The C Preprocessor

The C preprocessor modifies a source code file before handing it over to the compiler. You're most likely used to using the preprocessor to include files directly into other files, or #define constants, but the preprocessor can also be used to create "inlined" code using macros expanded at compile time and to prevent code from being compiled twice.

There are essentially three uses of the preprocessor--directives, constants, and macros. Directives are commands that tell the preprocessor to skip part of a file, include another file, or define a constant or macro. Directives always begin with a sharp sign (#) and for readability should be placed flush to the left of the page. All other uses of the preprocessor involve processing #define'd constants or macros. Typically, constants and macros are written in ALL CAPS to indicate they are special (as we will see).

Header Files

The #include directive tells the preprocessor to grab the text of a file and place it directly into the current file. Typically, such statements are placed at the top of a program--hence the name "header file" for files thus included.

Constants

If we write

```
#define [identifier name] [value]
```

whenever [identifier name] shows up in the file, it will be replaced by [value].

If you are defining a constant in terms of a mathematical expression, it is wise to surround the entire value in parentheses:

```
#define PI_PLUS_ONE (3.14 + 1)
```

By doing so, you avoid the possibility that an order of operations issue will destroy the meaning of your constant:

```
x = PI_PLUS_ONE * 5;
```

Without parentheses, the above would be converted to

```
x = 3.14 + 1 * 5;
```

which would result in 1 * 5 being evaluated before the addition, not after. Oops!

It is also possible to write simply

```
#define [identifier name]
```

which defines [identifier name] without giving it a value. This can be useful in conjunction with another set of directives that allow conditional compilation.

Conditional Compilation

There are a whole set of options that can be used to determine whether the preprocessor will remove lines of code before handing the file to the compiler. They include #if, #elif, #else, #ifdef, and #ifndef. An #if or #if/#elif/#else block or a #ifdef or #ifndef block must be terminated with a closing #endif.

The #if directive takes a numerical argument that evaluates to true if it's non-zero. If its argument is false, then code until the closing #else, #elif, of #endif will be excluded.

Commenting out Code

Conditional compilation is a particularly useful way to comment out a block of code that contains multi-line comments (which cannot be nested).

```
#if 0
/* comment ...
*/

// code

/* comment */
#endif
```

Include Guards

Another common problem is that a header file is required in multiple other header files that are later included into a source code file, with the result often being that variables, structs, classes or functions appear to be defined multiple times (once for each time the header file is included). This can result in a lot of compile-time headaches. Fortunately, the preprocessor provides an easy technique for ensuring that any given file is included once and only once.

By using the #ifndef directive, you can include a block of text only if a particular expression is undefined; then, within the header file, you can define the expression. This ensures that the code in the #ifndef is included only the first time the file is loaded.

```
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_

/* code */

#endif // #ifndef _FILE_NAME_H_
```

Notice that it's not necessary to actually give a value to the expression _FILE_NAME_H_. It's sufficient to include the line "#define _FILE_NAME_H_" to make it "defined". (Note that there is an **n** in #ifndef--it stands for "if not defined").

A similar tactic can be used for defining specific constants, such as NULL:

```
#ifndef NULL
#define NULL (void *)0
#endif // #ifndef NULL
```

Notice that it's useful to comment which conditional statement a particular #endif terminates. This is particularly true because preprocessor directives are rarely indented, so it can be hard to follow the flow of execution.

On many compilers, the #pragma once directive can be used intead of include guards.

Macros

The other major use of the preprocessor is to define macros. The advantage of a macro is that it can be type-neutral (this can also be a disadvantage, of course), and it's inlined directly into the code, so there isn't any function call overhead. (Note that in C++, it's possible to get around both of these issues with templated functions and the inline keyword.)

A macro definition is usually of the following form:

```
#define MACRO_NAME(arg1, arg2, ...) [code to expand to]
```

For instance, a simple increment macro might look like this:

```
#define INCREMENT(x) x++
```

They look a lot like function calls, but they're not so simple. There are actually a couple of tricky points when it comes to working with macros. First, remember that the exact text of the macro argument is "pasted in" to the macro. For instance, if you wrote something like this:

```
#define MULT(x, y) x * y
```

and then wrote

```
int z = MULT(3 + 2, 4 + 2);
```

what value do you expect z to end up with? The obvious answer, 30, is wrong! That's because what happens when the macro MULT expands is that it looks like this:

```
int z = 3 + 2 * 4 + 2;    // 2 * 4 will be evaluated first!
```

So z would end up with the value 13! This is almost certainly not what you want to happen. The way to avoid it is to force the arguments themselves to be evaluated before the rest of the macro body. You can do this by surrounding them by parentheses in the macro definition:

```
#define MULT(x, y) (x) * (y)
// now MULT(3 + 2, 4 + 2) will expand to (3 + 2) * (4 + 2)
```

But this isn't the only gotcha! It is also generally a good idea to surround the macro's code in parentheses if you expect it to return a value. Otherwise, you can get similar problems as when you define a constant. For instance, the following macro, which adds 5 to a given argument, has problems when embedded within a larger statement:

```
#define ADD_FIVE(a) (a) + 5

int x = ADD_FIVE(3) * 3;
// this expands to (3) + 5 * 3, so 5 * 3 is evaluated first
// Now x is 18, not 24!
```

