbitrary arity. Keep your eyes peeled for the next release of C++ – it will be far more functional than the current version!

Practice Problems

We've covered a lot of programming techniques in this chapter and there are no shortage of applications for the material. Here are some problems to get you thinking about how functors and adaptable functions can influence your programming style:

1. What is a functor?
2. What restrictions, if any, exist on the parameter or return types of `operator()`?
3. Why are functors more powerful than regular functions?
4. How do you define a function that can accept both functions and functors as parameters?
5. What is an adaptable function?
6. How do you convert a regular C++ function into an adaptable function?
7. What does the `bind1st` function do?
8. What does the `not2` function do?
9. The STL algorithm `for_each` accepts as parameters a range of iterators and a unary function, then calls the function on each argument. Unusually, the return value of `for_each` is the unary function passed in as a parameter. Why might this be?

10. Using the fact that `for_each` returns the unary function passed as a parameter, write a function `MyAccumulate` that accepts as parameters a range of `vector<int>::iterator`s and an initial value, then returns the sum of all of the values in the range, starting at the specified value. Do not use any loops – instead, use `for_each` and a custom functor class that performs the addition.

11. Write a function `AdvancedBiasedSort` that accepts as parameters a `vector<string>` and a `string` "winner" value, then sorts the range, except that all strings equal to the winner are at the front of the `vector`. Do not use any loops. *(Hint: Use the STL `sort` algorithm and functor that stores the "winner" parameter.)*

12. Modify the above implementation of `AdvancedBiasedSort` so that it works over an arbitrary range of iterators over strings, not just a `vector<string>`. Then modify it once more so that the iterators can iterate over any type of value.

13. The STL `generate` algorithm is defined as `void generate(ForwardIterator start, ForwardIterator end, NullaryFunction fn)` and iterates over the specified range storing the return value of the zero-parameter function `fn` as it goes. For example, calling `generate(v.begin(), v.end(), rand)` would fill the range [`v.begin()` to `v.end()`) with random values. Write a function `FillAscending` that accepts an iterator range, then sets the first element in the range to zero, the second to one, etc. Do not use any loops.

14. Write a function `ExpungeLetter` that accepts four parameters – two iterators delineating an input range of `string`s, one iterator delineating the start of an output range, and a character – then copies the strings in the input range that do not contain the specified character into the output range. The function should then return an iterator one past the last location written. Do not use loops. *(Hint: Use the `remove_copy_if` algorithm and a custom functor).*

15. The *standard deviation* of a set of data is a measure of how much the data varies from its average value. Data with a small standard deviation tends to cluster around a point, while data with large standard deviation will be more spread out.

    The formula for the standard deviation of a set of data $\{x_1, x_2, ..., x_n\}$ is

    $$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2}$$

    Here, $\bar{x}$ is the average of the data points.

    To give a feeling for this formula, given the data points 1, 2, 3, the average of the data points is 2, so the standard deviation is given by

    $$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2}=\sqrt{\frac{1}{3}\sum_{i=1}^{3}(x_i-2)^2}=\sqrt{\frac{1}{3}((1-2)^2+(2-2)^2+(3-2)^2)}=\sqrt{\frac{1}{3}(1+0+1)}=\sqrt{\frac{2}{3}}$$

    Write a function `StandardDeviation` that accepts an input range of iterators over `double`s (or values implicitly convertible to `double`s) and returns its standard deviation. Do not use any loops – instead use the `accumulate` function to compute the average, then use `accumulate` once more to compute the sum. *(Hint: To get the number of elements in the range, you can use the `distance` function)*

16. Write a function `ClearAllStrings` that accepts as input a range of iterators over `string`s that sets each string to be the empty string. If you harness the `<functional>` library correctly here, the function body will be only a single line of code.

17. The ROT128 cipher is a weak encryption cipher that works by adding 128 to the value of each character in a string to produce a garbled string. Since `char` can only hold 256 different values, two successive applications of ROT128 will produce the original string. Write a function `ApplyROT128` that accepts a `string` and returns the `string`'s ROT128 cipher equivalent.

18. Write a template function `CapAtValue` that accepts a range of iterators and a value by reference-to-`const` and replaces all elements in the range that compare greater than the parameter with a copy of the parameter. *(Hint: use the `replace_if` algorithm)*

19. One piece of functionality missing from the `<functional>` library is the ability to bind the first parameter of a unary function to form a nullary function. In this practice problem, we'll implement a function called `BindOnly` that transforms a unary adaptable function into a nullary function.

    a. Write a template functor class `BinderOnly` parameterized whose constructor accepts an adaptable function and a value to bind and whose `operator()` function calls the stored function passing in the stored value as a parameter. Your class should have this interface:

```
template <typename UnaryFunction> class BinderOnly {
public:
    BinderOnly(const UnaryFunction& fn,
               const typename UnaryFunction::argument_type& value);
    RetType operator() () const;
};
```

    b. Write a template function `BindOnly` that accepts the same parameters as the `BinderOnly` constructor and returns a `BinderOnly` of the proper type. The signature for this function should be

```
template <typename UnaryFunction>
    BinderOnly<UnaryFunction>
        BindOnly(const UnaryFunction &fn,
                 const typename UnaryFunction::argument_type& value);
```

20. Another operation not supported by the <functional> library is *function composition*. For example, given two functions **f** and **g**, the composition **g** ∘ **f** is a function such that **g** ∘ **f**(x) = **g**(**f**(x)). In this example, we'll write a function `Compose` that lets us compose two unary functions of compatible types.

    a. Write a template functor `UnaryCompose` parameterized over two adaptable function types whose constructor accepts and stores two unary adaptable functions and whose `operator()` accepts a single parameter and returns the composition of the two functions applied to that argument. Make sure that `UnaryCompose` is an adaptable unary function.

    b. Write a wrapper function `Compose` that takes in the same parameters as `UnaryCompose` and returns a properly-constructed `UnaryCompose` object.

    c. Explain how to implement `not1` using `Compose` and `logical_not`, a unary adaptable function exported by `<functional>` that returns the logical inverse of its argument.

Part Three

More to Explore

Whew! You've made it through the first three sections and are now a seasoned and competent C++ programmer. But your journey has just begun. There are many parts of the C++ programming language that we have not covered, and it's now up to you to begin the rest of your journey.

This last section of the book contains two chapters. The first, on C++0x, discusses what changes are expected for the C++ programming language over the next few years. Now that you've seen C++'s strengths and weaknesses, I hope that this chapter proves enlightening and exciting. The second chapter is all about how to continue your journey into further C++ mastery and hopefully can give you a boost in the right direction.

Chapter 14: C++0x

C++0x feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever. If you timidly approach C++ as just a better C or as an object-oriented language, you are going to miss the point. The abstractions are simply more flexible and affordable than before. Rely on the old mantra: If you think [o]f it *as a separate idea or object, represent it directly in the program; model real-world objects, concepts, and abstractions directly in code. It's easier now: Your ideas will map to enumerations, objects, classes (e.g. control of defaults), class hierarchies (e.g. inherited constructors), templates, concepts, concept maps, axioms, aliases, exceptions, loops, threads, etc., rather than to a single "one size fits all" abstraction mechanism.*

My ideal is to use programming language facilities to help programmers think differently about system design and implementation. I think C++0x can do that – and do it not just for C++ programmers but for programmers used to a variety of modern programming languages in the general and very broad area of systems programming.

In other words, I'm still an optimist.

– Bjarne Stroustrup, inventor of C++. [Str09.3]

C++ is constantly evolving. Over the past few years the C++ standards body has been developing the next revision of C++, nicknamed *C++0x*. C++0x is a major upgrade to the C++ programming language and as we wrap up our tour of C++, I thought it appropriate to conclude by exploring what C++0x will have in store. This chapter covers some of the more impressive features of C++0x and what to expect in the future.

Be aware that C++0x has not yet been finalized, and the material in this chapter may not match the final C++0x specification. However, it should be a great launching point so that you know where to look to learn more about the next release of C++.

Automatic Type Inference

Consider the following piece of code:

```
void DoSomething(const multimap<string, vector<int> >& myMap) {
    const pair<multimap<string, vector<int> >::const_iterator,
               multimap<string, vector<int> >::const_iterator> eq =
        myMap.equal_range("String!");
    for(multimap<string, vector<int> >::const_iterator itr = eq.first;
        itr != eq.second; ++itr)
        cout << itr->size() << endl;
}
```

This above code takes in a `multimap` mapping from `string`s to `vector<int>`s and prints out the length of all vectors in the `multimap` whose key is "String!" While the code is perfectly legal C++, it is extremely difficult to follow because more than half of the code is spent listing the types of two variables, `eq` and `itr`. If you'll notice, these variables can only take on one type – the type of the expression used to initialize them. Since the compiler knows all of the types of the other variables in this code snippet, couldn't we just ask the compiler to give `eq` and `itr` the right types? Fortunately, in C++0x, the answer is yes thanks to a

new language feature called *type inference*. Using type inference, we can rewrite the above function in about half as much space:

```
void DoSomething(const multimap<string, vector<int>>& myMap) {
    const auto eq = myMap.equal_range("String!");
    for(auto itr = eq.first; itr != eq.second; ++itr)
        cout << itr->size() << endl;
}
```

Notice that we've replaced all of the bulky types in this expression with the keyword `auto`, which tells the C++0x compiler that it should infer the proper type for a variable. The standard iterator loop is now considerably easier to write, since we can replace the clunky `multimap<string, vector<int> >::const_iterator` with the much simpler `auto`. Similarly, the hideous return type associated with `equal_range` is entirely absent.

Because `auto` must be able to infer the type of a variable from the expression that initializes it, you can only use `auto` when there is a clear type to assign to a variable. For example, the following is illegal:

```
auto x;
```

Since `x` could theoretically be of any type.

`auto` is also useful because it allows complex libraries to hide implementation details behind-the-scenes. For example, recall that the `ptr_fun` function from the STL `<functional>` library takes as a parameter a regular C++ function and returns an adaptable version of that function. In our discussion of the library's implementation, we saw that the return type of `ptr_fun` is either `pointer_to_unary_function<Arg, Ret>` or `pointer_to_binary_function<Arg1, Arg2, Ret>`, depending on whether the parameter is a unary or binary function. This means that if you want to use `ptr_fun` to create an adaptable function and want to store the result for later use, using current C++ you'd have to write something to the effect of

```
pointer_to_unary_function<int, bool> ouchies = ptr_fun(SomeFunction);
```

This is terribly hard to read but more importantly breaks the wall of abstraction of `ptr_fun`. The entire purpose of `ptr_fun` is to hide the transformation from function to functor, and as soon as you are required to know the return type of `ptr_fun` the benefits of the automatic wrapping facilities vanish. Fortunately, `auto` can help maintain the abstraction, since we can rewrite the above as

```
auto howNice = ptr_fun(SomeFunction);
```

C++0x will provide a companion operator to `auto` called `decltype` that returns the type of a given expression. For example, `decltype(1 + 2)` will evaluate to `int`, while `decltype(new char)` will be `char *`. `decltype` does not evaluate its argument – it simply yields its type – and thus incurs no cost at runtime.

One potential use of `decltype` arises when writing template functions. For example, suppose that we want to write a template function as follows:

```
template <typename T> /* some type */ MyFunction(const T& val) {
    return val.doSomething();
}
```

This function accepts a `T` as a template argument, invokes that object's `doSomething` member function, then returns its value (note that if the type `T` doesn't have a member function `doSomething`, this results in

a compile-time error). What should we use as the return type of this function? We can't tell by simply looking at the type `T`, since the `doSomething` member function could theoretically return any type. However, by using `decltype` and a new function declaration syntax, we can rewrite this as

```
template <typename T>
auto MyFunction(const T& val) -> decltype(val.doSomething()) {
    return val.doSomething();
}
```

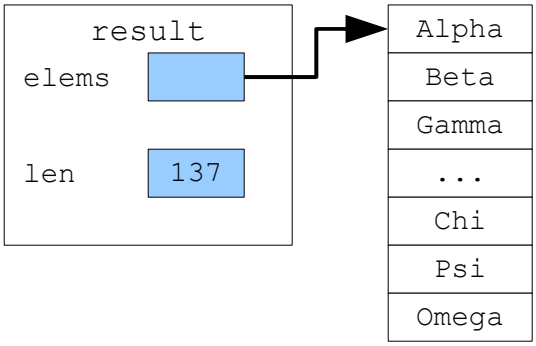
Notice that we defined the function's return type as `auto`, and then after the parameter list said that the return type is `decltype(val.doSomething())`. This new syntax for function declarations is optional, but will make complicated function prototypes easier to read.

Move Semantics

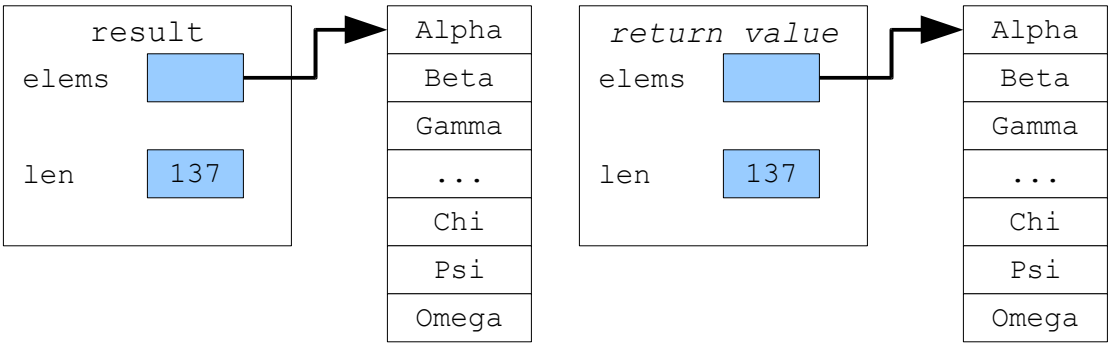
If you'll recall from our discussion of copy constructors and assignment operators, when returning a value from a function, C++ initializes the return value by invoking the class's copy constructor. While this method guarantees that the returned value is always valid, it can be grossly inefficient. For example, consider the following code:

```
vector<string> LoadAllWords(const string& filename) {
    ifstream input(filename.c_str());
    if(!input.is_open())
        throw runtime_error("File not found!");

    /* Use the vector's insert function, plus some istream_iterators, to
     * load the contents of the file.
     */
    vector<string> result;
    result.insert(result.begin(), istream_iterator<string>(input),
                  istream_iterator<string>());

    return result;
}
```

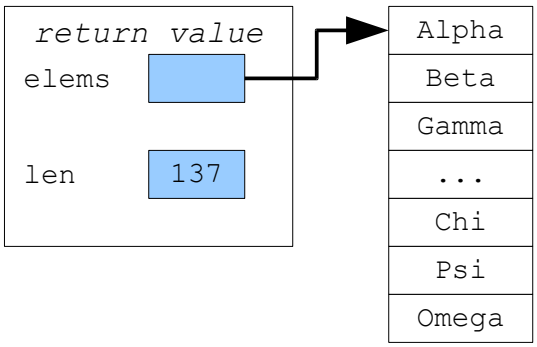
Here, we open the file specified by `filename`, then use a pair of `istream_iterator`s to load the contents of the file into the `vector`. At the end of this function, before the `return result` statement executes, the memory associated with the `result vector` looks something like this (assuming a `vector` is implemented as a pointer to a raw C++ array):



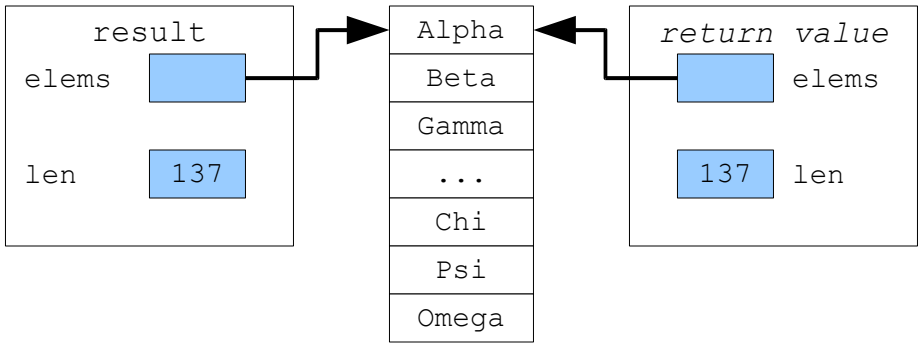
Now, the statement `return result` executes and C++ initializes the return value by invoking the `vector` copy constructor. After the copy the program's memory looks like this:



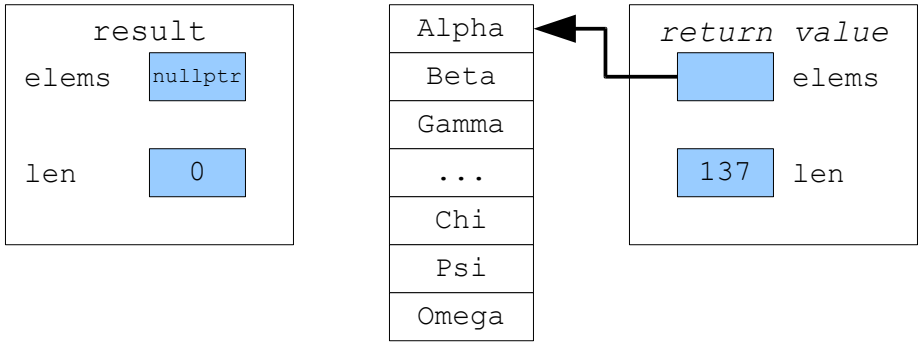
After the return value is initialized, `result` will go out of scope and its destructor will clean up its memory. Memory now looks like this:



Here, we made a full deep copy of the contents of the returned object, then deallocated all of the original memory. This is inefficient, since we needlessly copied a long list of strings. There is a much better way to return the `vector` from the function. Instead of initializing the return value by making a deep copy, instead we'll make it a shallow copy of `vector` we're returning. The in-memory representations of these two vectors thus look like this:



Although the two vectors share the same memory, the returned `vector` has the same contents as the source `vector` and is in fact indistinguishable from the original. If we then modify the original `vector` by detaching its pointer from the array and having it point to `NULL` (or, since this is C++0x, the special value `nullptr`), then we end up with a picture like this:



Now, `result` is an empty `vector` whose destructor will not clean up any memory, and the calling function will end up with a `vector` whose contents are exactly those returned by the function. We've successfully returned the value from the function, but avoided the expensive copy. In our case, if we have a `vector` of *n* `string`s of length at most *m*, then the algorithm for copying the `vector` will take O(*mn*). The algorithm for simply transferring the pointer from the source `vector` to the destination, on the other hand, is O(1) for the pointer manipulations.

The difference between the current method of returning a value and this improved version of returning a value is the difference between copy semantics and move semantics. An object has *copy semantics* if it can be duplicated in another location. An object has *move semantics* (a feature introduced in C++0x) if it can be moved from one variable into another, destructively modifying the original. The key difference between the two is the number of copies at any point. Copying an object duplicates its data, while moving an object transfers the contents from one object to another without making a copy.

To support move semantics, C++0x introduces a new variable type called an *rvalue reference* whose syntax is `Type &&`. For example, an rvalue reference to a `vector<int>` would be a `vector<int> &&`. Informally, you can view an rvalue reference as a reference to a temporary object, especially one whose contents are to be moved from one location to another.

Let's return to the above example with returning a `vector` from a function. In the current version of C++, we'd define a copy constructor and assignment operator for `vector` to allow us to return `vector`s from functions and to pass `vector`s as parameters. In C++0x, we can optionally define another special function, called a *move constructor*, that initializes a new `vector` by moving data out of one `vector` into another. In the above example, we might define a move constructor for the `vector` as follows:

```
/* Move constructor takes a vector&& as a parameter, since we want to move
 * data from the parameter into this vector.
 */
template <typename T> vector<T>::vector(vector&& other) {
    /* We point to the same array as other and have the same length. */
    elems = other.elems;
    len = other.len;

    /* Destructively modify the source vector to stop sharing the array. */
    other.elems = nullptr;
    other.len = 0;
}
```

Now, if we return a `vector` from a function, the new `vector` will be initialized using the move constructor rather than the regular copy constructor.

We can similarly define a *move assignment operator* (as opposed to the traditional *copy* assignment operator), as shown here:

```
template <typename T> vector<T>& vector<T>::operator= (vector&& other) {
    if(this != &other) {
        delete [] elems;

        elems = other.elems;
        len = other.len;

        /* Modify the source vector to stop sharing the array. */
        other.elems = nullptr;
        other.len = 0;
    }
    return *this;
}
```

The similarity between a copy constructor and copy assignment operator is also noticeable here in the move constructor and move assignment operator. In fact, we can rewrite the pair using helper functions `clear` and `moveOther`:

```
template <typename T> void vector<T>::moveOther(vector&& other) {
    /* We point to the same array as the other vector and have the same
     * length.
     */
    elems = other.elems;
    len = other.len;

    /* Modify the source vector to stop sharing the array. */
    other.elems = nullptr;
    other.len = 0;
}

template <typename T> void vector<T>::clear() {
    delete [] elems;
    len = 0;
}

template <typename T> vector<T>::vector(vector&& other) {
    moveOther(move(other)); // See later section for move
}

template <typename T> vector<T>& vector<T>::operator =(vector&& other) {
    if(this != &other) {
        clear();
        moveOther(move(other));
    }
    return *this;
}
```

Move semantics are also useful in situations other than returning objects from functions. For example, suppose that we want to insert an element into an array, shuffling all of the other values down one spot to make room for the new value. Using current C++, the code for this operation is as follows:

```
template <typename T>
void InsertIntoArray(T* elems, int size, int position, const T& toAdd) {
    for(int i = size; i > position; ++i)
        elems[i] = elems[i - 1]; // Shuffle elements down.
    elems[i] = toAdd;
}
```

There is nothing wrong *per se* with this code as it's written, but if you'll notice we're using copy semantics to shuffle the elements down when move semantics is more appropriate. After all, we don't want to *copy* the elements into the spot one element down; we want to *move* them.

In C++0x, we can use an object's move semantics (if any) by using the special helper function `move`, exported by `<utility>`, which simply returns an rvalue reference to an object. Now, if we write

```
a = move(b);
```

If `a` has support for move semantics, this will move the contents of `b` into `a`. If `a` does *not* have support for move semantics, however, C++ will simply fall back to the default object copy behavior using the assignment operator. In other words, supporting move operations is purely optional and a class can still use the old fashioned copy constructor and assignment operator pair for all of its copying needs.

