Storage In C

Matt Bishop

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, CA 94035

Introduction

In a previous article we discussed the scope of C variables. Intimately bound with the idea of scope is that of *storage*. When a program defines a variable, the compiler will decide how to store the value of that variable, and create a space in memory for the value. The compiler uses the declaration to determine how much space to allocate and how to allocate it – as temporary storage (such as on a stack) or as more permanent storage (in data space.)

Recall that the format of a C variable declaration is:

storage-class type identifier

The part of this declaration relevant to the storage is, not surprisingly, *storage-class*. 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.)

In each of the following sections we shall discuss one type of storage declaration, its effects, and when it is legal. In the last section, we shall look at a simple program written in a variety of ways and show how the results of changing storage classes affects the way the compiler handles storage.

Auto

The simplest class of storage is *automatic* storage, indicated by the storage class keyword *auto*. Any variables defined at the beginning of a block with no explicit storage class specifier are assumed to be automatic. The storage for these variables is created when the block is entered, and exists until the block is left. For example, in the routine

```
show()
{
    auto int i;

    i = 7;
    printf("In show, i = %d\n", i);
}
```

the storage space for *i* is allocated when the routine is entered, and deallocated when the program leaves the routine.

Most compilers do this by way of a *stack*. A stack is just a list onto which things are *pushed* (or put), and from which things are *popped* (or removed.) The useful property of a stack is that the last thing pushed onto the stack is the first thing removed (hence, a stack is sometimes called a ‘‘LIFO’’ list, for ‘‘Last In, First Out’’ list.) If you think of a stack of trays in a cafeteria, you will see why this data structure is called a stack; the last tray put on top of the pile is the first one removed.

The space for storing automatic variables is created by using a stack. The easiest way to see how is by an example. Here is a routine to call *show*, above:

```
main()
{
    auto int j;

    j = 10;
    show();
    printf("In main, j = %d\n", j);
}
```

When *main* is called, the stack of storage is empty:

(bottom)

Then space for the variable *j* is allocated, and *j* is set to 10. The stack now looks like (reading left to right):

10 *(bottom)*

At this point, *show* is called, and space need to be created for the variable *i*, which is set to 7. Once this occurs, the stack looks like

7 10 *(bottom)*

Now, the program leaves the routine *show*; the storage for *i* was automatic, so it is deallocated. Once the program has returned to *main*, the stack reverts to

10 *(bottom)*

When the routine *main* exits, again the space for the automatic variable *j* is released; the stack is now empty:

(bottom)

Note that the storage class specifier is almost always omitted when a variable is automatic; since automatic variables only occur in blocks, and never outside them, this does not pose any problems. However, another version of automatic storage requires a storage class specifier, and we shall discuss this class, called *register*, in the next section.

Register

The storage class specifier *register* is different from other storage class specifiers, in that compilers are at liberty to obey or ignore them. This storage class is the same as automatic so far as the programmer is concerned; however, rather than allocating storage on a stack (as with variables declared *auto*) the compiler arranges for the variables to be stored in registers. This makes accessing their values very quick, usually much quicker than if the variables were on a stack.

Because of the nature of machine registers, declaring a variable as a register variable entails some restrictions. Many machines cannot use the address of a register in the same way they use a memory address, so the address operator **&** cannot be applied to a register variable. Some compilers will not allow certain types of variables to be assigned to registers. Also, compilers accept only a limited number of register declarations (the number varies from machine to machine and even from compiler to compiler) and do not give messages indicating when the *register* keyword is being ignored.

It is time for an example! Let us return to the routines *main* and *show*, above, but change *show*:

```
show()
{
	auto int ir;
	register int *address_of_ir;

	ir = 7;
	address_of_ir = &ir;
	printf("In show, i = %d\n", *address_of_ir);
}
```

In this case, the stack used for storage looks just like it did in the previous section. However, the contents of a new variable, *address_of_ir*, will be put into a register. This register variable will be assigned as its value the address of *ir*. Note that references using pointers is legal with registers, so the *printf* will print the value stored in *ir* correctly. However, if *ir* were declared as a register variable rather than an *auto*, the compiler would have printed an error message for the line

```
address_of_ir = &ir;
```

since that line involves taking the address of the register variable.

Both the storage classes we have discussed are transient; they go away when the block finishes executing. But often a program needs values to remain throughout the life of the program; the next two sections will deal with storage classes for this case.

