Chapter 12 Understanding Variable Scope

12-1. What Is Scope?

The *scope* of a variable refers to the extent to which different parts of a program have access to the variable--in other words, where the variable is *visible*. When referring to C variables, the terms *accessibility* and *visibility* are used interchangeably. When speaking about scope, the term *variable* refers to all C data types: simple variables, arrays, structures, pointers, and so forth. It also refers to symbolic constants defined with the const keyword.

Scope also affects a variable's *lifetime:* how long the variable persists in memory, or when the variable's storage is allocated and deallocated. First, this chapter examines visibility.

12-1-1. A Demonstration of Scope

Look at the program in Listing 12.1. It defines the variable x in line 5, uses printf() to display the value of x in line 11, and then calls the function print_value() to display the value of x again. Note that the function print_value() is not passed the value of x as an argument; it simply uses x as an argument to printf() in line 19.

Listing 12.1. The variable x is accessible within the function print_value().

```
1:  /* Illustrates variable scope. */
2:
3:  #include <stdio.h>
4:
5:  int x = 999;
6:
7:  void print_value(void);
8:
9:  main()
10: {
11:     printf("%d\n", x);
12:     print_value();
13:
14:     return 0;
15: }
16:
17: void print_value(void)
18: {
19:     printf("%d\n", x);
20: }
999
999
```

This program compiles and runs with no problems. Now make a minor modification in the program, moving the definition of the variable x to a location within the main() function. The new source code is shown in Listing 12.2.

Listing 12.2. The variable x is not accessible within the function print_value().

```
1:  /* Illustrates variable scope. */
2:
3:  #include <stdio.h>
```

```
4:
5:  void print_value(void);
6:
7:  main()
8:  {
9:     int x = 999;
10:
11:     printf("%d\n", x);
12:     print_value();
13:
14:     return 0;
15: }
16:
17: void print_value(void)
18: {
19:     printf("%d\n", x);
20: }
```

ANALYSIS: If you try to compile Listing 12.2, the compiler generates an error message similar to the following:

```
list1202.c(19) : Error: undefined identifier `x'.
```

Remember that in an error message, the number in parentheses refers to the program line where the error was found. Line 19 is the call to printf() within the print_value() function.

This error message tells you that within the print_value() function, the variable x is undefined or, in other words, not visible. Note, however, that the call to printf() in line 11 doesn't generate an error message; in this part of the program, the variable x *is* visible.

The only difference between Listings 12.1 and 12.2 is where variable x is defined. By moving the definition of x, you change its scope. In Listing 12.1, x is an *external variable,* and its scope is the entire program. It is accessible within both the main() function and the print_value() function. In Listing 12.2, x is a *local variable,* and its scope is limited to within the main() function. As far as print_value() is concerned, x doesn't exist. Later in this chapter, you'll learn more about local and external variables, but first you need to understand the importance of scope.

12-1-2. Why Is Scope Important?

To understand the importance of variable scope, you need to recall the discussion of structured programming on Day 5. The structured approach, you might remember, divides the program into independent functions that perform a specific task. The key word here is *independent*. For true independence, it's necessary for each function's variables to be isolated from interference caused by other functions. Only by isolating each function's data can you make sure that the function goes about its job without some other part of the program throwing a monkey wrench into the works.

If you're thinking that complete data isolation between functions isn't always desirable, you are correct. You will soon realize that by specifying the scope of variables, a programmer has a great deal of control over the degree of data isolation.

12-2. External Variables

An *external variable* is a variable defined outside of any function. This means outside of main() as well, because main() is a function, too. Until now, most of the variable definitions in this book have been external, placed in the source code before the start of main(). External variables are sometimes referred to as *global variables*.

NOTE: If you don't explicitly initialize an external variable when it's defined, the compiler initializes it to 0.

12-2-1. External Variable Scope

The scope of an external variable is the entire program. This means that an external variable is visible throughout main() and every other function in the program. For example, the variable x in Listing 12.1 is an external variable. As you saw when you compiled and ran the program, x is visible within both functions, main() and print_value().

