Sams Teach Yourself C in 24 Hours

Hour 18 - More Data Types and Functions

That's all there is, there isn't any more.

—E. Barrymore

In Hour 4, "Data Types and Names in C," you learned about most of the data types, such as char, int, float, and double. In Hour 15, "Functions in C," you learned the basics of using functions in C. In this hour, you'll learn more about data types and functions from the following topics:

- The enum data type
- The typedef statement
- Function recursion
- Command-line arguments

The enum Data Type

The C language provides you with an additional data type—the enum data type. enum is short for enumerated. The enumerated data type can be used to declare named integer constants. The enum data type makes the C program more readable and easier to maintain. (Another way to declare a named constant is to use the #define directive, which is introduced later in this book.)

Declaring the enum Data Type

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. enumeration_list contains defined enumerated names that are used to represent integer constants. (Both tag_name and variable_list are optional.)

For instance, the following declares an enum data type with the tag name of automobile:

```
enum automobile {sedan, pick_up, sport_utility};
```

Given this, you can define enum variables like this:

```
enum automobile  domestic, foreign;
```

Here the two enum variables, domestic and foreign, are defined.

Of course, you can always declare and define a list of enum variables in a single statement, as shown in the general form of the enum declaration. Therefore, you can rewrite the enum declaration of domestic and foreign like this:

```
enum automobile {sedan, pick_up, sport_utility} domestic, foreign;
```

Assigning Values to enum Names

By default, the integer value associated with the leftmost name in the enumeration list field, surrounded by the braces ({ and }), starts with 0, and the value of each name in the rest of the list increases by one from left to right. Therefore, in the previous example, sedan, pick_up, and sport_utility have the values of 0, 1, and 2, respectively.

In fact, you can assign integer values to enum names. Considering the previous example, you can initialize the enumerated names like this:

```
enum automobile {sedan = 60, pick_up = 30, sport_utility = 10};
```

Now, sedan represents the value of 60, pick_up has the value of 30, and sport_utility assumes the value of 10.

The program shown in Listing 18.1 prints out the values of enum names.

TYPE

Listing 18.1. Defining enum data types.

```
1:  /* 18L01.c: Defining enum data types */
2:  #include <stdio.h>
3:  /* main() function */
4:  main()
5:  {
6:     enum language {human=100,
7:                    animal=50,
8:                    computer};
9:     enum days{SUN,
10:              MON,
11:              TUE,
12:              WED,
13:              THU,
14:              FRI,
15:              SAT};
```

```
16:
17:    printf("human: %d,  animal: %d,  computer: %d\n",
18:       human, animal, computer);
19:    printf("SUN: %d\n", SUN);
20:    printf("MON: %d\n", MON);
21:    printf("TUE: %d\n", TUE);
22:    printf("WED: %d\n", WED);
23:    printf("THU: %d\n", THU);
24:    printf("FRI: %d\n", FRI);
25:    printf("SAT: %d\n", SAT);
26:
27:    return 0;
28: }
```

The following output is shown on the screen after the executable, 18L01.exe, of the program in Listing 18.1 is created and executed:

OUTPUT

```
C:\app>18L01
human: 100,  animal: 50,  computer: 51
SUN: 0
MON: 1
TUE: 2
WED: 3
THU: 4
FRI: 5
SAT: 6
C:\app>
```

ANALYSIS

The purpose of the program in Listing 18.1 is to show you the default values of the enum names, as well as the values assigned to some enum names by the programmer.

As you can tell, there are two enum declarations, in lines 6_8 and lines 9_15, respectively. Note that the variable lists in the two enum declarations are omitted because there is no need for the variable lists in the program.

The first declaration has a tag name called language and three enumerated names, human, animal, and computer. In addition, human is assigned the value of 100; animal is initialized with 50. According to the enum definition, the default value of computer is the value of animal increased by 1. Therefore, in this case, the default value of computer is 51.