Extern

When applied to a variable, the storage class *extern* indicates the type of the variable and that the definition of the variable is located in another file. Hence, any statements of this class are declarations rather than definitions; the *extern* class does not cause any memory to be allocated.

An *extern* declaration may appear anywhere before the declared variable is referenced. In fact, the variable declared in one of these statements need not be referenced at all, in which case the compiler will treat the program as though the *extern* declaration were not present. An *extern* declaration can appear in the same file as the corresponding definition; a very common practise is to put the *extern* declarations in a header file and include that file in all source files using the *#include* mechanism.

An *extern* declaration can also be used with a function; in this case it indicates the function is defined elsewhere. (The keyword *extern* is often omitted here.) The compiler uses this declaration for type information and nothing else. As an example, look at the program

```
main()
{
        double x;
        x = sqrt(2.0);
        printf("The square root of 2 is %f\n", x);
}
```

When the compiler encounters a function in source code, it assumes that function returns an integer value unless that function has been declared previously. So, in the above program, the compiler assumes *sqrt* returns an integer. It therefore generates code to coerce the returned integer into floating point format at the assignment statement. This leads to a result that is quite wrong:

```
The square root of 2 is 1070596096.000000
```

(The precise answer is machine dependent; but it will be wrong.) However, if the line

```
extern double sqrt();
```

is placed before the assignment statement, the compiler will understand that the function *sqrt* returns a *double* and will not do any type coercion at the assignment statement; the result in this case is

```
The square root of 2 is 1.414214
```

Every *extern* declaration must have a corresponding definition (unless the variable or function in it is never referenced.) Precisely what constitutes an acceptable definition varies among compilers. Usually, a compiler will take one of two flavors of definitions.

With some compilers, the single definition at the top level (that is, not contained inside any function's body) is taken to be the definition associated with *extern* declarations. If more than one such definition occurs, the compiler (actually, the linker) will report an error. These errors are of the form "*variable or function* multiply defined".

Other compilers follow the ANSI C standard and use a more complex scheme. They consider a top-level declaration of a variable to be a *tentative definition* if the storage class is *static* or omitted. If a tentative definition is found in which the variable is initialized, that is taken to be *the* definition and the other tentative definitions become declarations. Otherwise, the first tentative definition becomes the definition and the rest become declarations.

Let us use an example to explain the differences. Suppose there are two source files to a program; one, *main.c*, contains the routine

```
int testcalled = 0;

main()
{
	test();
	printf("test() called, testcalled = %d\n", testcalled);
	test();
	printf("test() called, testcalled = %d\n", testcalled);
	exit(0);
}
```

and the second, *test.c*, contains the routine

```
int testcalled;

test()
{
	testcalled++;
}
```

If the first rule of defining variables is followed, each source file will compile correctly, but when they are linked, the linker will find two definitions of *testcalled*, and report that *testcalled* is multiply defined. If the second rule of defining variables is defined, the statement

```
int testcalled;
```

in *test.c* is considered a tentative definition, and the statement

```
int testcalled = 0;
```

in *main.c* is considered the real definition, because it is a tentative definition in which *testcalled* is initialized. Hence, the tentative definition in *test.c* becomes a declaration, so the linking procedure succeeds. Note that if the statement in *test.c* had been

```
int testcalled = 0;
```

there would have been two definitions, not one, and the linker would have complained that *testcalled* had been multiply defined.

The rules of global definition also apply to another storage class, which on first glance seems to be the most complicated.

Static

The *static* storage class means simply that the variable will retain its value throughout the life of the program. When used outside a function definition, it has the side effect of not allowing the variable or function to be referenced anywhere except within that source file.

Let us deal with *static* variables first. Look at the program defined in the previous section in *main.c* and *test.c*. Recall *testcalled* was defined globally. Let us rewrite these two routines slightly; the first, in *main.c*, becomes

```
main()
{
	register int result;

	result = test();
	printf("test() called, testcalled = %d\n", result);
	result = test();
	printf("test() called, testcalled = %d\n", result);
	exit(0);
}
```

and the second, in *test.c*, will be

```
test()
{
	static int testcalled = 0;
	testcalled++;
	return(testcalled);
}
```

When we compile, link, and execute this program, we get

```
test() called, testcalled = 1
test() called, testcalled = 2
```

Now, let us redo this program, omitting the storage class keyword *static* from the declaration of *testcalled* in *test*(). This means *testcalled* will be an automatic variable, so it will be created each time *test*() is called, and discarded each time *test*() returns. Hence, we get the following result:

```
test() called, testcalled = 1
test() called, testcalled = 1
```

This graphically points out that regardless of their scope, *static* variables retain whatever value they are assigned throughout the life of the program, whereas automatic variables do not.