Strictly speaking, it's not accurate to say that the scope of an external variable is the entire program. Instead, the scope is the entire source code file that contains the variable definition. If the entire program is contained in one source code file, the two scope definitions are equivalent. Most small-to-medium-sized C programs are contained in one file, and that's certainly true of the programs you're writing now.

It's possible, however, for a program's source code to be contained in two or more separate files. You'll learn how and why this is done on Day 21, "Advanced Compiler Use," and you'll see what special handling is required for external variables in these situations.

12-2-2. When to Use External Variables

Although the sample programs to this point have used external variables, in actual practice you should use them rarely. Why? Because when you use external variables, you are violating the principle of *modular independence* that is central to structured programming. Modular independence is the idea that each function, or module, in a program contains all the code and data it needs to do its job. With the relatively small programs you're writing now, this might not seem important, but as you progress to larger and more complex programs, overreliance on external variables can start to cause problems.

When should you use external variables? Make a variable external only when all or most of the program's functions need access to the variable. Symbolic constants defined with the const keyword are often good candidates for external status. If only some of your functions need access to a variable, pass the variable to the functions as an argument rather than making it external.

12-2-3. The extern Keyword

When a function uses an external variable, it is good programming practice to declare the variable within the function using the extern keyword. The declaration takes the form

```
extern type name;
```

in which *type* is the variable type and *name* is the variable name. For example, you would add the declaration of x to the functions main() and print_value() in Listing 12.1. The resulting program is shown in Listing 12.3.

Listing 12.3. The external variable x is declared as extern within the functions main() and print_value().

```
1:  /* Illustrates declaring external variables. */
2:
3:  #include <stdio.h>
4:
5:  int x = 999;
6:
```

```
7:  void print_value(void);
8:
9:  main()
10: {
11:     extern int x;
12:
13:     printf("%d\n", x);
14:     print_value();
15:
16:     return 0;
17: }
18:
19: void print_value(void)
20: {
21:     extern int x;
22:     printf("%d\n", x);
23: }
999
999
```

ANALYSIS: This program prints the value of x twice, first in line 13 as a part of main(), and then in line 21 as a part of print_value(). Line 5 defines x as a type int variable equal to 999. Lines 11 and 21 declare x as an extern int. Note the distinction between a variable definition, which sets aside storage for the variable, and an extern declaration. The latter says: "This function uses an external variable with such-and-such a name and type that is defined elsewhere." In this case, the extern declaration isn't needed, strictly speaking--the program will work the same without lines 11 and 21. However, if the function print_value() were in a different code module than the global declaration of the variable x (in line 5), the extern declaration would be required.

12-3. Local Variables

A *local variable* is one that is defined within a function. The scope of a local variable is limited to the function in which it is defined. Day 5 describes local variables within functions, how to define them, and what their advantages are. Local variables aren't automatically initialized to 0 by the compiler. If you don't initialize a local variable when it's defined, it has an undefined or *garbage* value. You must explicitly assign a value to local variables before they're used for the first time.

A variable can be local to the main() function as well. This is the case for x in Listing 12.2. It is defined within main(), and as compiling and executing that program illustrates, it's also only visible within main().

DO use local variables for items such as loop counters.

DON'T use external variables if they aren't needed by a majority of the program's functions.

DO use local variables to isolate the values the variables contain from the rest of the program.

12-3-1. Static Versus Automatic Variables

Local variables are *automatic* by default. This means that local variables are created anew each time the function is called, and they are destroyed when execution leaves the function. What this means, in practical terms, is that an automatic variable doesn't retain its value between calls to the function in which it is defined.

Suppose your program has a function that uses a local variable x. Also suppose that the first time it is called, the function assigns the value 100 to x. Execution returns to the calling program, and the function is called again later. Does the variable x still hold the value 100? No, it does not. The first instance of variable x was destroyed when execution left the function after the first call. When the function was called again, a new instance of x had to be created. The old x is gone.