The output made by the statement in line 17 shows that the values of human, animal, and computer are indeed 100, 50, and 51.

The second enum declaration in the program contains seven items with their default values. Then, lines 19_25 print out these default values one at a time. It is not surprising to see that the values represented by the enumerated names, SUN, MON, TUE, WED, THU, FRI, and SAT, are 0, 1, 2, 3, 4, 5, and 6, respectively.

Now, let's look at another example, shown in Listing 18.2, that demonstrates how to use the enum data type.

TYPE

Listing 18.2. Using the enum data type.

```
1:  /* 18L02.c: Using the enum data type */
2:  #include <stdio.h>
3:  /* main() function */
4:  main()
5:  {
6:     enum units{penny = 1,
7:                nickel = 5,
8:                dime = 10,
9:                quarter = 25,
10:               dollar = 100};
11:    int money_units[5] = {
12:               dollar,
13:               quarter,
14:               dime,
15:               nickel,
16:               penny};
17:    char *unit_name[5] = {
18:              "dollar(s)",
19:              "quarter(s)",
20:              "dime(s)",
21:              "nickel(s)",
22:              "penny(s)"};
23:    int cent, tmp, i;
24:
25:    printf("Enter a monetary value in cents:\n");
26:    scanf("%d", &cent);  /* get input from the user */
27:    printf("Which is equivalent to:\n");
28:    tmp = 0;
29:    for (i=0; i<5; i++){
30:       tmp = cent / money_units[i];
31:       cent -= tmp * money_units[i];
32:       if (tmp)
33:         printf("%d %s ", tmp, unit_name[i]);
34:    }
35:    printf("\n");
36:    return 0;
37: }
```

While the executable (18L02.exe) is being executed, I enter 141 (for 141 cents) and obtain the following output from the screen:

OUTPUT

```
C:\app>18L02
Enter a monetary value in cents:
141
Which is equivalent to:
1 dollar(s) 1 quarter(s) 1 dime(s) 1 nickel(s) 1 penny(s)
C:\app>
```

ANALYSIS

The purpose of the program in Listing 18.2 is to use the enum data type to represent the value of the amount of money entered by the user.

Inside the main() function, an enum declaration with a tag name of units is made in lines 6_10. The numbers assigned to the enumerated names are based on their ratios to the unit of cent. For instance, one dollar is equal to 100 cents. Therefore, the enum name dollar is assigned the value of 100.

After the enum declaration, an int array, called money_units, is declared, and is initialized with the enumerated names from the enum declaration. According to the definition of the enum data type, the declaration of the money_units array in the program is actually equivalent to the following one:

```
int money_units[5] = {
              100,
              25,
              10,
              5,
              1};
```

So now you see that you can use enumerated names, instead of integer numbers, to make up other expressions or declarations in your program.

In lines 17_22, an array of pointers, unit_name, is declared and initialized. (The usage of arrays of pointers was introduced in Hour 16, "Applying Pointers.")

Then, the statement in line 15 asks the user to enter an integer number in the unit of cent. The scanf() function in line 26 stores the number entered by the user to an int variable called cent.

The for loop in lines 29_34 divides the entered number and represents it in a dollar-quarter-dime-nickel-penny format.

Note that the integer constants represented by the enumerated names are used in lines 30 and 31, through the money_units array. If the value of a unit is not 0, a corresponding string pointed to by the array of pointers, unit_name, is printed out in line 33. Therefore, when I enter 141 (in unit of cent), I see its equivalent in the output: 1 dollar(s) 1 quarter(s) 1 dime(s) 1 nickel(s) 1 penny(s).

Making typedef Definitions

You can create your own names for data types with the help of the typedef keyword in C, and make those name synonyms for the data types. Then, you can use the name synonyms, instead of the data types themselves, in your programs. Often, the name synonyms defined by typedef can make your program more readable.

