Scope In C

Matt Bishop

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

Introduction

The *scope* of a variable or function is that part of the program in which it can be referenced. A *global* variable is one that can be used anywhere in the program; a *local* variable is one that can only be used within a particular function.

Figuring out the scope of a variable in C is more complicated than in most other languages. For one thing, C programs are rarely compiled as a unit; most have several files of source code, each file being compiled separately and the resulting objects linked together to form an executable module. Variables can be made global to all functions in all files, or to the functions in one file only; local variables can be made local to a function or to a block, and the compiler can be instructed to reserve storage for them which will remain allocated after the function or block exits. Most languages do not have these features.

Because of all this complexity, we would do well to review the definition of a few terms, and how C variables are declared.

Declarations and Definitions

A *declaration* is a C statement that gives the compiler information about the variable; a *definition* is a C statement that instructs the compiler to reserve storage. While a variable may be declared many times throughout a C program, especially if the program has its source in several files, there can only be one definition. Note also that a definition is invariably a declaration, although a declaration need not be a definition.

A variable cannot be used unless it has been declared first. Variables may be declared in three places: at the top level of the program (that is, outside any function), as a formal parameter to a function, or at the beginning of a block. Each of these has different consequences for the scope of the declared variable.

The format of a C variable declaration is:

storage-class type identifier

Type is the type of variable, and is one of *char*, *short*, *int*, *long*, *unsigned*, *float*, *double*, *struct* name, or *union* name. *Identifier* is the variable name. *Storage-class* indicates how the compiler is to allocate storage for the variable; this is the part of the declaration relevant to scope. The storage classes are *static*, *auto*, *register*, and *extern*. (There is a fifth keyword, *typedef*, that for syntactic reasons is considered a storage class; but it plays no part in anything that follows.)

The class *register* informs the compiler that the variable will be heavily used, and the compiler should attempt to keep the variable stored in a register (since register accesses are almost always quicker and more compact to express than other types of accesses.) It can only be used to declare parameters to a function or variables defined in a block, and compilers have a limit on the number of *register* declarations they will honor. (Excess *register* declarations are simply ignored.)

The class *static* tells the compiler that the storage allocated to that variable is not to be released until the process exits. Storage for parameters and variables declared at the top of a block is usually allocated in such a way that it is deallocated when the executing block or function exits (For example, storage is often created on a stack; when execution completes, the stack is popped.) The *static* declaration tells the compiler to allocate permanent storage for the variable. As a side effect, if the variable is being defined at the top level, *static* informs the compiler not to allow the variable to be accessed from any other file (this usually involves marking it as local or not putting the name in the symbol table.) For obvious reasons, variables which are parameters to functions cannot be declared *static*.

The class *extern* indicates that no storage is to be allocated, because the variable is defined elsewhere; it simply informs the compiler of the type of the variable. In fact, the compiler does not have to know where the variable's storage is actually located; *extern* variables are usually defined in another file, and the linker patches up the references. This class is used a great deal, since in C a variable cannot be used until it is declared. Like *static*, however, *extern* cannot be used to declare function parameters.

The final class, *auto*, is rarely (if ever) explicitly stated. It indicates that the storage allocated is to be released at the end of the block. Hence, the only place it can be used is in declaring variables at the beginning of blocks. This is the most common declaration used when defining variables local to a block or function.

Top Level Declarations

Variables declared at the top level are called *external* variables and may be of storage class *extern* or *static*. If the storage class is *static*, the variable may not be accessed from another file. (Incidentally, functions may be declared *static*, too.) If the storage class is *extern*, the statement is a declaration rather than a definition; the compiler will not reserve storage for the variable at that point, but when the linker links all the modules of the program together, there must be storage reserved for that variable in exactly one module. If the storage class is omitted, the declaration is a global definition of the variable; storage is reserved for that variable, and all *extern* declarations of that variable will refer to that storage location.