What if the function needs to retain the value of a local variable between calls? For example, a printing function might need to remember the number of lines already sent to the printer to determine when a new page is needed. In order for a local variable to retain its value between calls, it must be defined as *static* with the static keyword. For example:

```
void func1(int x)
{
    static int a;
    /* Additional code goes here */
}
```

Listing 12.4 illustrates the difference between automatic and static local variables.

Listing 12.4. The difference between automatic and static local variables.

```
1:  /* Demonstrates automatic and static local variables. */
2:  #include <stdio.h>
3:  void func1(void);
4:  main()
5:  {
6:     int count;
7:
8:     for (count = 0; count < 20; count++)
9:     {
10:         printf("At iteration %d: ", count);
11:         func1();
12:     }
13:
14:     return 0;
15: }
16:
17: void func1(void)
18: {
19:     static int x = 0;
20:     int y = 0;
21:
22:     printf("x = %d, y = %d\n", x++, y++);
23: }
At iteration 0: x = 0, y = 0
At iteration 1: x = 1, y = 0
At iteration 2: x = 2, y = 0
At iteration 3: x = 3, y = 0
At iteration 4: x = 4, y = 0
At iteration 5: x = 5, y = 0
At iteration 6: x = 6, y = 0
At iteration 7: x = 7, y = 0
At iteration 8: x = 8, y = 0
```

```
At iteration 9: x = 9, y = 0
At iteration 10: x = 10, y = 0
At iteration 11: x = 11, y = 0
At iteration 12: x = 12, y = 0
At iteration 13: x = 13, y = 0
At iteration 14: x = 14, y = 0
At iteration 15: x = 15, y = 0
At iteration 16: x = 16, y = 0
At iteration 17: x = 17, y = 0
At iteration 18: x = 18, y = 0
At iteration 19: x = 19, y = 0
```

ANALYSIS: This program has a function that defines and initializes one variable of each type. This function is func1() in lines 17 through 23. Each time the function is called, both variables are displayed on-screen and incremented (line 22). The main() function in lines 4 through 15 contains a for loop (lines 8 through 12) that prints a message (line 10) and then calls func1() (line 11). The for loop iterates 20 times.

In the output, note that x, the static variable, increases with each iteration because it retains its value between calls. The automatic variable y, on the other hand, is reinitialized to 0 with each call.

This program also illustrates a difference in the way explicit variable initialization is handled (that is, when a variable is initialized at the time of definition). A static variable is initialized only the first time the function is called. At later calls, the program remembers that the variable has already been initialized and therefore doesn't reinitialize. Instead, the variable retains the value it had when execution last exited the function. In contrast, an automatic variable is initialized to the specified value every time the function is called.

If you experiment with automatic variables, you might get results that disagree with what you've read here. For example, if you modify Listing 12.4 so that the two local variables aren't initialized when they're defined, the function func1() in lines 17 through 23 reads

```
17: void func1(void)
18: {
19:     static int x;
20:     int y;
21:
22:     printf("x = %d, y = %d\n", x++, y++);
23: }
```

When you run the modified program, you might find that the value of y increases by 1 with each iteration. This means that y is keeping its value between calls to the function. Is what you've read here about automatic variables losing their value a bunch of malarkey?

No, what you read is true (Have faith!). If you get the results described earlier, in which an automatic variable keeps its value during repeated calls to the function, it's only by chance. Here's what happens: Each time the function is called, a new y is created. The compiler might use the same memory location for the new y that was used for y the preceding time the function was called. If y isn't explicitly initialized by the function, the storage location might contain the value that y had during the preceding call. The variable seems to have kept its old value, but it's just a chance occurrence; you definitely can't count on it happening every time.