For instance, you can declare TWO_BYTE as a synonym for the int data type:

```
typedef  int  TWO_BYTE;
```

Then, you can start to 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 created in the definition is used in any declarations in your program.

Why Use typedef?

There are several advantages to using typedef definitions. First, you can consolidate complex data types into a single word and then use the word in variable declarations in your program. In this way, you don't need to type a complex declaration over and over, which helps to avoid typing errors.

The second advantage is that you just need to update a typedef definition, which fixes every use of that typedef definition if the data type is changed in the future.

typedef is so useful, in fact, that there is a header file called stddef.h included in the ANSI-standard C that contains a dozen typedef definitions. For instance, size_t is a typedef for the value returned by the sizeof operator.

The program shown in Listing 18.3 is an example of using typedef definitions.

TYPE

Listing 18.3. Using typedef definitions.

```
1:  /* 18L03.c: Using typedef definitions */
2:  #include <stdio.h>
3:  #include <stdlib.h>
4:  #include <string.h>
5:
6:  enum constants{ITEM_NUM = 3,
7:                 DELT='a'-'A'};
8:  typedef char *STRING[ITEM_NUM];
9:  typedef char *PTR_STR;
10: typedef char BIT8;
11: typedef int BIT16;
12:
13: void Convert2Upper(PTR_STR str1, PTR_STR str2);
14:
15: main()
16: {
17:    STRING str;
18:    STRING moon = {"Whatever we wear",
19:                   "we become beautiful",
20:                   "moon viewing!"};
21:    BIT16 i;
22:    BIT16 term = 0;
23:
24:    for (i=0; i<ITEM_NUM; i++){
25:      str[i] = malloc((strlen(moon[i])+1) * sizeof(BIT8));
26:      if (str[i] == NULL){
27:        printf("malloc() failed.\n");
28:        term = 1;
29:        i = ITEM_NUM;  /* break the for loop */
30:      }
31:      Convert2Upper(moon[i], str[i]);
32:      printf("%s\n", moon[i]);
33:    }
34:    for (i=0; i<ITEM_NUM; i++){
35:      printf("\n%s", str[i]);
36:      free (str[i]);
37:    }
38:
39:    return term;
40: }
41: /* function definition */
42: void Convert2Upper(PTR_STR str1, PTR_STR str2)
43: {
44:    BIT16 i;
45:
46:    for (i=0; str1[i]; i++){
47:      if ((str1[i] >= `a') &&
48:          (str1[i] <= `z'))
49:        str2[i] = str1[i] - DELT;
50:      else
51:        str2[i] = str1[i];
52:    }
53:    str2[i] = `\0';  /* add null character */
54: }
```

I have the following output displayed on the screen after running the executable, 18L03.exe, of the program in Listing 18.3:

OUTPUT

```
C:\app>18L03
Whatever we wear
we become beautiful
moon viewing!

WHATEVER WE WEAR
WE BECOME BEAUTIFUL
MOON VIEWING!
C:\app>
```

ANALYSIS

The purpose of the program in Listing 18.3 is to show you how to create your own names for data types such as char and int. The program in Listing 18.3 converts all characters in a Japanese haiku into their uppercase counterparts.

In lines 3 and 4, two more header files, stdlib.h and string.h, are included because the malloc() and strlen() functions are invoked later in the program.

An enum declaration is made in lines 6 and 7 with two enumerated names, ITEM_NUM and DELT. In addition, ITEM_NUM is assigned the value of 3 because there are three strings in the haiku. DELT contains the value of the difference between a lowercase character and its uppercase counterpart in the ASCII code. In line 7, the values of `a' and `A' are used to calculate the difference.

In lines 8_11, I define names, STRING, PTR_STR, BIT8, and BIT16, for a char array of pointers with three elements, a char pointer, a char, and an int data type, respectively, so that I can use these names as synonyms to these data types in the program.

For instance, the prototype of the Convert2Upper() function in line 13 contains two arguments that are all char pointers declared with PTR_STR.

