Hour 24 - What You Can Do Now

It's not what you know, but what you can.

—A. Alekhine

Congratulations! You're now in the last chapter of this book. You just need to spend one more hour to complete your 24-hour journey. In this lesson you'll learn more about the C language from the following topics:

- Programming style
- Modular programming
- Debugging

Also, a brief review of what you've learned from this book is included in this lesson. Before we start to cover these topics, let's have a look at the last example in this book.

Creating a Linked List

In this section, I'm going to build functions that can create a linked list, and add items to or delete items from that linked list. I save those functions into a source file (that is, a module; refer to the section "Modular Programming" in this lesson). In addition, I will set up an interface between the module file and the user. In other words, the user can call one of the functions saved in the module via the interface. The interface is invoked in the main() function that is saved in another source file. I will put data declarations and function prototypes in a separate header file.

A linked list is a chain of nodes (or elements). Each node consists of data items and a pointer that points to the next node in the list. A linked list with N nodes is shown in Figure 24.1.

As you can see from Figure 24.1, the first node in the list is pointed to by another pointer that is a start pointer for the list. The pointer in the last (Nth) node is a null pointer.

Figure 24.1. *A linked list with N nodes.*

The linked list I'm going to build is a very simple one, in which each element contains only two items: student name and ID number. Listing 24.1 contains the module program, which is saved in the source file named 24L01.c.

TYPE
Listing 24.1. Putting cohesive functions in the module program.

```
1:   /* 24L01.c: A module file */
2:   #include "24L02.h"
3:
4:   static NODE *head_ptr = NULL;
5:
6:   /**
7:    ** main_interface()
8:    **/
9:   void main_interface(int ch)
10:  {
11:     switch (ch){
12:       case `a':
13:          list_node_add();
14:          break;
15:       case `d':
16:          if (!list_node_delete())
17:             list_node_print();
18:          break;
19:       case `p':
20:          list_node_print();
21:          break;
22:       default:
23:          break;
24:     }
25:  }
26:  /**
27:   ** list_node_create()
28:   **/
29:  NODE *list_node_create(void)
30:  {
31:     NODE *ptr;
32:
33:     if ((ptr=(NODE *)malloc(sizeof(NODE))) == NULL)
34:        ErrorExit("malloc() failed.\n");
35:
36:     ptr->next_ptr = NULL;  /* set the next pointer to NULL */
37:     ptr->id = 0; /* initialization */
38:     return ptr;
39:  }
40:
41:  /**
42:   ** list_node_add()
43:   **/
44:  void list_node_add(void)
```

```
45:  {
46:     NODE *new_ptr, *ptr;
47:
48:     new_ptr = list_node_create();
49:     printf("Enter the student name and ID: ");
50:     scanf("%s%ld", new_ptr->name, &new_ptr->id);
51:
52:     if (head_ptr == NULL){
53:        head_ptr = new_ptr;
54:     } else {
55:        /* find the last node in the list */
56:        for (ptr=head_ptr;
57:             ptr->next_ptr != NULL;
58:             ptr=ptr->next_ptr)
59:           ; /* doing nothing here */
60:        /* link to the last node */
61:        ptr->next_ptr = new_ptr;
62:     }
63:  }
64:  /**
65:   ** list_node_delete()
66:   **/
67:  int list_node_delete(void)
68:  {
69:     NODE *ptr, *ptr_saved;
70:     unsigned long id;
71:     int deleted = 0;
72:     int reval = 0;
73:
74:     if (head_ptr == NULL){
75:        printf("Sorry, nothing to delete.\n");
76:        reval = 1;
77:     } else {
78:        printf("Enter the student ID: ");
79:        scanf("%ld", &id);
80:
81:        if (head_ptr->id == id){
82:           ptr_saved = head_ptr->next_ptr;
83:           free(head_ptr);
84:           head_ptr = ptr_saved;
85:           if (head_ptr == NULL){
86:              printf("All nodes have been deleted.\n");
87:              reval = 1;
88:           }
89:        } else {
90:           for (ptr=head_ptr;
91:                ptr->next_ptr != NULL;
92:                ptr=ptr->next_ptr){
93:              if (ptr->next_ptr->id == id){
94:                 ptr_saved = ptr->next_ptr->next_ptr;
95:                 free(ptr->next_ptr);
96:                 ptr->next_ptr = ptr_saved;
97:                 deleted = 1;
98:                 break;
99:              }
100:          }
101:          if (!deleted){
102:             printf("Can not find the student ID.\n");
103:          }
104:       }
105:    }
106:    return reval;
107: }
108: /**
109:  ** list_node_print()
110:  **/
111: void list_node_print(void)
112: {
113:    NODE *ptr;
114:
115:    if (head_ptr == NULL){
116:       printf("Nothing to display.\n");
117:    } else {
118:       printf("The content of the linked list:\n");
119:       for (ptr = head_ptr;
120:            ptr->next_ptr != NULL;
121:            ptr = ptr->next_ptr){
122:          printf("%s:%d -> ",
123:             ptr->name,
124:             ptr->id);
125:       }
126:       printf("%s:%d ->|",
127:          ptr->name,
128:          ptr->id);
129:       printf("\n");
130:    }
131: }
132: /**
133:  ** list_node_free()
134:  **/
135: void list_node_free()
136: {
137:    NODE *ptr, *ptr_saved;
138:
139:    for (ptr=head_ptr; ptr != NULL; ){
140:       ptr_saved = ptr->next_ptr;
141:       free(ptr);
```

```
142:       ptr = ptr_saved;
143:    }
144:    free(ptr);
145: }
146: /**
147:  ** ErrorExit()
148:  **/
149: void ErrorExit(char *str)
150: {
151:    printf("%s\n", str);
152:    exit(ERR_FLAG);
153: }
```