Here's the rewritten version of `InsertIntoArray`, this time using move semantics:

```
template <typename T>
void InsertIntoArray(T* elems, int size, int position, const T& toAdd) {
    for(int i = size; i > position; ++i)
        elems[i] = move(elems[i - 1]); // Move elements down.
    elems[i] = toAdd;
}
```

Curiously, we can potentially take this one step further by moving the new element into the array rather than copying it. We thus provide a similar function, which we'll call `MoveIntoArray`, which moves the parameter into the specified position:

```
template <typename T>
void MoveIntoArray(T* elems, int size, int position, T&& toAdd) {
    for(int i = size; i > position; ++i)
        elems[i] = move(elems[i - 1]); // Move elements down.

    /* Note that even though toAdd is an rvalue reference, we still must
     * explicitly move it in.  This prevents us from accidentally using
     * move semantics in a few edge cases.
     */
    elems[i] = move(toAdd);
}
```

Move semantics and copy semantics are independent and in C++0x it will be possible to construct objects that can be moved but not copied or vice-versa. Initially this might seem strange, but there are several cases where this is exactly the behavior we want. For example, it is illegal to copy an `ofstream` because the behavior associated with the copy is undefined – should we duplicate the file? If so, where? Or should we just share the file? However, it is perfectly legitimate to *move* an `ofstream` from one variable to another, since at any instant only one `ofstream` variable will actually hold a reference to the file stream. Thus functions like this one:

```
ofstream GetTemporaryOutputFile() {
    /* Use the tmpnam() function from <cstdio> to get the name of a
     * temporary file.  Consult a reference for more detail.
     */
    char tmpnamBuffer[L_tmpnam];
    ofstream result(tmpnam(tmpnamBuffer));
    return result; // Uses move constructor, not copy constructor!
}
```

Will be perfectly legal in C++0x because of move constructors, though the same code will not compile in current C++ because `ofstream` has no copy constructor.

Another example of an object that has well-defined move behavior but no copy behavior is the C++ `auto_ptr` class. If you'll recall, assigning one `auto_ptr` to another destructively modifies the original `auto_ptr`. This is exactly the definition of move semantics. However, under current C++ rules, implementing `auto_ptr` is extremely difficult and leads to all sorts of unexpected side effects. Using move constructors, however, we can eliminate these problems. C++0x will introduce a replacement to `auto_ptr` called `unique_ptr` which, like `auto_ptr`, represents a smart pointer that automatically cleans up its underlying resource when it goes out of scope. Unlike `auto_ptr`, however, `unique_ptr` cannot be copied or assigned but can be moved freely. Thus code of this sort:

```
unique_ptr<int> myPtr(new int);
unique_ptr<int> other = myPtr; // Error!  Can't copy unique_ptr.
```

Will not compile. However, by explicitly indicating that the operation is a move, we can transfer the contents from one `unique_ptr` to another:

```
unique_ptr<int> myPtr(new int);
unique_ptr<int> other = move(myPtr); // Legal; myPtr is now empty
```

Move semantics and rvalue references may seem confusing at first, but promise to be a powerful and welcome addition to the C++ family.

Lambda Expressions

Last chapter, we considered the problem of counting the number of `string`s in a `vector` whose lengths were less than some value determined at runtime. We explored how to solve this problem using the `count_if` algorithm and a functor. Our solution was as follows:

```
class ShorterThan {
public:
    explicit ShorterThan(int maxLength) : length(maxLength) {}
    bool operator() (const string& str) const {
        return str.length() < length;
    }
private:
    int length;
};

const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(), ShorterThan(myValue));
```

This functor-based approach works correctly, but has a huge amount of boilerplate code that obscures the actual mechanics of the solution. What we'd prefer instead is the ability to write code to this effect:

```
const int myValue = GetInteger()
count_if(myVector.begin(), myVector.end(), the string is shorter than myValue);
```

Using a new C++0x language feature known as *lambda expressions* (a term those of you familiar with languages like Scheme, ML, or Haskell might recognize), we can write code that very closely mirrors this structure. One possibility looks like this:

```
const int myValue = GetInteger();
count_if(myVector.begin(), myVector.end(),
         [myValue](const string& x) { return x.length() < myValue; });
```

The construct in the final line of code is a *lambda expression*, an unnamed ("anonymous") function that exists only as a parameter to `count_if`. In this example, we pass as the final parameter to `count_if` a temporary function that accepts a single `string` parameter and returns a `bool` indicating whether or not its length is less than `myValue`. The bracket syntax `[myValue]` before the parameter declaration `(int x)` is called the *capture list* and indicates to C++ that the lambda expression can access the value of `myValue` in its body.

Behind the scenes, C++ converts lambda expressions such as the one above into uniquely-named functors, so the above code is identical to the functor-based approach outlined above.

For those of you with experience in a functional programming language, the example outlined above should strike you as an extraordinarily powerful addition to the C++ programming language. Lambda expressions greatly simplify many tasks and represent an entirely different way of thinking about programming. It will be interesting to see how rapidly lambda expressions are adopted in professional code.

Variadic Templates

In the previous chapter we implemented a class called `Function` that wrapped an arbitrary unary function. Recall that the definition of `Function` is as follows:

```
template <typename ArgType, typename ReturnType> class Function {
public:
    /* Constructor and destructor. */
    template <typename UnaryFunction> Function(UnaryFunction);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgType value) const;
private:
    /* ... */
};
```

What if we want to generalize `Function` to work with functions of arbitrary arity? That is, what if we want to create a class that encapsulates a binary, nullary, or ternary function? Using standard C++, we could do this by introducing new classes `BinaryFunction`, `NullaryFunction`, and `TernaryFunction` that were implemented similarly to `Function` but which accepted a different number of parameters. For example, here's one possible interface for `BinaryFunction`:

```
template <typename ArgType1, typename ArgType2, typename ReturnType>
class BinaryFunction {
public:
    /* Constructor and destructor. */
    template <typename BinaryFn> BinaryFunction(BinaryFn);
    ~BinaryFunction();

    /* Copy support. */
    BinaryFunction(const BinaryFunction& other);
    BinaryFunction& operator= (const BinaryFunction& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgType1 arg2, ArgType2 arg2) const;
private:
    /* ... */
};
```

Writing different class templates for functions of each arity is troublesome. If we write `Function`-like classes for a fixed number of arities (say, functions between zero and ten arguments) and then discover that we need a wrapper for a function with more arguments, we we'll have to write that class from scratch. Moreover, the structure of each function wrapper is almost identical. Compare the `BinaryFunction` and `Function` class interfaces mentioned above. If you'll notice, the only difference between the classes is the number of template arguments and the number of arguments to `operator()`. Is there some way that we can use this commonality to implement a single class that works with functions of arbitrary arity? Using the current incarnation of C++ this is not possible, but using a C++0x feature called *variadic templates* we can do just this.

A *variadic template* is a template that can accept an arbitrary number of template arguments. These arguments are grouped together into arguments called *parameter packs* that can be expanded out to code for each argument in the pack. For example, the following class is parameterized over an arbitrary number of arguments:

```
template <typename... Args> class Tuple {
    /* ... */
};
```

The syntax `typename... Args` indicates that `Args` is a parameter pack that represents an arbitrary number of arguments. Since `Args` represents a list of arguments rather than an argument itself, it is illegal to use `Args` in an expression by itself. Instead, `Args` must be used in a *pattern expression* indicating what operation should be applied to each argument in the pack. For example, if we want to create a constructor for `Tuple` that accepts a list of arguments with one argument for each type in `Args`, we could write the following:

```
template <typename... Args> class Tuple {
public:
    Tuple(const Args& ...);
};
```

Here, the syntax `const Args& ...` is a pattern expression indicating that for each argument in `Args`, there should be a parameter to the constructor that's passed by reference-to-`const`. For example, if we created a `Tuple<int>`, the constructor would be `Tuple<int>(const int&)`, and if we create a `Tuple<int, double>`, it would be `Tuple<int, double>(const int&, const double&)`.

Let's return to the example of `Function`. Suppose that we want to convert `Function` from encoding a unary function to encoding a function of arbitrary arity. Then we could change the class interface to look like this:

```
template <typename ReturnType, typename... ArgumentTypes> class Function {
public:
    /* Constructor and destructor. */
    template <typename Callable> Function(Callable);
    ~Function();

    /* Copy support. */
    Function(const Function& other);
    Function& operator= (const Function& other);

    /* Function is a functor that calls into the stored resource. */
    ReturnType operator() (ArgumentTypes... args) const;
private:
    /* ... */
};
```

Function is now parameterized such that the first argument is the return type and the remaining arguments are argument types. For example, a `Function<int, string>` is a function that accepts a string and returns an `int`, while a `Function<bool, int, int>` would be a function accepting two `int`s and returning a `bool`.

We've just seen how the interface for `Function` looks with `variadic` templates, but what about the implementation? If you'll recall, the original implementation of `Function`'s `operator()` function looked as follows:

```
template <typename ArgType, typename ReturnType>
ReturnType Function<ArgType, ReturnType>::operator()(ArgType param) const {
    return function->execute(param);
}
```

Let's begin converting this to use variadic templates. The first step is to adjust the signature of the function, as shown here:

```
template <typename RetType, typename... ArgTypes>
RetType Function<RetType, ArgTypes...>::operator()(ArgTypes... args) const {
    /* ... */
}
```

Notice that we've specified that this is a member of `Function<RetType, ArgTypes...>`.

In the unary version of `Function`, we implemented `operator()` by calling a stored function object's execute member function, passing in the parameter given to `operator()`. But how can we now call execute passing in an arbitrary number of parameters? The syntax for this again uses `...` to tell C++ to expand the `args` parameters to the function into an actual list of parameters. This is shown here:

```
template <typename RetType, typename... ArgTypes>
RetType Function<RetType, ArgTypes...>::operator()(ArgTypes... args) const {
    return function->execute(args...);
}
```

Just as using `...` expands out a parameter pack into its individual parameters, using `...` here expands out the variable-length argument list `args` into each of its individual parameters. This syntax might seem a bit tricky at first, but is easy to pick up with practice.

Library Extensions

In addition to all of the language extensions mentioned in the above sections, C++0x will provide a new set of libraries that should make certain common tasks much easier to perform:

- **Enhanced Smart Pointers**. C++0x will support a wide variety of smart pointers, such as the reference-counted `shared_ptr` and the aforementioned `unique_ptr`.

- **New STL Containers**. The current STL associative containers (`map`, `set`, etc.) are layered on top of balanced binary trees, which means that traversing the `map` and `set` always produce elements in sorted order. However, the sorted nature of these containers means that insertion, lookup, and deletion are all O(lg *n*), where *n* is the size of the container. In C++0x, the STL will be enhanced with `unordered_map`, `unordered_set`, and multicontainer equivalents thereof. These containers are layered on top of hash tables, which have O(1) lookup and are useful when ordering is not important.

- **Multithreading Support**. Virtually all major C++ programs these days contain some amount of multithreading and concurrency, but the C++ language itself provides no support for concurrent programming. The next incarnation of C++ will support a threading library, along with atomic operations, locks, and all of the bells and whistles needed to write robust multithreaded code.

- **Regular Expressions**. The combination of C++ `string`s and the STL algorithms encompasses a good deal of string processing functionality but falls short of the features provided by other languages like Java, Python, and (especially) Perl. C++0x will augment the strings library with full support for regular expressions, which should make string processing and compiler-authoring considerably easier in C++.

- **Upgraded `<functional>` library**. C++0x will expand on `<functional>` with a generic `function` type akin to the one described above, as well as a supercharged `bind` function that can bind arbitrary parameters in a function with arbitrary values.

- **Random Number Generation**. C++'s only random number generator is `rand`, which has extremely low randomness (on some implementations numbers toggle between even and odd) and is not particularly useful in statistics and machine learning applications. C++0x, however, will support a rich random number generator library, complete with a host of random number generators and probability distribution functors.

- **Metaprogramming Traits Classes**. C++0x will provide a large number of classes called *traits classes* that can help generic programmers write optimized code. Want to know if a template argument is an abstract class? Just check if `is_abstract<T>::type` evaluates to `true_type` or `false_type`.

Other Key Language Features

Here's a small sampling of the other upgrades you might find useful:

- **Unified Initialization Syntax**: It will be possible to initialize C++ classes by using the curly brace syntax (e.g. `vector<int> v = {1, 2, 3, 4, 5};`)

- **Delegating Constructors**: Currently, if several constructors all need to access the same code, they must call a shared member function to do the work. In C++0x, constructors can invoke each other in initializer lists.

- **Better Enumerations**: Currently, `enum` can only be used to create integral constants, and those constants can be freely compared against each other. In C++0x, you will be able to specify what type to use in an enumeration, and can disallow automatic conversions to `int`.

- **Angle Brackets**: It is currently illegal to terminate a nested template by writing two close brackets consecutively, since the compiler confuses it with the stream insertion `operator >>`. This will be fixed in C++0x.

- **C99 Compatibility**: C++0x will formally introduce the `long long` type, which many current C++ compilers support, along with various preprocessor enhancements.

C++0x Today

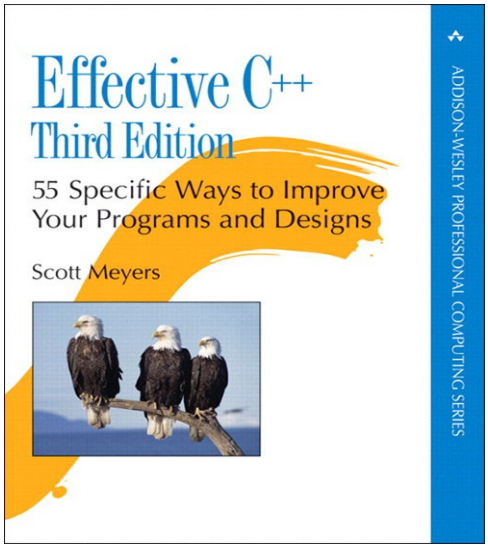
Although C++0x has not yet been adopted as a standard, there are several freely-available compilers that support a subset of C++0x features. For example, `g++` versions 4.4 and up have support for much of C++0x, and Microsoft Visual Studio 2010 has a fair number of features implemented, including lambda expressions and the `auto` keyword. If you want to experience the future of C++ today, consider downloading one of these compilers.

Chapter 15: Where to Go From Here

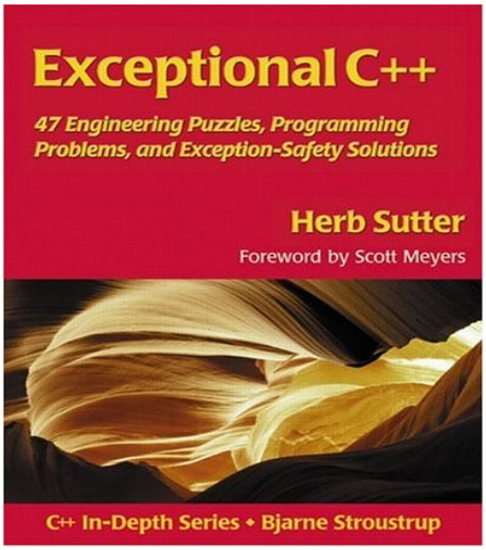
Congratulations! You've made it through CS106L. You've taken the first step on your journey toward a mastery of the C++ programming language. This is no easy feat! In the course of reading through this far, you now have a command of the following concepts:

- The streams library, including how to interface with it through operator overloading.
- STL containers, iterators, algorithms, adapters, and functional programming constructs, including a working knowledge of how these objects are put together.
- Pointer arithmetic and how objects are laid out in memory.
- The preprocessor and how to harness it to automatically generate C++ code.
- Generic programming in C++ and just how powerful the C++ template system can be.
- The `const` keyword and how to use it to communicate function side-effects to other programmers.
- Object layout and in-memory representation.
- Copy semantics and how to define implicit conversions between types.
- Operator overloading and how to make a C++ class act like a primitive type.
- What a functor is and how surprisingly useful and flexible they are.
- Exception handling and how to use objects to automatically manage resources.
- C++0x and what C++ will look like in the future.
- ... and a whole host of real-world examples of each of these techniques.

Despite all of the material we've covered here, there is *much* more to learn in the world of C++ and your journey has just begun. I feel that it is a fitting conclusion to this course reader to direct you toward other C++ resources that will prove invaluable along your journey into the wondrous realm of this language. In particular, there are several excellent C++ resources I would be remiss to omit:

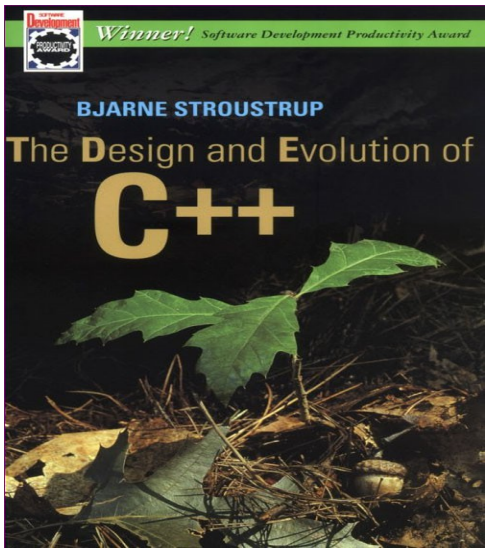


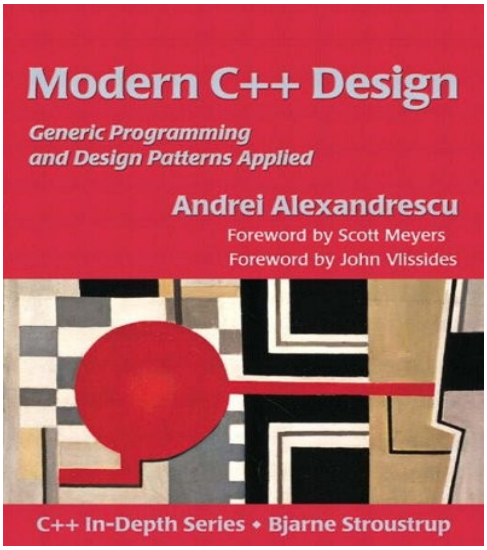
Effective C++, *More Effective C++*, and *Effective STL* by Scott Meyers. Picking up and reading this trio of books is perhaps the best thing you can do for yourself as a C++ programmer. The books in the *Effective C++* series will help you transition from a solid C++ programmer into an excellent C++ programmer and are widely regarded as among the best C++ books on the market. What separates the *Effective C++* series from most other C++ books is that *Effective C++* focuses almost exclusively on correct usage of core C++ language features and how to avoid common pitfalls. If you plan on using C++ in the professional world, you should own copies of this book.



Exceptional C++ by Herb Sutter. This book is an invaluable tool in any C++ programmer's arsenal. The book is largely organized as a set of puzzles that give you a chance to think about the best way to solve a problem and what C++ issues you'll encounter in the process. Along with *Effective C++*, *Exceptional C++* is one of the most highly-recommended C++ books out there. Herb Sutter's personal website is also an excellent resource for all your C++ needs.

The Design and Evolution of C++ by Bjarne Stroustrup. This book, affectionately known to hardcore C++ programmers as *D&E*, is a glimpse into Bjarne Stroustrup's thought processes as he went about designing C++. D&E is not a programming guide, but rather a history of the evolution of C++ from the small language C with Classes into the modern language we know and love today. D&E was written before C++ had been ISO standardized and even predates the STL, meaning that it can offer a new perspective on some of the language features and libraries you may take for granted. If you want an interesting glimpse into the mind of the man behind C++, this is the book for you.





Modern C++ Design: Generic Programming and Design Patterns Applied by Andrei Alexandrescu. Considered the seminal work in modern C++ programming, this book is an excellent introduction into an entirely new way of thinking in C++. Alexandrescu takes many advanced language features like templates and multiple inheritance, then shows how to harness them to achieve synergistic effects that are far more powerful than any of the individual features used. As an example, the first chapter shows how to write a single smart pointer class that is capable of storing any type of value, performing any sort of resource management, and having any copy behavior that the client desires. The book is very language-intensive and requires you to have a grasp of C++ slightly beyond the scope of this reader, but is a most wonderful text for all who are interested.

Final Thoughts

It's been quite a trip since we first started with the streams library. You now know how to program with the STL, write well-behaved C++ objects, and even how to use functional programming constructs. But despite the immense amount of material we've covered, we have barely scratched the surface of C++. There are volumes of articles and books out there that cover all sorts of amazing C++ tips and tricks, and by taking the initiative and exploring what's out there you can hone your C++ skills until problem solving in C++ transforms from "how do I solve this problem?" to "which of these many options is best for solving this problem?"

C++ is an amazing language. It has some of the most expressive syntax of any modern programming language, and affords an enormous latitude in programming styles. Of course, it has its flaws, as critics are eager to point out, but despite the advent of more modern languages like Java and Python C++ still occupies a prime position in the software world.

I hope that you've enjoyed reading this course reader as much as I enjoyed writing it. If you have any comments, suggestions, or criticisms, feel free to email me at htiek@cs.stanford.edu. Like the C++ language, CS106L and this course reader are constantly evolving, and if there's anything I can do to make the class more enjoyable, be sure to let me know!

Have fun with C++, and I wish you the best of luck wherever it takes you!

- Keith

Part Four

Object-Oriented Programming

Chapter 16: Introduction to Inheritance

It's impossible to learn C++ or any other object-oriented language without encountering *inheritance*, a mechanism that lets different classes share implementation and interface design. However, inheritance has evolved greatly since it was first introduced to C++, and consequently C++ supports several different inheritance schemes. This chapter introduces and motivates inheritance, then discusses how inheritance interacts with other language features.

Inheritance of Implementation and Interface

C++ started off as a language called "C with Classes," so named because it was essentially the C programming language with support for classes and object-oriented programming. C++ is the more modern incarnation of C with Classes, so most (but not all) of the features of C with Classes also appear in C++.

The inheritance introduced in C with Classes allows you to define new classes in terms of older ones. For example, suppose you are using a third-party library that exports a `Printer` class, as shown below:

```
class Printer
{
public:
    /* Constructor, destructor, etc. */

    void setFont(const string& fontName, int size);
    void setColor(const string& color);
    void printDocument(const string& document);
private:
    /* Implementation details */
};
```

This `Printer` class exports several formatting functions, plus `printDocument`, which accepts a string of the document to print. Let's assume that `printDocument` is implemented synchronously – that is, `printDocument` will not return until the document has finished printing. In some cases this behavior is fine, but in others it's simply not acceptable. For example, suppose you're writing database software for a large library and want to give users the option to print out call numbers. Chances are that people using your software will print call numbers for multiple books and will be irritated if they have to sit and wait for their documents to finish printing before continuing their search. To address this problem, you decide to add a new feature to the printer that lets the users enqueue several documents and print them in a single batch job. That way, users searching for books can enqueue call numbers without long pauses, then print them all in one operation. However, you don't want to force users to queue up documents and then do a batch print job at the end – after all, maybe they're just looking for one book – so you want to retain all of the original features of the `Printer` class. How can you elegantly solve this problem in software?

