Sams Teach Yourself C in 24 Hours

Previous | Table of Contents | Next

Hour 14 - Scope and Storage Classes in C

Nobody owns anything and all anyone has is the use of his presumed possessions.

—P. Wylie

In the previous hours, you've learned how to declare variables of different data types, as well as to initialize and use those variables. It's been assumed that you can access variables from anywhere. Now, the question is: Can we declare variables that are accessible only to certain portions of a program? In this lesson you'll learn about the scope and storage classes of data in C. The main topics covered in this lesson are

- Block scope
- Function scope
- File scope
- Program scope
- The auto specifier
- The static specifier
- The register specifier
- The extern specifier
- The const modifier
- The volatile modifier

Hiding Data

To solve a complex problem in practice, the programmer normally breaks the problem into smaller pieces and deals with each piece of the problem by writing one or two functions (or routines). Then, all the functions are put together to form a complete program that can be used to solve the complex problem.

In the complete program, there might be variables that have to be shared by all the functions. On the other hand, the use of some other variables may be limited to only certain functions. That is, the visibility of those variables is limited, and values assigned to those variables are hidden from many functions.

Limiting the scope of variables is very useful when several programmers are working on different pieces of the same program. If they limit the scope of their variables to their pieces of code, they do not have to worry about conflicting with variables of the same name used by others in other parts of the program.

In C, you can declare a variable and indicate its visibility level by designating its scope. Thus, variables with local scope can only be accessed within the block in which they are declared.

The following sections teach you how to declare variables with different scopes.

Block Scope

In this section, a block refers to any sets of statements enclosed in braces ({ and }). A variable declared within a block has block scope. Thus, the variable is active and accessible from its declaration point to the end of the block. Sometimes, block scope is also called local scope.

For example, the variable i declared within the block of the following main function has block scope:

```
int main()
{
   int i;   /* block scope */
   .
   .
   .
   return 0;
}
```

Usually, a variable with block scope is called a local variable.

Nested Block Scope

You can also declare variables within a nested block. If a variable declared in the outer block shares the same name with one of the variables in the inner block, the variable within the outer block is hidden by the one within the inner block for the scope of the inner block.

Listing 14.1 gives an example of variable scopes in nested blocks.

TYPE

Listing 14.1. Printing out variables with different scope levels.

```
/* 14L01.c: Scopes in nested block */
#include <stdio.h>

main()
{
   int i = 32;   /* block scope 1*/

   printf("Within the outer block: i=%d\n", i);
9:
10:    {    /* the beginning of the inner block */
11:      int i, j;    /* block scope 2, int i hides the outer int i*/
12:
13:      printf("Within the inner block:\n");
14:      for (i=0, j=10; i<=10; i++, j--)
15:          printf("i=%2d, j=%2d\n", i, j);
16:    }   /* the end of the inner block */
17:    printf("Within the outer block:  i=%d\n", i);
18:    return 0;
19: }
```

The following output is displayed on the screen after the executable (14L01.exe) of the program in Listing 14.1 is created and run from a DOS prompt:

OUTPUT

```
C:\app>14L01
Within the outer block: i=32
Within the inner block:
i= 0, j=10
i= 1, j= 9
i= 2, j= 8
i= 3, j= 7
i= 4, j= 6
i= 5, j= 5
i= 6, j= 4
i= 7, j= 3
i= 8, j= 2
i= 9, j= 1
i=10, j= 0
Within the outer block: i=32
C:\app>
```

ANALYSIS

The purpose of the program in Listing 14.1 is to show you the different scopes of variables in nested blocks. As you can see, there are two nested blocks in Listing 14.1. The integer variable i declared in line 6 is visible within the outer block enclosed by the braces ({ and }) in lines 5 and 19. Another two integer variables, i and j, declared in line 11, are visible only within the inner block from line 10 to line 16.

Although the integer variable i within the outer block has the same name as one of the integer variables within the inner block, the two integer variables can not be accessed at the same time due to their different scopes.

To prove this, line 8 prints out the value, 32, contained by i within the outer block for the first time. Then, the for loop in lines 14 and 15 displays 10 pairs of values assigned to i and j within the inner block. At this point, there is no sign showing that the integer variable i within the outer block has any effects on the one within the inner block. When the inner block is exited, the variables within the inner block are no longer accessible.

Finally, the statement in line 17 prints out the value of i within the outer block again to find out whether the value has been changed due to the integer variable i within the inner block. The result shows that these two integer variables hide from each other, and no conflict occurs.