ANALYSIS
There is no direct output from the module program in Listing 24.1.

The purpose of the program in Listing 24.1 is to provide a module program that contains all cohesive functions for linked list creation, node addition, and node reduction. Figure 24.2 demonstrates the tasks performed by functions, such as list_node_create(), list_node_add(), and list_node_delete(), from the program.

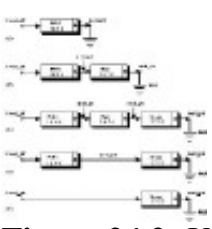

Figure 24.2. *Use functions defined in 24L01.c (refer to the following paragraphs for explanation).*

As you can see in Figure 24.2 (a), the first linked list node is created by calling the list_node_create() function, and the data items are added with the help of the list_node_add() function. Also, the node is pointed to by the head_ptr pointer. Here Peter is the student name; 1234 is his ID number. Because there are no more nodes linked, the next_ptr pointer of the first node is set to be null.

In Figure 24.2 (b), another node is added to the linked list, with Paul as the student name and 5678 as the ID number. Note that the next_ptr pointer of the first node is reset to point to the second node, while the next_ptr pointer of the second node is set to be null.

Likewise, in Figure 24.2 (c) , the third node is added to the linked list. The next_ptr pointer of the third node is a null pointer. The pointer in the second node is reset to point to the third node.

If I want to delete one of the nodes, I can call the list_node_delete() function. As shown in Figure 24.2 (d) , the second node is deleted, so the pointer of the first node has to be reset to point to the former third node that contains the student name Mary and her ID number, 7777.

In Figure 24.2 (e), the first node is deleted by applying the list_node_delete() function again. There is only one node left in the linked list. The head_ptr pointer has to be reset to point to the last node.

The header file, 24L02.h, included in the module program 24L01.c, is shown in Listing 24.2. (The header file is also included by the driver program in Listing 24.3.)

TYPE
Listing 24.2. Putting data declarations and function prototypes into the header file.

```
1:  /* lnk_list.h: the header file */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:
5:  #ifndef LNK_LIST_H
6:  #define LNK_LIST_H
7:  #define ERR_FLAG  1
8:  #define MAX_LEN   16
9:
10: struct lnk_list_struct
11: {
12:    char name[MAX_LEN];
13:    unsigned long id;
14:    struct lnk_list_struct *next_ptr;
15: };
16:
17: typedef struct lnk_list_struct NODE;
18:
19: NODE *list_node_create(void);
20: void list_node_add(void);
21: int list_node_delete(void);
22: void list_node_print(void);
23: void list_node_free(void);
24: void ErrorExit(char *);
25: void main_interface(int);
26:
27: #endif /* for LNK_LIST_H */
```

ANALYSIS
There is no direct output from the program in Listing 24.2.