Let's consider the above problem from a programming perspective. The important points are:

- We are provided the `Printer` class from an external source, so we cannot modify the `Printer` interface.
- We want to preserve all of the existing functionality from the `Printer` class.
- We want to extend the `Printer` class to include extra functionality.

This is an ideal spot to use *inheritance*, a means for defining a new class in terms of an older one. We have an existing class that contains most of our needed functionality, but we'd like to add some extra features.

Let's define a `BatchPrinter` class that supports two new functions, `enqueueDocument` and `printAllDocuments`, on top of all of the regular `Printer` functionality. In C++, we write this as

```
class BatchPrinter: public Printer // Inherit from Printer
{
public:
    void enqueueDocument(const string& document);
    void printAllDocuments();
private:
    queue<string> documents; // Document queue
};
```

Here, the class declaration `class BatchPrinter: public Printer` indicates that the new class `BatchPrinter` inherits the functionality of the `Printer` class. Although we haven't explicitly provided the `printDocument` or `setFont` functions, since those functions are defined in `Printer`, they are also part of `BatchPrinter`. For example:

```
BatchPrinter myPrinter;
myPrinter.setColor("Red");                              // Inherited from Printer
myPrinter.printDocument("This is a document!");         // Same
myPrinter.enqueueDocument("Print this one later."); // Defined in BatchPrinter
myPrinter.printAllDocuments();                          // Same
```

While the `BatchPrinter` can do everything that a `Printer` can do, the converse is not true – a `Printer` cannot call `enqueueDocument` or `printAllDocuments`, since we did not modify the `Printer` class interface.

In the above setup, `Printer` is called a *base class* of `BatchPrinter`, which is a *derived class*. In C++ jargon, the relationship between a derived class and its base class is the *is-a* relationship. That is, a `BatchPrinter` *is-a* `Printer` because everything the `Printer` can do, the `BatchPrinter` can do as well. The converse is not true, though, since a `Printer` is not necessarily a `BatchPrinter`.

Because `BatchPrinter` *is-a* `Printer`, anywhere that a `Printer` is expected we can instead provide a `BatchPrinter`. For example, suppose we have a function that accepts a `Printer` object, perhaps to configure its font rendering, as shown here:

```
void InitializePrinter(Printer& p);
```

Then the following code is perfectly legal:

```
BatchPrinter batch;
InitializePrinter(batch);
```

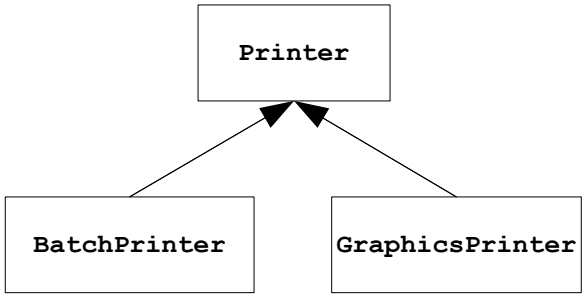
Although `InitializePrinter` expects an argument of type `Printer&`, we can instead provide it a `BatchPrinter`. This operations is well-defined and perfectly safe because the `BatchPrinter` contains all of the functionality of a regular `Printer`. If we temporarily forget about all of the extra functionality provided by the `BatchPrinter` class, we still have a good old-fashioned `Printer`. When working with inheritance, you can think of the types of arguments to functions as specifying the minimum requirements for the parameter. A function accepting a `Printer&` or a `const Printer&` can take in a object of any type, provided of course that it ultimately derives from `Printer`.

Note that it is completely legal to have several classes inherit from a single base class. Thus, if we wanted to develop another printer that supported graphics printing in addition to text, we could write the following class definition:

```
class GraphicsPrinter: public Printer
{
public:
    /* Constructor, destructor, etc. */
    void printPicture(const Picture& picture); // For some Picture class
private:
    /* Implementation details */
};
```

Now, `GraphicsPrinter` can do everything a regular `Printer` can do, but can also print `Picture` objects. Again, `GraphicsPrinter` *is-a* `Printer`, but not vice-versa. Similarly, `GraphicsPrinter` is not a `BatchPrinter`. Although they are both derived classes of `Printer`, they have nothing else in common.

It sometimes help to visualize the inheritance relations between classes as a tree. We adopt the convention that if one class derives from another, the first class is represented as a child of the second. We also label all edges in the tree with arrows pointing from derived classes to base classes. For example, `Printer`, `BatchPrinter`, and `GraphicsPrinter` are all related as follows:



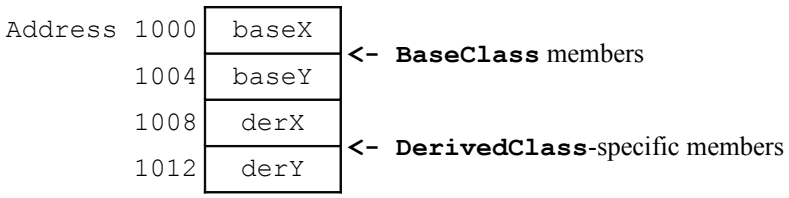
Runtime Costs of Basic Inheritance

The inheritance scheme outlined above incurs no runtime penalties. Programs using this type of inheritance will be just as fast as programs not using inheritance.

In memory, a derived class is simply a base class object with its extra data members tacked on the end. For example, suppose you have the following classes:

```
class BaseClass
{
private:
    int baseX, baseY;
};

class DerivedClass: public BaseClass
{
private:
   int derX, derY;
};
```

Then, in memory, a `DerivedClass` object looks like this:



Notice that the first eight bytes of this object are precisely the data members of a `BaseClass` object. This is in part the reason that you can treat instances of a derived class as instances of a base class. This in-memory representation of inheritance is extremely efficient and is one of the reasons that C++ was so popular in its infancy. Most competing object-oriented languages represented objects with considerably more complicated structures that required complex pointer lookups, so inheritance in those languages incurred a steep runtime penalty. C++, on the other hand, supported this simple form of inheritance with zero overhead.

Inheritance of Interface

The inheritance pattern outlined above uses inheritance to add *extensions* to existing classes. This is undoubtedly useful, but does not arise frequently in practice. A different version of inheritance, called *inheritance of interface*, is extraordinarily useful in modern programming.

Let's return to the `Printer` class. `Printer` exports a `printDocument` member function that accepts a `string` parameter, then sends the string to the printer. One of our other derived classes, `GraphicsPrinter`, has a `printPicture` member function that accepts some sort of `Picture` object, then sends the picture to the printer. What if we want to print a document containing a mix of text and pictures – for example, this course reader? We'd then need to introduce yet another subclass of `Printer`, perhaps a `MixedTextPrinter`, that supports a `printMixedText` member function that prints a combination of text and images. While we could continue to use the style of inheritance introduced above, it will quickly spiral out of control for several reasons. First, each printer can only print out one type of document. That is, a `MixedTextPrinter` cannot print out pictures, nor a `GraphicsPrinter` a mixed-text document. We could eliminate this problem by writing a single `MixedTextAndGraphicsPrinter` class, but this too has its problems if we then introduce another type of object to print (say, a high-resolution photo) that required its own special printing code. This leads to the second problem, a lack of extensibility. For any new type of object we want to print, we need to introduce another member function or class capable of handling that object. In our case this is inconvenient and does not scale well. We need to pick another plan of attack.

The problem is that everything that might get sent to the printer requires slightly different logic to print out. Text documents need to apply fonts and formatting transformations, graphics documents need to convert from some application-specific format into something readable by the printer, and mixed-text documents need to arrange their text and images into an appropriate layout before processing each piece individually. But each type of document has one piece of functionality in common. Most printers are programmed to accept as input a grid of dots representing the document to print. That is, whether you're printing text or a three-dimensional pie chart, the input to the printer is a grid of pixels representing the dots making up the image. A text document might end up producing different pixels than a high-resolution photo, but they both end up as pixels at some point. Provided that we can transform an object in memory into a mess of pixels, we can send it to the printer.

Consider the following class definition for a `GenericBatchPrinter` object:

```
class GenericBatchPrinter
{
public:
    /* Constructor, destructor, etc. */

    void enqueueDocument( /* What goes here? */ );
    void printAllDocuments();
private:
    /* Implementation details */
};
```

This `GenericBatchPrinter` object exports an `enqueueDocument` function that stores an arbitrary document in a print queue that can then be printed by calling `printAllDocuments`. There is one major question, though: what should the parameter type be? We can print any type of document we can think of, provided that we can convert it into a grid of pixels. We might be tempted to accept a grid of pixels as a parameter. This, however, has several drawbacks. First, pixel grids take up a huge amount of memory. Color printers usually store color information as quadruples of the cyan, magenta, yellow, and black (CYMK) color components, so a single pixel is usually a four-byte value. If you have a 200 DPI printer and want to print to an 8.5 x 11" page, you'd need to store around 75kb. That's a lot of memory, and if you wanted to enqueue a large number of documents this approach might strain or exhaust system resources. Plus, we don't actually need the pixels until we begin printing, and even then we only need pixel information for one document at a time. Second, what if later in design we realize that we need extra information about the print job? For example, suppose we want to implement a printing priority system where more urgent documents print before less important ones. In this case, we'd need to add an extra parameter to `enqueueDocument` representing that priority and all existing code using `enqueueDocument` would stop working. Finally, this approach exposes too much of the inner workings of `GenericBatchPrinter` to the client. By treating documents as masses of pixels instead of documents, the `GenericBatchPrinter` violates some of the fundamental rules of data abstraction.

Let's review these problems:

- The above approach is needlessly memory-intensive by catering to the lowest common denominator of all possible printable documents.
- The approach limits later extensions by fixing the parameter as an inflexible pixel array.
- The `GenericBatchPrinter` should work on documents, not pixels.

Is there a language feature that would let us solve all of these problems? The answer is yes. What if we simply create an object that looks like this:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    grid<pixelT> convertToPixelArray() const; // For some struct pixelT
    int getPriority() const;
private:
    /* Implementation details */
};
```

This `Document` class exports two functions – `convertToPixelArray()`, which converts the document from its current format into a `grid<pixelT>` of the pixels in the image, and `getPriority()`, which returns the relative priority of the document.

We can now have the `enqueueDocument` function from `GenericBatchPrinter` accept a `Document` as a parameter. That way, when the `GenericBatchPrinter` needs to get an array of pixels, it can simply call `convertToPixelArray` on any stored `Document`. Similarly, if the `GenericBatchPrinter` decides to implement a priority system, it can use the information provided by the `Document`'s `getPriority()` function. Moreover, if later on during implementation we realize that `GenericBatchPrinter` needs access to additional information, we can simply add extra member functions to the `Document` class. While this still requires us to rewrite code to add these member functions, the actual calls to `enqueueDocument` will still work correctly, and the only people who need to modify any code is the `Document` class implementer, not the `Document` class clients.

While this solution might seem elegant, it still has a major problem – how can we write a `Document` class that encompasses all of the possible documents we can try to print? The answer is simple: we can't. Using the language features we've covered so far, it simply isn't possible to solve this problem.

Consider for a minute what form our problem looks like. We need to provide a `Document` object that represents a printable document, but we cannot write a single umbrella class representing every conceivable document. Instead of creating a single `Document` class, what if we could create *several different* `Document` classes, each of which provided a working implementation of the `convertToPixelArray` and `getDocumentName` functions? That is, we might have a `TextDocument` class that stores a `string` and whose `convertToPixelArray` converts the string into a grid of pixels representing that string. We could also have a `GraphicsDocument` object with member functions like `addCircle` or `addImage` whose `convertToPixelArray` function generates a graphical representation of the stored image. In other words, we want to make the `Document` class represent an *interface* rather than an *implementation*. `Document` should simply outline what member functions are common to other classes like `GraphicsDocument` or `TextDocument` without specifying how those functions should work. In C++ code, this means that we will rewrite the `Document` class to look like this:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual int* convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    /* Implementation details */
};
```

If you'll notice, we tagged both of the member functions with the `virtual` keyword, and put an `= 0` after each function declaration. What does this strange syntax mean? The `= 0` syntax is an odd bit of C++ syntax that says “this function does not actually exist.” In other words, we've prototyped a function that we have no intention of ever writing. Why would we ever want to do this? The reason is simple. Because we've prototyped the function, other pieces of C++ code know what the parameter and return types are for the `convertToPixelArray` and `getDocumentName` functions. However, since there is no meaningful implementation for either of these functions, we add the `= 0` to tell C++ not to expect one.

To understand the `virtual` keyword, consider this `TextDocument` class outlined below:

```
class TextDocument: public Document // Inherit from Document
{
public:
    /* Constructor, destructor, etc. */
    virtual grid<pixelT> convertToPixelArray() const; // Has an implementation
    virtual int getPriority() const;   // Has an actual implementation

    void setText(const string& text); // Text-specific formatting functions
    void setFont(const string& font);
    void setSize(int size);
private:
    /* Implementation details */
};
```

This `TextDocument` class inherits from `Document`, and although `Document` has a declaration of the `convertToPixelArray` and `getPriority` functions, `TextDocument` has specified that it too contains these functions. They are marked `virtual`, as in the `Document` class, but unlike `Document`'s versions of these functions the `TextDocument` functions do not have the `= 0` notation after them. This indicates that the functions actually exist and do have implementations. We won't cover how these functions are implemented since it's irrelevant to our discussion, but because they have actual implementations code like this is perfectly legal:

```
TextDocument myDocument;
grid<pixelT> array = myDocument.convertToPixelArray();
```

We've covered the `= 0` notation, but what does the `virtual` keyword mean? To understand how `virtual` works, consider the following code snippet:

```
TextDocument* myDocument = new TextDocument;
grid<pixelT> array = myDocument->convertToPixelArray();
```

This code should not be at all surprising – we've just rewritten the above code using a pointer to a `TextDocument` rather than a stack-based `TextDocument`. However, consider this code snippet below:

```
Document* myDocument = new TextDocument;        // Note: pointer is a Document *
grid<pixelT> array = myDocument->convertToPixelArray();
```

This code looks similar to the above code but represents a fundamentally different operation. In the first line, we allocate a new `TextDocument` object, but store it in a pointer of type `Document *`. Initially, this might seem nonsensical – pointers of type `Document *` should only be able to point to objects of type `Document`. However, because `TextDocument` is a derived class of `Document`, *`TextDocument` is-a `Document`*. The *is-a* relation applies literally here – since `TextDocument` *is-a* `Document`, we can point to objects of type `TextDocument` using pointers of type `Document *`.

Even if we can point to objects of type `TextDocument` with objects of type `Document *`, why is the line `myDocument->convertToPixelArray()` legal? As mentioned earlier, the `Document` class definition says that `Document` does not have an implementation of `convertToPixelArray`, so it seems like this code should not compile. This is where the `virtual` keyword comes in. Since we marked `convertToPixelArray` `virtual`, when we call the `convertToPixelArray` function through a `Document *` object, C++ will call the function named `convertToPixelArray` for the class that's *actually being pointed at*, not the `convertToPixelArray` function defined for objects of the type of the pointer. In this case, since our `Document *` is actually pointing at a `TextDocument`, the call to `convertToPixelArray` will call the `TextDocument`'s version of `convertToPixelArray`.

The above approach to the problem is known as *polymorphism*. We define a base class (in this case `Document`) that exports several functions marked `virtual`. In our program, we pass around pointers to objects of this base class, which may in fact be pointing to a base class object or to some derived class. Whenever we make member function calls to the virtual functions of the base class, C++ figures out at runtime what type of object is being pointed at and calls its implementation of the virtual function.

Let's return to our `GenericBatchPrinter` class, which now in its final form looks something like this:

```
class GenericBatchPrinter
{
public:
    /* Constructor, destructor, etc. */

    void enqueueDocument(Document* doc);
    void printAllDocuments();
private:
    queue<Document *> documents;
};
```

Our `enqueueDocument` function now accepts a `Document *`, and its private data members include an STL `queue` of `Document *`s. We can now implement `enqueueDocument` and `printAllDocuments` using code like this:

```
void GenericBatchPrinter::enqueueDocument(Document* doc)
{
    documents.push(doc); // Recall STL queue uses push instead of enqueue
}

void GenericBatchPrinter::printAllDocuments()
{
    /* Print all enqueued documents */
    while(!documents.empty())
    {
        Document* nextDocument = documents.front();
        documents.pop(); // Recall STL queue requires explicit pop operation

        sendToPrinter(nextDocument->convertToPixelArray());
        delete nextDocument; // Assume it was allocated with new
    }
}
```

The `enqueueDocument` function accepts a `Document *` and enqueues it in the document queue, and the `printAllDocuments` function continuously dequeues documents, converts them to pixel arrays, then sends them to the printer. But while this above code might seem simple, it's actually working some wonders behind the scenes. Notice that when we call `nextDocument->convertToPixelArray()`, the object pointed at by `nextDocument` could be of *any type* derived from `Document`. That is, the above code will work whether we've enqueued `TextDocument`s, `GraphicsDocument`s, or even `MixedTextDocument`s. Moreover, the `GenericBatchPrinter` class does not even need to know of the existence of these types of documents; as long as `GenericBatchPrinter` knows the generic `Document` interface, C++ can determine which functions to call. This is the main strength of inheritance – we can write code that works with objects of arbitrary types by identifying the common functionality across those types and writing code solely in terms of these operations.

Virtual Functions, Pure Virtual Functions, and Abstract Classes

In the above example with the `Document` class, we defined `Document` as

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual grid<pixelT> convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    /* Implementation details */
};
```

Here, all of the `Document` member functions are marked `virtual` and have the `= 0` syntax to indicate that the functions are not actually defined. Functions marked `virtual` with `= 0` are called *pure virtual functions* and represent functions that exist solely to define how other pieces of C++ code should interact with derived classes.*

Classes that contain pure virtual functions are called *abstract classes*. Because abstract classes contain code for which there is no implementation, it is illegal to directly instantiate abstract classes. In the case of our document example, this means that both of the following are illegal:

```
Document myDocument; // Error!
Document* myDocument = new Document; // Error!
```

Of course, it's still legal to declare `Document *` variables, since those are pointers to abstract classes rather than abstract classes themselves.

A derived class whose base class is abstract may or may not implement all of the pure virtual functions defined in the base class. If the derived class does implement each function, then the derived class is non-abstract (unless, of course, it introduces its own pure virtual functions). Otherwise, if there is at least one pure virtual function declared in the base class and not defined in the derived class, the derived class itself will be an abstract class.

There is no requirement that functions marked `virtual` be pure virtual functions. That is, you can provide `virtual` functions that have implementations. For example, consider the following class representing a roller-blader:

```
class RollerBlader
{
public:
    /* Constructor, destructor, etc. */

    virtual void slowDown(); // Virtual, not pure virtual
private:
    /* Implementation details */
};
```

* Those of you familiar with inheritance in other languages like Java might wonder why C++ uses the awkward `= 0` syntax instead of a clearer keyword like `abstract` or `pure`. The reason was mostly political. Bjarne Stroustrup introduced pure virtual functions to the C++ language several weeks before the planned release of the next set of revisions to C++. Adding a new keyword would have delayed the next language release, so to ensure that C++ had support for pure virtual functions, he chose the `= 0` syntax.