In lines 17_20, two arrays of pointers, str and moon, are declared with STRING. moon is initialized to point to the strings of the Japanese haiku. In lines 21 and 22, two int variables, i and term, are declared with BIT16.

The for loop in lines 24_33 allocates enough memory space dynamically based on the size of the haiku. The Conver2Upper() function is then called in line 31 to copy strings referenced by moon to the memory locations pointed to by str and to convert all lowercase characters to

their uppercase counterparts as well. Line 32 prints out the strings referenced by moon. The definition of the Conver2Upper() function is shown in lines 42_54.

In lines 34_37, another for loop is made to print out the content from the memory locations referenced by str. There are a total of three strings with uppercase characters in the content. After a string is displayed on the screen, the memory space allocated for the string is released by calling the free() function.

On the screen, you see two copies of the haiku—the original one and the one with all-uppercase characters.

Recursive Functions

You already know that in C a function can be called by another function. But can a function call itself? The answer is yes. A function can call itself from a statement inside the body of the function itself. Such a function is said to be recursive.

Listing 18.4 contains an example of calling a recursive function to add integers from 1 to 100.

TYPE

Listing 18.4. Calling a recursive function.

```
1:  /* 18L04.c: Calling a recursive function */
2:  #include <stdio.h>
3:
4:  enum con{MIN_NUM = 0,
5:           MAX_NUM = 100};
6:
7:  int fRecur(int n);
8:
9:  main()
10: {
11:    int i, sum1, sum2;
12:
13:    sum1 = sum2 = 0;
14:    for (i=1; i<=MAX_NUM; i++)
15:      sum1 += i;
16:    printf("The value of sum1 is %d.\n", sum1);
17:    sum2 = fRecur(MAX_NUM);
18:    printf("The value returned by fRecur() is %d.\n", sum2);
19:
20:    return 0;
21: }
22: /* function definition */
23: int fRecur(int n)
24: {
25:    if (n == MIN_NUM)
26:      return 0;
27:    return  fRecur(n - 1) + n;
28: }
```

After the executable 18L04.exe is created and executed, the following output is displayed on the screen:

OUTPUT

```
C:\app>18L04
The value of sum1 is 5050.
The value returned by fRecur() is 5050.
C:\app>
```

ANALYSIS

In the program in Listing 18.4, a recursive function, fRecur(), is declared in line 7 and defined in lines 23_28.

You can see from the definition of the fRecur() function that the recursion is stopped in line 26 if the incoming int variable, n, is equal to the value contained by the enum name MIN_NUM. Otherwise, the fRecur() function is called by itself over and over in line 27. Note that each time the fRecur() function is called, the integer argument passed to the function is decreased by one.

Now, let's have a look at the main() function of the program. The for loop, shown in lines 14 and 15, adds integers from 1 to the value represented by another enum name, MAX_NUM. In lines 4 and 5, MIN_NUM and MAX_NUM are respectively assigned 0 and 100 in an enum declaration. The printf() function in line 16 then prints out the sum of the addition made by the for loop.

In line 17, the recursive function, fRecur(), is called and passed with an integer argument starting at the value of MAX_NUM. The value returned by the fRecur() function is then assigned to an int variable, sum2.

Eventually, the value saved by sum2 is printed out in line 18. From the output, you can see that the execution of the recursive function fRecur() actually produces the same result as the for loop inside the main() function.

NOTE

> Recursive functions are useful in making clearer and simpler implementations of algorithms. On the other hand, however, recursive functions may run slower than their iterative equivalents due to the overhead of repeated function calls. Function arguments and local variables of a program are stored temporarily in a block of memory called the stack. Each call to a recursive function makes a new copy of the arguments and local variables. The new copy is then put on the stack. If you see your recursive function behaving strangely, it's probably overwriting other data stored on the stack.

Revisiting the main() Function

As you've learned, each C program should have one and only one main() function. The execution of a program starts and ends at its main() function.