The purpose of the program in Listing 24.2 is to declare a structure with the tag name of lnk_list_struct in lines 10_15, and define a new variable name, of the structure NODE, in line 17.

The prototypes of the functions defined in the module program in Listing 24.1, such as list_node_create(), list_node_add(), and list_node_delete(), are listed in lines 19_25.

Note that the #ifndef and #endif preprocessor directives are used in lines 5 and 27. The declarations and definitions located between the two directives are compiled only if the macro name LNK_LIST_H has not been defined. Also, line 6 defines the macro name if it's not been

defined. It's a good idea to put the #ifndef and #endif directives in a header file so as to avoid cross-inclusions when the header file is included by more than one source file. In this case, the declarations and definitions in the 24L02.h header file will not be included more than one time.

The module program in Listing 24.3 provides an interface that the user can use to call the functions saved in the source file (24L01.c).

TYPE
Listing 24.3. Calling functions saved in the module file.

```
1:  /* 24L03.c:  The driver file */
2:  #include "24L02.h"    /* include header file */
3:
4:  main(void)
5:  {
6:    int ch;
7:
8:    printf("Enter a for adding, d for deleting,\n");
9:    printf("p for displaying, and q for exit:\n");
10:   while ((ch=getchar()) != `q'){
11:      main_interface(ch);   /* process input from the user */
12:   }
13:
14:   list_node_free();
15:   printf("\nBye!\n");
16:
17:   return 0;
18: }
```

OUTPUT
I compile the source files, 24L01.c and 24L03.c, separately with Microsoft Visual C++, and then link their object files and C library functions together to produce a single executable program called 24L03.exe. I have the following output shown on the screen after I run the executable 24L03.exe, and enter or delete several student names and their ID numbers (the bold characters or numbers in the output section are what I entered from the keyboard):

```
C:\app>24L03
Enter a for adding, d for deleting,
p for displaying, and q for exit:
a
Enter the student name and ID: Peter 1234
a
Enter the student name and ID: Paul 5678
a
Enter the student name and ID: Mary 7777
p
The content of the linked list:
Peter:1234 -> Paul:5678 -> Mary:7777 ->|
d
Enter the student ID: 1234
The content of the linked list:
Paul:5678 -> Mary:7777 ->|
d
Enter the student ID: 5678
The content of the linked list:
Mary:7777 ->|
d
Enter the student ID: 7777
All nodes have been deleted.
q

Bye!
C:\app>
```

ANALYSIS
The purpose of the program in Listing 24.3 is to provide the user with an interface. The functions, such as list_node_create(), list_node_add(), and list_node_delete(), can be invoked through the interface. Also, the main() function is located inside the program of Listing 24.3.

The content of a linked list node can be printed out in the format of

```
name:id ->
```

The following is an example:

```
Peter:1234 -> Paul:5678 -> Mary:7777 ->|
```

Here the sign | is used to indicate that the pointer of the last node is a null pointer.

Figure 24.3 shows the relationship among the 24L01.c, 24L02.h, and 24L03.c files.

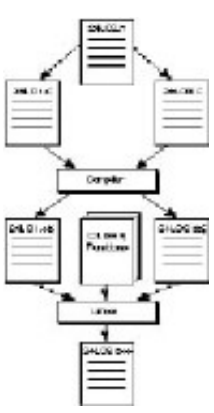

Figure 24.3. *The relationship among the 24L01.c, 24L02.h, and 24L03.c files.*

To learn to compile separate source files and link their object files together to make a single executable program, you need to check the technical reference from your C compiler vendor.

Programming Style

In this section, I'd like to briefly highlight some points that will help you write clean programs that can easily be read, understood, and maintained.

First, make sure the variable or function names in your program describe the meanings of the variables or tasks of the functions precisely and concisely.

Put comments into your code so that you or the other readers can have clues as to what your code is doing, or at least what the code intends to do, but might do incorrectly.

Whenever possible, keep using local variables, not global variables. Try to avoid sharing global data among functions; instead, pass the shared data as arguments to functions.

You should be careful when using C operators, especially the assignment operator (=) and the conditional operator (==), because misuse of these two operators can lead to an unexpected result and make the debugging very difficult.

