Chapter: 8

C Preprocessors and Advanced topics

#define, #include, #erroe, #undef, #line, #pragma, built-in macros, #,## operators, function pointers

8.1 Advanced #define and #include

We use ***#define*** to define a ***macro name*** which will be substituted by the character sequence associated with that ***macro***.

❒ ***Create function like macros using #define :*** We can use #define to create function like macros. In a function-like macro, arguments can be passed to the macro when it is expanded by the preprocessor. For example, consider this program:

```
#include<stdio.h>
#define SUM(i, j) i+j  /* when sum(i, j) appears in program it wll replaced by the operation "i+j" */
      int main(void){     int sum;
                          sum = SUM(10, 20);
                          printf("%d", sum);
                          return 0; }
```

Here the line ***sum = SUM(10, 20);*** is transformed into ***sum = 10+20;*** by the ***preprocessor***. As you can see, the values ***10*** and ***20*** are ***automatically substituted*** for the parameters ***i*** and ***j***.

Again observe how a "***function-like macro : RANGE***" performing several range check a well as controls the ***Do-While loop*** as follows:

```
#define RANGE(i, min, max) (i<min)||(i>max) ?  1:0
                    . . . . .
                    . . . . .
/* Forced to find a random number between 1 & 100 */
do {   r = rand();   } while(RANGE(r, 1, 100));
```

☞ The advantage to using ***function-like macros*** instead of functions is that ***in-line*** code is generated by the macro, thus ***avoiding the time it takes to call and return from a function***.

➲ Only relatively simple operations can be made into function-like macros.

➲ Also, because code is duplicated, the resulting program might be longer than it would be if a function were used.

❒ ***Usage of #include :*** The #include directive has these two general forms: ***#include <filename>*** ***#include "filename"***

☞ ***#include <filename> (used for search standered Header file comes with compiler) :*** If you specify the file name between ***angle brackets***, you are instructing the ***compiler*** to search for the file in some ***implementation defined manner***. For most compilers, this means ***searching a special directory devoted to the standard header files***. This is why the sample programs have been using this form to include the header files required by the standard library functions.

☞ ***#include "filename" (used for search user-defined HEADER file) :*** If you enclose the file name between ***quotation marks***, the ***compiler*** searches for the file in ***another implementation-defined manner***. If that search ***fails***, the search is ***restarted*** as ***if*** you ***had specified*** the file name between ***angle brackets***. For the majority of compilers, enclosing the name between quotation marks causes ***the current working directory to be searched first***. Typically, you will use quotation marks to include ***header files that you create***.

NOTE

[1] ***#include "stdio.h"*** uses ***quotes*** in the ***#include*** directive. While not as efficient as using the angle brackets, the ***#include*** statement will still find and include the ***STDIO.H*** header file.

[2] It is ***permissible*** to use ***both forms*** of the ***#include*** directive in the same program. For example,

```
#include <stdio.h>
#include "stdlib.h"
```

8.2 Conditional COMPILATlON

The C preprocessor includes several directives that allow ***parts of the source code of a program to be selectively compiled***. This is called ***conditional compilatlon***. These directives are

#if ***#elif*** ***#ifdef***
#else ***#endif*** ***#ifndef***

#if : The general form of **#if** is shown here:

```
#if constant-expression
      statement-sequence
#endif
```

If the value of the ***constant-expression*** is ***true*** the statement or statements between ***#if*** and ***#endif*** are compiled. If the ***constant-expression*** is ***false***, the compiler ***skips*** the statement or statements.

☛ The ***preprocessing*** stage is the ***first stage*** of ***compilation***,

☛ So the ***constant-expression*** means exactly that. No variables may be used.

#else : The ***#else*** can be used to form an ***alternative*** to the ***#if***. Its general form is shown here:

```
#if constant-expression
statement-sequence
#else
statement-sequence
#endif
```

- Notice that there is only one ***#endif***. The ***#else*** automatically terminates the ***#if*** block of statements.
- If the ***constant-expression*** is ***false***, the statement or statements associated with the ***#else*** are compiled.

#elif: You can create an if-else-if ladder using the ***#elif*** directive, as shown here:

```
#if constant-expression-1
statement-sequence
#elif constant-expression-2
statement-sequence
#elif constant-expression-3
statement-sequence
#endif
```

- As soon as the first expression is true, the lines of code associated with that expression are compiled, and the rest of the code is, skipped.

#ifdef: Another approach to conditional compilation is the ***#ifdef*** directive. It is used to check if a ***macro*** has defined. It has this general form:

```
#ifdef macro-name
statement-sequence
#endif
```

- If the ***macro-name*** is currently defined, then the ***statement-sequence*** associated with the ***#ifdef*** directive will be compiled. Otherwise, it is skipped.
- The ***#else*** may also be used with ***#ifdef*** to provide an ***alternative***.
- In addition to ***#ifdef***, there is a second way to determine if a ***macro name*** is defined. You can use the ***#if*** directive in ***conjunction*** with the ***defined*** compile-time operator. The defined operator has this general form:

```
#if  defined macro-name
```

  - If ***macro-name*** is defined, then the outcome is ***true***. Otherwise, it is ***false***. For example, the following two preprocessor directives are equivalent:

```
#ifdef WIN32
#if defined WIN32   /* defined is a compile-time operator */
```

  - In this case you can also apply the **!** operator to defined to ***reverse*** the ***condition***.

#ifndef: The complement of ***#ifdef*** is ***#ifndef***. It has the same general form as ***#ifdef***. The only difference is that the statement sequence associated with an ***#ifndef*** directive is compiled only ***if the macro-name is not defined***. (alternative to ***!defined***).