Because automatic is the default for local variables, it doesn't need to be specified in the variable definition. If you want to, you can include the auto keyword in the definition before the type keyword, as shown here:

```
void func1(int y)
{
    auto int count;
```

```
    /* Additional code goes here */
}
```

12-3-2. The Scope of Function Parameters

A variable that is contained in a function heading's parameter list has *local scope*. For example, look at the following function:

```
void func1(int x)
{
    int y;
    /* Additional code goes here */
}
```

Both x and y are local variables with a scope that is the entire function func1(). Of course, x initially contains whatever value was passed to the function by the calling program. Once you've made use of that value, you can use x like any other local variable.

Because parameter variables always start with the value passed as the corresponding argument, it's meaningless to think of them as being either static or automatic.

12-3-3. External Static Variables

You can make an external variable static by including the static keyword in its definition:

```
static float rate;
main()
{
    /* Additional code goes here */
}
```

The difference between an ordinary external variable and a static external variable is one of scope. An ordinary external variable is visible to all functions in the file and can be used by functions in other files. A static external variable is visible only to functions in its own file and below the point of definition.

These distinctions obviously apply mostly to programs with source code that is contained in two or more files. This topic is covered on Day 21.

12-3-4. Register Variables

The register keyword is used to suggest to the compiler that an automatic local variable be stored in a *processor register* rather than in regular memory. What is a processor register, and what are the advantages of using it?

The central processing unit (CPU) of your computer contains a few data storage locations called *registers*. It is in the CPU registers that actual data operations, such as addition and division, take place. To manipulate data, the CPU must move the data from memory to its registers, perform the manipulations, and then move the data back to memory. Moving data to and from memory takes a finite amount of time. If a particular variable could be kept in a register to begin with, manipulations of the variable would proceed much faster.

By using the register keyword in the definition of an automatic variable, you ask the compiler to store that variable in a register. Look at the following example:

```
void func1(void)
{
```

```
    register int x;
    /* Additional code goes here */
}
```

Note that I said *ask,* not *tell*. Depending on the program's needs, a register might not be available for the variable. In this case, the compiler treats it as an ordinary automatic variable. The register keyword is a suggestion, not an order. The benefits of the register storage class are greatest for variables that the function uses frequently, such as the counter variable for a loop.

The register keyword can be used only with simple numeric variables, not arrays or structures. Also, it can't be used with either static or external storage classes. You can't define a pointer to a register variable.

DO initialize local variables, or you won't know what value they will contain.

DO initialize global variables even though they're initialized to 0 by default. If you always initialize your variables, you'll avoid problems such as forgetting to initialize local variables.

DO pass data items as function parameters instead of declaring them as global if they're needed in only a few functions.

DON'T use register variables for nonnumeric values, structures, or arrays.

12-4. Local Variables and the main() Function

Everything said so far about local variables applies to main() as well as to all other functions. Strictly speaking, main() is a function like any other. The main() function is called when the program is started from your operating system, and control is returned to the operating system from main() when the program terminates.

This means that local variables defined in main() are created when the program begins, and their lifetime is over when the program ends. The notion of a static local variable retaining its value between calls to main() really makes no sense: A variable can't remain in existence between program executions. Within main(), therefore, there is no difference between automatic and static local variables. You can define a local variable in main() as being static, but it has no effect.

DO remember that main() is a function similar in most respects to any other function.

DON'T declare static variables in main(), because doing so gains nothing.

12-5. Which Storage Class Should You Use?

When you're deciding which storage class to use for particular variables in your programs, it might be helpful to refer to Table 12.1, which summarizes the five storage classes available in C.

Table 12.1. C's five variable storage classes.

Storage Lifetime	Where It's Defined	Scope	Class	Keyword

Automatic	None[1]	Temporary	In a function	Local
Static	static	Temporary	In a function	Local
Register	register	Temporary	In a function	Local
External	None[2]	Permanent	Outside a function	Global (all files)
External	static	Permanent	Outside a function	Global (one file)
[1]The auto keyword is optional.				
[2]The extern keyword is used in functions to declare a static external variable that is defined elsewhere.				