Avoid using the goto statement; instead, use other control flow statements whenever needed.

Use named constants in your program, instead of numeric constants, because named constants can make your program more readable, and you will have to go to only one place to update the values of constants.

You should put parentheses around each constant expression or argument defined by a preprocessor directive to avoid side effects.

Also, you should set up a reasonable rule for spacing and indentation so that you can follow the rule consistently in all the programs you write. The rule should help make your programs easy to read.

Modular Programming

It's not a good programming practice to try to solve a complex problem with a single function. The proper way to approach it is to break the problem into several smaller and simpler pieces that can be understood in more details, and then start to define and build functions to solve those smaller and simpler problems. Keep in mind that each of your functions should do only one task, but do it well.

As your program continues to grow, you should consider breaking it into several source files, with each source file containing a small group of cohesive functions. Such source files are also called modules. Put data declarations and function prototypes into header files so that any changes to the declarations or prototypes can be automatically signified to all source files that include the header file.

For instance, in the section "Creating a Linked List," all functions that can be used to create a linked list and add or delete a node are put into the same module (24L01.c). Data structure and variable declarations, and function prototypes are saved into a header file (24L02.h). The main() function and the interface are saved into another module (24L03.c).

You can use a software-engineering technique known as information hiding to reduce the complexity of programming. Simply speaking, information hiding requires a module to withhold information from other modules unless it's necessary.

The C compiler enables you to compile and debug different source files separately. In this way, you can focus on one source file at a time, and complete the compiling before you move to the next one. With the separate compilation, you can compile only those source files that have been changed and leave the source files that have already been compiled

and debugged unattached.

If you're interested in knowing more about software engineering, study Ian Sommerville's classic book, Software Engineering, which is on the list of recommended books at the end of this lesson.

Debugging

I've mentioned debugging several times in this lesson. What is a bug, anyway?

A bug in this context refers to any erroneous behavior of a computer system or a software program. Debugging means finding bugs and fixing them. Please be aware that no computer system or software program is immune to bugs. Programmers, like you and I, make bugs, because we're human beings.

When you're debugging your program, learn how to isolate the erroneous behavior performed by your program. Many C compilers provide built-in debuggers that you can use. Also, there are quite a few debugging tools available from third-party software vendors.

As has been said, debugging requires patience, ingenuity, and experience. I recommend that you read a good book that will teach you all the techniques of debugging; in fact, I recommend one in the list of the books in the next section.

A Brief Review

The following subsections provide you with a brief review of the basics of the C language. The review is a summary that you will find useful to brush up on what you've learned in the previous hours.

C Keywords

In C, certain words have been reserved. These reserved words, called C keywords, have special meaning in the C language. The following are the C keywords:

auto	int
break	long
case	register
char	return
const	short
continue	signed
default	sizeof
do	static
double	struct
else	switch
enum	typedef
extern	union
float	unsigned
for	void
goto	volatile
if	while

Operators

Operators can help you manipulate data. C provides you with a rich set of operators. Table 24.1 contains a list of the operators used in C.

Table 24.1. The operators in C.

Operator	Description
=	Assignment operator
+=	Addition assignment operator
-=	Subtraction assignment operator
*=	Multiplication assignment operator
/=	Division assignment operator
%=	Remainder assignment operator
-	Unary minus operator
++	Increment operator
--	Decrement operator
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater or equal to
<=	Less or equal to
sizeof	Size-of operator
&&	Logical AND operator
||	Logical OR operator
!	Logical negation operator
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise exclusive OR (XOR) operator
~	Bitwise complement operator
>>	Right shift operator
<<	Left shift operator
?:	Conditional operator

Constants

Constants are elements whose values in the program do not change. In C, there are several different types of constants.

Integer Constants
Integer constants are decimal numbers. You can suffix an integer constant with u or U to specify that the constant is of the unsigned data type. An integer constant suffixed with l or L is a long int constant.

An integer constant is prefixed with a 0 (zero) to indicate that the constant is in the octal format. If an integer constant is prefixed with 0X or 0x, the constant is a hexadecimal number.

Character Constants
A character constant is a character enclosed by single quotes. For instance, `C' is a character constant.

In C, there are several character constants that represent certain special characters. (See Table 24.2.)