8.3 #error, #undef, #Iine, #pragma

C's preprocessor supports ***four*** special-use directives:

#error ***#undef*** ***#line*** ***#pragma***

#error: The ***#error*** directive has this general form:

```
#error error-message
```

It causes the ***compiler*** to ***stop compilation*** and issue the ***error-message*** along with other implementation-specific information, which will generally include the *number of the line* the ***#error directive*** is in and the *name of the file*.

- Note that the error-message is not enclosed between quotes.
- The principal use of the #error directive is in debugging.

#undef: The ***#undef*** directive ***undefines*** a ***macro*** name. Its general form is

```
#undef macro-name
```

- If the ***macro***-name is currently undefined, ***#undef*** has no effect.
- The principal use for ***#undef*** is to localize ***macro*** names.

#line: When a C compiler compiles a ***source file***, it maintains ***two*** pieces of ***information***: *the number of the line currently being compiled* and the *name of the source file currently being compiled*. The ***#line*** directive is ***used to change these values***. Its general form is

```
#line line_num "filename"
```

Here, ***line_num*** becomes the ***number*** of the *next line of source code*, and ***filename*** becomes the ***name*** the compiler will associate with the ***source file***.

- The value of ***line_num*** must be between ***1*** and ***32,767***.
- The ***filename*** may be a string consisting of any valid file name.
- The principal use for ***#line*** is for ***debugging*** and for ***managing large projects***.

#pragma: The ***#pragma*** directive allows a ***compiler's implementor*** *to define other preprocessing instructions to be given to the compiler*. It has this general form:

```
#pragma instructions
```

- ☛ If a compiler encounters a ***#pragma*** statement that it does not recognize, it ignores it.
- ☛ Whether your compiler supports any ***#pragmas*** depends on how your compiler was implemented.

8.4 C's built-in MACROS

ANSI C standard has at least ***five*** predefined ***macro*** names. They are

```
__LINE __          __ TIME __
__FILE __          __ STDC __
__ DATE __
```

- ➢ The ***__LINE__*** macro defines an integer value that is *equivalent to the line number of the source line* ***currently*** being ***compiled***.
- ➢ The ***__FILE__*** macro defines a string that is the *name of the file* ***currently*** being ***compiled***,
- ➢ The ***__DATE _*** macro defines a string that holds the ***current system date***, The string has this general form:

```
month/day/year
```

- ➢ The ***__TIME__*** macro defines a string that contains the ***time the compilation of a program began***. The string has this general form:

```
hours:minutes:seconds
```

- ➢ The ***__ STDC__*** macro is defined as the value 1 if the *compiler conforms to the ANSI standard*.

8.5 The # and ## operators

The C preprocessor contains two little-used but potentially valuable operators: **#** and **##**.

- ☛ The **#** operator turns the argument of a ***function-like macro*** (recall 8.1) into a ***quoted string***.

```
#include <stdio.h>
#define MKSTRING(str) # str
int main(void){     int value;
                    value = 10;
                    printf("%s is %d", MKSTRING(value), value);
                    return 0;}
```

It displays ***value is 10***. This output occurs because ***MKSTRING()*** causes the identifier "***value***" to be made into a ***quoted string***.

- ☛ The **##** operator ***concatenates*** two ***identifiers***.

Following creates the ***output() macro***, which translates into a call to ***printf().*** The value of two variables, which end in ***1*** or ***2***, is displayed.

```
#include<stdio.h>
#define output(i) printf("%d %d\n", i##1, i##2)
      /* two variables, which end in 1 or 2, eg. a1, a2 */
int main (void) {   int count1, count2;
                    int i1, i2;
                    count1= 10;  count2 = 20;
                    i1 = 99;     i2 = -10;
                    output(count);
                    output (i) ;
                 return 0;}
```

The program displays ***10 20 99 -10***. In the calls to ***output()***, ***count*** and ***i*** are ***concatenated*** with ***1*** and ***2*** to form the variable names ***count1, count2, i1*** and ***i2*** in the ***printf()*** statements.

8.6 DYNAMIC ALLOCATION

Dynamic allocation is the process by which memory is allocated as needed during runtime. This allocated memory can be used for a variety of purposes. Most commonly, memory is allocated by applications that need to take full advantage of all the memory in the computer.

For example, a word processor will want to let the user edit documents that are as large as possible. However, if the word processor uses a normal character array, it *must fix its size at compile time*. Thus, it would have to be compiled to run in computers with the minimum amount memory ***not allowing users with more memory to edit larger documents***.

If memory is allocated dynamically (as needed until memory is exhausted), however, any user may make full use of the memory in the system. Other uses for dynamic allocation include linked lists and binary trees.

The core of C's ***dynamic-allocation*** functions are ***malloc()***, which allocates memory. And ***free()***, which releases previously allocated memory. Both functions use the header file ***STDLIB.H***. Their ***prototypes*** are

```
void *malloc(size_t numbytes);
void free(void *ptr);
```

Here, ***numbytes*** is the number of bytes of memory you wish to allocate.

- ❒ The ***malloc()*** function returns a ***pointer*** to the start of the allocated piece of memory. If ***malloc()*** cannot fulfill the memory request—for example, there may be insufficient memory available—it returns a ***null pointer***.
- ❒ To free memory, call ***free()*** with a pointer to the start the block of memory (previously allocated using ***malloc()***) you wanna free.

NOTE

[1] Memory is allocated from a region called the ***heap***. Although the actual physical layout of memory may differ, conceptually the ***heap*** lies between your program and the ***stack***. Since this is a finite area, an allocation request c an fail when memory is exhausted.

[2] When a program terminates, all allocated memory is automatically released.