As an example, suppose we have a simple program which prints the integers between its two command-line arguments. We shall put the main routine in the file ‘‘main.c’’ and the counting routine in the file ‘‘count.c’’. Note that no error checking is done (it is omitted because we are only interested in the scope of the variables, *not* because error checking is not worthwhile!):

```
/* main.c */
#include <stdio.h>

int first, last;    /* start, end points of counting */

main(argc, argv)
int argc;           /* argument count */
char **argv;        /* argument list */
{
        /*
         * set up the endpoints
         */
        first = atoi(argv[1]);
        last = atoi(argv[2]);

        /*
         * count from first to last
         */
        count();

        /*
         * exit nicely (0 = success)
         */
        exit(0);
}
```

```
/* count.c */
#include <stdio.h>

extern int first, last; /* start, end points of counting */

count()
{
        int i;          /* counter */

        /*
         * count from first to last
         */
        for(i = first; i <= last; i++)
                printf("%d ", i);
}
```

In ‘‘main.c’’, the definition

```
int first, last;
```

instructs the compiler to allocate storage for two integer variables. In ‘‘count.c’’, we need to use those values. We cannot just re-declare the variables there, however, since that would produce a conflict (specifically, there would be two variables named *first*.) So, we declare the two variables to be *extern* in ‘‘count.c’’ -- this informs the compiler that the two variables are integer variables, and that the storage is reserved elsewhere. Then, the compiler has sufficient information to generate correct references to them, which the linker will then resolve when the two modules are linked.

Assuming the operating system is UNIX, the way to compile these programs is:

```
% cc main.c count.c
main.c:
count.c:
%
```

(the italicized text is what you type.)

Contrast this with what would happen had we defined the two variables *first* and *last* to be static in ‘‘main.c’’:

```
/* main.c */
#include <stdio.h>

static int first, last;  /* start, end points of counting */

main(argc, argv)
int argc;          /* argument count */
char **argv;          /* argument list */
{
        /*
         * set up the endpoints
         */
        first = atoi(argv[1]);
        last = atoi(argv[2]);

        /*
         * count from first to last
         */
        count();

        /*
         * exit nicely (0 = success)
         */
        exit(0);
}
```

The keyword *static* informs the compiler that the variables *first* and *last* are to be available in ‘‘main.c’’, but not in any other file. So, if we tried compiling these files, we would get:

```
% cc main.c count.c
main.c:
count.c:
ld. error. undefined: <_last>
ld. error. undefined: <_first>
%
```

The storage classes *register* and *auto* are meaningless when used with variables declared at the top level, and produce error messages when the file containing them is compiled.

Recall that an identifier must be declared before it can be used. Sometimes, for organizational purposes, a variable is defined at the top level but after several functions. If a function defined earlier in the file references that variable, the variable must be declared before the function. In this case, the declaration has the storage class *extern*, just as though the variable were defined in another file.

Formal Parameters

When a function is defined, the order and types of the arguments are given by listing identifiers in parentheses after the function name, and then declaring each before the block containing the statements making up the function. For example, in our sample program, the following

```
main(argc, argv)
int argc;
char **argv;
```

identifies *main* as having two parameters, the first being an integer and the second a pointer to a pointer to characters. The scope of these variables is the body of the function in which they are declared. If any of these parameters has the same name as a global, using the name within the function will refer to the parameter (which is a local variable) and not the global variable.

If a storage class is specified for these parameters, it must be *register*. This informs the compiler that the parameter should be moved from the stack (or wherever the arguments are kept) to a register. As discussed earlier, specifying any of *extern*, *auto*, or *static* will cause a compiler error.

Variables Declared In A Block

Variable declarations follow the opening brace of a block. These variables are local to that block; attempting to use them once the block has ended produces an error. Note that any blocks contained in the block in which the variables are defined may access the variable.

Any of the four storage classes may be used in block declarations. If none is given, *auto* is assumed. If *register* is used, the compiler will attempt to keep the variable in a register; this makes use of the variable faster. If *static* is used, the variable will retain its value across new invocations of the same block. For example, the program

```
/* staticdemo.c */
main()
{
	demo();
	demo();
}
demo()
{
	static int timescalled = 0;

	timescalled++;
	printf("demo has been called %d times\n",
				timescalled);
}
```

produces the output

```
demo has been called 1 times
demo has been called 2 times
```

Of course, the variable cannot be referenced from outside the block. If *extern* rather than *static* is used, no storage is generated, and the compiler and linker expect the variable to be defined elsewhere.