Table 24.2. Special characters in C.

Character	Meaning
\a	Audible alert
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\"	Double quote
\'	Single quote
\0	Null
\\	Backslash
\N	Octal constant (Here N is an octal constant.)
\xN	Hexadecimal constant (Here N is a hexadecimal constant.)

Floating-Point Constants
Floating-point constants are decimal numbers that can be suffixed with f, F, l, or L to specify

float or long double. A floating-point constant without a suffix is of the double data type by default. For instance, the following statements declare and initialize a float variable (flt_num) and a double variable (db_num):

```
float  flt_num = 1234.56f;
double  db_num = 1234.56;
```

A floating-point can also be represented in scientific notation.

String Constants
A string constant is a sequence of characters enclosed by double quotes. For instance, "This is a string constant." is a string constant. Note that the double quotes are not part of the content of the string. Also, the C compiler automatically adds a null character (\0) at the end of a string constant to indicate the end of the string.

Data Types

The basic data types provided by the C language are char, int, float, and double. In addition, there are array, enum, struct, and union data types which you can declare and use in your C programs.

The general form to define a list of variables with a specified data type is

```
data_type   variable_name_list;
```

Here data_type can be one of the keywords of the data types. variable_name_list represents a list of variable names separated by commas.

The Array Data Type
An array is a collection of variables that are of the same data type. The following is the general form to declare an array:

```
data-type  array-name[array-size];
```

Here data-type is the type specifier that indicates the data type of the array elements. array-name is the name of the declared array. array-size defines how many elements the array can contain. Note that the brackets ([and]) are required in declaring an array. The pair of [and] is also called the array subscript operator.
In addition, C supports multidimensional arrays.

The enum Data Type
enum is a short name for enumerated. The enumerated data type is used to declare named integer constants. The general form of the enum data type declaration is

```
enum tag_name {enumeration_list} variable_list;
```

Here tag_name is the name of the enumeration. variable_list gives a list of variable names that are of the enum data type. Both tag_name and variable_list are optional. enumeration_list contains defined enumerated names that are used to represent integer constants. Names represented by variable_list or enumeration_list are separated by commas.

The struct Data Type
In C, a structure collects different data items in such a way that they can be referenced as a single unit. The general form to declare a structure is

```
struct struct_tag {
     data_type1 variable1;
     data_type2 variable2;
     data_type3 variable3;
     .
     .
     .
     };
```

Here struct is the keyword used in C to start a structure declaration. struct_tag is the tag name of the structure. variable1 , variable2, and variable3 are the members of the structure. Their data types are specified respectively by data_type1, data_type2, and data_type3. The declarations of the members have to be enclosed within the opening and closing braces ({ and }) in the structure declaration, and a semicolon

(;) has to be included at the end of the declaration.

The following is an example of a structure declaration:

```
struct automobile {
     int year;
     char model[8];
     int engine_power;
     float weight;
     };
```

Here struct is used to start a structure declaration. automobile is the tag name of the structure. In the example here, there are three types of variables: char, int, and float. The variables have their own names, such as year, model, engine_power, and weight. They are all members of the structure, and are declared within the braces ({ and }).

The union Data Type

A union is a block of memory that is used to hold data items of different types. In C, a union is similar to a structure, except that data items saved in the union are overlaid in order to share the same memory location. The syntax for declaring a union is similar to the syntax for a structure. The general form to declare a union is

```
union union_tag {
     data_type1 variable1;
     data_type2 variable2;
     data_type3 variable3;
     .
     .
     .
     };
```

Here union is the keyword used in C to start a union declaration. union_tag is the tag name of the union. variable1 , variable2, and variable3 are the members of the union. Their data types are specified respectively by data_type1, data_type2, and data_type3. The union declaration is ended with a semicolon (;).

The following is an example of a union declaration:

```
union automobile {
     int year;
     char model[8];
     int engine_power;
     float weight;
};
```

Here union specifies the union data type. automobile is the tag name of the union. The variables, such as year, model, engine_power, and weight, are the members of the union and are declared within the braces ({ and }).