```
void RollerBlader::slowDown() // Implementation doesn't have virtual keyword
{
    applyBrakes();
}
```

Here, `slowDown` is implemented as a virtual function that is not pure virtual. In the implementation of `slowDown`, you do not repeat the `virtual` keyword, and for all intents and purposes treat `slowDown` as a regular C++ function. Now, suppose we write a `InexperiencedRollerBlader` class, as shown here:

```
class InexperiencedRollerBlader: public RollerBlader
{
public:
    /* Constructor, destructor, etc. */

    virtual void slowDown();
private:
    /* Implementation details */
};

void InexperiencedRollerBlader::slowDown()
{
    fallDown();
}
```

This `InexperiencedRollerBlader` class provides its own implementation of `slowDown` that calls some `fallDown` function.* Now, consider the following code snippet:

```
RollerBlader* blader = new RollerBlader;
blader->slowDown();

RollerBlader* blader2 = new InexperiencedRollerBlader;
blader2->slowDown();
```

In both cases, we call the `slowDown` function through a pointer of type `RollerBlader *`, so C++ will call the version of `slowDown` for the class that's actually pointed at. In the first case, this will call the `RollerBlader`'s version of `slowDown`, which calls `applyBrakes`. In the second, since `blader2` points to an `InexperiencedRollerBlader`, the `slowDown` call will call `InexperiencedRollerBlader`'s `slowDown` function, which then calls `fallDown`. In general, when calling a virtual function, C++ will invoke the version of the function that corresponds to the most derived implementation available in the object being pointed at. Because the `InexperiencedRollerBlader` implementation of `slowDown` replaces the base class version, `InexperiencedRollerBlader`'s implementation `slowDown` is said to *override* `RollerBlader`'s.

When inheriting from non-abstract classes that contain virtual functions, there is no requirement to provide your own implementation of the virtual functions. For example, a `StuntRollerBlader` might be able to do tricks a regular `RollerBlader` can't, but still slows down the same way. In code we could write this as

```
class StuntRollerBlader: public RollerBlader
{
public:
    /* Note: no mention of slowDown */
    void backflip();
    void tripleAxel();
```

* Of course, this is not based on personal experience. ☺

```
};
```

If we then were to write code that used `StuntRollerBlader`, calling `slowDown` would invoke `RollerBlader`'s version of `slowDown` since it is the most derived implementation of `slowDown` available to `StuntRollerBlader`. For example:

```
RollerBlader* blader = new StuntRollerBlader;
blader->slowDown(); // Calls RollerBlader::slowDown
```

Similarly, if we were to create a class `TerriblyInexperiencedRollerBlader` that exports a `panic` function but no `slowDown` function, as shown here:

```
class TerriblyInexperiencedRollerBlader: public InexperiencedRollerBlader
{
public:
    /* Note: no reference to slowDown */
    void panic();
};
```

Then the following code will invoke `InexperiencedRollerBlader::slowDown`, causing the roller blader to fall down:

```
RollerBlader* blader = new TerriblyInexperiencedRollerBlader;
blader->slowDown();
```

In this last example we wrote a class that derived from a class which itself was a derived class. This is perfectly legal and arises commonly in programming practice.

A Word of Warning

Consider the following two classes:

```
class NotVirtual
{
public:
    void notAVirtualFunction();
};

class NotVirtualDerived: public NotVirtual
{
public:
    void notAVirtualFunction();
};
```

Here, the base class `NotVirtual` exports a function called `notAVirtualFunction` and its derived class, `NotVirtualDerived`, also provides a `notAVirtualFunction` function. Although these functions have the same name, since `notAVirtualFunction` is not marked `virtual`, the derived class version does *not* replace the base class version. Consider this code snippet:

```
NotVirtual* nv = new NotVirtualDerived;
nv->notAVirtualFunction();
```

Here, since `NotVirtualDerived` is-a `NotVirtual`, the above code will compile. However, since `notAVirtualFunction` is (as its name suggests) not a virtual function, the above code will call the `NotVirtual` version of `notAVirtualFunction`, not `NotVirtualDerived`'s `notAVirtualFunction`.

If you want to let derived classes override functions in a base class, you *must* mark the base class's function `virtual`. Otherwise, C++ won't treat the function call virtually and will always call the version of the function associated with the type of the pointer. For example:

```
NotVirtual* nv = new NotVirtualDerived;
nv->notAVirtualFunction();  // Calls NotVirtual::notAVirtualFunction()

NotVirtualDerived *nv2 = new NotVirtualDerived;
nv2->notAVirtualFunction(); // Calls NotVirtualDerived::notAVirtualFunction();
```

In general, it is considered bad programming practice to have a derived class implement a member function with the same name as a non-virtual function in its base class. Doing so leads to the sorts of odd behavior shown above and is an easy source of errors.

The `protected` Access Specifier

Let's return to the `Document` class from earlier in the chapter. Suppose that while designing some of the `Document` subclasses, we note that every single subclass ends up having a `width` and `height` field. To minimize code duplication, we decide to move the `width` and `height` fields from the derived classes into the `Document` base class. Since we don't want `Document` class clients directly accessing these fields, we decide to mark them `private`, as shown here:

```
class Document
{
public:
    /* Constructor, destructor, etc. */

    virtual grid<pixelT> convertToPixelArray() const = 0;
    virtual string getDocumentName() const = 0;
private:
    int width, height; // Warning: slight problem here
};
```

However, by moving `width` and `height` into the `Document` base class, we've accidentally introduced a problem into our code. Since `width` and `height` are `private`, even though `TextDocument` and the other subclasses inherit from `Document`, the subclasses will not be able to access the `width` and `height` fields. We want the `width` and `height` fields to be accessible only to the derived classes, but not to the outside world. Using only the C++ we've covered up to this point, this is impossible. However, there is a third access specifier beyond `public` and `private` called *`protected`* that does exactly what we want. Data members and functions marked `protected`, like `private` data members, cannot be accessed by class clients. However, unlike `private` variables, `protected` functions and data members *are* accessible by derived classes.

`protected` is a useful access specifier that in certain circumstances can make your code quite elegant. However, you should be very careful when granting derived classes `protected` access to data members. Like `public` data members, using `protected` data members locks your classes into a single implementation and can make code changes down the line difficult to impossible. Make sure that marking a data member `protected` is truly the right choice before proceeding. However, marking *member functions* `protected` is a common programming technique that lets you export member functions only usable by derived classes. We will see an example of `protected` member functions later in this chapter.

Virtual Destructors

An important topic we have ignored so far is *virtual destructors*. Consider the following two classes:

```
class BaseClass
{
public:
    BaseClass();
    ~BaseClass();
};

class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();
private:
   char* myString;
};

DerivedClass::DerivedClass()
{
    myString = new char[128]; // Allocate some memory
}

DerivedClass::~DerivedClass()
{
    delete [] myString; // Deallocate the memory
}
```

Here, we have a trivial constructor and destructor for `BaseClass`. `DerivedClass`, on the other hand, has a constructor and destructor that allocate and deallocate a block of memory. What happens if we write the following code?

```
BaseClass* myClass = new DerivedClass;
delete myClass;
```

Intuitively, you'd think that since `myClass` points to a `DerivedClass` object, the `DerivedClass` destructor would invoke and clean up the dynamically-allocated memory. Unfortunately, this is not the case. The `myClass` pointer is statically-typed as a `BaseClass *` but points to an object of type `DerivedClass`, so `delete myClass` results in *undefined behavior*. The reason for this is that we didn't let C++ know that it should check to see if the object pointed at by a `BaseClass *` is really a `DerivedClass` when calling the destructor. Undefined behavior is never a good thing, so to fix this we mark the `BaseClass` destructor `virtual`. Unlike the other virtual functions we've encountered, though, derived class destructors do not replace the base class destructors. Instead, when invoking a destructor virtually, C++ will first call the derived class destructor, then the base class destructor. We thus change the two class declarations to look like this:

```
class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass();
};
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();
private:
   char *myString;
};
```

There is one more point to address here, the *pure virtual destructor*. Because virtual destructors do not act like regular virtual functions, even if you mark a destructor pure virtual, you must still provide an implementation. Thus, if we rewrote `BaseClass` to look like

```
class BaseClass
{
public:
    BaseClass();
    virtual ~BaseClass() = 0;
};
```

We'd then need to write a trivial implementation for the `BaseClass` destructor, as shown here:

```
BaseClass::~BaseClass()
{
    // Do nothing
}
```

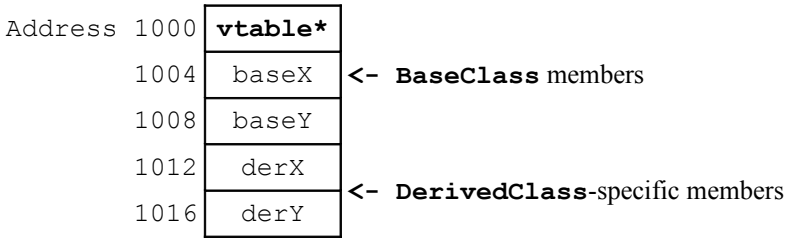
This is an unfortunate language quirk, but you should be aware of it since this will almost certainly come up in the future.

Runtime Costs of Virtual Functions

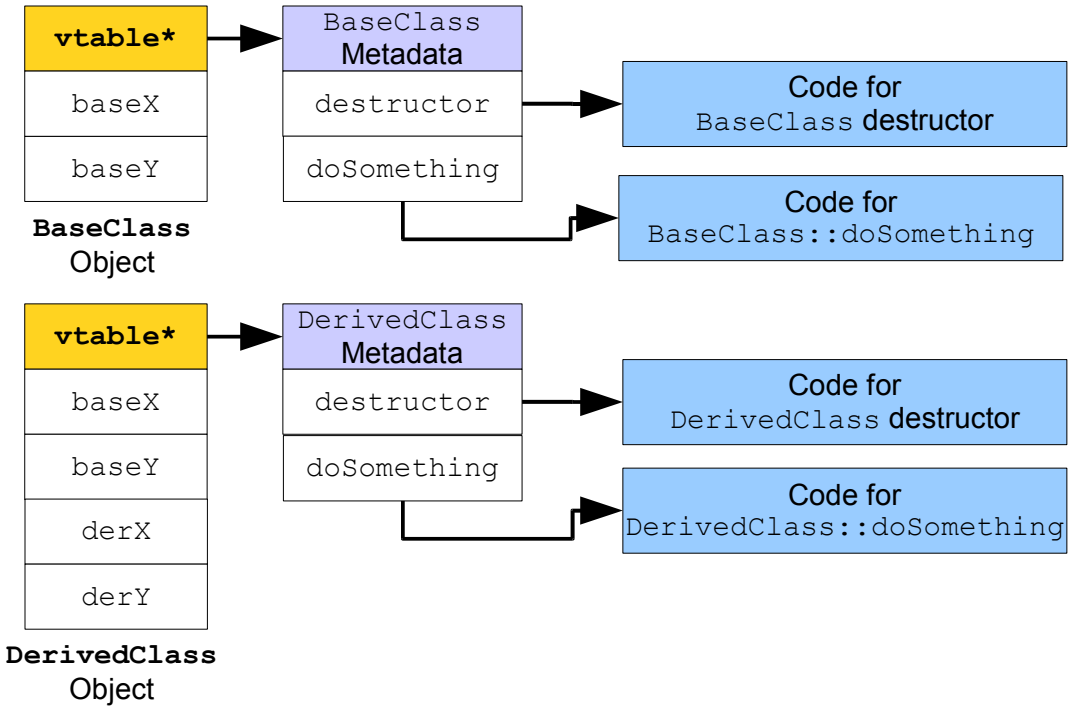
Virtual functions are incredibly useful and syntactically concise, but exactly how efficient are they? After all, a virtual function call invokes one of many possible functions, and somehow the compiler has to determine which version of the function to call. There could be an arbitrarily large number of derived classes overriding the particular virtual function, so a naïve `switch` statement that checks the type of the object would be prohibitively expensive. Fortunately, most C++ compilers use a particularly clever implementation of virtual functions that, while slower than regular function calls, are much faster than what you may have initially expected. Consider the following classes:

```
class BaseClass
{
public:
    virtual ~BaseClass() {} // Polymorphic classes need virtual destructors
    virtual void doSomething();
private:
    int baseX, baseY;
};

class DerivedClass: public BaseClass
{
public:
    virtual void doSomething(); // Override BaseClass version
private:
    int derX, derY;
};
```

Then the in-memory representation of a `DerivedClass` object looks like this:



As before, we see the `BaseClass` members followed by the `DerivedClass` members, but there is now another piece of data in this object: the **`vtable*`**, or *vtable-pointer*. This vtable-pointer is a *pointer to the virtual function table*. Whenever you create a class containing one or more virtual functions, C++ will create a table containing information about that class that includes metadata about the class along with a list of function pointers for each of the virtual functions in that class. For example, here's a diagram of a `BaseClass` object, a `DerivedClass` object, and their respective virtual function tables:



The virtual function table for `BaseClass` begins with metadata about the `BaseClass` type, then has two function pointers – one for the `BaseClass` destructor and one for `BaseClass`'s implementation of `doSomething`. The `DerivedClass` virtual function table similarly contains information about `DerivedClass`, as well as function pointers for the destructor and `doSomething` member functions. If you'll notice, the virtual function tables for `BaseClass` and `DerivedClass` have the member functions listed in the same order, with the destructor first and then `doSomething`. This allows C++ to invoke virtual functions quickly and efficiently. Suppose that we have the following code:

```
BaseClass* myPtr = RandomChance(0.5)? new BaseClass : new DerivedClass;
myPtr->doSomething();
delete myPtr;
```

We assign a random object to `myPtr` that is either a `BaseClass` or a `DerivedClass` using the `RandomChance` function we wrote in the chapter on Snake. We then invoke the `doSomething` member function on the object and then `delete` it. To implement this functionality, the C++ compiler compiles the second two lines into machine code that performs the following operations:

`// myPtr->doSomething();`

1. Look at the first four bytes of the object pointed at by `myPtr`; this is the `vtable*` for the object.
2. Follow the `vtable*` and retrieve the second function pointer from the table; this corresponds to `doSomething`.
3. Call this function.

`// delete myPtr;`

1. Look at the first four bytes of the object pointed at by `myPtr`.
2. Follow the `vtable*` and retrieve the first function pointer from the table; this corresponds to the destructor.
3. Call this function.
4. Deallocate the memory pointed at by `myPtr`.

This sequence of commands can be executed quickly and is efficient no matter how many subclasses of `BaseClass` exist. If there are millions of derived classes, this code still only has to make a single lookup through the virtual function table to call the proper function.

Although this above implementation of virtual function calls is considerably more efficient than a naïve approach, it is still noticeably slower than a regular function call because of the necessary virtual function table lookups. This extra overhead is the reason that C++ requires you to explicitly mark member functions you want to treat polymorphically `virtual` – if all functions were called this way, you would pay a performance hit irrespective of whether you actually used inheritance, a violation of the zero-overhead principle.

Invoking Virtual Member Functions Non-Virtually

From time to time, you will need to be able to explicitly invoke a base class's version of a virtual function. For example, suppose that you're designing a `HybridCar` that's a specialization of `Car`, both of which are defined below:

```
class Car
{
public:
    virtual ~Car(); // Polymorphic classes need virtual destructors
    virtual void applyBrakes();
    virtual void accelerate();
};

class HybridCar: public Car
{
public:
    virtual void applyBrakes();
    virtual void accelerate();
private:
    void chargeBattery();
    void dischargeBattery();
};
```

The `HybridCar` is exactly the same as a regular car, except that whenever a `HybridCar` slows down or speeds up, the `HybridCar` charges and discharges its electric motor to conserve fuel. We want to implement the `applyBrakes` and `accelerate` functions inside `HybridCar` such that they perform exactly the same tasks as the `Car`'s version of these functions, but in addition perform the extra motor management.

Initially, we might consider implementing these functions like this:

```
void HybridCar::applyBrakes()
{
    applyBrakes(); // Uh oh...
    chargeBattery();
}

void HybridCar::accelerate()
{
    accelerate(); // Uh oh...
    dischargeBattery();
}
```

The above code is well-intentioned but incorrect. At a high level, we *want* to have the hybrid car accelerate or apply its brakes by doing whatever a regular car does, then managing the motor. As written, though, these functions will cause a stack overflow, since the calls to `applyBrakes()` and `accelerate()` recursively invoke the `HybridCar`'s versions of these functions over and over. Since this doesn't work, what other approaches might we try? First, we could simply copy and paste the code from the `Car` class into the `HybridCar` class. This should cause you to cringe – a good solution to a problem should *never* involve copying and pasting code! More concretely, though, this approach has several problems. First, if we change the implementation of `accelerate()` or `applyBrakes()` in the `Car` class, we have to remember to make the same changes inside `HybridCar`. If we forget to do so, the code will compile but will be incorrect. Moreover, if the implementation of `accelerate()` or `applyBrakes()` in the `Car` class reference private data members or member functions of `Car`, the resulting code will be illegal. This clearly isn't the right way to solve this problem. What other options are available?

A second idea is to factor out the code for `applyBrakes` and `accelerate` into `protected`, non-virtual functions of the `Car` class. For example:

```
class Car
{
public:
    virtual ~Car();
    virtual void applyBrakes() { doApplyBrakes(); }
    virtual void accelerate()  { doAccelerate(); }
protected:
    void doApplyBrakes(); // Non-virtual function that actually slows down.
    void doAccelerate();  // Non-virtual function that actually accelerates.
};

class HybridCar: public Car
{
public:
    virtual void applyBrakes() { doApplyBrakes(); chargeBattery(); }
    virtual void accelerate()  { doAccelerate();  dischargeBattery(); }
private:
    void chargeBattery();
    void dischargeBattery();
};
```

Here, the virtual functions `applyBrakes` and `accelerate` are wrapped calls to non-virtual, protected functions written in the base class. To implement the derived versions of `applyBrakes` and `accelerate`, we can simply call these functions.

This approach is stylistically pleasing. The code that's common to `applyBrakes` and `accelerate` is factored out into helper member functions, so changes to one function appear in the other. But there's one minor problem with this approach: this solution only works if we can modify the `Car` class. In small projects this shouldn't be a problem, but if these classes are pieces in a much larger system the code may be off-limits – maybe it's being developed by another team, or perhaps it's been compiled into a program that expects the class definition to precisely match a specific pattern. This idea is clearly on the right track, but in some cases cannot work.

The optimal solution to this conundrum, however, is to simply have the `HybridCar`'s implementations of these functions directly call the versions of these functions defined in `Car`. When calling a virtual function through a pointer or reference, C++ ensures that the function call will "fall down" to the most derived class's implementation of that function. However, we can force C++ to call a specific version of a virtual function by calling it using the function's fully-qualified name. For example, consider this version of the `HybridCar`'s version of `applyBrakes`**:**

```
void HybridCar::applyBrakes()
{
    Car::applyBrakes(); // Call Car's version of applyBrakes, no polymorphism
    chargeBattery();
}
```

The syntax `Car::applyBrakes` instructs C++ to call the `Car` class's version of `applyBrakes`. Even though `applyBrakes` is virtual, since we've used the fully-qualified name, C++ will not resolve the call at runtime and we are guaranteed to invoke `Car`'s version of the function. We can write an `accelerate` function for `HybridCar` similarly.

When using the fully-qualified-name syntax, you're allowed to access *any* superclass's version of the function, not just the direct ancestor. For example, if `Car` were derived from the even more generic class `FourWheeledVehicle` that itself provides an `applyBrakes` method, we could invoke that version from `HybridCar` by writing `FourWheeledVehicle::applyBrakes()`. You can also use the fully-qualified name syntax as a class client, though it is rare to see this in practice.

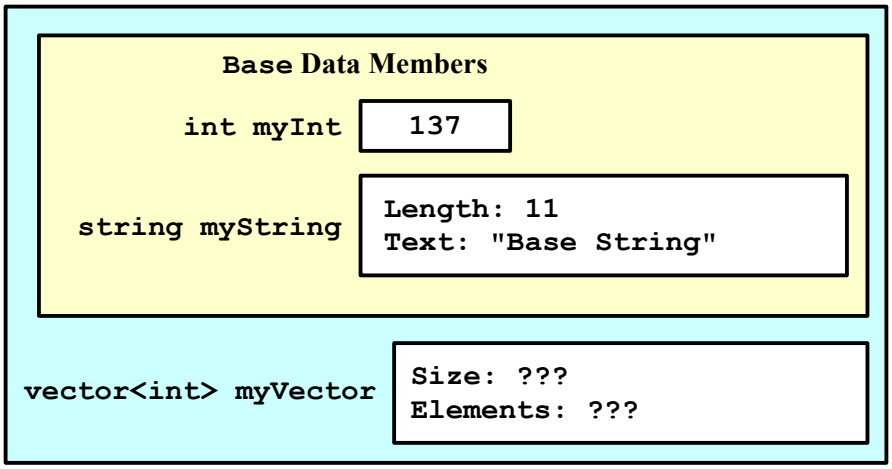
Object Initialization in Derived Classes

Recall from several chapters ago that class construction proceeds in three steps – allocating space to hold the object, calling the constructors of all data members, and invoking the object constructor. While this picture is mostly correct, it omits an important step – initialization of base classes. Let's suppose we have the following classes:

```
class Base
{
public:
    Base() : myInt(137), myString("Base string") {}
    virtual ~Base();
private:
    int myInt;
    string myString;
};

class Derived: public Base
{
private:
    vector<int> myVector;
};
```

Because we have not defined a constructor for `Derived`, C++ will automatically supply it with a default, zero-argument constructor that invokes the default constructor of the `Base` object. To see what this means, let's trace through the construction of a new `Derived` object. First, C++ gives the object a block of uninitialized memory with enough space to hold all of the parts of the `Derived`. This memory looks something like this:

At this point, C++ will initialize the `Base` class using its default constructor. Similarly, if `Base` has any parent classes, those parent classes would also be initialized. After this step, the object now looks like this:



From this point forward, construction will proceed as normal.

By default, derived class constructors invoke the default constructor for their base classes, or a zero-argument constructor if one has explicitly been defined. This is often, but not always, the desired behavior for a class. But what if you want to invoke a different constructor? For example, let's return to the `Car` example from earlier in this chapter. Suppose that `Car` exports a single constructor that accepts a `string` encoding the license number. For example:

```
class Car
{
public:
    explicit Car(const string& licenseNum) : license(licenseNum) {}
    virtual ~Car() {}

    virtual void accelerate();
    virtual void applyBrakes();
private:
    const string license;
};
```

Because `Car` no longer has a default constructor, the previous definition of `HybridCar` will cause a compile-time error because the `HybridCar` constructor cannot call the nonexistent default constructor for `Car`. How can we tell `HybridCar` to invoke the `Car` constructor with the proper arguments? The answer is similar to how we would construct a data member with a certain value – we use the initializer list. Here is a modified version of the `HybridCar` class that correctly initializes its `Car` base class:

```
class HybridCar: public Car
{
public:
    explicit HybridCar(const string& license) : Car(license) {}
    virtual void applyBrakes();
    virtual void accelerate();
private:
    void chargeBattery();
    void dischargeBattery();
};
```

Note that when using initializer lists to initialize base classes, you are only allowed to specify the names of *direct* base classes. As an example, suppose that we want to create a class called `ExperimentalHybridCar` that is similar to a `HybridCar` except that it contains extra instrumentation to monitor the state of the motor. Because `ExperimentalHybridCar` represents a prototype car, the car does not have a license plate, and so we want to communicate the string "None" up to `Car` to represent this information. Then if we define the `ExperimentalHybridCar` class as follows:

```
class ExperimentalHybridCar: public HybridCar
{
public:
    ExperimentalHybridCar();
    virtual void applyBrakes();
    virtual void accelerate();
};
```

It would be illegal to define the constructor as follows:

```
/* Note: This is not legal C++! */
ExperimentalHybridCar::ExperimentalHybridCar() : Car("None")
{
}
```

The problem with this code is that `Car` is an indirect base class of `ExperimentalHybridCar` and thus cannot be initialized from the `ExperimentalHybridCar` initializer list. The reason for this is simple. `ExperimentalHybridCar` inherits from `HybridCar`, which itself inherits from `Car`. What would happen if both `HybridCar` and `ExperimentalHybridCar` each tried to initialize `Car` in their initializer lists? Which constructor should take precedence? If it's `HybridCar`, then the initializer list for `ExperimentalHybridCar` would be ignored, leading to misleading code. If it's `ExperimentalHybridCar`, then if `HybridCar` needs to call the `Car` constructor with particular arguments, those arguments would be ignored and `HybridCar` might not be initialized correctly. To avoid this sort of confusion, C++ only lets you initialize direct base classes. Thus the proper version of the `ExperimentalHybridCar` constructor is as follows:

```
/* Tell HybridCar to initialize itself with the string "None" */
ExperimentalHybridCar::ExperimentalHybridCar() : HybridCar("None")
{
}
```

Since `HybridCar` forwards its constructor argument up to `Car`, this ends up producing the correct behavior.

Virtual Functions in Constructors

Let's take a quick look back at class construction for derived classes. If you'll recall, base classes are initialized before any of the derived class data members are set up. This means that there is a small window when the base class constructor executes where the base class is fully set up, but nothing in the derived class has yet been initialized. If the base class constructor could somehow access the data members of the derived class, it would read uninitialized memory and almost certainly crash the program. But this seems impossible – after all, the base class has no idea what's deriving from it, so how could it access any of the derived class's data members? Unfortunately, there is one way – virtual functions. Suppose the base class contains a virtual function and that one of the derived classes overrides that function to read a data member of the derived class. If the base class calls the virtual function in its constructor, it would be able to read the uninitialized value, causing a potential program crash.

The designers of C++ were well-aware of this edge case, and to prevent this error from occurring they added a restriction on the behavior of virtual function calls inside constructors. If you invoke a virtual function inside a class constructor, the function is *not* invoked polymorphically. That is, the virtual function call will always call the version of the function appropriate for the type of the base class rather than the type of the derived class. To see this in action, consider the following code:

```
class Base
{
public:
    Base()
    {
        fn();
    }
    virtual void fn()
    {
        cout << "Base" << endl;
    }
};
```

```
class Derived: public Base
{
public:
    virtual void fn()
    {
        cout << "Derived" << endl;
    }
};
```

Here, the `Base` constructor invokes its virtual function `fn`. While normally you would expect that this would invoke `Derived`'s version of `fn`, since we're inside the body of the `Base` constructor, the code will execute `Base`'s version of `fn`, which prints out "Base" instead of the expected "Derived." Cases where you would invoke a virtual function in a constructor are rare, but if you plan on doing so remember that it will not behave as you might expect.

Everything we've discussed in this section has focused on class *constructors*, but these same restrictions apply to class *destructors* as well. C++ destructs classes from the outside inward, cleaning up derived classes before base classes, and if virtual functions were treated polymorphically inside destructors it would be possible to access data members of a derived class after they'd already been cleaned up.

Copy Constructors and Assignment Operators for Derived Classes

Copy constructors and assignment operators are complicated beasts that are even more perilous when mixed with inheritance. In particular, you must make sure to invoke the copy constructor and assignment operator for any base classes in addition to any other behavior. As an example, consider the following base class, which has a well-defined copy constructor and assignment operator:

```
class Base
{
public:
    Base();
    Base(const Base& other);
    Base& operator= (const Base& other);
    virtual ~Base();
private:
    /* ... implementation specific ... */
};
```

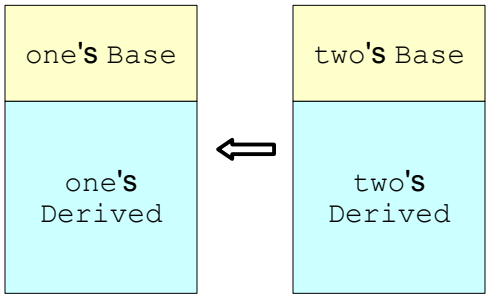
Now, consider the following derived class:

```
class Derived: public Base
{
public:
    Derived();
    Derived(const Derived& other);
    Derived& operator= (const Derived& other);
    virtual ~Derived();
private:
    char* theString; // Store a C string
    void copyOther(const Derived& other);
    void clear();
};
```

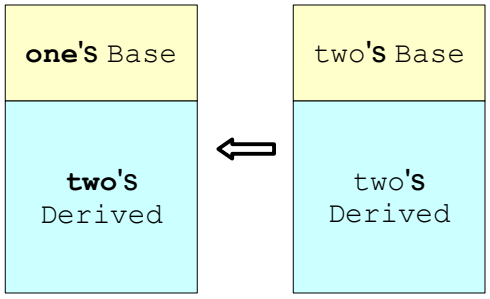
Using the template outlined in the chapter on copy functions, we might write the following code for the `Derived` assignment operator and copy constructor:

```
/* Generic "copy other" member function. */
void Derived::copyOther(const Derived& other)
{
    theString = new char[strlen(other.theString) + 1];
    strcpy(theString, other.theString);
}

/* Clear-out member function. */
void Derived::clear()
{
    delete [] theString;
    theString = NULL;
}

/* Copy constructor. */
Derived::Derived(const Derived& other) // Wrong!
{
    copyOther(other);
}

/* Assignment operator. */
Derived& Derived::operator= (const Derived& other) // Wrong!
{
    if(this != &other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}
```

Initially, it seems like this code should work, but, alas, it is seriously flawed. During this copy operation, we never instructed C++ to copy over the data from `other`'s base class into the receiver object's base class. As a result, we'll end up with half-copied data, where the data specific to `Derived` is correctly cloned but `Base`'s data hasn't changed. To see this visually, if we have two objects of type `Derived` that look like this:



After invoking the copy functions implemented above, the objects would end up in this state:



We now have a partially-copied object, which will almost certainly crash at some point down the line.

When writing assignment operators and copy constructors for derived classes, you must make sure to manually invoke the assignment operators and copy constructors for base classes to guarantee that the object is fully-copied. Fortunately, this is not particularly difficult. Let's first focus on the copy constructor. Somehow, we need to tell the receiver's base object that it should initialize itself as a copy of the parameter's base object. Because `Derived` *is-a* `Base`, so we can pass the parameter to the `Derived` copy constructor as a parameter to `Base`'s copy constructor inside the initializer list. The updated version of the `Derived` copy constructor looks like this:

```
/* Copy constructor. */
Derived::Derived(const Derived &other) : Base(other) // Correct
{
    copyOther(other);
}
```

The code we have so far for the assignment operator correctly clears out the `Derived` part of the `Derived` class, but leaves the `Base` portion untouched. How should we go about assigning the `Base` part of the receiver object the `Derived` part of the parameter? Simple – we'll invoke the `Base`'s assignment operator and have `Base` do its own copying work. The code for this is a bit odd and is shown below:

```
/* Assignment operator. */
Derived& Derived::operator= (const Derived &other)
{
    if(this != &other)
    {
        clear();
        Base::operator= (other); // Invoke the assignment operator from Base.
        copyOther(other);
    }
    return *this;
}
```

Here we've inserted a call to `Base`'s assignment operator using the full name of the `operator =` function. This is one of the rare situations where you will need to use the full name of an overloaded operator. In case you're curious why just writing `*this = other` won't work, remember that this calls `Derived`'s version of `operator =`, causing infinite recursion.

All of the above discussion has assumed that your classes require their own assignment operator and copy constructor. However, if your derived class does not contain any data members that require manual copying and assignment (for example, a derived class that simply holds an `int`), none of the above code will be necessary. C++'s default assignment operator and copy constructor automatically invoke the assignment operator and copy constructor of any base classes, which is exactly what you'd want it to do.

Disallowing Copying

Using inheritance, it's possible to elegantly and concisely disallow copying for objects of a certain type. As mentioned above, a class's default copy constructor and assignment operator automatically invoke the copy constructor and assignment operator for any base classes. But what if for some reason the derived class can't call those functions? For example, suppose that we have the following class:

```
class Uncopyable
{
public:
    /* ... */
private:
    Uncopyable(const Uncopyable&);
    Uncopyable& operator= (const Uncopyable&);
};
```

As mentioned in the chapter on copy constructors and assignment operators, this class cannot be copied because the copy constructor and assignment operator are marked private. What will happen if we then create a class that inherits from `Uncopyable`, as shown here:

```
class MyClass: public Uncopyable
{
    /* ... */
};
```

Let's assume that `MyClass` does not explicitly declare a copy constructor or assignment operator. This will cause C++ to try to create a default implementation for these functions. In the process of doing so, the compiler will realize that it needs to call the copy constructor and assignment operator of `Uncopyable`. But these functions are `private`, meaning that the derived class `MyClass` can't access them. Rather than reporting this as an error, instead the compiler doesn't create default implementations of these functions. This means that `MyClass` has no copy constructor or assignment operator, not even default implementations, and thus can't be copied or assigned. We've successfully disallowed copying!

However, by inheriting from `Uncopyable`, we've introduced some undesirable behavior. It is now legal for clients of `MyClass` to treat `MyClass` as though it were an `Uncopyable`, as shown here:

```
MyClass* mc = new MyClass;
Uncopyable* uPtr = mc;
```

This is unfortunate, since `Uncopyable` is an implementation detail of `MyClass`, not a supertype.

We are now in a rather interesting situation. We want to absorb the functionality provided by another class, but don't want to make our type a subtype of that class in the process. In other words, we want to absorb an *implementation* without its corresponding *interface*. Fortunately, using a technique called *private inheritance,* we can express this notion precisely.

So far, the inheritance you have seen has been *public inheritance*. When a class publicly inherits from a base class, it absorbs the public interface of the base class along with any implementations of the functions in that interface. In private inheritance, a derived class inherits from a base class solely to acquire its implementation. While the derived class retains the implementation of all `public` member functions from the base class, those functions become `private` in the derived class. For example, given these two classes:

```
class Base
{
public:
    void doSomething();
};

class Derived: private Base
{
    /* ... */
};
```

The following code is illegal:

```
Derived d;
d.doSomething(); // Error
```

Even though `doSomething` was declared `public` in `Base`, because `Derived` inherits privately from `Base` the `doSomething` member function is `private`.

Additionally, private inheritance does *not* define a subtyping relationship. That is, the following code is illegal:

```
Base* ptr = new Derived; // Error
```

While public inheritance models the *is-a* relationship, private inheritance represents the *is-implemented-in-terms-of* relationship. For example, we might use private inheritance to implement a `stack` in terms of a `deque`, since a `stack`'s entire functionality can be expressed through proper calls to `push_front`, `front`, and `pop_front`. This is shown here for a `stack` of integers:

```
class stack: private deque<int>
{
public:
    void push(int val)
    {
        push_front(val); // Calls deque<int>::push_front.
    }
    int pop()
    {
        const int result = front();
        pop_front();
        return result;
    }
    /* ... etc. ... */
};
```

Notice that `push` is implemented as a call to the `deque`'s `push_front` function, while `pop` is implemented through a series of calls to `front` and `pop_front`. Because we privately inherited from `deque<int>`, our class contains an implementation of all of the `deque`'s member functions, and it is as if we have our own private copy of a `deque` that we can work with.

Public and private inheritance are designed for entirely different purposes. We use public inheritance to design a collection of classes logically related to each other by some common behaviors. Private inheritance, on the other hand, is an implementation technique used to define one class's behaviors in terms of another's. One way to remember the difference between public and private inheritance is to recognize that they play entirely different roles in class design. Public inheritance is used during the design of a class *interface* (determining what behaviors the class should provide), while private inheritance is used during design of a class *implementation* (how those behaviors should be performed). This parallels the difference between a function prototype and a function definition – public inheritance defines a set of prototypes, while private inheritance provides implementations.

Private inheritance is not frequently encountered in practice because the *is-implemented-in-terms-of* relationship can be modeled more clearly through composition. If we wanted to implement a `stack` in terms of a `deque`, instead of using private inheritance, we could just have the `stack` contain a `deque` as a data member, as shown here:

```
class stack
{
public:
    void push(int val)
    {
        implementation.push_front(val);
    }
    int pop()
    {
        const int result = implementation.front();
        implementation.pop_front();
        return result;
    }
    /* ... etc. ... */
private:
    deque<int> implementation;
};
```

In practice it is recommended that you shy away from private inheritance in favor of this more explicit form of composition. However, there are several cases where private inheritance is precisely the tool for the job. Let's return to our discussion of the `Uncopyable` class. Recall that to make a class uncopyable, we had it publically inherit from a class `Uncopyable` that has its copy functions marked private. This led to problems where we could convert an object that inherited from `Uncopyable` into an `Uncopyable`. However, we can remedy this by having the derived class inherit *privately* from `Uncopyable`. That way, it is not considered a subtype of `Uncopyable` and instances of `MyClass` cannot be converted into instances of `Uncopyable`. For example:

```
class MyClass: private Uncopyable
{
    /* ... */
};
```

Now, `MyClass` cannot be copied, nor can it be treated as though it were an object of type `Uncopyable`. This is precisely the idea we want to express.

In C++, all inheritance is considered private inheritance unless explicitly mentioned otherwise; this is why you must write `public Base` to publicly inherit from `Base`. Thus we can rewrite the above class definition omitting the `private` keyword, as shown here:

```
class MyClass: Uncopyable
{
   /* ... */
};
```

This method of disallowing copying is particularly elegant – syntactically, we communicate that `MyClass` cannot be copied at the same time that we actually make it uncopyable through private inheritance.

Before concluding this section, let's make a quick change to the `Uncopyable` class by marking its constructor and destructor `protected`. This means that classes that inherit from `Uncopyable` can still access the constructor and destructor, but otherwise these functions are off-limits. This prevents us from accidentally instantiating `Uncopyable` directly and only lets us use it as a base class. The code for this is shown here:

```
class Uncopyable
{
protected:
    Uncopyable() {}
    ~Uncopyable() {}
private:
    Uncopyable(const Uncopyable&);
    Uncopyable& operator= (const Uncopyable&);
};
```

Classes like `Uncopyable` are sometimes referred to as *mixin classes* since they are designed to be "mixed" into other classes to provide a particular functionality.

Slicing

In our discussion of copy constructor and assignment operators for derived classes, we encountered problems when we copied over the the data of the `Derived` class but not the `Base` class. However, there's a far more serious problem we can run into called *slicing* where we copy only the base class of an object while leaving its derived classes unmodified.

Suppose we have two `Base *` pointers called `one` and `two` that point to objects either of type `Base` or of type `Derived`. What happens if we write code like `*one = *two`? Here, we're copying the value of the object pointed at by `two` into the variable pointed at by `one`. While at first glance this might seem harmless, the above statement is one of the most potentially dangerous mistakes you can make when working with C++ inheritance. The problem is that this line expands into a call to

```
one->operator =(*two);
```

Note that the version of `operator =` we're calling here is the one defined in `Base`, not `Derived`, so this line will only copy over the `Base` portion of `two` into the `Base` portion of `one`, resulting in half-formed objects that are almost certainly not in the correct state and may be entirely corrupted.

Slicing can be even more insidious in scenarios like this one:

```
void DoSomething(Base baseObject)
{
    // Do something
}

Derived myDerived
DoSomething(myDerived);
```

Recall that the parameter `baseObject` will be initialized using the `Base` copy constructor, not the `Derived` copy constructor, so the object in `DoSomething` will not be a correct copy `myDerived`. Instead, it will only hold a copy of the `Base` part of the `myDerived` object.

You should almost never assign a base class object the value of a derived class. The second you do, you will open the door to runtime errors as your code tries to use incompletely-formed objects. While it may sound simple to follow this rule, at many times it might not be clear that you're slicing an object. For example, consider this code snippet:

```
vector<Base> myBaseVector;
Base* myBasePtr = SomeFunction();
myBaseVector.push_back(*myBasePtr);
```

Here, the object pointed at by `myBasePtr` could be of type `Base` or any type inheriting from `Base`. When we call `myBaseVector.push_back(*myBasePtr)`, there is a good chance that we will slice the object pointed at by `myBasePtr`, storing only its `Base` component in the `vector` and dropping the rest. You'll need to be extra vigilant when working with derived classes, since it's easy to generate dangerous, difficult-to-track bugs.

The C++ Casting Operators

One of the most useful features of C++ inheritance is the ability to use an object of one type in place of another. For example, a pointer of type `Derived *` can be used whenever a `Base *` would be expected, and the conversion is automatic. However, in many circumstances, we may want to perform this conversion in reverse. Suppose that we have a pointer that's statically typed as a `Base *`, but we know that the object it points to is actually of typed `Derived *`. How can we use the `Derived` features of the pointee? Because the pointer to the object is a `Base *`, not a `Derived *`, we will have to use a typecast to convert the pointer from the base type to the derived type. Using the typecasts most familiar to us in C++, the code to perform this conversion looks as follows:

```
Base* myBasePtr; // Assume we know that this points to a Derived object.
Derived* myDerivedPtr = (Derived *)myBasePtr;
```

There is nothing wrong with the above code as written, but it is risky because of the typecast `(Derived *)myBasePtr`. In C++, using a typecast of the form `(Type)` is extremely dangerous because there are only minimal compile-time checks to ensure that the typecast makes any sense. For example, consider the following C++ code:

```
Base* myBasePtr; // Assume we know that this points to a Derived object.
vector<double>* myVectorPtr = (vector<double> *)myBasePtr; // Uh oh!
```

This code is completely nonsensical, since there is no reasonable way that a pointer of type `Base *` can end up pointing to an object of type `vector<double>`. However, because of the explicit pointer-to-pointer typecast, this code is entirely legal. In the above case, it's clear that the conversion we're performing is incorrect, but in others it might be more subtle. Consider the following code:

```
const Base* myBasePtr; // Assume we know that this points to a Derived object.
Derived* myDerivedPtr = (Derived *)myBasePtr;
```

This code again is totally legal and at first glance might seem correct, but unfortunately it contains a serious error. In this example, our initial pointer was a pointer to a `const Base` object, but in the second line we removed that `const`ness with a typecast and the resulting pointer is free to modify the object it points at. We've just subverted `const`, which could lead to a whole host of problems down the line.

The problem with the above style of C++ typecast is that it's just too powerful. If C++ can figure out a way to convert the source object to an object of the target type, it will, even if it's clear from the code that the conversion is an error. To resolve this issue, C++ has four special operators called *casting operators* that you can use to perform safer typecasts. When working with inheritance, two of these casting operators are particularly useful, the first of which is `static_cast`. The `static_cast` operator performs a typecast in the same way that the more familiar C++ typecast does, except that it checks at compile time that the cast "makes sense." More specifically, `static_cast` can be used to perform the following conversions:*

* There are several other conversions that you can perform with `static_cast`, especially when working with `void *` pointers, but we will not discuss them here.

1. Converting between primitive types (e.g. `int` to `float` or `char` to `double`).
2. Converting between pointers or references of a derived type to pointers or references of a base type (e.g. `Derived *` to `Base *`) where the target is at least as `const` as the source.
3. Converting between pointers or references of a base type to pointers or references of a derived type (e.g. `Base *` to `Derived *`) where the target is at least as `const` as the source.

If you'll notice, neither of the errors we made in the previous code snippets are possible with a `static_cast`. We can't convert a `Base *` to a `vector<double> *`, since `Base` and `vector<double>` are not related to each other via inheritance. Similarly, we cannot convert from a `const Base *` to a `Derived *`, since `Derived *` is less `const` than `const Base *`.

The syntax for the `static_cast` operator looks resembles that of templates and is illustrated below:

```
Base* myBasePtr; // Assume we know this points to a Derived object.
Derived* myDerivedPtr = static_cast<MyDerived *>(myBasePtr);
```

That is, `static_cast`, followed by the type to convert the pointer to, and finally the expression to convert enclosed in parentheses.

Throughout this discussion of typecasts, when converting between pointers of type `Base *` and `Derived *`, we have implicitly assumed that the `Base *` pointer we wanted to convert was pointing to an object of type `Derived`. If this isn't the case, however, the typecast can succeed but lead to a `Derived *` pointer pointing to a `Base` object, which can cause all sorts of problems at runtime when we try to invoke nonexistent member functions or access data members of the `Derived` class that aren't present in `Base`. The problem is that the `static_cast` operator doesn't check to see that the typecast it's performing makes any sense at runtime. To provide this functionality, you can use another of the C++ casting operators, *`dynamic_cast`*, which acts like `static_cast` but which performs additional checks before performing the cast. `dynamic_cast`, like `static_cast`, can be used to convert between pointer types related by inheritance (but not to convert between primitive types). However, if the typecast requested of `dynamic_cast` is invalid at runtime (e.g. attempting to convert a `Base` object to a `Derived` object), `dynamic_cast` will return a `NULL` pointer instead of a pointer to the derived type. For example, consider the following code:

```
Base* myBasePtr = new Base;
Derived* myDerivedPtr1 = (Derived *)myBasePtr;
Derived* myDerivedPtr2 = static_cast<Derived *>(myBasePtr);
Derived* myDerivedPtr3 = dynamic_cast<Derived *>(myBasePtr);
```

In this example, we use three different typecasts to convert a pointer that points to an object of type `Base` to a pointer to a `Derived`. In the above example, the first two casts will perform the type conversion, resulting in pointers of type `Derived *` that actually point to a `Base` object, which can be dangerous. However, the final typecast, which uses `dynamic_cast`, will return a `NULL` pointer because the cast cannot succeed.

When performing downcasts (casts from a base type to a derived type), unless you are absolutely sure that the cast will succeed, you should consider using `dynamic_cast` over a `static_cast` or raw C++ typecast. Because of the extra check at runtime, `dynamic_cast` is slower than the other two casts, but the extra safety is well worth the cost.

There are two more interesting points to take note of when working with `dynamic_cast`. First, you can only use `dynamic_cast` to convert between types if the base class type contains at least one virtual member function. This may seem like a strange requirement, but greatly improves the efficiency of the

language as a whole and makes sense when you consider that it's rare to hold a pointer to a `Derived` in a pointer of type `Base` when `Base` isn't polymorphic. The other important note is that if you use `dynamic_cast` to convert between *references* rather than pointers, `dynamic_cast` will throw an object of type `bad_cast` rather than returning a "`NULL` reference" if the cast fails. Consult a reference for more information on `bad_cast`.

Implicit Interfaces and Explicit Interfaces

In the chapter on templates we discussed the concept of an *implicit interface*, behaviors required of a template argument. For example, consider a function that returns the average of the values in an STL container of `double`s, as shown here:

```
template <typename ContainerType> double GetAverage(const ContainerType& c)
{
    return accumulate(c.begin(), c.end(), 0.0) / c.size();
}
```

The function `GetAverage` may be parameterized over an arbitrary type, but will only compile if the type `ContainerType` exports functions `begin` and `end` that return iterators over `double`s (or objects implicitly convertible to `double`s) and a `size` function that returns some integer type.

Contrast this with a similar function that uses inheritance:

```
class Container
{
public:
    typedef something-dereferencable-to-a-double const_iterator;
    virtual const_iterator begin() const = 0;
    virtual const_iterator end() const = 0;
};

double GetAverage(const Container& c)
{
    return accumulate(c.begin(), c.end(), 0.0) / c.size();
}
```

This function is no longer a template an instead accepts as an argument an object that derives from `Container`.

In many aspects these functions are similar. Both of the implementations have a set of constraints enforced on their parameter, which can be of any type satisfying these constraints. But these similarities obscure a fundamental difference between how the two functions work – at what point the function calls are resolved. In the inheritance-based version of `GetAverage`, the calls to `begin`, `end`, and `size` are virtual function calls which are resolved at *runtime* using the system described earlier in this chapter. With the template-based version of `GetAverage`, the version of the `begin`, `end`, and `size` functions to call are resolved at *compile-time*.

When you call a template function, C++ instantiates the template by replacing all instances of the template argument with a concrete type. Thus if we call the template function `GetAverage` on a `vector<int>`, the compiler will instantiate `GetAverage` on `vector<int>` to yield the following code:

```
double GetAverage<vector<int> >(const vector<int>& c)
{
    return accumulate(c.begin(), c.end(), 0.0) / c.size();
}
```

Now that the template has been instantiated, it's clear what functions `c.begin()` and the like correspond to – they're the `vector<int>`'s versions of those functions. Since those functions aren't virtual, the compiler can hardcode which function is being called and can optimize appropriately.

The template version of this function is desirable from a performance perspective but is not always the best option. In particular, while we can pass as a parameter to `GetAverage` any object that conforms to the implicit interface, we cannot treat those classes polymorphically. For example, in the above case it's perfectly legal to call `GetAverage` on a `vector<double>` or a `set<double>`, but we cannot write code like this:

```
Container* ptr = RandomChance(0.5) ? new vector<double> : new set<double>;
```

Templates and inheritance are designed to solve fundamentally different problems. If you want to write code that operates over any type that meets some minimum requirements, the template system can help you do so efficiently and concisely provided that you know what types are being used at compile-time. If the types cannot be determined at compile-time, you can use inheritance to describe what behaviors are expected of function arguments.

More to Explore

1. **Multiple Inheritance**: Unlike other object-oriented languages like Java, C++ lets classes inherit from multiple base classes. You can use this build classes that act like objects of multiple types, or in conjunction with mixin classes to build highly-customizable classes. In most cases multiple inheritance is straightforward and simple, but there are many situations where it acts counterintuitively. If you plan on pursuing C++ more seriously, be sure to read up on multiple inheritance.
2. **`const_cast` and `reinterpret_cast`**: C++ has two other conversion operators, `const_cast`, which can add or remove `const` from a pointer, and `reinterpret_cast`, which performs fundamentally unsafe typecasts (such as converting an `int *` to a `string *`). While the use \cases of these operators are far beyond the scope of this class, they do arise in practice and you should be aware of their existences. Consult a reference for more information.
3. **The Curiously Recurring Template Pattern** (CRTP): Virtual functions make your programs run slightly slower than programs with non-virtual functions because of the extra overhead of the dynamic lookup. In certain situations where you want the benefits of inheritance without the cost of virtual functions, you can use an advanced C++ trick called the *curiously recurring template pattern*, or CRTP. The CRTP is also known as "static interfacing" and lets you get some of the benefits of inheritance without any runtime cost.
4. **Policy Classes**: A nascent but popular C++ technique called *policy classes* combines multiple inheritance and templates to design classes that are simultaneously customizable and efficient. A full treatment of policy classes is far beyond the scope of this reader, but if you are interested in seeing exactly how customizable C++ can be I strongly encourage you to read more about them.

Practice Problems

1. In the `GenericBatchPrinter` example from earlier in this chapter, recall that the `Document` base class had several methods defined purely virtually, meaning that they don't actually have any code for those member functions. Inside `GenericBatchPrinter`, why don't we need to worry that the `Document *` pointer from the `queue` points to an object of type `Document` and thus might cause problems if we tried invoking those purely virtual functions? ♦
2. In the next exercises, we'll explore a set of classes that let you build and modify functions at runtime using tools similar to those in the STL `<functional>` programming library.

Consider the following abstract class:

```
class Function
{
public:
    virtual ~Function() = 0;
    virtual double evaluateAt(double value) = 0;
};
```

This class exports a single function, `evaluateAt`, that accepts a `double` as a parameter and returns the value of some function evaluated at that point. Write a derived class of `Function`, `SimpleFunction`, whose constructor accepts a regular C++ function that accepts and returns a `double` and whose `evaluateAt` function returns the value of the stored function evaluated at the parameter.

3. The composition of two functions **F** and **G** is defined as **F**(**G**(**x**)) – that is, the function **F** applied to the value of **G** applied to **x**. Write a class `CompositionFunction` whose constructor accepts two `Function *` pointers and whose `evaluateAt` returns the composition of the two functions evaluated at a point.
4. The derivative of a function is the slope of the tangent line to that function at a point. The derivative of a function **F** can be approximated as $\mathbf{F'(x)} \approx \frac{\mathbf{F(x + \Delta x)} - \mathbf{F(x - \Delta x)}}{\mathbf{2\Delta x}}$ for small values of **Δx**. Write a class `DerivativeFunction` whose constructor accepts a `Function *` pointer and a `double` representing **Δx** and whose `evaluateAt` approximates the derivative of the stored function using the initial value of **Δx**. ♦
5. For the above classes, why did we make a function named `evaluateAt` instead of providing an implementation of `operator()`? *(Hint: Will we be using actual `Function` objects, or pointers to them?)*
6. A common mistake is to try to avoid problems with slicing by declaring `operator =` as a virtual function in a base class. Why won't this solve the slicing problem? *(Hint: what is the parameter type to `operator =`?)*
7. Suppose you have three classes, `Base`, `Derived`, and `VeryDerived` where `Derived` inherits from `Base` and `VeryDerived` inherits from `Derived`. Assume all three have copy constructors and assignment operators. Inside the body of `VeryDerived`'s assignment operator, why shouldn't it invoke `Base::operator =` on the other object? What does this tell you about long inheritance chains, copying, and assignment?
8. The C++ casting operators were deliberately designed to take up space to discourage programmers from using typecasts. Explain why the language designers frown upon the use of typecasts.
9. The `unary_function` and `binary_function` classes from `<functional>` do not define `operator()` as a `virtual` member function. Considering that every adaptable function must be a subclass of one of these two classes, it seems logical for the two classes to do so. Why don't they? (Hint: the STL is designed for maximum possible efficiency. What would happen if `operator()` was `virtual`?)

Appendices

Appendix 0: Moving from C to C++

C++ owes a great debt to the C programming language. Had it not been rooted in C syntax, C++ would have attracted fewer earlier adopters and almost certainly would have vanished into the mists of history. Had it not kept C's emphasis on runtime efficiency, C++ would have lost relevance over time and would have gone extinct. But despite C++'s history in C, C and C++ are very different languages with their own idioms and patterns. It is a common mistake to think that knowledge of C entails a knowledge of C++ or vice-versa, and experience with one language often leads to suboptimal coding skills in the other. In particular, programmers with a background in pure C often use C constructs in C++ code where there is a safer or more elegant alternative. This is not to say that C programmers are somehow worse coders than C++ programmers, but rather that some patterns engrained into the C mentality are often incompatible with the language design goals of C++.

This appendix lists ten idiomatic C patterns that are either deprecated or unsafe in C++ and suggests replacements. There is no new C++ content here that isn't already covered in the main body of the course reader, but I highly recommend reading through it anyway if you have significant background in C. This by no means an exhaustive list of differences between C and C++, but should nonetheless help you transition between the languages.

Tip 0: Prefer streams to `stdio.h`

C++ contains the C runtime library in its standard library, so all of the I/O functions you've seen in `<stdio.h>` (`printf`, `scanf`, `fopen`) are available in C++ through the `<cstdio>` header file. While you're free to use `printf` and `scanf` for input and output in C++, I strongly advise you against doing so because the functions are inherently unsafe. For example, consider the following C code:

```
char myString[1024] = {'\0'};
int  myInt;

printf("Enter an integer and a string: ");
scanf("%d %1023s", &myInt, myString);
printf("You entered %d and %s\n", myInt, myString);
```

Here, we prompt the user for an integer and a string, then print them back out if the user entered them correctly. As written there is nothing wrong with this code. However, consider the portions of the code I've highlighted here:

```
char myString[1024] = {'\0'};
int  myInt;

printf("Enter an integer and a string: ");
scanf("%d %1023s", &myInt, myString);
printf("You entered %d and %s\n", myInt, myString);
```

Consider the size of the buffer, 1024. When reading input from the user, if we don't explicitly specify that we want to read at most 1023 characters of input, we risk a buffer overrun that can trash the stack and allow an attacker to fully compromise the system. What's worse, if there is a mismatch between the declared size of the buffer (1024) and the number of characters specified for reading (1023), the compiler will not provide any warnings. In fact, the only way we would discover the problem is if we were very careful to read over the code checking for this sort of mistake, or to run an advanced tool to double-check the code for consistency.

Similarly, consider the highlighted bits here:

```
char myString[1024] = {'\0'};
int  myInt;

printf("Enter an integer and a string: ");
scanf("%d %1023s", &myInt, myString);
printf("You entered %d and %s\n", myInt, myString);
```

Notice that when reading values from the user or writing values to the console, we have to explicitly mention what types of variables we are reading and writing. The fact that `myInt` is an `int` and `myString` is a `char*` is insufficient for `printf` and `scanf`; we have to mention to read in an `int` with `%d` and a string with `%s`. If we get these backwards or omit one, the program contains a bug but will compile with no errors.* Another vexing point along these lines is the parameter list in `scanf` – we must pass in a pointer to `myInt`, but can just specify `myString` by itself. Confusing these or getting them backwards will cause a crash or a compiler warning, which is quite a price to pay for use input.

The problem with the C I/O libraries is that they completely bypass the type system. Recall that the signatures of `printf` and `scanf` are

```
int printf(const char* formatting, ...);
int scanf (const char* formatting, ...);
```

The `...` here means that the caller can pass in any number of arguments of any type, and in particular this means that the C/C++ compiler cannot do any type analysis to confirm that you're using the arguments correctly. Don't get the impression that C or C++ are type-safe – they're not – but the static type systems they have are designed to prevent runtime errors from occuring and subverting this system opens the door for particularly nasty errors.

In pure C, code like the above is the norm. In C++, however, we can write the following code instead:

```
int myInt;
string myString;

cout << "Enter an int and a string: ";
cin >> myInt >> myString;
cout << "You entered " << myInt << " and " << myString << endl;
```

If you'll notice, the only time that the types of `myInt` and `myString` are mentioned is at the point of declaration. When reading and writing `myInt` and `myString`, the C++ can automatically infer which version of `operator >>` and `operator <<` to call to perform I/O and thus there is no chance that we can accidentally read a `string` value into an `int` or vice-versa. Moreover, since we're using a C++-style string, there is no chance that we'll encounter a buffer overflow. In short, the C++ streams library is just plain safer than the routines in `<stdio.h>`.

When working in pure C++, be wary of the `<stdio.h>` functions. You are missing out on the chance to use the streams library and are exposing yourself and your code to all sorts of potential security vulnerabilities.

Tip 1: Use C++ `string`s instead of C-style strings

Life is short, nasty, and brutish, and with C strings it will be even worse. C strings are notoriously tricky to get right, have a cryptic API, and are the cause of all sorts of security bugs. C++ `string`s, on the other hand, are elegant, pretty, and difficult to use incorrectly. If you try truncating a C++ `string` at an invalid index with `erase`, the `string` will throw an exception rather than clobbering memory. If you append data

* Many compilers will report errors if you make this sort of mistake, but they are not required to do so.

to a C++ `string`, you don't need to worry about reallocating any memory – the object does that for you. In short, C strings are tricky to get *right*, and C++ `string`s are tricky to get *wrong*. "But wait!," you might exclaim, "Because C strings are so low-level, I can sometimes outperform the heavyweight C++ `string`." This is absolutely true – because C strings are so exposed, you have a great deal of flexibility and control over how the memory is managed and what operations go on behind the scenes. But is it really worth it? Here's a small sampling of what can go wrong if you're not careful with C strings:

1. You might write off the end of a buffer, clobbering other data in memory and paving the way for a massive security breach.
2. You might forget to deallocate the memory, causing a memory leak.
3. You might overwrite the terminating null character, leading to a runtime error or incomprehensible program outputs.

Are C strings faster than their C++ counterparts? Of course. But should you nonetheless sacrifice a little speed for the peace of mind that your program isn't going to let hackers take down your system? Absolutely.

Tip 2: Use C++ typecasts instead of C typecasts

Both C and C++ have static type systems – that is, if you try to use a variable of one type where a variable of another type is expected, the compiler will report an error. Both C and C++ let you use typecasts to convert between types when needed, sometimes safely and sometimes unsafely.

C has only one style of typecast, which is conveniently dubbed a "C-style typecast." As mentioned in the chapter on inheritance, C-style typecasts are powerful almost to a fault. Converting between a `double` and an `int` uses the same syntax for unsafe operations like converting pointers to integers, integers to pointers, `const` variables to non-`const` variables, and pointers of one type to pointers of another type. As a result, it is easy to accidentally perform a typecast other than the one you wanted. For example, suppose we want to convert a `char*` pointer to an `int*` pointer, perhaps because we're manually walking over a block of memory. We write the following code:

```
const char* myPtr = /* ... */
int* myIntPtr = (int *)myPtr;
```

Notice that in this typecast we've converted a `const char*` to an `int*`, subverting `const`ness. Is this deliberate? Is this a mistake? Given the above code there's no way to know because the typecast does not communicate what sort of cast is intended. Did we mean to strip off `const`ness, convert from a `char*` to an `int*`, or both?

C++ provides three casting operators (`const_cast`, `static_cast`, `reinterpret_cast`) that are designed to clarify the sorts of typecasts performed in your code. Each performs exactly one function and causes a compile-time error if used incorrectly. For example, if in the above code we only meant to convert from a `const char*` to a `const int*` without stripping `const`ness, we could write it like this:

```
const char* myPtr = /* ... */
const int* myIntPtr = reinterpret_cast<const int*>(myPtr);
```

Now, if we leave off the `const` in the typecast, we'll get a compile-time error because `reinterpret_cast` can't strip off `const`ness. If, on the other hand, we want to convert the pointer from a `const char*` to a regular `int*`, we could write it as follows:

```
const char* myPtr = /* ... */
int* myIntPtr = const_cast<int*>(reinterpret_cast<const int*>(myPtr));
```

This is admittedly much longer and bulkier than the original C version, but it is also more explicit about exactly what it's doing. It also is safer, since the compiler can check that the casts are being used correctly.

When writing C++ code that uses typecasts, make sure that you use the C++-style casting operators. Are they lengthy and verbose? Absolutely. But the safety and clarity guarantees they provide will more than make up for it.

Tip 3: Prefer `new` and `delete` to `malloc` and `free`

In C++, you can allocate and deallocate memory either using `new` and `delete` or using `malloc` and `free`. If you're used to C programming, you may be tempted to use `malloc` and `free` as you have in the past. This can lead to very subtle errors because `new` and `delete` do *not* act the same as `malloc` and `free`. For example, consider the following code:

```
string* one = new string;
string* two = static_cast<string*>(malloc(sizeof string));
```

Here, we create two `string` objects on the heap – one using `new` and one using `malloc`. Unfortunately, the string allocated with `malloc` is a ticking timebomb waiting to explode. Why is this? The answer has to do with a subtle but critical difference between the two allocation routines.

When you write `new string`, C++ performs two steps. First, it conjures up memory from the heap so that the new `string` object has a place to go. Second, it calls the `string` constructor on the new memory location to initialize the `string` data members. On the other hand, if you write `malloc(sizeof string)`, you only perform the memory allocation. In the above example, this means that the `string` object pointed at by `two` has the right size for a `string` object, but isn't actually a `string` because none of its data members have been set appropriately. If you then try using the `string` pointed at by `two`, you'll get a nasty crash since the object is in a garbage state. To avoid problems like this, make sure that you always allocate objects using `new` rather than `malloc`.

If you do end up using both `new` and `malloc` in a C++ program (perhaps because you're working with legacy code), make sure that you are careful to deallocate memory with the appropriate deallocator function. That is, don't `free` an object allocated with `new`, and don't delete an object allocated with `malloc`. `malloc` and `new` are not the same thing, and memory allocated with one is not necessarily safe to clean up with the other. In fact, doing so leads to undefined behavior, which can really ruin your day.

Tip 4: Avoid `void*` Pointers

Code in pure C abounds with `void*` pointers, particularly in situations where a function needs to work with data of any type. For example, the C library function `qsort` is prototyped as

```
void qsort(void* elems, size_t numElems, size_t elemSize,
           int (*cmpFn)(const void*, const void*));
```

That's quite a mouthful and uses `void*` three times – once for the input array and twice in the comparison function. The reason for the `void*` here is that C lacks language-level support for generic programming and consequently algorithms that need to operate on arbitrary data have to cater to the lowest common denominator – raw bits and bytes.

When using C's `qsort`, you have to be extremely careful to pass in all of the arguments correctly. When sorting an array of `int`s, you must take care to specify that `elemSize` is `sizeof(int)` and that your comparison function knows to interpret its arguments as pointers to `int`s. Passing in a comparison function which tries to treat its arguments as being of some other type (perhaps `char**`s or `double*`s) will cause runtime errors, and specifying the size of the elements in the array incorrectly will probably cause incorrect behavior or a bus error.

Contrast this with C++'s `sort` algorithm:

```
template <typename RandomAccessIterator, typename Comparator>
void sort(RandomAccessIterator begin, RandomAccessIterator end, Comparator c);
```

With C++'s `sort`, the compiler can determine what types of elements are stored in the range [`begin`, `end`) by looking at the type of the iterator passed as a parameter. The compiler can thus automatically figure out the size of the elements in the range. Moreover, if there is a type mismatch between what values the `Comparator` parameter accepts and what values are actually stored in the range, you'll get a compile-time error directing you to the particular template instantiation instead of a difficult-to-diagnose runtime error.

This example highlights the key weakness of `void*` – it completely subverts the C/C++ type system. When using a `void*` pointer, you are telling the compiler to forget all type information about what's being pointed at and therefore have to explicitly keep track of all of the relevant type information yourself. If you make a mistake, the compiler won't recognize your error and you'll have to diagnose the problem at runtime. Contrast this with C++'s template system. C++ templates are strongly-typed and the compiler will ensure that everything type-checks. If the program has a type error, it won't compile, and you can diagnose and fix the problem without having to run the program.

Whenever you're thinking about using a `void*` in C++ programming, make sure that it's really what you want to do. There's almost always a way to replace the `void*` with a template. Then again, if you want to directly manipulate raw bits and bytes, `void*` is still your best option.

One point worth noting: In pure C, you can implicitly convert between a `void*` pointer and a pointer of any type. In C++, you can implicitly convert any pointer into a `void*`, but you'll have to use an explicit typecast to convert the other way. For example, the C code

```
int* myArray = malloc(numElems * sizeof(int));
```

Does not compile in C++ since `malloc` returns a `void*`. Instead, you'll need to write

```
int* myArray = (int *)malloc(numElems * sizeof(int));
```

Or, even better, as

```
int* myArray = static_cast<int *>(malloc(numElems * sizeof(int)));
```

Using the C++ `static_cast` operator.

Tip 5: Prefer `vector` to raw arrays

Arrays live in a sort of nether universe. They aren't quite variables, since you can't assign them to one another, and they're not quite pointers, since you can't reassign where they're pointing. Arrays can't remember how big they are, but when given a static array you can use `sizeof` to get the total space it occupies. Functions that operate on arrays have to either guess the correct size or rely on the caller to supply it. In short, arrays are a bit of a mess in C and C++.

Contrast this with the C++ `vector`. vectors know exactly how large they are, and can tell you if you ask. They are first-class variables that you can assign to one another, and aren't implicitly convertible to pointers. On top of that, they clean up their own messes, so you don't need to worry about doing your own memory management. In short, the `vector` is everything that the array isn't.

In addition to being safer than regular arrays, vectors can also be much cleaner and easier to read. For example, consider the following C code:

```
void MyFunction(int size)
{
    int* arr = malloc(size * sizeof(int));
    memset(arr, 0, size * sizeof(int));

    /* ... */

    free(arr);
};
```

Compare this to the equivalent C++ code:

```
void MyFunction(int size)
{
    vector<int> vec(size);

    /* ... */
};
```

No ugly computations about the size of each element, no messy cleanup code at the end, and no `memset`s. Just a single line and you get the same effect. Moreover, since the `vector` always cleans up its memory after it goes out of scope, the compiler will ensure that no memory is leaked. Isn't that nicer?

Of course, there are times that you might want to use a fixed-size array, such as if you know the size in advance and can fit the array into a struct. But in general, when given the choice between using arrays and using vectors, the `vector` is the more natural choice in C++.

Tip 6: Avoid `goto`

> *For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code)*
>
> – Edsger Dijkstra [Dij68]

The `goto` keyword been widely criticised since Dijkstra published "Go To Statement Considered Harmful" in 1968, yet still managed to make its way into C and consequently C++. Despite its apparent simplicity, `goto` can cause all sorts of programming nightmares because it is inherently *unstructured*. `goto` can jump pretty much anywhere and consequently can lead to unintuitive or even counterintuitive code. For example, here's some code using `goto`:

```
int x = 0;
start:
    if (x == 10) goto out;
    printf("%d\n", x);
    ++x;
    goto start;
out:
    printf("Done!\n");
```

This is completely equivalent to the *much* more readable

```
for(int x = 0; x < 10; ++x)
```

```
    printf("%d\n", x);
```

Despite `goto`'s bad reputation, modern C programming still has several places in which `goto` can still be useful. First, `goto` can be used as a sort of "super `break`" to break out of multiple levels of loop nesting. This use is still legitimate in C++, but is frowned upon stylistically. Second, `goto` can be used as a way of performing necessary cleanup in an error condition. For example:

```
/* Returns a string of the first numChars characters from a file or NULL in an
 * error case.
 */
char* ReadFromFile(const char* filename, size_t numChars)
{
    FILE* f;
    char* buffer;

    /* Allocate some space. */
    buffer = malloc(numChars + 1);
    if(buffer == NULL) return NULL;

    /* Open the file, abort on error. */
    f = fopen(filename, "rb");
    if(f == NULL)
        goto error;

    /* Read the first numChars characters, failing if we don't read enough. */
    if(fread(buffer, numChars, 1, f) != numChars)
        goto error;

    /* Close the file, null-terminate the string, and return. */
    fclose(f);
    buffer[numChars] = '\0';
    return buffer;

    /* On error, clean up the resources we opened. */
error:
    free(buffer);
    if(f != NULL)
        fclose(f);

    return NULL;
}
```

Here, there are several error conditions in which we need to clean up the temporary buffer and potentially close an open file. When this happens, rather than duplicating the cleanup code, we use `goto` to jump to the error-handling subroutine.

In pure C this is perfectly fine, but in C++ would be considered a gross error because there are much better alternatives. As mentioned in the chapter on exception handling, we could instead use a catch-and-rethrow
strategy to get the exact same effect without `goto`, as shown here:

```
/* Returns a string of the first numChars characters from a file.
 * Throws a runtime_error on error.
 */
char* ReadFromFile(const char* filename, size_t numChars)
{
    FILE* f;
    char* buffer = NULL;

    try
    {
        /* Allocate some space.  This will throw on error rather than returning
         * NULL.
         */
        buffer = new char[numChars + 1];

        /* Open the file, abort on error. */
        f = fopen(filename, "rb");
        if(f == NULL)
            throw runtime_error("Can't open file!");

        /* Read the first numChars characters, failing if we don't read enough. */
        if(fread(buffer, numChars, 1, f) != numChars)
            throw runtime_error("Can't read enough characters!");