As with other functions in C, you can pass arguments to a main() function. So far, I've been using the void keyword in the definition of the main() function to indicate that there are no arguments passed to the function. Now, the question is how to do it if you want to pass information to the main() function.

Command-Line Arguments

Because each C program starts at its main() function, information is usually passed to the main() function via command-line arguments.

A command-line argument is a parameter that follows a program's name when the program is invoked from the operating system's command line. For instance, given a C program, test.c, whose executable file is called test.exe, if you run the program from a DOS prompt like this,

```
C:\app>test.exe argument1 argument2 argument3
```

argument1, argument2, and argument3 are called command-line arguments to the main() function in the test.c program.

Of course, you can simply omit an executable file's extension, .exe, when you run it from a DOS prompt.

The next subsection teaches you how to receive command-line arguments.

Receiving Command-Line Arguments

There are two built-in arguments in the main() function that can be used to receive command-line arguments. Usually, the name of the first argument is argc, and it is used to store the number of arguments on the command line. The second argument is called argv and is a pointer to an array of char pointers. Each element in the array of pointers points to a command-line argument that is treated as a string.

In order to use argc and argv, you have to declare them in the main() function in your program like this:

```
data_type_specifier main(int argc, char *argv[])
{
   . . .
}
```

Here data_type_specifier specifies the data type returned by the main() function. By default, the data type returned by the main() function is int. If the main() does not return any value, you should put the void keyword in front of the main() function definition.

Let's continue to use the example shown in the last section. Suppose that the main() function defined in the test.c program looks like this:

```
void main(int argc, char *argv[])
{
   . . .
}
```

If you run the executable file of the program from a DOS prompt,

```
C:\app>test.exe argument1 argument2 argument3
```

the value received by argc is 4, because the name of the program itself is counted as the first command-line argument. Accordingly, argv[0] holds the string of the path and program name C:\app\test.exe, and argv[1], argv[2], and argv[3] contain the strings of argument1, argument2, and argument3, respectively. (Note that C:\app\ is the path to the executable file on my machine.)

The program in Listing 18.5 is another example of passing command-line arguments to the main() function.

TYPE

Listing 18.5. Passing command-line arguments to the main() function.

```
1:  /* 18L05.c: Command-line arguments */
2:  #include <stdio.h>
3:
4:  main (int argc, char *argv[])
5:  {
6:     int i;
7:
8:     printf("The value received by argc is %d.\n", argc);
9:     printf("There are %d command-line arguments passed to main().\n",
10:           argc);
11:
12:    printf("The first command-line argument is: %s\n", argv[0]);
13:    printf("The rest of the command-line arguments are:\n");
14:    for (i=1; i<argc; i++)
15:       printf("%s\n", argv[i]);
16:
17:    return 0;
18: }
```

After the executable, 18L05.exe, is executed and passed with several command-line arguments, the following output is displayed on the screen:

OUTPUT

```
C:\app>18L05 Hello, world!
The value received by argc is 3.
There are 3 command-line arguments passed to main().
The first command-line argument is: C:\app\18L05.EXE
The rest of the command-line arguments are:
Hello,
world!
C:\app>
```

ANALYSIS

The purpose of the program in Listing 18.5 is to show you how to check the number of command-line arguments and print out the strings that hold the arguments entered by the user.

Note that there are two arguments, argc and argv, that are declared in line 4 for the main() function. Then, the statements in lines 8 and 9 print out the value of the total number of arguments held by argc. If there is no command-line argument entered by the user, argc contains the default value of 1 because the name of the program itself is counted as the first argument.

Line 12 prints out the first string saved in the memory location pointed to by argv[0]. As you can see from the output, the content of the first string is the executable file name of the program in Listing 18.5, plus the path to the executable file.

The for loop in lines 14 and 15 displays the rest of the strings that contain the command-line arguments entered by the user. In this example, I enter two command-line argument strings, "Hello," and "world!", which are shown back on the screen after the execution of the for loop.