Defining New Type Names with typedef

You can create your own names for data types with the help of the typedef keyword in C, and use those names as synonyms for the data types. For instance, you can declare TWO_BYTE as a synonym for the int data type:

```
typedef  int  TWO_BYTE;
```

Then, you can use TWO_BYTE to declare integer variables like this,

```
TWO_BYTE i, j;
```

which is equivalent to

```
int i, j;
```

Remember that a typedef definition must be made before the synonym made in the definition is used in any declarations in your program.

Expressions and Statements

An expression is a combination of constants or variables that is used to denote computations.

For instance,

```
(2 + 3) * 10
```

is an expression that adds 2 and 3 first, and then multiplies the sum by 10.

In the C language, a statement is a complete instruction, ended with a semicolon. In many cases, you can turn an expression into a statement by simply adding a semicolon at the end of the expression.

A null statement is represented by an isolated semicolon.

A group of statements can form a statement block that starts with an opening brace ({) and ends with a closing brace (}). The C compiler treats a statement block as a single statement.

Control Flow Statements

In C, there is a set of control flow statements that can be divided into two categories: looping and conditional branching.

The for, while, and do-while Loops

The general form of the for statement is

```
for (expression1; expression2; expression3) {
   statement1;
   statement2;
   .
   .
   .
}
```

The for statement first evaluates expression1, which is usually an expression that initializes one or more variables. The second expression, expression2, is the conditional part that is evaluated and tested by the for statement for each looping. If expression2 returns a nonzero value, the statements within the braces, such as statement1 and statement2, are executed. Usually, the nonzero value is 1 (one). If expression2 returns 0 (zero), the looping is stopped and the execution of the for statement is finished. The third expression in the for statement, expression3, is evaluated after each looping before the statement goes back to test expression2 again.

The following for statement makes an infinite loop:

```
for ( ; ; ){
  /* statement block */
}
```

The general form of the while statement is

```
while (expression) {
   statement1;
   statement2;
   .
   .
   .
}
```

Here expression is the field of the expression in the while statement. The expression is evaluated first. If it returns a nonzero value, the looping continues; that is, the statements inside the statement block, such as statement1 and statement2, are executed. After the execution, the expression is evaluated again. Then the statements are executed one more time if the expression still returns a nonzero value. The process is repeated over and over until the expression returns zero.

You can also make a while loop infinite by putting 1 (one) in the expression field like this:

```
while (1) {
   /* statement block */
}
```

The general form for the do-while statement is

```
do {
   statement1;
   statement2;
   .
   .
   .
} while (expression);
```

Here expression is the field for the expression that is evaluated in order to determine whether the statements inside the statement block are executed one more time. If the expression returns a nonzero value, the do-while loop continues; otherwise, the looping stops. Note that the do-while statement ends with a semicolon, which is an important distinction. The statements controlled by the do-while statement are executed at least once before the expression is evaluated. Note that a do-while loop ends with a semicolon (;).

Conditional Branching
The if, if-else, switch, break, continue, and goto statements fall into the conditional branching category.

The general form of the if statement is

```
if (expression) {
   statement1;
   statement2;
   .
   .
   .
}
```

Here expression is the conditional criterion. If expression is logical TRUE (that is, nonzero), the statements inside the braces ({ and }), such as statement1 and statement2, are executed. If expression is logical FALSE (0), the statements are skipped.

As an expansion of the if statement, the if-else statement has the following form:

```
if (expression) {
   statement1;
   statement2;
   .
   .
   .
}
else {
   statement_A;
   statement_B;
   .
   .
   .
}
```

Here if expression is logical TRUE, the statements controlled by if, including statement1 and statement2, are executed. Otherwise, the statements, such as statement_A and statement_B, inside the statement block following the else keyword are executed, if expression is

logical FALSE.

The general form of the switch statement is

```
switch (expression) {
   case expression1:
        statement1;
   case expression2:
        statement2;
   .
   .
   .
   default:
        statement-default;
}
```

Here the conditional expression, expression, is evaluated first. If the return value of expression is equal to the constant expression expression1, then execution begins at the statement statement1. If the value of expression is the same as the value of expression2, execution then begins at statement2. If, however, the value of expression is not equal to any values of the constant expressions labeled by the case keyword, the statement, statement-default, following the default keyword is executed.

You can add a break statement at the end of the statement list following each case label if you want to exit the switch construct after the statements within a selected case have been executed.

Also, the break statement can be used to break an infinite loop.