Function Scope

Function scope indicates that a variable is active and visible from the beginning to the end of a function.

In C, only the goto label has function scope. For example, the goto label, start, shown in the following code portion has function scope:

```
int main()
{
   int i;   /* block scope */
   .
   .
   .
   start:   /* A goto label has function scope */
   .
   .
   .
   goto  start;  /* the goto statement */
   .
   .
   .
   return 0;
}
```

Here the label start is visible from the beginning to the end of the main() function. Therefore, there should not be more than one label having the same name within the main() function.

Program Scope

A variable is said to have program scope when it is declared outside a function. For instance, look at the following code:

```
int x = 0;        /* program scope */
float y = 0.0;    /* program scope */
int main()
{
   int i;   /* block scope */
   .
   .
   .
   return 0;
}
```

Here the int variable x and the float variable y have program scope.

Variables with program scope are also called global variables, which are visible among different files. These files are the entire source files that make up an executable program. Note that a global variable is declared with an initializer outside a function.

The program in Listing 14.2 demonstrates the relationship between variables with program scope and variables with block scope.

TYPE

Listing 14.2. The relationship between program scope and block scope.

```
1:  /* 14L02.c: Program scope vs block scope */
2:  #include <stdio.h>
3:
4:  int x = 1234;          /* program scope */
5:  double y = 1.234567;  /* program scope */
6:
7:  void function_1()
8:  {
9:     printf("From function_1:\n  x=%d, y=%f\n", x, y);
10: }
11:
12: main()
13: {
14:    int x = 4321;   /* block scope 1*/
15:
16:    function_1();
17:    printf("Within the main block:\n  x=%d, y=%f\n", x, y);
18:    /* a nested block */
19:    {
20:       double y = 7.654321;  /* block scope 2 */
21:       function_1();
22:       printf("Within the nested block:\n  x=%d, y=%f\n", x, y);
23:    }
24:    return 0;
25: }
```

I have the following output shown on the screen after the executable 14L02.exe is created and run from a DOS prompt:

OUTPUT

```
C:\app>14L02
From function_1:
  x=1234, y=1.234567
Within the main block:
  x=4321, y=1.234567
From function_1:
  x=1234, y=1.234567
Within the nested block:
  x=4321, y=7.654321
C:\app>
```

ANALYSIS

As you can see in Listing 14.2, there are two global variables, x and y, with program scope; they are declared in lines 4 and 5.

In lines 7_10, a function, called function_1(), is declared. (More details about function declarations and prototypes are taught in the next hour.) The function_1() function contains only one statement; it prints out the values held by both x and y. Because there is no variable declaration made for x or y within the function block, the values of the global variables x and y are used for the statement inside the function. To prove this, the function_1() function is called twice in lines 16 and 21, respectively, from two nested blocks. The output shows that the values of the two global variables x and y are passed to printf() enclosed in the function_1() function body.

Then, line 14 declares another integer variable, x, with block scope, which can replace the global variable x within the block of the main() function. The result made by the statement in line 17 shows that the value of x is the value of the local variable x with block scope, while the value of y is still that of the global variable y.

There is a nested block in lines 19 and 23, within which another double variable y, with block scope, is declared and initialized. Like the variable x within the main() block, this variable, y, within the nested block replaces the global variable y. The statement in line 22 puts the values of the local variables x and y on the screen.

TIP

Since a global variable is visible among different source files of a program, using global variables increases your program's complexity, which in turn makes your program hard to maintain or debug. Generally, it's not recommended that you declare and use global variables, unless it's very necessary. For instance, you can declare a global variable whose value is used but never changed by several subroutines in your program. (In Hour 23, "The C Preprocessor," you'll learn to use the #define directive to define constants that are used in many places in a program.)

Before I introduce file scope, let's talk about the storage class specifiers.

The Storage Class Specifiers

In C, the storage class of a variable refers to the combination of its spatial and temporal regions.

You've learned about scope, which specifies the spatial region of a variable. Now, let's focus on duration, which indicates the temporal region of a variable.

There are four specifiers and two modifiers that can be used to indicate the duration of a variable. These specifiers and modifiers are introduced in the following sections.

The auto Specifier

The auto specifier indicates that the memory location of a variable is temporary. In other words, a variable's reserved space in the memory can be erased or relocated when the variable is out of its scope.