        /* Close the file, null-terminate the string, and return. */
        fclose(f);
        buffer[numChars] = '\0';
        return buffer;
    }
    catch(...)
    {
        /* On error, clean up the resources we opened. */
        delete [] buffer;
        if(f != NULL)
            fclose(f);

        throw;
    }
}
```

Now that we're using exception-handling instead of `goto`, the code is easier to read and allows the caller to get additional error information out of the function.

An even better alternative here would be to use an `ifstream` and a `string` to accomplish the same result. Since the `ifstream` and `string` classes have their own destructors, we don't need to explicitly clean up any memory. This is shown here:

```
/* Returns a string of the first numChars characters from a file.
 * Throws a runtime_error on error.
 */
string ReadFromFile(const char* filename, size_t numChars)
{
    string buffer(numChars);

    /* Open the file, abort on error. */
    ifstream input(filename);
    if(input.fail())
        throw runtime_error("Can't open the file!");

    /* Read the first numChars characters, failing if we don't read enough. */
    input.read(&buffer[0], numChars);
    if(input.fail())
        throw runtime_error("Couldn't read enough data");

    return buffer;
}
```

This version is very clean and concise and doesn't require any `goto`-like structure at all. Since the object destructors take care of all of the cleanup, we don't need to worry about doing that ourselves.

My advice against `goto` also applies to `setjmp` and `longjmp`. These functions are best replaced with C++'s exception-handling system, which is far safer and easier to use.

Tip 7: Use C++'s `bool` type when applicable

Prior to C99, the C programming language lacked a standard `bool` type and it was common to find idioms such as

```
enum bool {true, false};
```

Or

```
#define bool int
#define true 1
#define false 0
```

Similarly, to loop indefinitely, it was common to write

```
while(1)
{
   /* ... */
}
```

Or

```
for(;;)
{
   /* ... */
}
```

Defining your own custom `bool` type is risky in C++ because a custom type will not interact with language features like templates and overloading correctly. Similarly, while both of the "loop forever" loop constructs listed above are legal C++ code, they are both less readable than the simpler

```
while(true)
{
   /* ... */
}
```

If you aren't already used to working with `bool`s, I suggest that you begin doing so when working in C++. Sure, you can emulate the functionality of a `bool` using an `int`, but doing so obscures your intentions and leads to all sorts of other messes. For example, if you try to emulate `bool`s using `int`s, you can get into nasty scrapes where two `int`s each representing `true` don't compare equal because they hold different nonzero values. This isn't possible with the `bool` type. To avoid subtle sources of error and to make your code more readable, try to use `bool` whenever applicable.

Tip 8: Avoid "`typedef struct`"

In pure C, if you define a struct like this:

```
struct pointT
{
    int x, y;
};
```

Then to create an instance of the struct you would declare a variable as follows:

```
struct pointT myPoint;
```

In C++, this use of `struct` is unnecessary. It is also bad practice, since veteran C++ programmers will almost certainly have to pause and think about exactly what the code means. Of course, most C programmers are also not particularly fond of this syntax, and to avoid having to spell out `struct` each time would write

```
typedef struct pointT_
{
    int x, y;
} pointT;
```

This syntax is still valid C++, but is entirely unnecessary and makes the code significantly trickier to read. Moreover, if you want to add constructors or destructors to the `struct` you would have to use the name `pointT_` even though externally the object would be called `pointT` without the underscore. This makes the code more difficult to read and may confuse class clients. In the interests of clarity, avoid this use of `typedef` when writing C++ code.

Tip 9: Avoid `memcpy` and `memset`

In pure C, code like the following is perfectly normal:

```
struct pointT
{
    int x, y;
};
struct pointT myPoint;
memset(&myPoint, 0, sizeof(pointT));
```

Here, the call to `memset` is used to initialize all the variables in the `pointT` to zero. Since C lacks constructors and destructors, this code is a reasonably good way to ensure that the `pointT` is initialized before use.

Because C++ absorbed C's standard library, the functions `memset`, `memcpy`, and the like are all available in C++. However, using these functions can lead to subtle but dangerous errors that can cause all sorts of runtime woes. For example, consider the following code:

```
string one = "This is a string!";
string two = "I like this string more.";
memcpy(&one, &two, sizeof(string)); // Set one equal to two - does this work?
```

Here, we use `memcpy` to set `one` equal to `two`. Initially, it might seem like this code works, but unfortunately this `memcpy` results in undefined behavior and will almost certainly cause a runtime crash. The reason is that the `string` object contains pointers to dynamically-allocated memory, and when `memcpy`ing the data from `two` into `one` we've made both of the string objects point to the same memory. After each pointer goes out of scope, both will try to reclaim the memory, causing problems when the underlying string buffer is doubly-`delete`d. Moreover, if this *doesn't* immediately crash the program, we've also leaked the memory `one` was originally using since all of its data members were overwritten without first being cleaned up.

If we wanted to set `one` equal to `two`, we could have just written this instead:

```
string one = "This is a string!";
string two = "I like this string more.";
two = one;
```

This uses the `string`'s assignment operator, which is designed to safely perform a deep copy.

In general, mixing `memcpy` with C++ classes is just asking for trouble. Most classes maintain some complex invariants about their data members and what memory they reference, and if you `memcpy` a block of data over those data members you might destroy those invariants. `memcpy` doesn't respect `public` and `private`, and thus completely subverts the encapsulation safeguards C++ tries to enforce.

But the problem runs deeper than this. Suppose that we have a polymorphic class representing a binary tree node:

```
class BinaryTreeNode
{
public:
    BinaryTreeNode();
    virtual ~BinaryTreeNode(); // Polymorphic classes need virtual destructors

    /* ... etc. ... */
private:
    BinaryTreeNode* left, *right;
};
```

We want to implement the constructor so that it sets `left` and `right` to `NULL`, indicating that the node has no children. Initially, you might think that the following code is safe:

```
BinaryTreeNode::BinaryTreeNode()
{
    /* Zero out this object.  Is this safe? */
    memset(this, 0, sizeof(BinaryTreeNode));
}
```

Since the null pointer has value zero, it seems like this should work – after all, if we overwrite the entire object with zeros, we've effectively nulled out all of its pointer data members. But this code is a recipe for

disaster because the class contains more than just a `left` and `right` pointer. In the chapter on inheritance, we outlined how virtual functions are implemented using a virtual function table pointer that sits at the beginning of the class. If we use `memset` to clear out the object, we'll overwrite this virtual function table pointer with `NULL`, meaning that any attempt to call a virtual function on this object will result in a null pointer dereference and a program crash.*

The key problem with `memset` and `memcpy` is that they completely subvert the abstractions C++ provides to increase program safety. Encapsulation is supposed to prevent clients from clobbering critical class components and object layout is done automatically specifically to prevent programmers from having to explicitly manipulate low-level machinery. `memset` and `memcpy` remove these barriers and expose you to dangerous you could otherwise do without.

This is not to say, of course, that `memset` and `memcpy` have no place in C++ code – they certainly do – but their role is considerably less prominent than in pure C. Before you use low-level manipulation routines, make sure that there isn't a better way to accomplish the same goal through more "legitimate" C++ channels.

With that said, welcome to C++! Enjoy your stay!

* This example is based on a conversation I had with Edward Luong, who encountered this very problem when writing a large program in C++.

Appendix 1: Solutions to Practice Problems

Streams

Problem 2. There are two steps necessary to get `HasHexLetters` working. First, we transform the input number into a string representation of its hexadecimal value. Next, using techniques similar to that for `GetInteger`, we check to see if this string can be interpreted as an `int` when read in base 10. If so, the hexadecimal representation of the number must not contain any letters (since letters can't be interpreted as a decimal value), otherwise the representation has at least one letter in it.

One possible implementation is given here:

```
bool HasHexLetters(int value)
{
    /* Funnel the data into a stringstream, using the hex manipulator to represent
     * it in hexadecimal.
     */
    stringstream converter;
    converter << hex << value;

    /* Now, try extracting the string as an int, using the dec manipulator to read
     * it in decimal.
     */
    int dummy;
    converter >> dec >> dummy;

    /* If the stream failed, we couldn't read an int and we're done. */
    if(converter.fail()) return true;

    /* Otherwise, try reading something else from the stream.  If we succeed, it
     * must have been a letter and we know that the integer has letters in its hex
     * representation.
     */
    char leftover;
    converter >> leftover;

    return !converter.fail();
}
```

STL Containers, Part I

Problem 5a. If we want to cycle the elements of a container, then our best options are the `deque` and the `queue`. Both of these choices let us quickly move the front element of the container to the back; the `deque` with `pop_front` and `push_back` and the `queue` with `push` and `pop`.

Cycling the elements of a `stack` is impossible without having some external structure that can store the `stack`'s data, so this is not a good choice. While it's possible to cycle the elements of a `vector` using `push_back` and `erase`, doing so is very inefficient because the `vector` will have to shuffle all of its elements down to fill in the gap at the beginning of the container. Remember, if you ever want to add and remove elements at the beginning or end of a `vector`, the `deque` is a better choice.

Problem 5b. For this solution we'll use an STL `queue`, since we don't need access to any element of the key list except the first. Then one solution is as follows:

```
string VigenereCipher(string toEncode, queue<int> values)
{
    for(int k = 0; k < toEncode.length(); ++k)
    {
        toEncode[k] += values.front(); // Encrypt the current character
        values.push(values.front());   // Add the current key to the back.
        values.pop();                  // Remove the current key from the front.
    }
}
```

STL Iterators

Problem 5. One possible implementation of the function is as follows:

```
vector<string> LoadAllTokens(string filename)
{
    vector<string> result;

    /* Open the file, if we can't, just return the empty vector. */
    ifstream input(filename.c_str());
    if(input.fail()) return result;

    /* Using the istream_iterator iterator adapter, read everything out of the
     * file.  Since by default the streams library uses whitespace as a separator
     * character, this reads in all of the tokens.
     */
    result.insert(result.begin(),
                  istream_iterator<string>(input), istream_iterator<string>());

    return result;
}
```

STL Containers, Part II

Problem 2. We can solve this problem by loading all of the values in the `map` into a `map<string, int>` associating a value in the initial `map` with its frequency. We then can get the number of duplicates by adding up all of the entries in the second `map` whose value is not one (i.e. at least two elements have the same value). This can be implemented as follows:

```
int NumDuplicateValues(map<string, string>& input)
{
    map<string, int> counter;

    /* Iterate over the map updating the frequency count.  Notice that we are
     * checking for the number of duplicate values, so we'll index into the map by
     * looking at itr->second.  Also note that we don't check for the case where
     * the map doesn't already contain this key.  Since STL containers initialize
     * all stored integers to zero, if the key doesn't exist a fresh pair will be
     * created with value zero.
     */
    for(map<string, string>::iterator itr = input.begin();
        itr != input.end(); ++itr)
        ++counter[itr->second];

    int result = 0;
    /* Now iterate over the entries and accumulate those that have at least value
     * two.
     */
    for(map<string, string>::iterator itr = input.begin();
        itr != input.end(); ++itr)
        if(itr->second > 1) result += itr->second;

    return result;
}
```

Problem 5. There are many good solutions to this problem. My personal favorite is this one:

```
void CountLetters(ifstream& input, map<char, int>& freqMap)
{
    char ch;
    while(input.get(ch)) ++freqMap[ch];
}
```

This code is dense and relies on several properties of the stream library and the STL. First, the member function `get` accepts as input a `char` by reference, then reads in a single character from the stream. On success, the function fills the `char` with the read value. On failure, the value is unchanged and the stream goes into a fail state. The `get` function then returns a reference to the stream object that did the reading, meaning that `while(input.get(ch))` is equivalent to

```
while(true)
{
    input.get(ch);
    if(!input) break;

    /* ... body of loop ... */
}
```

And since `!input` is equivalent to `input.fail()`, this one-liner will read in a character from the file, then break out of the loop if the read failed.

Once we've read in the character, we can simply write `++freqMap[ch]`, since if the key already exists we're incrementing the older value and if not a new key/value pair will be created with value 0, which is then incremented up to one.

Problem 6. As mentioned in the hint, the trick is to use the structure of lexicographic comparisons to construct a pair of strings lower- and upper-bounding all strings with the given prefix. For example, suppose that we want to find all words beginning with the prefix `anti`. Now, any word beginning with `anti` must compare lexicographically greater than or equal to `anti`, since the first four characters will match but the word beginning with `anti` must also be longer than `anti`. For example, `antigen` and `antidisestablishmentarianism` both compare greater than `anti` since they have the same prefix as `anti` but are longer.

The next observation is that any word that *doesn't* start with `anti` falls into one of two categories – those that compare lexicographically less than `anti` and those that compare lexicographically greater than `anti`. The first of these sets can be ignored, but how can we filter out words with non-`anti` prefixes that compare lexicographically greater than `anti`? The trick is to note that if the word doesn't have `anti` as a prefix, then somewhere in its first four letters it must disagree with `anti`. If we take the next lexicographically-higher prefix than `anti` (which is formed by incrementing the last letter), we get `antj`. This is the smallest possible prefix any word not starting by `anti` can have. Moreover, every word that starts with `anti` compares lexicographically less than `antj`, and so if we only look at words that compare lexicographically greater than or equal to `anti` and lexicographically less than `antj`, we have all of the words that start with `anti`. Using the `set`'s `lower_bound` function, we can find which words in the set match these criteria efficiently (in O(lg n) time) using the following code:

```
void PrintMatchingPrefixes(set<string>& lexicon, string prefix)
{
    /* We'll assume the prefix is nonempty in this next step. */
    string nextPrefix = prefix;
    ++nextPrefix[nextPrefix.size() - 1];

    /* Compute the range to iterate over.  We store these iterators outside of the
     * loop so that we don't have to recompute them every time.
     */
    set<string>::iterator end = lexicon.lower_bound(nextPrefix);
    for(set<string>::iterator itr = lexicon.lower_bound(prefix); itr != end; ++itr)
        cout << *itr << endl;
}
```

STL Algorithms

Problem 1. Printing out a `vector` is easy thanks to `ostream_iterator` and `copy`. Recall that an `ostream_iterator` is an iterator which prints out the values stored in it to `cout`, and that the `copy` algorithm accepts three inputs – two iterators defining a range to copy and one iterator representing the destination – then copies the input range to the output source. Thus we can print out a `vector` as follows:

```
void PrintVector(vector<int>& v)
{
    copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
}
```

Problem 4. By default, if you compare two strings to one another using <, the result is whether the first string lexicographically precedes the second string. Thus if we have a `vector<string>` called `v`, calling `sort(v.begin(), v.end())` will sort the input lexicographically. In our case, though, we want to "hack" the `sort` function so that it always puts "Me First!" at the front of the sorted ordering. To do this, we'll write a custom callback function that performs a default string comparison if neither string is "Me First!" but which skews the result otherwise. Here's one implementation:

```
const string kWinnerString = "Me First!";
bool BiasedSortHelper(string one, string two)
{
    /* Case one: Neither string is the winner string.  Just do a default
     * comparison.
     */
    if(one != kWinnerString && two != kWinnerString)
        return one < two;

    /* Case two: Both strings are the winner string.  Then return false because the
     * string isn't less than itself.
     */
    if(one == kWinnerString && two == kWinnerString)
        return false;

    /* Case three: one is the winner string, two isn't.  Return true to bias
     * the sort so that the winner string comes first.
     */
    if(one == kWinnerString)
        return true;

    /* Otherwise, two is the winner string and one isn't */
    return false;
}
```

The implementation of `BiasedSort` is then

```
void BiasedSort(vector<string>& v)
{
    sort(v.begin(), v.end(), BiasedSortHelper)
}
```

Problem 6: One implementation is as follows:

```
int count(vector<int>::iterator start, vector<int>::iterator end, int value)
{
    int result = 0;
    for(; start != end; ++start)
        if(*start == value) ++result;

    return result;
}
```

In the chapter on templates, you'll see how to generalize this function to operate over any type of iterators.

Pointers and References

Problem 5. If we allocate memory by writing `int* myIntPtr = new int`, C++ will only give us space to hold a single integer. However, the code `myIntPtr[0] = 42` is perfectly safe. Recall that `myIntPtr[0]` means to go to the address pointed at by `myIntPtr`, move forward zero elements, then return the value there. But this is equivalent to just looking up the object directly pointed at by `myIntPtr`, and in fact `myIntPtr[0] = 42` is completely equivalent to `*myIntPtr = 42`.

However, `myIntPtr[1] = 42` instructs C++ to access the element one after the element pointed at by `myIntPtr`. We don't own the memory after our single `int`, so this line writes 42 into a memory location. It probably will result in some sort of crash, either immediately or later on when the memory allocator gets confused.

C Strings

Problem 1. The code `myString = "String" + '!'` just *looks* right, doesn't it? It seems like we're concatenating an exclamation point on to the end of a string and then assigning the new string to `myString`. Had we been working with C++ `string` objects, this would be true, but remember that `"String"` is a C string and that `+` means pointer arithmetic rather than concatenation. In fact, what this code does is obtain a pointer to the string `"String"` somewhere in memory, then advance that pointer by the numerical value of the exclamation point character (thirty-three) and store the resulting pointer in `myString`. This results in a string that's pointing into random memory we don't own, resulting in (surprise!) undefined behavior.

The Preprocessor

Problem 5. This restriction exists because the preprocessor is a *compile-time* construct whereas functions are a *runtime* construct. That is, the code that you write for a function is executed only when the program already runs, and preprocessor directives execute before the program begins running (or has even finished compiling, for that matter). If a preprocessor directive were to execute a C++ function, the compiler would need to compile and run that function during preprocessing, which might cause dependency issues if the function referenced code that hadn't yet been preprocessed, or would have to defer preprocessing to runtime, defeating the entire purpose of the preprocessor.

Problem 9. The code for a not-reached macro is actually simpler than that for an assert macro because we don't need to verify any conditions and instead can immediately abort. We can begin by writing the code that actually performs the action associated with the not-reached statement:

```
void DoCS106LNotReached(string message, int line, string filename)
{
    cerr << "CS106LNotReached failed: " << message << endl;
    cerr << "Line number: " << line << endl;
    cerr << "Filename: " << filename << endl;
    abort();
}

#define CS106LNotReached(msg) DoCS106LNotReached(msg, __LINE__, __FILE__)
```

Next, we need to disable the macro in case `NO_CS106L_NOTREACHED` is defined. This can be done as follows:

```
#ifndef NO_CS106L_NOTREACHED

void DoCS106LNotReached(string message, int line, string filename)
{
    cerr << "CS106LNotReached failed: " << message << endl;
    cerr << "Line number: " << line << endl;
    cerr << "Filename: " << filename << endl;
    abort();
}

#define CS106LNotReached(msg) DoCS106LNotReached(msg, __LINE__, __FILENAME__)

#else

#define CS106LNotReached(msg) /* Nothing */

#endif
```

Problem 11. Adding a `NOT_A_COLOR` sentinel to the `Color` enumeration is much easier than it sounds. The X Macro code we have for generating the `Color` enumeration is currently

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
};
```

These preprocessor directives expand out to the full list of colors with a comma following the name of the last color. Thus all we need to do is change the code to look like this:

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
    NOT_A_COLOR
};
```

Now, the `enum` contains a constant called `NOT_A_COLOR` that follows all other colors.

Problem 13. We want to change the definition of the `Color` enumeration so that the names of the colors are prefaced with `eColor_`. Thus the code we want to generate should look like this:

```
enum Color {
    eColor_Red,
    eColor_Green,
    /* ... */
    eColor_White
};
```

Recall that the original X Macro code we had for automatically generating the `Color` enumeration was

```
enum Color {
    #define DEFINE_COLOR(color, opposite) color, // Name followed by comma
    #include "color.h"
    #undef DEFINE_COLOR
};
```

We can modify this to generate the above code by using the token-pasting operator ## to concatenate `eColor_` before the name of each of the colors. The resulting code is

```
enum Color {
    #define DEFINE_COLOR(color, opposite) eColor_##color,
    #include "color.h"
    #undef DEFINE_COLOR
};
```

Introduction to Templates

Problem 1. `copy_if` accepts four parameters – two iterators defining an input range, one iterator defining the output range, and a predicate function determining whether we should copy a particular element. Since we can provide any sort of input iterator, any sort of output iterator, and a predicate that could theoretically accept any type, we'll templatize `copy_if` over the types of the arguments, as shown here:

```
template <typename InputIterator, typename OutputIterator, typename Predicate>
inline OutputIterator copy_if(InputIterator start,
                              InputIterator end,
                              OutputIterator where,
                              Predicate fn)
{
    /* ... */
}
```

Note that although the type of the predicate function depends on the type of iterators (that is, if we're iterating over `string`s we can't give a function accepting an `int`), we've templatized the function with respect to the predicate. This gives the client more leeway in what predicates they can provide. For example, if the iterators iterate over a `vector<int>`, they could provide a predicate function accepting a `double`. Later when we cover functors you'll see a more general reason to templatize the predicate parameter.

Now all that's left to do is write the function body. Fortunately this isn't too tricky – we just keep advancing the `start` iterator forward, checking if the element iterated over passes the predicate and copying the element if necessary. The final code looks like this:

```
template <typename InputIterator, typename OutputIterator, typename Predicate>
inline OutputIterator copy_if(InputIterator start,
                              InputIterator end,
                              OutputIterator where,
                              Predicate fn)
{
    for(; start != end; ++start)
    {
        if(fn(*start))
        {
            *where = *start;
            ++where;
        }
    }
    return where;
}
```

As an FYI, you will sometimes see the code

```
*where = *start;
++where;
```

Written as

```
*where++ = *start;
```

This is a trick that relies on the fact that the `++` operator binds more tightly than the `*` operator. Thus the code is interpreted as `*(where++) = *start`, meaning that we advance the `where` iterator by one step, then store in the location it use to point at the value of `*start`. I personally find this syntax more attractive than the longhand version, but it is admittedly more confusing.

Problem 6. Recall that the code for `GetInteger` is as follows:

```
int GetInteger()
{
    while(true)
    {
        stringstream converter(GetLine());
        int result;

        converter >> result;
        if(!converter.fail())
        {
            char leftover;
            converter >> leftover;

            if(converter.fail()) return result;
            cout << "Unexpected character: " << leftover << endl;
        }
        else cout << "Please enter an integer." << endl;

        cout << "Retry: ";
    }
}
```

To templatize this function over an arbitrary type, we need to change the return type, the type of `result`, and the error message about entering an integer. The modified code is shown here:

```
template <typename ValueType> ValueType GetValue(string type)
{
    while(true)
    {
        stringstream converter(GetLine());
        ValueType result;

        converter >> result;
        if(!converter.fail())
        {
            char leftover;
            converter >> leftover;

            if(converter.fail()) return result;
            cout << "Unexpected character: " << leftover << endl;
        }
        else cout << "Please enter " << type << endl;

        cout << "Retry: ";
    }
}
```

const

Problem 5. At first it seems like it should be safe to convert an `int **` into a `const int **`. After all, we're just adding more `const`s to the pointer, which restricts what we should be able to do with the pointer. How could we possibly use this to subvert the type system? The answer is the following chain of assignments that allow us to overwrite a `const int`:

```
const int myConstant = 137; // Legal
int * evilPtr; // Legal, uninitialized

/* This next line is not legal C++ because &evilPtr is an int** and badPtr is a
 * const int **.  Watch what happens if we allow this.
 */
const int** badPtr = &evilPtr;

/* Dereference badPtr and assign it the address of myConstant.  &myConstant is a
 * const int * and badPtr is a const int*, so the assignment is legal.  However,
 * since badPtr points to evilPtr, this assigns evilPtr the address of the
 * myConstant variable, which is const!
 */
*badPtr = &myConstant;

/* This would overwrite a const variable. */
*evilPtr = 42;
```

This is a subtle edge case, but because it's possible C++ explicitly disallows it.

Problem 6. The initial interface for the CS106B/X `Vector` class is reprinted here:

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size();
    bool isEmpty();

    ElemType getAt(int index);
    void setAt(int index, ElemType value);

    ElemType & operator[](int index);

    void add(ElemType elem);
    void insertAt(int index, ElemType elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(ElemType elem));
    template <typename ClientDataType>
        void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
                    ClientDataType & data);

    Iterator iterator();
};
```

The first task in `const`-correcting this is marking non-mutating operations `const`. This is reasonably straightforward for most of the functions. The interesting case is the `operator[]` function, which returns a reference to the element at a given position. This function needs to be `const`-overloaded since if the `Vector` is `const` we want to hand back a `const` reference and if the `Vector` is non-`const` we want to hand back a non-`const` reference. The updated interface is shown here:

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size() const;
    bool isEmpty() const;

    ElemType getAt(int index) const;
    void setAt(int index, ElemType value);

    ElemType & operator[](int index);
    const ElemType & operator[](int index) const;

    void add(ElemType elem);
    void insertAt(int index, ElemType elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(ElemType elem)) const;
    template <typename ClientDataType>
        void mapAll(void (*fn)(ElemType elem, ClientDataType & data),
                    ClientDataType & data) const;

    Iterator iterator() const;
};
```

Next, we'll update the class by passing all appropriate parameters by reference-to-`const` rather than by value. This results in the following interface:

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size() const;
    bool isEmpty() const;

    ElemType getAt(int index) const;
    void setAt(int index, const ElemType& value);

    ElemType & operator[](int index);
    const ElemType & operator[](int index) const;

    void add(const ElemType& elem);
    void insertAt(int index, const ElemType& elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(const ElemType& elem)) const;
    template <typename ClientDataType>
        void mapAll(void (*fn)(const ElemType& elem, ClientDataType & data),
                    ClientDataType & data) const;

    Iterator iterator() const;
};
```

Finally, we'll change the return type of `getAt` to return a `const` reference to the element in the `Vector` rather than a full copy, since this reduces the cost of the function. The final version of the `Vector` is shown below:

```
template <typename ElemType> class Vector
{
public:
    Vector(int sizeHint = 0);

    int size() const;
    bool isEmpty() const;

    const ElemType& getAt(int index) const;
    void setAt(int index, const ElemType& value);

    ElemType & operator[](int index);
    const ElemType & operator[](int index) const;

    void add(const ElemType& elem);
    void insertAt(int index, const ElemType& elem);
    void removeAt(int index);

    void clear();

    void mapAll(void (*fn)(const ElemType& elem)) const;
    template <typename ClientDataType>
        void mapAll(void (*fn)(const ElemType& elem, ClientDataType & data),
                    ClientDataType & data) const;

    Iterator iterator() const;
};
```

static

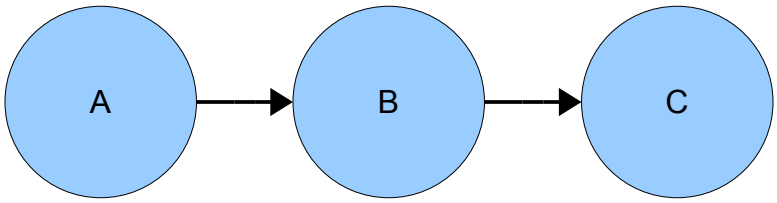
Problem 2. The `UniquelyIdentified` class can be implemented by having a static variable inside the class that tracks the most recently used ID, as well as a non-static variable for each instance that tracks the particular class's unique ID. This can be implemented as follows:

```
class UniquelyIdentified
{
public:
    UniquelyIdentified();

    int getUniqueID() const;
private:
    static int lastUsedID;
    const int currentID;
};

/* Remember, this must go outside the class! */
int UniquelyIdentified::lastUsedID = 1;

UniquelyIdentified::UniquelyIdentified() : currentID(lastUsedID)
{
    ++lastUsedID;
}
```

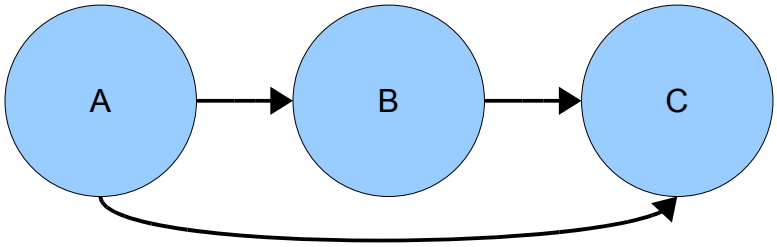
```
int UniquelyIdentified::getUniqueID() const
{
    return currentID;
}
```

Conversion Constructors

Problem 5. C++ will only apply at most one conversion constructor at a time to avoid getting into ambiguous situations. For example, suppose that we start off with types A, B, and C such that A is convertible to B and B is convertible to C. Then we can think of this graphically as follows:



Now, suppose we introduce another conversion into this mix, this time directly from A to C, as shown here:



If we try to implicitly convert from an A to a C, which sequence of conversions is correct? Do we first convert the A to a B and then convert that to a C, or just directly convert the A to a C? There's no clear right answer here, and to avoid confusions like this C++ sidesteps the issue by only applying one implicit conversion at a time.

Copy Constructors and Assignment Operators

Problem 1. The reason that this code is problematic is that at some point in the assignment operator, all of the resources that were allocated by the current class instance need to be cleaned up. However, we never allocated any resources, and none of the class's data members have been initialized. Trying to clean up garbage almost always results in a program crash. In general, you should not implement copy constructors in terms of assignment operators

Problem 2. It is illegal to write a copy constructor that accepts its parameter by value because this causes a circular dependency. Recall that the copy constructor is invoked whenever a function is passed by value into a function, so if the copy constructor itself took its parameter by value it would need to call itself to initialize the parameter by value, but then it would have to initialize the parameter in that function call by invoking itself, etc. This causes a compile-time error rather than a runtime error, by the way.

The assignment operator doesn't have this problem because if the assignment operator were to accept its parameter by value, the copy is made by the copy constructor, which is an entirely different function.

Operator Overloading

Problem 3. The other operators can be defined as follows:

```
A < B     □     A < B
A <= B    □     !(B < A)
A == B    □     !(A < B || B < A)
A != B    □     A < B || B < A
A >= B    □     !(A < B)
A > B     □     B < A
```

Problem 7. For reference, here's the broken interface of the `iterator` class:

```
class iterator
{
public:
    bool operator== (const iterator& other);
    bool operator!= (const iterator& other);

    iterator operator++ ();

    ElemType* operator* () const;
    ElemType* operator-> () const;
};
```

There are five problems in this implementation. The first two have to do with the declarations of the `==` and `!=` operators. Since these functions don't modify the state of the `iterator` (or at least, no sensible implementation should), they should both be marked `const`, as shown here:

```
class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator operator++ ();

    ElemType* operator* () const;
    ElemType* operator-> () const;
};
```

Next, take a look at the return type of `operator*`. Remember that `operator*` is called whenever the iterator is dereferenced. Thus if the iterator is pretending to point to an element of type `ElemType`, this function should return an `ElemType&` (a reference to the value) rather than an `ElemType*` (a pointer to the value). Otherwise code like this:

```
*myItr = 137;
```

Wouldn't compile. The updated interface now looks like this:

```
class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator operator++ ();

    ElemType& operator* () const;
    ElemType* operator-> () const;
};
```

Next, look at the return type of `operator++`. Recall that `operator++` is the prefix `++` operator, meaning that we should be able to write code like this:

```
++(++itr)
```

To increment the iterator twice. Unfortunately, with the above interface this code compiles but does something totally different. Since the return type is `iterator`, the returned object is a *copy* of the receiver object rather than the receiver object itself. Thus the code `++(++itr)` means to increment `itr` by one step, then to increment the *temporary `iterator` object* by one step. This isn't at all what we want to do, so we'll fix this by having `operator++` return a reference to an iterator, as shown here:

```
class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator& operator++ ();

    ElemType& operator* () const;
    ElemType* operator-> () const;
};
```

We've now fixed four of the five errors, so what's left? The answer's a bit subtle – there's nothing *technically* wrong with this interface any more, but we've left out an important function that makes the interface unintuitive. In particular, we've only defined a prefix `operator++` function, meaning that code like `++itr` is legal but not code like `itr++`. We should thus add support for a postfix `operator++` function. The final version of the `iterator` class thus looks like this:

```
class iterator
{
public:
    bool operator== (const iterator& other) const;
    bool operator!= (const iterator& other) const;

    iterator& operator++ ();
    const iterator operator++ (int);

    ElemType& operator* () const;
    ElemType* operator-> () const;
};
```

Functors

Problem 1. `for_each` returns the function passed in as a final parameter so that you can pass in a functor that updates internal state during the loop and then retrieve the updated functor at the end of the loop. For example, suppose that we want to compute the minimum, maximum, and average of a range of data in a single pass. Then we could write the following functor class:

```
class DataSample
{
public:
    /* Constructor initializes minVal and maxVal so that they're always updated,
     * sets the total to zero, and the number of elements to zero.
     */
    DataSample() : minVal(INT_MAX), maxVal(INT_MIN), total(0.0), count(0) {}

    /* Accessor methods. */
    int getMin() const
    {
         return minVal;
    }
    int getMax() const
    {
        return maxVal;
    }
    double getAverage() const
    {
        return total / count;
    }
    int getNumElems() const
    {
        return count;
    }

    /* operator() accepts a piece of input and updates state appropriately. */
    void operator() (int val)
    {
        minVal = min(minVal, val);
        maxVal = max(maxVal, val);
        total += val;
        ++count;
    }
private:
    int minVal, maxVal;
    double total;
    int count;
};
```

Then we can write a function which accepts a range of iterators (presumed to be iterating over `int` values) and then returns a `DataSample` object which contains a summary of that data:

```
template <typename InputIterator>
DataSample AnalyzeSample(InputIterator begin, InputIterator end)
{
    /* Create a temporary DataSample object and pass it through for_each.  This
     * then invokes operator() once for each element in the range, resulting in a
     * DataSample with its values set appropriately.  We then return the result of
     * for_each, which is the updated DataSample.  Isn't that nifty?
     */
    return for_each(begin, end, DataSample());
}
```

Problem 3. `AdvancedBiasedSort` is similar to the `BiasedSort` example from the chapter on STL algorithms. If you'll recall, if we assume that the string that should be the winner is stored in a constant `kWinnerString`, the following comparison function can be used to bias the sort so that `kWinnerString` always comes in front:

```
bool BiasedSortHelper(const string& one, const string& two)
{
    /* Case one: Neither string is the winner string.  Just do a default
     * comparison.
     */
    if(one != kWinnerString && two != kWinnerString)
        return one < two;

    /* Case two: Both strings are the winner string.  Then return false because the
     * string isn't less than itself.
     */
    if(one == kWinnerString && two == kWinnerString)
        return false;

    /* Case three: one is the winner string, two isn't.  Then return true to bias
     * the sort.
     */
    if(one == kWinnerString)
        return true;

    /* Otherwise, two is the winner string and one isn't, so return false to bias
     * the sort.
     */
    return false;
}
```

To update this code so that *any* string can be set as the winner string, we'll need to convert this function into a functor that stores the string as a data member. This can be done as follows:

```
class BiasedSortHelper
{
public:
    explicit BiasedSortHelper(const string& winner) : winnerString(winner) {}
    bool operator() (const string& one, const string& two) const;
private:
    string winnerString;
};
```

```
bool BiasedSortHelper::operator()(const string& one, const string& two) const
{
    /* Case one: Neither string is the winner string.  Just do a default
     * comparison.
     */
    if(one != winnerString && two != winnerString)
        return one < two;

    /* Case two: Both strings are the winner string.  Then return false because the
     * string isn't less than itself.
     */
    if(one == winnerString && two == winnerString)
        return false;

    /* Case three: one is the winner string, two isn't.  Then return true to bias
     * the sort.
     */
    if(one == winnerString)
        return true;

    /* Otherwise, two is the winner string and one isn't, so return false to bias
     * the sort.
     */
    return false;
}
```

We can then implement `AdvancedBiasedSort` as follows:

```
void AdvancedBiasedSort(vector<string>& v, const string& winner)
{
    sort(v.begin(), v.end(), BiasedSortHelper(winner));
}
```

Problem 7. We want to compute the value of $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$ as applied to a range of data. This breaks down into four smaller tasks:

1. Compute the average of the data set.
2. Compute the value of the inner sum.
3. Divide the total by the number of elements.
4. Return the square root of this value.

We can use the `sqrt` function exported by `<cmath>` to perform the square root and `accumulate` for the rest of the work. We'll begin by computing the average of the data, as shown here:

```
template <typename ForwardIterator>
double StandardDeviation(ForwardIterator begin, ForwardIterator end)
{
    const ptrdiff_t numElems = distance(begin, end);
    const double average = accumulate(begin, end, 0.0) / numElems;
    /* ... */
}
```

Note that we've computed the size of the range using the `distance` function, which efficiently returns the number of elements between two iterators. We've stored the number of elements in a variable of type `ptrdiff_t`, which, as the name suggests, is a type designed to hold the distance between two pointers.

Technically speaking we should query the iterator for the type of value returned when computing the distance between the iterators, but doing so is beyond the scope of this text.

Now, let's see how we can use `accumulate` to evaluate the sum. Recall that the four-parameter version of `accumulate` allows us to specify what operation to apply pairwise to the current accumulator and any element of the range. In our case, we'll want to add up the sum of the values of $(x_i - \bar{x})^2$ for each element, so we'll construct a functor that stores the value of the average and whose `operator()` function takes the accumulator and the current element, computes $(x_i - \bar{x})^2$, then adds it to the current accumulator. This is shown here:

```
class StandardDeviationHelper
{
public:
    explicit StandardDeviationHelper(double average) : mean(average) {}
    double operator() (double accumulator, double currentValue) const
    {
        const double expr = currentValue - mean; //  xi - x̄
        return accumulator + expr * expr;        // (xi - x̄)²
    }
private:
    const double mean;
};
```

Given this helper functor, we can now write the following:

```
template <typename ForwardIterator>
double StandardDeviation(ForwardIterator begin, ForwardIterator end)
{
    const ptrdiff_t numElems = distance(begin, end);
    const double average = accumulate(begin, end, 0.0) / numElems;
    const double sum =
          accumulate(begin, end, 0.0, StandardDeviationHelper(average));
    return sqrt(sum / numElems);
}
```

Problem 8. If we were given an arbitrary C string and told to set it to the empty string, we could do so by calling `strcpy(myStr, "")`, where `myStr` is the string variable. To do this to every string in a range, we can use `for_each` to apply the function everywhere and `bind2nd` to lock the second parameter of `strcpy` in place. This is shown here:

```
template <typename ForwardItr>
void ClearAllStrings(ForwardItr begin, ForwardItr end)
{
    for_each(begin, end, bind2nd(ptr_fun(strcpy), ""));
}
```

This works because `bind2nd(ptr_fun(strcpy), "")` is a one-parameter function that passes the argument as the first parameter into `strcpy` with the second parameter locked as the empty string.

Problem 9. Now that we're dealing with regular C++ `string`s, we can clear the strings by calling the `.clear()` member function on each of them. Recalling that the `mem_fun_ref` function transforms a member function into a one-parameter function that calls the specified function on the argument, we can write `ClearAllStrings` as follows:

```
template <typename ForwardItr>
void ClearAllStrings(ForwardItr begin, ForwardItr end)
{
    for_each(begin, end, mem_fun_ref(&string::clear));
}
```

Problem 11. The `replace_if` algorithm takes four parameters – two iterators defining a range of elements, a predicate function, and a value – then replaces all elements in the range for which the predicate returns true with the specified value. In our case, we're given a value as a parameter to the `CapAtValue` function and want to replace elements in the range that compare greater than the parameter with the parameter. Using `bind2nd` and the `greater` operator function, we have the following:

```
template <typename ForwardItr, typename ValueType>
void CapAtValue(ForwardItr begin, ForwardItr end, ValueType maxValue)
{
    replace_if(begin, end, bind2nd(greater<ValueType>(), maxValue), maxValue);
}
```

Introduction to Exception Handling

Problem 1. If you put a `catch(...)` clause at the top of a `catch` cascade, then none of the other `catch` handlers will ever catch exceptions thrown by the `try` block. C++ evaluates each `catch` clause in sequence until it discovers a match, and if the first is a `catch(...)` it will always choose that one first.

Problem 4. We want to modify the code

```
void ManipulateStack(stack<string>& myStack)
{
    if(myStack.empty())
        throw invalid_argument("Empty stack!");

    string topElem = myStack.top();
    myStack.pop();

    /* This might throw an exception! */
    DoSomething(myStack);

    myStack.push(topElem);
}
```

So that if the `DoSomething` function throws an exception we're sure to put the element `topElem` back on the `stack` before propagating the exception. This can be done as follows:

```
void ManipulateStack(stack<string>& myStack)
{
    if(myStack.empty())
        throw invalid_argument("Empty stack!");

    string topElem = myStack.top();
    myStack.pop();

    try
    {
        /* This might throw an exception! */
        DoSomething(myStack);
        myStack.push(topElem);
    }
    catch(...)
    {
        myStack.push(topElem);
        throw;
    }

     myStack.push(topElem);
}
```

Problem 5. We want to solve the same problem as in the above case, but by using RAII to manage the resource instead of manually catching and rethrowing any exceptions. We'll begin by defining an `AutomaticStackManager` class that takes in a reference to a `stack`, then pops the top and stores the result internally. When the destructor executes, we'll push the element back on. This looks like this:

```
template <typename ElemType> class AutomaticStackManager
{
public:
    explicit AutomaticStackManager(stack<ElemType>& s) :
        toManage(s), topElem(s.top())
    {
        toManage.pop();
    }

    ~AutomaticStackManager()
    {
        toManage.push(topElem);
    }
private:
    stack<ElemType>& toManage;
    const ElemType topElem;
};
```

Notice that we've templatized this class with respect to the type of element stored in the `stack`, since there's nothing special about `string`. This is in general a good design philosophy – if you don't need to specialize your code over a single type, make it a template.

We can then rewrite the function as follows:

```
void ManipulateStack(stack<string>& myStack)
{
    if(myStack.empty())
        throw invalid_argument("Empty stack!");

    AutomaticStackManager<string> autoCleanup(myStack);
    DoSomething(myStack);
}
```

Notice how much cleaner and shorter this code is than before – by having objects manage our resources we're sure that we won't leak any resources here. True, we had to write a bit of code for `AutomaticStackManager`, but provided that we use it in more than one circumstance the savings in code simplicity over the manual catch-and-rethrow approach are impressive.

Introduction to Inheritance

Problem 1. We don't have to worry that a pointer of type `Document*` points to an object of concrete type `Document` because `Document` is an abstract class and thus can't be instantiated. You do not need to worry about a pure virtual function call realizing that there's no actual code to execute since C++ will raise a compile-time error if you try to instantiate an abstract class.

Problem 4. One possible implementation for `DerivativeFunction` is shown here:

```
class DerivativeFunction: public Function
{
public:
    explicit DerivativeFunction(Function* toCall) : function(toCall) {}

    virtual double evaluateAt(double where) const
    {
        const double kEpsilon = 0.00001; // Small Δx
        return (function->evaluateAt(where + kEpsilon) -
                function->evaluateAt(where - kEpsilon)) / (2 * kEpsilon);
    }
private:
    Function *const function;
}
```

Here, the constructor accepts and stores a pointer to an arbitary `Function`, and the `evaluateAt` function invokes the virtual `evaluateAt` function of the stored function at the proper points to approximate the derivative.

Bibliography

[Str94]: Bjarne Stroustrup, *The Design and Evolution of C++*. Addison-Wesley: 1994.
[Str09]: Bjarne Stroustrup. "Stroustrup: FAQ" URL: http://www.research.att.com/~bs/bs_faq.html#decline. Accessed 7 Jul 2009.
[Ste07]: Alexander Stepanov. "Short History of STL" URL: http://www.stepanovpapers.com/history%20of%20STL.pdf. Accessed 2 Jul 2009.
[Intel]: Intel Corporation. "965_diagram.gif" URL: http://www.intel.com/assets/image/diagram/965_diagram.gif. Accessed 7 Jul 2009.
[Pic96]: NeilFred Picciotto. "Neil/Fred's Gigantic List of Palindromes" URL: http://www.derf.net/palindromes/old.palindrome.html. Accessed 10 Jul 2009.
[Str09.2]: Bjarne Stroustrup. "Stroustrup: C++" URL: http://www.research.att.com/~bs/C++.html. Accessed 24 Jun 2009.
[Spo08]: Joel Spolsky, *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress: 2008.
[GNU]: GNU Project. "strfry" URL: http://www.gnu.org/s/libc/manual/html_node/strfry.html#strfry. Accessed 4 Aug 2009.
[Sut98]: Herb Sutter. "Advice from the C++ Experts: Be Const-Correct" URL: http://www.gotw.ca/publications/advice98.htm. Accessed 10 Jun 2009.
[MCO]: *Mars Climate Orbiter Mishap Investigation Board Phase I Report*. November 10, 1999. URL: ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf
[Mey05]: Scott Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley: 2005.
[Str09.3]: Bjarne Stroustrup. "C++0x FAQ" URL: http://www.research.att.com/~bs/C++0xFAQ.html#think. Accessed 10 Jul 2009.
[Dij68]: Edsger Dijkstra. "Go To Statement Considered Harmful." *Communications of the ACM*, Vol. 11, No. 3, Mar 1968, pg. 147-148

Index