To fix this, you generally want to surround the whole macro body with parentheses to prevent the surrounding context from affecting the macro body.

```
#define ADD_FIVE(a) ((a) + 5)

int x = ADD_FIVE(3) * 3;
```

On the other hand, if you have a multiline macro that you are using for its side effects, rather than to compute a value, you probably want to wrap it within curly braces so you don't have problems when using it following an if statement.

```
// We use a trick involving exclusive-or to swap two variables
#define SWAP(a, b)  a ^= b; b ^= a; a ^= b;

int x = 10;
int y = 5;

// works OK
SWAP(x, y);

// What happens now?
if(x < 0)
    SWAP(x, y);
```

When SWAP is expanded in the second example, only the first statement, a ^= b, is governed by the conditional; the other two statements will always execute. What we really meant was that all of the statements should be grouped together, which we can enforce using curly braces:

```
#define SWAP(a, b)  {a ^= b; b ^= a; a ^= b;}
```

Now, there is still a bit more to our story! What if you write code like so:

```
#define SWAP(a, b)  { a ^= b; b ^= a; a ^= b; }

int x = 10;
int y = 5;
int z = 4;

// What happens now?
if(x < 0)
    SWAP(x, y);
else
    SWAP(x, z);
```

Then it will not compile because semicolon after the closing curly brace will break the flow between if and else. The solution? Use a do-while loop:

```
#define SWAP(a, b)  do { a ^= b; b ^= a; a ^= b; } while ( 0 )

int x = 10;
int y = 5;
int z = 4;

// What happens now?
if(x < 0)
    SWAP(x, y);
else
    SWAP(x, z);
```

Now the semi-colon doesn't break anything because it is part of the expression. (By the way, note that we didn't surround the arguments in parentheses because we don't expect anyone to pass an expression into swap!)

More Gotchas

By now, you've probably realized why people don't really like using macros. They're dangerous, they're picky, and they're just not that safe. Perhaps the most irritating problem with macros is that you don't want to pass arguments with "side effects" to macros. By side effects, I mean any expression that does something besides evaluate to a value. For instance, ++x evaluates to x+1, but it also increments x. This increment operation is a side effect.

The problem with side effects is that macros don't evaluate their arguments; they just paste them into the macro text when performing the substitution. So something like

```
#define MAX(a, b) ((a) < (b) ? (b) : (a))
int x = 5, y = 10;
int z = MAX(x++, y++);
```

will end up looking like this:

```
int z = (x++ < y++ ? y++ : x++)
```

The problem here is that y++ ends up being evaluated twice! The nasty consequence is that after this expression, y will have a value of 12 rather than the expected 11. This can be a real pain to debug!

Multiline macros

Until now, we've seen only short, one line macros (possibly taking advantage of the semicolon to put multiple statements on one line.) It turns out that by using a the "\" to indicate a line continuation, we can write our macros across multiple lines to make them a bit more readable.

For instance, we could rewrite swap as

```
#define SWAP(a, b)  {                   \
                        a ^= b;         \
                        b ^= a;         \
                        a ^= b;         \
                    }
```

Notice that you **do not** need a slash at the end of the last line! The slash tells the preprocessor that the macro continues to the next line, not that the line is a continuation from a previous line.

Aside from readability, writing multi-line macros may make it more obvious that you need to use curly braces to surround the body because it's more clear that multiple effects are happening at once.

Advanced Macro Tricks

In addition to simple substitution, the preprocessor can also perform a bit of extra work on macro arguments, such as turning them into strings or pasting them together.

Pasting Tokens

Each argument passed to a macro is a token, and sometimes it might be expedient to paste arguments together to form a new token. This could come in handy if you have a complicated structure and you'd like to debug your program by printing out different fields. Instead of writing out the whole structure each time, you might use a macro to pass in the field of the structure to print.

To paste tokens in a macro, use ## between the two things to paste together.

For instance

```
#define BUILD_FIELD(field) my_struct.inner_struct.union_a.##field
```

Now, when used with a particular field name, it will expand to something like

```
my_struct.inner_struct.union_a.field1
```

The tokens are literally pasted together.

String-izing Tokens

Another potentially useful macro option is to turn a token into a string containing the literal text of the token. This might be useful for printing out the token. The syntax is simple--simply prefix the token with a pound sign (#).

```
#define PRINT_TOKEN(token) printf(#token " is %d", token)
```

For instance, PRINT_TOKEN(foo) would expand to

```
printf("<foo>" " is %d" <foo>)
```

(Note that in C, string literals next to each other are concatenated, so something like "token" " is " " this " will effectively become "token is this". This can be useful for formatting printf statements.)

For instance, you might use it to print the value of an expression as well as the expression itself (for debugging purposes).

```
PRINT_TOKEN(x + y);
```

Avoiding Macros in C++

In C++, you should generally avoid macros when possible. You won't be able to avoid them entirely if you need the ability to paste tokens together, but with templated classes and type inference for templated functions, you shouldn't need to use macros to create type-neutral code. Inline functions should also get rid of the need for macros for efficiency reasons. (Though you aren't guaranteed that the compiler will inline your code.)

Moreover, you should use const to declare typed constants rather than #define to create untyped (and therefore less safe) constants. Const should work in pretty much all contexts where you would want to use a #define, including declaring static sized arrays or as template parameters.