Only variables with block scope can be declared with the auto specifier. The auto keyword is rarely used, however, because the duration of a variable with block scope is temporary by default.

The static Specifier

The static specifier, on the other hand, can be applied to variables with either block scope or program scope. When a variable within a function is declared with the static specifier, the variable has a permanent duration. In other words, the memory storage allocated for the variable is not destroyed when the scope of the variable is exited, the value of the variable is maintained outside the scope, and if execution ever returns to the scope of the variable, the last value stored in the variable is still there.

For instance, in the following code portion:

```
int main()
{
   int i;         /* block scope and temporary duration */
   static int j;  /* block scope and permanent duration */
   .
   .
   return 0;
}
```

the integer variable i has temporary duration by default. But the other integer variable, j, has permanent duration due to the storage class specifier static.

The program in Listing 14.3 shows the effect of the static specifier on variables.

TYPE

Listing 14.3. Using the static specifier.

```
/* 14L03.c: Using the static specifier */
#include <stdio.h>
/* the add_two function */
int add_two(int x, int y)
{
   static int counter = 1;

   printf("This is the function call of %d,\n", counter++);
   return (x + y);
}
/* the main function */
main()
{
   int i, j;

   for (i=0, j=5; i<5; i++, j--)
      printf("the addition of %d and %d is %d.\n\n",
             i, j, add_two(i, j));
   return 0;
}
```

The following output is displayed on the screen after the executable (14L03.exe) is run from a DOS prompt:

OUTPUT

```
C:\app>14L03
This is the function call of 1,
the addition of 0 and 5 is 5.

This is the function call of 2,
the addition of 1 and 4 is 5.

This is the function call of 3,
the addition of 2 and 3 is 5.

This is the function call of 4,
the addition of 3 and 2 is 5.

This is the function call of 5,
the addition of 4 and 1 is 5.
C:\app>
```

ANALYSIS

The purpose of the program in Listing 14.3 is to call a function to add two integers and then print out the result returned by the function on the screen. The function is called several times. A counter is set to keep track of how many times the function has been called.

This function, called add_two(), is declared in lines 4_10. There are two int arguments, x and y, that are passed to the function, and the addition of the two arguments is returned in line 9. Note that there is an integer variable, counter, that is declared with the static specifier in line 6. Values stored by counter are retained because the duration of the variable is permanent. In other words, although the scope of counter is within the block of the add_two() function, the memory location of counter and value saved in the location are not changed after the add_two() function is called and the execution control is returned back to the main() function.

Therefore, the counter variable is used as a counter to keep the number of calls received by the add_two() function. In fact, the statement of the printf() function in line 8 prints out the value saved by the counter variable each time the add_two() function is called. In addition, counter is incremented by one each time after the printf() function is executed.

The for loop, declared in lines 16_18 within the main() function, calls the add_two() function five times. The values of the two integer variables, i and j, are passed to the add_two() function for the operation of addition. Then, the return value from the add_two() function is displayed on the screen by the printf() function in lines 17 and 18.

From the output, you can see that the value saved by counter is indeed incremented by one each time the add_two() function is called, and is retained after the function exits because the integer variable counter is declared with static. Note that counter is only initialized once when the add_two() function is called for the first time.

File Scope and the Hierarchy of Scopes

In the first part of this hour, I mentioned three of the four types of scopes: block scope, function scope, and program scope. It's time now to introduce the fourth scope—file scope.

In C, a global variable declared with the static specifier is said to have file scope. A variable with file scope is visible from its declaration point to the end of the file. Here the file refers to the program file that contains the source code. Most large programs consist of several program files.

The following portion of source code shows variables with file scope:

```
int x = 0;             /* program scope */
static int y = 0;      /* file scope */
static float z = 0.0;  /* file scope */
int main()
{
   int i;   /* block scope */
   .
   .
   .
   return 0;
}
```

Here the int variable y and the float variable z both have file scope.

Figure 14.1 shows the hierarchy of the four scopes. As you can see, a variable with block scope is the most limited and is not visible outside the block within which the variable is

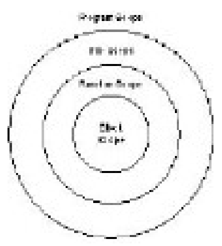

Figure 14.1. *The hierarchy of the four scopes.*

declared. On the other hand, a variable with program scope is visible within all files, functions, and other blocks that make up the program.

