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 ex-

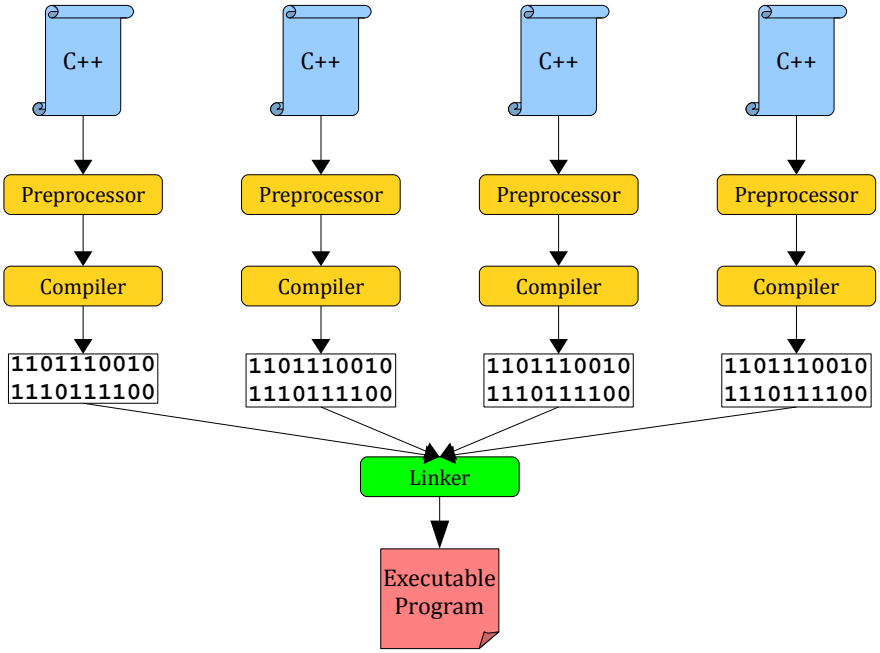
ecutable file that you can run and distribute. During this phase, the linker may report some final errors that prevent it from generating a working C++ program. For example, consider the following C++ program:

```
#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

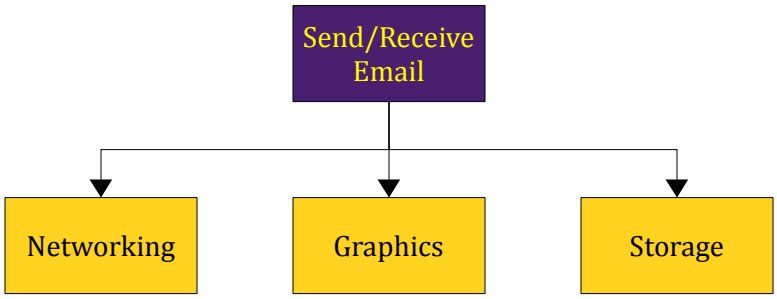
Because compilation and linking are separate steps in C++, it is possible to split up a C++ program across multiple files. To do so, we must first answer two questions:

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?

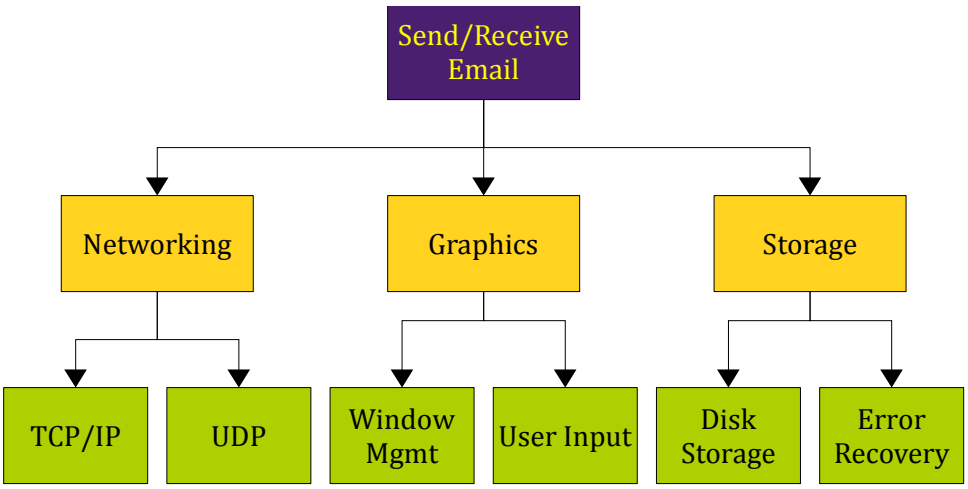
To answer these questions, we first must take a minute to reflect on the structure of most C++ programs.* When writing a C++ program to perform a particular task or solve a particular problem, one usually begins by starting with a large, difficult problem and then solves that problem by breaking it down 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 `#defined` 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 `#defined` 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 `#defined` 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 `#defined` subexpression is treated as a complete entity when used in arithmetic expressions. When writing a `#define` expression in terms of other `#defines`,

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 pre-processed 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 math-

* 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.

ematical 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 pro-

gramming 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 `GetOpposite-Color` 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.