There are times when you want to stay in a loop but skip over some of the statements within the loop. To do this, you can use the continue statement provided by C.

The following gives the general form of the goto statement:

```
label-name:
   statement1;
   statement2;
   .
   .
   .
   goto  label-name;
```

Here label-name is a label name that tells the goto statement where to jump. You have to place label-name in two places: at the place where the goto statement is going to jump and at the place following the goto keyword. Also, the place for the goto statement to jump to can appear either before or after the statement. Note that a colon (:) must follow the label name at the place where the goto statement will jump to.

Pointers

A pointer is a variable whose value is used to point to another variable. The general form of a pointer declaration is

```
data-type  *pointer-name;
```

Here data-type specifies the type of data to which the pointer points. pointer-name is the name of the pointer variable, which can be any valid variable name in C. When the compiler sees the asterisk (*) prefixed to the variable name in the declaration, it makes a note in its symbol table so that the variable can be used as a pointer.

Usually, the address associated with a variable name is called the left value of the variable. When a variable is assigned with a value, the value is stored in the reserved memory location of the variable as the content. The content is also called the right value of the variable.

A pointer is said to be a null pointer when its right value is 0. Remember that a null pointer can never point to valid data.

The dereference operator (*) is a unary operator that requires only one operand. For instance, the *ptr_name expression returns the value pointed to by the pointer variable ptr_name, where ptr_name can be any valid variable name in C.

The & operator is called the address-of operator because it can return the address (that is, left value) of a variable.

Several pointers can point to the same location of a variable in the memory. In C, you can move the position of a pointer by adding or subtracting integers to or from the pointer.

Note that for pointers of different data types, the integers added to or subtracted from the pointers have different scalar sizes.

Pointing to Objects

You can access an element in an array by using a pointer. For instance, given an array, an_array, and a pointer, ptr_array, if an_array and ptr_array are of the same data type, and ptr_array is assigned with the start address of the array like this:

```
ptr_array = an_array;
```

the expression

```
an_array[n]
```

is equivalent to the expression

```
*(ptr_array + n)
```

Here n is a subscript number in the array.

In many cases, it's useful to declare an array of pointers and access the contents pointed to by the array through dereferencing each pointer. For instance, the following declaration declares an int array of pointers:

```
int *ptr_int[3];
```

In other words, the variable ptr_int is a three-element array of pointers with the int type.

Also, you can define a pointer of struct and refer to an item in the structure via the pointer. For example, given the following structure declaration:

```
struct computer {
   float cost;
   int year;
   int cpu_speed;
   char cpu_type[16];
};
```

a pointer can be defined like this:

```
struct computer *ptr_s;
```

Then, the items in the structure can be accessed by dereferencing the pointer. For instance, to assign the value of 1997 to the int variable year in the computer structure, you can have the following assignment statement:

```
(*ptr_s).year = 1997;
```

Or, you can use the arrow operator (->) for the assignment, like this:

```
ptr_s->year = 1997;
```

Note that the arrow operator (->) is commonly used to reference a structure member with a pointer.

Functions

Functions are the building blocks of C programs. Besides the standard C library functions, you can also use some other functions made by you or by another programmer in your C program. The opening brace ({) signifies the start of a function body, while (}), the closing brace, marks the end of the function body.

According to the ANSI standard, the declaration of a variable or function specifies the interpretation and attributes of a set of identifiers. The definition, on the other hand, requires the C compiler to reserve storage for a variable or function named by an identifier.

In fact, a variable declaration is a definition. But the same is not true for functions. A function declaration alludes to a function that is defined elsewhere, and specifies what kind of value returned by the function. A function definition defines what the function does, as well as the number and type of arguments passed to the function.

With the ANSI standard, the number and types of arguments passed to a function are allowed to be added into the function declaration. The number and types of argument are called the function prototype.

The general form of a function declaration, including its prototype, is as follows:

```
data_type_specifier  function_name(
      data_type_specifier argument_name1,
      data_type_specifier argument_name2,
      data_type_specifier argument_name3,
      .
      .
      .
      data_type_specifier argument_nameN,
);
```

Here data_type_specifier determines the type of the return value made by the function or specifies the data types of arguments, such as argument_name1, argument_name2, argument_name3, and argument_nameN, passed to the function with the name of function_name.