If declared outside functions, the storage class *static* has one additional side effect: it limits the scope of the variable or function so declared to the file in which it is declared. For example, going back to the files *main.c* and *test.c*, suppose *main.c* were written as it was originally (see the section **extern**) and *test.c* contained

```
static int testcalled;

test()
{
    testcalled++;
}
```

When run, this program produces

```
test() called, testcalled = 0
test() called, testcalled = 0
```

because the variable *testcalled* in *test.c* is not visible to any other file. So, it and the variable *testcalled* in *main.c* are completely different.

As an example of what happens when a function is declared *static*, let us combine *main.c* and *test.c* into one file, *maintest.c*, and make *test*() *static*:

```
int testcalled = 0;
main()
{
    test();
    printf("test() called, testcalled = %d\n", testcalled);
    test();
    printf("test() called, testcalled = %d\n", testcalled);
    exit(0);
}
static test()
{
    testcalled++;
}
```

The result is what we expect:

```
test() called, testcalled = 1
test() called, testcalled = 2
```

Now let us put *test*() into a separate file:

```
int testcalled;

static test()
{
    testcalled++;
}
```

Both files, *main.c* and *test.c*, will compile, but the linker will complain that ‘‘*test* is undefined’’. Since *test* is declared as *static*, it is only visible in the file *test.c*; it cannot be accessed by anything in the file *main.c*, and so the linker can find nothing to link the calls to the routine *test*() in *main.c* to. Hence, the error message.

Now that we have discussed these classes, let us take a look at a program and see how different storage classes affect the way the program works.

An Example

Here is a very simple program that sets a variable to 20, and loops, adding 20 each time through, until the variable's value is more than 1000 or 1000 loops have been made:

```
 1A.main()
 2A.{
 3A.      int counter = 0;
 4A.      while (counter < 1000){
 5A.              if (dowork() > 1000)
 6A.                      break;
 7A.              printf("Number of loops: %d\n", counter++);
 8A.      }
 9A.      printf("Done!\n");
10A.}
11A.dowork()
12A.{
13A.      static int total = 20;
14A.      total += 20;
15A.      printf("total is %d ... ", total);
16A.      return (total);
17A.}
```

First, note that the variable *counter* is declared automatic. It could just have easily been declared static; the effect would be the same, since it is local to *main* and exists until that routine exits (at which time the program exits.) So, *counter* could be any storage class except *extern*.

However, notice that *total* is defined as *static* on line 13A. Hence, as *dowork* is called, *total* will increase in value and eventually the program would exit with total being 1020. However, if the word *static* were replaced by *auto* (or *register,* or just omitted) then each time *dowork* were called, *total* would be recreated and reinitialized to 20. Hence *dowork* would always return 40, and the program would loop 1000 times before exiting.

We could also move *total* outside the function *dowork* and put it either above or below *main*. We could even put it after *dowork,* but then we would need the statement

```
extern int total;
```

somewhere before line 14A (otherwise *total* would be undefined and undeclared at that point, causing a compile-time error.)

Let us split this example into two files now. The first, *main.c*, looks like:

```
 1B. main()
 2B. {
 3B.     int counter = 0;
 4B.     while (counter < 1000){
 5B.         if (dowork() > 1000)
 6B.             break;
 7B.         printf("Number of loops: %d\n", counter++);
 8B.     }
 9B.     printf("Done!\n");
10B. }
```

and the second is *dowork.c*:

```
11B. dowork()
12B. {
13B.     static int total = 20;
14B.     total += 20;
15B.     printf("total is %d ... ", total);
16B.     return (total);
17B. }
```

Now, if the function *dowork* were declared *static*, the program would not compile correctly because the function *dowork* would not be defined anywhere so far as the function *main* is concerned.

Note line 13B could be moved before line 11B without changing anything in this program. In fact, line 3B could be put before line 1B, and every occurrence of the variable *counter* could be renamed *total* without any problem, because the variable *total* in the file *dowork.c* is declared *static* and hence is defined for that file only.

Conclusion

Choosing storage classes with care enables a programmer to balance speed with memory used. Because of this balance, it is very important that users of C know the effects of each storage class. In addition, scope and allocation of storage are very closely related. In a previous article we discussed the scope of C variables. This article completes the discussion begun there, by describing how variables are stored.

∞