NOTE

argc and argv are normally used as the two built-in arguments in the main() function, but you can use other names to replace them in their declarations.
In addition to these two arguments, some compilers may support another argument to the main() function. The third argument is a pointer to an array of pointers that are used to point to memory locations containing the environmental parameters, such as the paths, the Windows boot directory name, the temporary directory name, and soon.

Summary

In this lesson you've learned the following:

- The enum (that is, enumerated) data type can be used to declare named integer constants.
- By default, the first enum name starts with the value of 0. Each name in the rest of the list increases by one from the value contained by the name on its left side.
- If needed, you can assign any integer values to enumerated names.
- You can create your own names for data types with the help of the typedef keyword. Those names can then be used as synonyms for the data types.
- In ANSI C, there is a header file called stddef.h that contains a dozen typedef definitions.
- A function in C can be made to call itself. Such a function is said to be recursive.
- You can use command-line arguments to pass information to the main() function in your program.
- There are two built-in arguments to the main() function. The executable filename you entered from the operating system's command line is counted as the first command-line argument.
- The first built-in argument receives the number of command-line arguments entered by the user. The second built-in argument is a pointer to an array of pointers that refers to the strings of command-line arguments.

In the next lesson you'll learn about collecting variables of different types with structures.

Q&A

Q What can you do with the enum data type?

A The enum data type can be used to declare names that represent integer constants. You can use the default values held by enum names, or you can assign values to enum names and use them later in the program. The enum data type makes the C program more readable and easier to maintain, because you can use words which you understand as the names of enum, and you will only have to go to one place to update the values when needed.

Q Why do you need to use the typedef keyword?

A By using the typedef keyword, you can define your own names to represent the data types in C. You can represent complex data types in a single word and then use that word in subsequent variable declarations. In this way, you can avoid typing errors when writing a complex declaration over and over. Also, if a data type is changed in the future, you just need to update the typedef definition of the data type, which fixes every use of the typedef definition.

Q Does a recursive function help to improve the performance of a program?

A Not really. Normally, a recursive function only makes the implementations of some algorithms clearer and simpler. A recursive function may slow down the speed of a program because of the overhead of repeated function calls.

Q What is the first command-line argument passed to a main() function?

A The first command-line argument passed to a main() function is the executable filename entered by the user, plus the path to the executable file. The executable file is created from the program that contains the main() function. The first command-line argument is stored in the memory location referenced by the first element in the array of pointers that is declared as the second argument to the main() function.

Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the Workshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

Quiz

1. What are the values represented by the following enum names?

```
enum months { Jan, Feb, Mar, Apr,
              May, Jun, Jul, Aug,
              Sep, Oct, Nov, Dec };
```

2. What are the values represented by the following enum names?

```
enum tag { name1,
           name2 = 10,
           name3,
           name4 };
```

3. Which statements in the following are equivalent in the variable declaration?
   - typedef long int BYTE32; BYTE32 x, y, z;
   - typedef char *STRING[16]; STRING str1, str2, str3;
   - long int x, y, z;
   - char *str1[16], *str2[16], *str3[16];
4. Can you pass some command-line arguments to a main() function that has the following definition?

```
int main(void)
{
   . . .
}
```

Exercises

1. Write a program to print out the values represented by the enumerated names declared in Quiz question 2.
2. Given the following declarations:

```
typedef char WORD;
typedef int SHORT;
typedef long LONG;
typedef float FLOAT;
typedef double DFLOAT;
```

   write a program to measure the sizes of the synonyms to the data types.

3. Rewrite the program in Listing 18.4. This time, add integers starting at the value of MIN_NUM instead of the value of MAX_NUM.
4. Write a program that accepts command-line arguments. If the number of command-line arguments, not including the name of the executable itself, is less than two, print out the usage format of the program and ask the user to reenter the command-line arguments. Otherwise, display all command-line arguments entered by the user.