When you're deciding on a storage class, you should use an automatic storage class whenever possible and use other classes only when needed. Here are some guidelines to follow:

- Give each variable an automatic local storage class to begin with.
- If the variable will be manipulated frequently, add the register keyword to its definition.
- In functions other than main(), make a variable static if its value must be retained between calls to the function.
- If a variable is used by most or all of the program's functions, define it with the external storage class.

12-6. Local Variables and Blocks

So far, this chapter has discussed only variables that are local to a function. This is the primary way local variables are used, but you can define variables that are local to any program block (any section enclosed in braces). When declaring variables within the block, you must remember that the declarations must be first. Listing 12.5 shows an example.

Listing 12.5. Defining local variables within a program block.

```
1:  /* Demonstrates local variables within blocks. */
2:
3:  #include <stdio.h>
4:
5:  main()
6:  {
7:      /* Define a variable local to main(). */
8:
9:      int count = 0;
10:
11:     printf("\nOutside the block, count = %d", count);
12:
13:     /* Start a block. */
14:     {
15:       /* Define a variable local to the block. */
16:
17:       int count = 999;
18:       printf("\nWithin the block, count = %d", count);
19:     }
20:
21:     printf("\nOutside the block again, count = %d\n", count);
22:     return 0;
23: }
```

```
Outside the block, count = 0
Within the block, count = 999
Outside the block again, count = 0
```

ANALYSIS: From this program, you can see that the count defined within the block is independent of the count defined outside the block. Line 9 defines count as a type int variable equal to 0. Because it is declared at the beginning of main(), it can be used throughout the entire main() function. The code in line 11 shows that the variable count has been initialized to 0 by printing its value. A block is declared in lines 14 through 19, and within the block, another count variable is defined as a type int variable. This count variable is initialized to 999 in line 17. Line 18 prints the block's count variable value of 999. Because the block ends on line 19, the print statement in line 21 uses the original count initially declared in line 9 of main().

The use of this type of local variable isn't common in C programming, and you may never find a need for it. Its most common use is probably when a programmer tries to isolate a problem within a program. You can temporarily isolate sections of code in braces and establish local variables to assist in tracking down the problem. Another advantage is that the variable declaration/initialization can be placed closer to the point where it's used, which can help in understanding the program.

DON'T try to put variable definitions anywhere other than at the beginning of a function or at the beginning of a block.

DON'T use variables at the beginning of a block unless it makes the program clearer.

DO use variables at the beginning of a block (temporarily) to help track down problems.

12-7. Summary

This chapter covered C's variable storage classes. Every C variable, whether a simple variable, an array, a structure, or whatever, has a specific storage class that determines two things: its scope, or where in the program it's visible; and its lifetime, or how long the variable persists in memory.

Proper use of storage classes is an important aspect of structured programming. By keeping most variables local to the function that uses them, you enhance functions' independence from each other. A variable should be given automatic storage class unless there is a specific reason to make it external or static.

Q&A

Q If global variables can be used anywhere in the program, why not make all variables global?
A As your programs get bigger, you will begin to declare more and more variables. As stated in this chapter, there are limits on the amount of memory available. Variables declared as global take up memory for the entire time the program is running; however, local variables don't. For the most part, a local variable takes up memory only while the function to which it is local is active. (A static variable takes up memory from the time it is first used to the end of the program.) Additionally, global variables are subject to unintentional alteration by other functions. If this occurs, the variables might not contain the values you expect them to when they're used in the functions for which they were created.
Q Day 11, "Structures," stated that scope affects a structure instance but not a structure tag or body. Why doesn't scope affect the structure tag or body?
A When you declare a structure without instances, you are creating a template. You don't actually declare any variables. It isn't until you create an instance of the structure that you declare a variable. For this reason, you can leave a structure body external to any functions with no real effect on external memory. Many programmers put commonly used structure bodies with tags into header files and then include these header files when they need to create an instance of the structure. (Header files are covered on Day 21.)

