The C++ Preprocessor

A C++ (or C) compiler begins by invoking the *preprocessor*, a program that uses special statements, known as *directives* or *control statements*, that cause special compiler actions such as:

- *file inclusion*, in which the file being preprocessed incorporates the contents of another file, exactly as if the included file's text were actually part of the including file;
- *macro substitution*, in which one sequence of text is replaced by another;
- *conditional compilation*, in which parts of the source file's code can be eliminated at compile time under certain circumstances.

All preprocessor directives begin with the # symbol (known as *pound* or *hash*), which **must** occur in the leftmost column of the line. A preprocessor directive that takes up more than one line needs a *continuation* symbol, \ (backslash), as the very last character of every line except the last.

File Inclusion

To include a file inside the file that you are currently compiling, use the `#include` directive, which takes either of two forms:

1. `#include <`*filename*`> // Note:  `*filename*` in pointy brackets`
   which includes the given file from one of the standard directories (e.g., the directory `/usr/include/cxx` on `ecnalpha`);
2. `#include "`*filename*`" // Note:  `*filename*` in double quotes`
   which includes the given file from the current working directory.

The `#include` directive is replaced by the entire contents of the requested file.

File inclusion can be nested; that is, if you have a file named `myprogram.h` that includes the standard input/output stream file named `iostream.h`, and you have a file named `myprogram.cpp` that includes the file `myprogram.h`, then when you compile the file `myprogram.cpp`, the files `myprogram.h` and `iostream.h` will both get included, in the appropriate places.

Macro Substitution

A *macro* is a sequence of text that is defined as another (perhaps empty) sequence of text. After a macro has been defined, whenever the preprocessor encounters that macro in the text of the source file being preprocessed, it replaces the macro with the substitution text.

To define a macro, you use the `#define` preprocessor directive, like so:

```
#define MACRO_NAME text_to_be_substituted
```

Once `MACRO_NAME` has been defined, then for the rest of the source file being preprocessed, the preprocessor will replace any occurrence of `MACRO_NAME` with `text_to_be_substituted`. For example, if your source file has:

```
#define Integer2Byte                                  short
#define NUMBER_OF_STUDENTS_IN_CS2413_SUMMER_2000    45
...
Integer2Byte numberOfStudents =
                  NUMBER_OF_STUDENTS_IN_CS2413_SUMMER_2000;
```

the preprocessor will delete the macro definition lines and replace the declaration statement with:

```
short numberOfStudents =
                  45;
```

Note that you can define a macro with no substitution text:

```
#define THIS_MACRO_HAS_NO_SUBSTITUTION_TEXT
```

In this case, the preprocessor simply takes note of the fact that the macro exists; if it encounters the macro within the source text, it simply deletes it. This property may seem, at first blush, to be pretty useless, but it turns out to be very handy in conditional compilation (below).

Note that you can "undefine" (eliminate) a macro definition using the `#undef` directive:

```
#undef NUMBER_OF_STUDENTS_IN_CS2413_SUMMER_2000
```

Conditional Compilation

To compile conditionally, define a macro or set of macros and then use one of these:

- The `#ifdef`-`#endif` directive pair:

  ```
  #ifdef INCREMENT_A
      a = a + 1;
  #endif
  ```

  Note that `#ifdef` stands for "if **has** been defined;" in this case, the statement

  ```
      a = a + 1;
  ```

  will only be compiled if the macro named `INCREMENT_A` **has** been defined (regardless of what substitution text, if any, is associated with `INCREMENT_A`); if `INCREMENT_A` **has not** been defined, then the statement will be deleted.

- The `#ifndef`-`#endif` directive pair:

  ```
  #ifndef DECREMENT_B_AND_C
      b = b - 1;
      c = c - 1;
  #endif
  ```

  Note that `#ifndef` stands for "if **has not** been defined;" in this case, the statements

  ```
      b = b - 1;
      c = c - 1;
  ```

  will only be compiled if `DECREMENT_B_AND_C` **has not** been defined (regardless of substitution text, if any); if `DECREMENT_B_AND_C` **has** been defined, then the statements will be deleted.

- The `#if`-`#endif` directive pair:

  ```
  Point::Point ()
  {
  #if DEBUGGING_VERBOSITY > 1
      cout << "Default Point constructor called" << endl;
  #endif
      _x = 0; _y = 0;
  }
  ```

  In this case, the statement

  ```
      cout << "Default Point constructor called" << endl;
  ```

will be compiled only if the integer-valued expression `DEBUGGING_VERBOSITY > 1` evaluates to the Boolean value `true` (i.e., any non-zero integer value); if the expression evaluates to the Boolean value `false` (i.e., integer zero), then the statement will be deleted. For example, if the macro `DEBUGGING_VERBOSITY` has been defined as 2 (which is greater than 1), then the statement will be compiled, but if `DEBUGGING_VERBOSITY` has been defined as 1 (which is not greater than 1), then the statement will be deleted.