The purpose of using a function prototype is to help the compiler to check whether the data types of arguments passed to a function match what the function expects. The compiler issues an error message if the data types do not match. The void data type is needed in the declaration of a function that takes no argument.

To declare a function that takes a variable number of arguments, you have to specify at least the first argument and use the ellipsis (...) to represent the rest of the arguments passed to the function.

A function call is an expression that can be used as a single statement or within other expressions or statements.

It's more efficient to pass the address of an argument, instead of its copy, to a function so that the function can access and manipulate the original value of the argument. Therefore, it's a good idea to pass the name of a pointer, which points to an array, as an argument to a function, instead of the array elements themselves.

You can also call a function via a pointer that holds the address of the function.

Input and Output (I/O)

In C, a file refers to a disk file, a terminal, a printer, or a tape drive. In other words, a file represents a concrete device with which you want to exchange information. A stream, on the other hand, is a series of bytes through which you read or write data to a file. Unlike a file, a stream is device-independent. All streams have the same behavior.

In addition, there are three file streams that are pre-opened for you:

- stdin—The standard input for reading.
- stdout—The standard output for writing.
- stderr—The standard error for writing error message.

Usually, the standard input stdin links to the keyboard, while the standard output stdout and the standard error stderr point to the screen. Also, many operating systems allow you to redirect these file streams.

By default, all I/O streams are buffered. The buffered I/O is also called the high-level I/O.

The FILE structure is the file control structure defined in the header file stdio.h. A pointer of type FILE is called a file pointer and references a disk file. A file pointer is used by a stream to conduct the operation of the I/O functions. For instance, the following defines a file pointer called fptr:

```
FILE *fptr;
```

In the FILE structure, there is a member, called the file position indicator, that points to the position in a file where data will be read from or written to.

The C language provides a set of rich library functions to perform I/O operations. Those functions can read or write any types of data to files. Among them, fopen(), fclose(), fgetc(), fputc(), fgets(), fputs(), fread(), fwrite(), feof(), fscanf(), fprintf(), fseek(), ftell(), rewind(), and freopen() have been introduced in this book.

The C Preprocessor

The C preprocessor is not part of the C compiler. The C preprocessor runs before the compiler. During preprocessing, all occurrences of a macro name are replaced by the macro body that is associated with the macro name. Note that a macro statement ends with a newline character, not a semicolon.

The C preprocessor also enables you to include additional source files to the program or compile sections of C code conditionally.

The #define directive tells the preprocessor to replace every occurrence of a macro name defined by the directive with a macro body that is associated with the macro name. You can specify one or more arguments to a macro name defined by the #define directive.

The #undef directive is used to remove the definition of a macro name that has been previously defined.

The #ifdef directive controls whether a given group of statements is to be included as part of the program. The #ifndef directive is a mirror directive to the #ifdef directive; it enables you to define code that is to be included when a particular macro name is not defined.

The #if, #elif, and #else directives enable you to filter out portions of code to compile. #endif is used to mark the end of an #ifdef, #ifndef, or #if block because the statements under the control of these preprocessor directives are not enclosed in braces.

The Road Ahead…

I believe that you can start to run now after you learn to walk in the world of the C language through this book. Although you're on your own, you're not alone. You can revisit this book whenever you feel like it. Besides, the following books, which I recommend to you, can send you in the right direction to continue your journey in the C world:

> The C Programming Language
> by Brian Kernighan and Dennis Ritchie, published by Prentice Hall
>
> C Interfaces and Implementations
> by David Hanson, published by Addison-Wesley
>
> Practical C Programming
> by Steve Oualline, published by O'Reilly & Associates, Inc.
>
> No Bugs!—Delivering Error-Free Code in C and C++
> by David Thielen, published by Addison-Wesley
>
> Software Engineering
> by Ian Sommerville, published by Addison-Wesley
>
> The Mythical Man-Month: Essays on Software Engineering
> by F. P. Brooks, Jr., published by Addison-Wesley

Summary

Before you close this book, I'd like to thank you for your patience and the effort you have put into learning the basics of the C language in the past 24 hours. Now, it's your turn to apply what you've learned from this book to solving the problems in the real world. Good luck!

Previous | Table of Contents