Q How does the computer know the difference between a global variable and a local variable that have the same name?
A The answer to this question is beyond the scope of this chapter. The important thing to know is that when a local variable is declared with the same name as a global variable, the program temporarily ignores the global variable. It continues to ignore the global variable until the local variable goes out of scope.
Q Can I declare a local variable and a global variable that have the same name, as long as they have different variable types?
A Yes. When you declare a local variable with the same name as a global variable, it is a completely different variable. This means that you can make it whatever type you want. You should be careful, however, when declaring global and local variables that have the same name.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered, and exercises to provide you with experience in using what you've learned.

Quiz

1. What does scope refer to?
2. What is the most important difference between local storage class and external storage class?
3. How does the location of a variable definition affect its storage class?
4. When defining a local variable, what are the two options for the variable's lifetime?
5. Your program can initialize both automatic and static local variables when they are defined. When do the initializations take place?
6. True or false: A register variable will always be placed in a register.
7. What value does an uninitialized global variable contain?
8. What value does an uninitialized local variable contain?
9. What will line 21 of Listing 12.5 print if lines 9 and 11 are removed? Think about this, and then try the program to see what happens.
10. If a function needs to remember the value of a local type int variable between calls, how should the variable be declared?
11. What does the extern keyword do?
12. What does the static keyword do?

Exercises

1. Write a declaration for a variable to be placed in a CPU register.
2. Change Listing 12.2 to prevent the error. Do this without using any external variables.
3. Write a program that declares a global variable of type int called var. Initialize var to any value. The program should print the value of var in a function (not main()). Do you need to pass var as a parameter to the function?
4. Change the program in exercise 3. Instead of declaring var as a global variable, change it to a local variable in main(). The program should still print var in a separate function. Do you need to pass var as a parameter to the function?
5. Can a program have a global and a local variable with the same name? Write a program that uses a global and a local variable with the same name to prove your answer.
6. BUG BUSTER: Can you spot the problem in this code? Hint: It has to do with where a variable is declared.

```
void a_sample_function( void )
{
   int ctr1;
   for ( ctr1 = 0; ctr1 < 25; ctr1++ )
      printf( "*" );
   puts( "\nThis is a sample function" );
   {
      char star = `*';
      puts( "\nIt has a problem\n" );
```

```
      for ( int ctr2 = 0; ctr2 < 25; ctr2++ )
      {
         printf( "%c", star);
      }
   }
}
```

7. BUG BUSTER: What is wrong with the following code?

```
/*Count the number of even numbers between 0 and 100. */
#include <stdio.h>
main()
{
   int x = 1;
   static int tally = 0;
   for (x = 0; x < 101; x++)
   {
      if (x % 2 == 0)  /*if x is even...*/
      tally++;..  /*add 1 to tally.*/
   }
   printf("There are %d even numbers.\n", tally);
   return 0;
}
```

8. BUG BUSTER: Is anything wrong with the following program?

```
#include <stdio.h>
void print_function( char star );
int ctr;
main()
{
   char star;
   print_function( star );
   return 0;
}
void print_function( char star )
{
   char dash;
   for ( ctr = 0; ctr < 25; ctr++ )
   {
      printf( "%c%c", star, dash );
   }
}
```

9. What does the following program print? Don't run the program--try to figure it out by reading the code.

```
#include <stdio.h>
void print_letter2(void);           /* function prototype */
int ctr;
char letter1 = `X';
char letter2 = `=';
main()
{
   for( ctr = 0; ctr < 10; ctr++ )
   {
      printf( "%c", letter1 );
      print_letter2();
   }
```

```
   return 0;
}
void print_letter2(void)
{
   for( ctr = 0; ctr < 2; ctr++ )
      printf( "%c", letter2 );
}
```

10. BUG BUSTER: Will the preceding program run? If not, what's the problem? Rewrite it so that it is correct.