Note that the expression that follows `#if` can have any legal C++ syntax for integer-valued expressions, but its atomic terms must be either (a) integer-valued literal constants (such as 2, 19, -303984, etc.), or (b) macros that reduce to integer-valued literal constants; that is, no variables, function calls or method invocations are permitted, because the preprocessor (as opposed to the actual compiler) doesn't know anything about them.

Note that all three of these conditional compilation forms have an associated `#else` directive, whose text will be compiled **if and only if** the associated `#ifdef`, `#ifndef` or `#if` clause is **not** compiled; for example:

```
#ifdef LOOP_BACKWARDS
    for (i = last; i >= first; i--)
#else
    for (i = first; i <= last; i++)
#endif
```

In addition to defining macros inside of source files, you can also define them in the command line of the `g++` compiler (and most other Unix-based C++ compilers), using the `-D` compiler option (note that the percent sign `%` is the Unix prompt):

```
% g++ -DDEBUG_VERBOSITY=2 Point.cpp
```

Notice that there is no space between the `-D` and the name of the macro being defined, and that its substitution text, if any, comes after an equals sign, with no spaces in between.

In this case, the compiler behaves exactly as if `Point.cpp` began with a directive

```
#define DEBUG_VERBOSITY    2
```

Likewise, macros can be undefined in the compile command using the `-U` compiler option:

```
% g++ -UDEBUG_VERBOSITY Point.cpp
```

In this case, the compiler behaves exactly as if `Point.cpp` began with a directive

```
#undef DEBUG_VERBOSITY
```

Header Files

In many cases, we want to isolate a set of macro definitions and class declarations from the executable code of the associated methods or functions, because we need the definitions and declarations to be known in a variety of places throughout a large program, but we only need to compile the executable code once.

For example, suppose that we have a class named `Point`, and suppose that we want, in our `main` routine, to declare a variable that is an instance of `Point`; e.g.,

```
int main ()
{
    Point p;
    ...
}
```

In this case, we need for `main` to know the definition of the `Point` class, but we don't need `main` to know the particulars of how the methods of `Point` are implemented.

When this situation comes up, we create a special file, known as a *header file*, that contains the appropriate macros and declarations, but not the associated method or function implementations. We include this file at the top of the associated file of executable statements, as well as at the top of the file that contains `main`.

For example, we might have a file named `Point.h` that looks like this:

```
#ifndef Point_h
#define Point_h

class Point {
    friend ostream& operator<< (ostream& s, Point& p);
    protected:
        double _x, _y;
    public:
        Point();
        virtual ~Point();
        void display();
        ...
} ; // class Point

#endif // #ifndef Point_h
```

Associated with this file would be a file named `Point.cpp`, containing the source code of the definitions of the methods of the class `Point`, that looks like this:

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

Point::Point ()
{
    _x = 0; _y = 0;
}

Point::~Point () { }

void Point::display ()
{
    cout << "x = " << _x << ", y = " << _y << endl;
}
...
```

Notice that the file `iostream.h` is asserted to be in a standard include directory, because in its `#include` directive it's between pointy brackets, while `Point.h` is asserted to be in the current working directory, because in its `#include` directive it's between double quotes.

Also notice that, in the text of `Point.h`, there are several directives that may at first seem odd:

```
#ifndef Point_h
#define Point_h
...
#endif // #ifndef Point_h
```

What purpose do these directives serve? They insure that the file `Point.h` is included no more than once during any given compilation.

How does this work? Well, preprocessor directives are executed in the order in which the preprocessor encounters them. Thus, the first time that `Point.h` is included, the macro `Point_h` has not yet been defined. So, the directive `#ifndef Point_h` ("if `Point_h` has not yet been defined") evaluates to `true`, and therefore all of the statements (including preprocessor directives) between the `#ifndef Point_h` directive and the `#endif` directive will be compiled.

The very first statement after the `#ifndef Point_h` directive is the `#define Point_h` directive. If, later during this current compilation, there is another attempt to include `Point.h`, then the `#ifndef Point_h` will evaluate to `false`, because `Point.h` will by then have been defined. In this case, the rest of the file `Point.h` will not be included in that place – which is fine, because it already has been included, earlier during this compilation, so the preprocessor already knows everything that `Point.h` contains.

Summary

The C++ preprocessor is a powerful tool for manipulating text files. While file inclusion, macro substitution and conditional compilation are its most popular features, some versions of the preprocessor provide additional directives such as `#pragma`, which can be used for implementation-specific features such as parallelization (for multiprocessing) and optimization of executable statements.

References

S. Radhakrishnan, L. Wise & C. N. Sekharan, *Object-Oriented Data Structures Featuring C++*, 1999.

B. Stroustrup, *The C++ Programming Language* (1st ed.), Addison-Wesley Publishing Co., Massachusetts, 1986.