Chapter 7: The Preprocessor

Consider the prototypical C++ “Hello, World!” program, which is reprinted here:

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

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

When you first started off your tour of C++, this code was almost certainly a mystery. But as you've seen more and more C++ in action, this program has probably begun to look less and less mysterious. The declaration of the `main` function is not at all complicated compared to other functions that you've written, and the code to print text to the console is nothing compared to some of the more advanced streams hackery we've seen over the last few chapters. But two lines of this code still remain an enigma – `using namespace std`, which we touched on briefly in an earlier chapter, and `#include <iostream>`. It is this line of code, along with related structures, which are the focus of this chapter.

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. But what exactly goes on behind the scenes that makes this transition possible? There are several steps involved in compilation, among the first of which 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. 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 Since then, both C and C++ have introduced new language features that have obsoleted or superseded much of the preprocessor's functionality and consequently you should attempt to minimize your use of the preprocessor.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

So far, every program you've encountered has begun with several lines using the `#include` directive; for example, `#include <iostream>` or `#include "mail.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 <iostream>` at the top of your C++ programs, it is as if you had literally copied and pasted the contents of the file `iostream` 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 custom libraries, you've surrounded the name of the file in double quotes (e.g. `"nstream.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 the 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);
```

And the computation will be incorrect. The problem is that the preprocessor treats the replacement for `TOTAL_MARGIN` as a string, not a mathematic 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)
```

[*] Technically speaking, the preprocessor operates on "preprocessor tokens," which are slightly different from the whitespace-delineated 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.

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.

Compile-time Conditional Expressions

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

MyFile.h

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

What happens when we try to compile the following program?

```
#include "myfile.h"
#include "myfile.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 `myfile.h` into the program without checking whether or not it has already included the file. Consequently, it will copy the contents of `myfile.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 "myfile.h"` appearing twice in one file. Here is a slightly modified version of the `myfile.h` file that introduces some conditional directives:

MyFile.h, version 2

```
#ifndef MyFile_included
#define MyFile_included

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

#endif
```

Here, we've surrounded the entire file in a block `#ifndef MyFile_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 "myfile.h"
#include "myfile.h" // #include the same file twice

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

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

```
#ifndef MyFile_included
#define MyFile_included

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

#endif

#ifndef MyFile_included
#define MyFile_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 `MyFile_included` hasn't been defined yet. The code then `#define`s `MyFile_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`,

* 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.

this is the value of the parameter plus one because the replacement is interpreted as a 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 #defined symbols. Programmers expect certain structures when reading C++ code, and by obscuring those structures behind walls of #defines 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 #defined 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 section:

```
#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).

Practical Applications of the Preprocessor I: Diagnostic Debugging Functions

When writing a program, at times you may want to ensure that certain invariants about your program hold true – for example, that certain pointers cannot be NULL, that a value is always less than some constant, etc. While in many cases these conditions should be checked using a language feature called *exception handling*, in several cases it is acceptable to check them at runtime using a standard library macro called `assert`. `assert`, exported by the header `<cassert>`, is a macro that 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, `as-sert` is usually used to express programmer assumptions about the state of execution. For example, assuming we have some enumerated type `Color`, suppose we want to write a function that returns whether a color is a primary color. Here's one possible 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 re-turn 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.

You should not, however, use `assert` to check that input from `GetLine` is correctly-formed, for example, since it makes far more sense to reprompt the user than to terminate the program.

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 problematic input, 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, leading to different behavior in debug and release builds. This is a persistent source of debugging headaches.

Using the tools outlined in this chapter, it's possible for us to write our own version of the `assert` macro, which we'll call `CS106LAssert`, to see how to use the preprocessor to design such utilities. We'll split the work into two parts – a function called `DoCS106LAssert`, which actually performs the testing and error-printing, and the macro `CS106LAssert`, which will set up the parameters to this function appropriately. The `DoCS106LAssert` function will look like this:

```
#include <cstdlib> // for abort();

/* These parameters will be assumed to be correct. */
void DoCS106LAssert(bool invariant, string statement, string file, int line)
{
    if(!invariant)
    {
        cerr << "CS106LAssert Failed!" << endl;
        cerr << "Expression: " << statement << endl;
        cerr << "File:       " << file << endl;
        cerr << "Line:       " << line << endl;
        abort(); // Quits program and signals error to the OS.
    }
}
```

This function takes in the expression to evaluate, along with a string representation of that statement, the name of the file it is found in, and the line number of the initial expression. It then checks the invariant, and, if it fails, signals an error and quits the program by calling `abort()`. Since these parameters are rather bulky, we'll hide them behind the scenes by writing the `CS106LAssert` macro as follows:

```
#define CS106LAssert(expr) DoCS106LAssert(expr, #expr, __FILE__, __LINE__)
```

This macro takes in a single parameter, an expression `expr`, and passes it in to the `DoCS106LAssert` function. To set up the second parameter to `DoCS106LAssert`, we get a string representation of the expression using the stringizing operator on `expr`. Finally, to get the file and line numbers, we use the special preprocessor symbols `__FILE__` and `__LINE__`. Note that since the macro is expanded at the call site, `__FILE__` and `__LINE__` refer to the file and line where the macro is used, not where it was declared.

* In practice, this isn't always the case. But it's still a nice theory!

To see CS106LAssert in action, suppose we make the following call to CS106LAssert in myfile.cpp at line 137. Given this code:

```
CS106LAssert(myPtr != NULL);
```

The macro expands out to

```
DoCS106LAssert(myPtr != NULL, "myPtr != NULL", __FILE__, __LINE__);
```

Which in turn expands to

```
DoCS106LAssert(myPtr != NULL, "myPtr != NULL", "myfile.cpp", 137);
```

Which is exactly what we want.

Now, suppose that we've used CS106LAssert throughout a C++ program and have successfully debugged many of its parts. In this case, we want to disable CS106LAssert for a release build, so that the final program doesn't have the overhead of all the runtime checks. To allow the user to toggle whether CS106LAssert has any effect, we'll let them #define a constant, NO_CS106L_ASSERT, that disables the assertion. If the user does not define NO_CS106L_ASSERT, we'll use #define to have the CS106LAssert macro perform the runtime checks. Otherwise, we'll have the macro do nothing. This is easily accomplished using #ifndef ... #else ... #endif to determine the behavior of CS106LAssert. This smart version of CS106LAssert is shown below:

```
#ifndef NO_CS106L_ASSERT // Enable assertions

#include <cstdlib> // for abort();

/* Note that we define DoCS106LAssert inside this block, since if
 * the macro is disabled there's no reason to leave this function sitting
 * around.
 */
void DoCS106LAssert(bool invariant, string statement, string file, int line)
{
    if(!invariant)
    {
        cerr << "CS106LAssert Failed!" << endl;
        cerr << "Expression: " << statement << endl;
        cerr << "File:  " << file << endl;
        cerr << "Line: " << line << endl;
        abort(); // Quits program and signals error to the OS.
    }
}

#define CS106LAssert(expr) DoCS106LAssert(expr, #expr, __FILE__, __LINE__)

#else // Disable assertions

/* Define CS106LAssert as a macro that expands to nothing.  Now, if we call
 * CS106LAssert in our code, it has absolutely no effect.
 */
#define CS106LAssert(expr) /* nothing */

#endif
```

Practical Applications of the Preprocessor II: The X Macro Trick

That macros give C++ programs access to their own source code can be used in other ways as well. 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. What is the difference between angle brackets and double-quotes in the context of `#include`?

2. Why is it a bad idea to use `#define` to create constants? What should you do instead?

3. 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?)*

4. What is an include guard? How do you write one? When and why are they necessary?

5. 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.

6. 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)`?)*

7. 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. ♦

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

9. 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.

[*] 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.

10. A common but nonstandard variant of the `assert` macro is the `verify` macro which, like `assert`, checks a condition at runtime and prints and error and terminates if the condition is false. However, in optimized builds, `verify` is not disabled, but simply does not abort at runtime if the condition is false. This allows you to use `verify` to check the return value of functions directly (Do you see why?). Create a function called `CS106LVerify` that, unless the symbol `NO_CS106L_VERIFY` is defined, checks the parameter and aborts the program if it is false. Otherwise, if `NO_CS106L_VERIFY` is defined, check the condition, but do not terminate the program if it is false.

11. Another common debugging macro is a "not reached" macro which automatically terminates the program if executed. "Not reached" macros are useful inside constructs such as `switch` statements where the `default` label should never be encountered. Write a macro, `CS106LNotReached`, that takes in a string parameter and, if executed, prints out the string, file name, and line number, then calls `abort` to end the program. As with `CS106LAssert` and `CS106LVerify`, if the user `#define`s the symbol `NO_CS106L_NOTREACHED`, change the behavior of `CS106LNotReached` so that it has no effect. ♦

12. If you've done the two previous problems, you'll notice that we now have three constants, `NO_CS106L_ASSERT`, `NO_CS106L_VERIFY`, and `NO_CS106L_NOTREACHED`, that all must be `#define`d to disable them at runtime. This can be a hassle and could potentially be incorrect if we accidentally omit one of these symbols. Write a code snippet that checks to see if a symbol named `DISABLE_ALL_CS106L_DEBUG` is defined and, if so, disables all of the three aforementioned debugging tools. However, still give the user the option to selectively disable the individual functions.

13. 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?)* ♦

14. 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`.

15. 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)* ♦

16. 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.

17. Suppose that you are designing a control system for an autonomous vehicle in the spirit of the DARPA Grand Challenge. As part of its initialization process, the program needs to call a function named `InitCriticalInfrastructure()` to set up the car's sensor arrays. In order for the rest of the program to respond in the event that the startup fails, `InitCriticalInfrastructure()` returns a `bool` indicating whether or not it succeeded. To ensure that the function call succeeds, you check the return value of `InitCriticalInfrastructure()` as follows:

    ```
    assert(InitCriticalInfrastructure());
    ```

    During testing, your software behaves admirably and you manage to secure funding, fame, and prestige. You then compile your program in release mode, install it in a production car, and to your horror find that the car immediately drives off a cliff. Later analysis determines that the cause of the problem was that `InitCriticalInfrastructure` had not been called and that consequently the sensor array had failed to initialize.

18. Why did the release version of the program not call `InitCriticalInfrastructure`? How would you rewrite the code that checks for an error so that this problem doesn't occur?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.