Intro to the Preprocessor

Introduction

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 handout introduces the preprocessor, illustrates its basic syntax, and highlights potential sources of error.

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 Since then, both C and C++ have introduced new language features that have obsoleted or superseded much of the preprocessor's functionality and consequently you should attempt to minimize your use of the preprocessor.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 in the next handout – but do consider other options before adding a hastily-crafted directive.

`#include` Explained

In both CS106X 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 CS106X 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 CS106X-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 be at start
                                     // of 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 directive!
```

`#define`

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 from your code.

Note that the preprocessor treats C++ source code as strings, rather than representations of higher-level C++ constructs. For example, the preprocessor treats `int x = 137` as the string "`int x = 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 Latin 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," string representations of the individual elements of a C++ program. Thus the above example would be treated as the series of tokens "`int`" "`x`" "`=`" "`137`." However, in this handout, it's safe to treat preprocessor tokens and complete strings interchangeably.

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)
```

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; // Ooops-- 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. Most 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 I 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.

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 completely 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 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*, a sort of compile-time function that accepts parameters. 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 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;
}
```

---

* 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.

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 in the next handout, 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 do it with a regular C++ function. If you can, use the C++ function instead of the macro – you'll save yourself hours of debugging nightmares.

A `#define` Cautionary Tale

`#define` is a powerful directive that enables you to completely transform C++. You can rename every keyword to something else, convert operators into one another, and create simple shorthands that expand out into complex structures. However, virtually every C/C++ expert agrees that you should not (note the double-underline!) 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 almost certainly confuse anyone reading 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.

More to Explore

While this handout has covered some of the more common uses of the preprocessor, there are several features of the preprocessor that we have not discussed. If you're interested in learning more about the preprocessor, consider reading into the `#pragma` directive. If you ever browse the header files that ship with your compiler, you may notice frequent use of the `#pragma` directive. `#pragma` is a compiler-specific preprocessor directive that helps to fine-tune compiler behavior. Since the functionality available in `#pragma` changes from compiler to compiler, we have not covered it in this class. Consult your compiler documentation for more details.

Practice Problems

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 handout.
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)`?)*