If a variable is declared in a block, and it has the same name as another variable declared in an outer block, a global variable, or a parameter, references to that variable name will access the innermost definition. For example, the following program:

```
main()
{
	int i = 1;
	printf("i = %d at level 1\n", i);
	{
		int i = 2;
		printf("i = %d at level 2\n", i);
	}
	printf("i = %d at level 1\n", i);
}
```

produces the output

```
i = 1 at level 1
i = 2 at level 2
i = 1 at level 1
```

How To Figure Out Which Variable You Are Talking About

One of the most common complaints from people learning C is that they get confused about which variable is being used. In C, however, this is very easy to determine, if you remember that every variable must be declared before it is used.

When you find a statement containing a variable, and want to find the variable's declaration, locate the beginning of the block in which the statement occurs. If the variable is not defined there, look for the beginning of the next outermost block. Continue this procedure until you reach the outermost block. If the variable is not defined there, it is either a parameter or a global of some type. As you are now at the outermost block, you should be able to find the function name and parameter list immediately before the block. If the variable is not listed as a parameter, it is a global. (Be sure you check the parameter list rather than the declarations, since a parameter listed but not declared is an integer.) Start at the function name and move backwards through the file, towards the top. Look only at the top-level declarations, not in any functions. If the variable is not defined within that file, check any included files for its declaration. If you still have not located the variable's declaration, the variable is an undefined variable. The algorithm below summarizes this procedure:

```
while( in a block ){
    find the top of the current block
    if ( var is declared there )
        exit /* found it! */
    go to next outer block
}
if ( var in function parameter list )
    exit /* found it! */
while( not at top of the file ){
    if ( at #include line )
        look in #include-ed file
    else if ( function definition )
        skip to before the function
    else if ( var is declared )
        exit /* found it */
    else
        go to previous line in file
}
/* not found -- var is undeclared variable */
```

As an example, here is a very simple program:

```
 1  #include <stdio.h>

 2  int var = 0;

 3  main(argc, argv)
 4  int argc;
 5  char **argv;
 6  {
 7          int i = 1;

 8          printf("at level 1: i = %d, var = %d\n", i, var);
 9          {
10                  int i = 2;

11                  printf("at level 2: i = %d, var = %d\n", i, var);
12          }
13          printf("back at level 1: i = %d, var = %d\n", i, var);
14  }
```

To find out which variable *i* the *i* in line 11 refers to, begin there, and move to the top of the block containing it (line 9). Check the list of variables declared for that block. In this case, an *i* is defined there (line 10). This is the desired *i*.

How about the *i* in line 13? The top of the block containing that statement is at line 6; looking at the list of variables immediately following, we find the appropriate definition at line 7.

For a more complicated case, consider the variable *var* on line 11. We find the beginning of the block at line 9; checking the variable declarations there, we do not find one for *var*. This block is contained in another, which begins at line 6; checking the declarations within that block, we still do not see any for *var*. Since this is the outermost block, we next check the parameter list for the function (line 3) to see if *var* is defined there. It is not. Hence, it must be defined globally. Moving backwards towards the top, we find the definition we seek at line 2.

The rule for locating function declarations is the same as for variable declarations, with one modification. If at the end of the algorithm, the the function declaration has not been found, it is to be taken as a function returning an integer and declared *extern*. Thus, every function has an implicit declaration.

Conclusion

The basic rules of scope are:

- Variables declared outside functions may be used from the declaration on, throughout the file.
- Variables declared as parameters may be used within that function definition.
- Variables declared at the beginning of a block may be used within that block and any enclosed blocks.
- Variables declared in blocks take precedence over variables declared as parameters, which in turn take precedence over global variables.

This summarizes all one needs to know about C scope.