The register Specifier

The word register is borrowed from the world of computer hardware. Each computer has a certain number of registers to hold data and perform arithmetic or logical calculations. Because registers are located within the CPU (central processing unit) chip, it's much quicker to access a register than a memory location. Therefore, storing variables in registers may help to speed up your program.

The C language provides you with the register specifier. You can apply this specifier to variables when you think it's necessary to put the variables into the computer registers.

However, the register specifier only gives the compiler a suggestion. In other words, a variable specified by the register keyword is not guaranteed to be stored in a register. The compiler can ignore the suggestion if there is no register available, or if some other restrictions have to apply.

It's illegal to take the address of a variable that is declared with the register specifier because the variable is intended to be stored in a register, not in the memory.

In the following portion of code, the integer variable i is declared with the register specifier:

```
int main()
{
   /* block scope with the register specifier */
   register int i;
   . . .
   for (i=0; i<MAX_NUM; i++){
      /* some statements */
   }
   . . .
   return 0;
}
```

The declaration of i suggests that the compiler store the variable in a register. Because i is intensively used in the for loop, storing i in a register may increase the speed of the code shown here.

The extern Specifier

As stated in the section titled "Program Scope," a variable with program scope is visible through all source files that make up an executable program. A variable with program scope is also called a global variable.

Here is a question: How can a global variable declared in file A, for instance, be seen in file B? In other words, how does the compiler know that the variable used in file B is actually the same variable declared in file A?

The answer is: Use the extern specifier provided by the C language to allude to a global variable defined elsewhere. In this case, we declare a global variable in file A, and then declare the variable again using the extern specifier in file B.

For instance, suppose you have two global int variables, y and z, that are defined in one file, and then, in another file, you may have the following declarations:

```
int x = 0;          /* a global variable */
extern int y;       /* an allusion to a global variable y */
int main()
{
   extern int z;  /* an allusion to a global variable z */
   int i;         /* a local variable */
   .
   .
   .
   return 0;
}
```

As you can see, there are two integer variables, y and z, that are declared with the extern specifier, outside and inside the main() function, respectively. When the compiler sees the two declarations, it knows that the declarations are actually allusions to the global variables y and z that are defined elsewhere.

NOTE

To make your program portable across different computer platforms, you can apply the following rules in your program when you declare or allude to global variables:

- You can ignore the extern specifier, but include an initializer, when you declare a global variable.
- You should use the extern specifier (without an initializer) when you allude to a global variable defined elsewhere.

The Storage Class Modifiers

Besides the four storage class specifiers introduced in the previous sections, C also provides you with two storage class modifiers (or qualifiers, as they're sometimes called) that you can use to indicate to the C compiler how variables may be accessed.

The const Modifier

If you declare a variable with the const modifier, the content of the variable cannot be changed after it is initialized.

For instance, the following expression indicates to the compiler that circle_ratio is a variable whose value should not be changed:

```
const double circle_ratio = 3.141593;
```

Likewise, the value of the character array str declared in the following statement cannot be changed, either:

```
const char str[] = "A string constant";
```

Therefore, it's illegal to do something like this:

```
str[0] = `a';  /* It's not allowed here. */
```

In addition, you can declare a pointer variable with the const modifier so that an object pointed to by the pointer cannot be changed. For example, consider the following pointer declaration with the const modifier:

```
char const *ptr_str = "A string constant";
```

After the initialization, you cannot change the content of the string pointed to by the pointer ptr_str. For instance, the following statement is not allowed:

```
*ptr_str = `a';     /* It's not allowed here. */
```

However, the ptr_str pointer itself can be assigned a different address of a string that is declared with char const.

The volatile Modifier

Sometimes, you want to declare a variable whose value can be changed without any explicit assignment statement in your program. For instance, you might declare a global variable that contains characters entered by the user. The address of the variable is passed to a device register that accepts characters from the keyboard. However, when the C compiler optimizes your program automatically, it intends to not update the value held by the variable unless the variable is on the left side of an assignment operator (=). In other words, the value of the variable is likely not changed even though the user is typing in characters from the keyboard.

