C PREPROCESSOR

The *preprocessor* is a part of C compilation process that recognizes statements that are preceded by a pound sign (#), as already introduced in Chapter 2. Although most C compilers have the preprocessor integrated into it, the preprocessing is considered independent, since it works on the source code before compilation. Preprocessor statements have a different syntax from that of normal C statements, and are used for the following purposes:

1. Including header files
2. Conditional compilation
3. Macro definitions

Including files

The *# include* directive is used to include both system and user header files. This directive instructs the C preprocessor to scan the included file before proceeding with the current file. Following are the two ways for including a file:

```
#include <systemfile.h>
#include "userfile.h"
```

The first statement searches for an include file by name, *systemfile.h* in the example, in the standard system directories. A programmer can specify the directories to search for include files by using the compiler option -I. You can also direct the compiler to omit searching the standard system directories by using the compiler switch *-noinstdinc*

The second statement is used for including a header file defined by a programmer, the name is userfile.h in the example, in the current directory. The preprocessor will first search for the file in the directory where the program exists, and then if it does not find such an include file, then it looks for it in the standard system directories.

This preprocessor directive must appear in your program before any of the definitions contained in the header file are referenced. The preprocessor searches for this file on the system and includes its contents to the program at the point where the #include statement appears.

Conditional compilation

In case you are developing a software package that needs to run on different flavors of UNIX, you have to write the programs in such a way that they are able to execute successfully on all platforms without errors. This is known as a portable application. It is very common to find portions of the program which can run only one operating system. In such a case where you have to switch on or off various statements in a program, conditional compilation is used. This applies to running a program in debug mode or in release mode.

Some of the most common statements used for implementing conditional compilation are:

- #define, #ifdef, #ifndef

  Consider the following example:

  ```
  #ifdef _WIN32_
  #define INT_SIZE 32
  #else
  #define INT_SIZE 16
  #endif
  ```

  The above statements check whether it is a Windows NT operating system by checking if the constant WIN32 has been previously defined. If so, then the constant INT_SIZE is set to 32 else for other operating systems it is set to 16.

- #if

  #if *expression*

lines
#endif
The preprocessor evaluates *expression* and if it is different from zero, *lines* are compiled otherwise they are ignored.

- #if with #else

  #if *expression*
  lines1
  #else
  lines2
  #endif
  The preprocessor evaluates *expression* and if it is different from zero, *lines1* are compiled and *lines2* ignored. Otherwise if it is equal to zero, *lines2* are compiled and*lines1* ignored.

- #if with #elif

  elif corresponds to else if.
  #if *expression1*
  lines1
  #elif *expression2*
  lines2
  ...
  #elif *expressionN*
  linesN
  #else
  linesF
  #endif
  The preprocessor evaluates *expressioni* till it finds one that is different from zero, in this case *linesi* will be compiled. Otherwise if they are all equal to zero, *linesF* will be compiled.

The *#error* directive displays an error message which is defined on the line past this directive. This directive stops compilation. This is used to alert a programmer that there is an error in the compilation of the program.

```
#ifdef DEBUG
#ifdef RELEASE
#error DEBUG and RELEASE both are defined
#endif
#endif
```

The *#warning* directive is similar to #error, but this issues a warning without stopping compilation.

The *#pragma* directive has no proper definition, it is compiler vendor specific. In the GNU C preprocessor, the #pragma is not used except for *#pragma once*. This was used to include a header file only once. But this directive is now obsolete.

Macros

Macros are used for assigning symbolic names to program constants, when these are repetitively used within a program. Using macros improves readability of a program and also makes it more portable.

#define

As we have seen, #define macro is the most commonly used macro. Its purpose is to define a name for a value or constant.

```
#define TRUE 1
#define MAX 100
```

Whenever the above names appear in a program they are substituted with their respective values by the preprocessor.

Note: #define statements can appear anywhere in the program, but they are generally grouped at the beginning of a program.

The #define macro makes a program extendable, meaning the value of the definition can be changed in just one place, and this gets reflected in all the places the value is used.

#define macros can include expressions, or identifiers that are previously declared in a macro. Macros can also contain parameters just like functions. Consider the following:

```
#define SQUARE(x) ((x) * (x))
#define HEIGHT 25
#define WIDTH 30
#define AREA (HEIGHT * WIDTH)
```

It is recommended to use parentheses for macros including mathematical expressions in order to prevent problems of orders of operations. For example, if we don't use parentheses in the above definition of the SQUARE macro, we will obtain a different result:

```
#define SQUARE(x) x * x
#define VALUE 10
```

If x = VALUE + 2, we will have SQUARE(x) = VALUE + 2 * VALUE + 2 = 10 + 20 + 2 = 32.

If we use parentheses, we will have SQUARE(x) = (VALUE + 2) * (VALUE + 2) = 144.

You can also include a function within a macro definition by using the line continuation character namely \, and surrounding the statements with curly braces. If you forget and omit the braces, then only the first statement will be included in the macro.

Macro definition may even include arguments, just like a function. The arguments are enclosed in parenthesis. The left parenthesis must immediately follow the macro name.Macros are used instead of functions to eliminate the time delay associated with function calls. But too many macros instead of function calls will increase the size of the compiled program. So these should

be used in a program that requires a single function called repetitively. Also when you use a macro in a program, the compiler checks for the number of arguments but not the type, so there is potential for errors. The most important point to note is that you can have a function pointer, but not a macro pointer.

Predefined macros

The following is a list of predefined macros:

__LINE__ is an integer representing the current line number in the program.

__FILE__ is a string representing the current filename being compiled.

__DATE__ is a string representing the current date in "MMM DD YYYY" format.

__TIME__ is a string representing the current time in "HH:MM:SS" format.

__STDC__ is equal to 1 if the compiler is compliant with ANSI C standard, 0 if not.

Source : http://www.peoi.org/Courses/Coursesen/cprog/frame14.html