To ask the compiler to turn off certain optimizations on a variable, you can declare the variable with the volatile specifier. For instance, in the following code portion, a variable, keyboard_ch, declared with the volatile specifier, tells the compiler not to optimize any expressions of the variable because the value saved by the variable may be changed without execution of any explicit assignment statement:

```
void read_keyboard()
{
   volatile char keyboard_ch;  /* a volatile variable */
   .
   .
   .
}
```

Summary

In this lesson you've learned the following:

- A variable declared within a block has block scope. Such a variable is also called a local variable and is only visible within the block
- The goto label has function scope, which means that it is visible through the whole block of the function within which the label is placed. No two goto labels share the same name within a function block.

- A variable declared with the static specifier outside a function has file scope, which means that it is visible throughout the entire source file in which the variable is declared.
- A variable declared outside a function is said to have program scope. Such a variable is also called a global variable. A global variable is visible in all source files that make up an executable program.
- A variable with block scope has the most limited visibility. On the other hand, a variable with program block is the most visible through all files, functions, and other blocks that make up the program.
- The storage class of a variable refers to the combination of its spatial and temporal regions (that is, its scope and duration.)
- By default, a variable with block scope has an auto duration, and its memory storage is temporary.
- A variable declared with the static specifier has permanent memory storage, even though the function in which the variable is declared has been called and the function scope has exited.
- A variable declared with the register specifier may be stored in a register to speed up the performance of a program; however, the compiler can ignore the specifier if there is no register available or if some other restrictions have to apply.
- You can also allude to a global variable defined elsewhere by using the extern specifier from the current source file.
- To make sure the value saved by a variable cannot be changed, you can declare the variable with the const modifier.
- If you want to let the compiler know that the value of a variable can be changed without an explicit assignment statement, declare the variable with the volatile modifier so that the compiler will turn off optimizations on expressions involving the variable.

In the next lesson you'll learn about function declarations and prototypes in C.

Q&A

Q Can a global variable be hidden by a local variable with block scope?

A Yes. If a local variable shares the same name with a global variable, the global

variable can be hidden by the local variable for the scope of the block within which the local variable is defined with block scope. However, outside the block, the local variable cannot be seen, but the global variable becomes visible again.

Q Why do you need the static specifier?

A In many cases, the value of a variable is needed, even if the scope of the block, in which the variable is declared, has exited. By default, a variable with block scope has a temporary memory storage—that is, the lifetime of the variable starts when the block is executed and the variable is declared, and ends when the execution is finished. Therefore, to declare a variable with permanent duration, you have to use the static specifier to indicate to the compiler that the memory location of the variable and the value stored in the memory location should be retained after the execution of the block.

Q Does using the register specifier guarantee to improve the performance of a program?

A Not really. Declaring a variable with the register specifier only suggests to the compiler that the variable be stored in a register. But there is no guarantee that the variable will be stored in a register. The compiler can ignore the request based on the availability of registers or other restrictions.

Q When you declare a variable with the extern specifier, do you define the variable or allude to a global variable elsewhere?

A When a variable is declared with the extern specifier, the compiler considers the declaration of the variable as an allusion rather than a definition. The compiler will therefore look somewhere else to find a global variable to which the variable with extern alludes.

Workshop

To help solidify your understanding of this 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. Given the following code portion, which variables are global variables, and which ones are local variables with block scope?

```
int x = 0;
float y = 0.0;
int myFunction()
{
   int i, j;
```

```
      float y;
      . . .
      {
         int x, y;
         . . .
      }
      . . .
   }
```

2. When two variables with the same name are defined, how does the compiler know which one to use?
3. Identify the storage class of each declaration in the following code portion:

```
int i = 0;
static int x;
extern float y;
int myFunction()
{
   int i, j;
   extern float z;
   register long s;
   static int index;
   const char str[] = "Warning message.";
   . . .
}
```

4. Given the following declaration:

```
const char ch_str[] = "The const specifier";
```

   is the ch_str[9] = `-'; statement legal?

Exercises

1. Given the following,
   - An int variable with block scope and temporary storage
   - A constant character variable with block scope
   - A float local variable with permanent storage
   - A register int variable
   - A char pointer initialized with a null character

   write declarations for all of them.

2. Rewrite the program in Listing 14.2. This time, pass the int variable x and the float variable y as arguments to the function_1() function. What do you get on your screen after running the program?
3. Compile and run the following program. What do you get on the screen, and why?

```
#include <stdio.h>
int main()
{
   int i;

   for (i=0; i<5; i++){
      int x = 0;
      static int y = 0;
      printf("x=%d, y=%d\n", x++, y++);
   }
   return 0;
}
```

4. Rewrite the add_two() function in Listing 14.3 to print out the previous result of