12

Storage Classes, Linkage, and Memory Management

You will learn about the following in this chapter:

- Keywords:

  `auto`, `extern`, `static`, `register`, `const`, `volatile`, `restricted`, `_Thread_local`, `_Atomic`

- Functions:

  `rand()`, `srand()`, `time()`, `malloc()`, `calloc()`, `free()`

- How C allows you to determine the scope of a variable (how widely known it is) and the lifetime of a variable (how long it remains in existence)
- Designing more complex programs

One of C's strengths is that it enables you to control a program's fine points. C's memory management system exemplifies that control by letting you determine which functions know which variables and for how long a variable persists in a program. Using memory storage is one more element of program design.

Storage Classes

C provides several different models, or *storage classes*, for storing data in memory. To understand the options, it's helpful to go over a few concepts and terms first.

Every programming example in this book stores data in memory. There is a hardware aspect to this—each stored value occupies physical memory. C literature uses the term *object* for such a chunk of memory. An object can hold one or more values. An object might not yet actually have a stored value, but it will be of the right size to hold an appropriate value. (The phrase *object-oriented programming* uses the word *object* in a more developed sense to indicate class objects, whose definitions encompass both data and permissible operations on the data; C is not an object-oriented programming language.)

There also is a software aspect—the program needs a way to access the object. This can be accomplished, for instance, by declaring a variable:

```
int entity = 3;
```

This declaration creates an *identifier* called `entity`. An identifier is a name, in this case one that can be used to designate the contents of a particular object. Identifiers satisfy the naming conventions for variables discussed in Chapter 2, "Introducing C." In this case, the identifier `entity` is how the software (the C program) designates the object that's stored in hardware memory. This declaration also provides a value to be stored in the object.

A variable name isn't the only way to designate an object. For instance, consider the following declarations:

```
int * pt = &entity;
int ranks[10];
```

In the first case, `pt` is an identifier. It designates an object that holds an address. Next, the expression `*pt` is not an identifier because it's not a name. However, it does designate an object, in this case the same object that `entity` designates. In general, as you may recall from Chapter 3, "Data and C," an expression that designates an object is called an lvalue. So `entity` is an identifier that is an lvalue, and `*pt` is an expression that is an lvalue. Along the same lines, the expression `ranks + 2 * entity` is neither an identifier (not a name) nor an lvalue (doesn't designate the contents of a memory location). But the expression `*(ranks + 2 * entity)` is an lvalue because it does designate the value of a particular memory location, the seventh element of the `ranks` array. The declaration of `ranks`, by the way, creates an object capable of holding ten `int`s, and each member of the array also is an object.

If, as with all these examples, you can use the lvalue to change the value in an object, it's a *modifiable lvalue*. Now consider this declaration:

```
const char * pc = "Behold a string literal!";
```

This causes the program to store the string literal contents in memory, and that array of character values is an object. Each character in the array also is an object, as it can be accessed individually. The declaration also creates an object having the identifier `pc` and holding the address of that string. The identifier `pc` is a modifiable lvalue because it can be reset to point to a different string. The `const` prevents you from altering the contents of a pointed-to string but not from changing which string is pointed to. So `*pc`, which designates the data object holding the `'B'` character, is an lvalue, but not a modifiable lvalue. Similarly, the string literal itself, because it designates the object holding the character string, is an lvalue, but not a modifiable one.

You can describe an object in terms of its *storage duration*, which is how long it stays in memory. You can describe an identifier used to access the object by its *scope* and its *linkage*, which together indicate which parts of a program can use it. The different storage classes offer different combinations of scope, linkage, and storage duration. You can have identifiers that can be shared over several files of source code, identifiers that can be used by any function in one particular file, identifiers that can be used only within a particular function, and even identifiers that can be used only within a subsection of a function. You can have objects that exist for the duration of a program and objects that exist only while the function containing them is executing. With concurrent programming, you can have objects that exist for the duration of a particular thread. You also can store data in memory that is allocated and freed explicitly by means of function calls.

Next, let's investigate the meaning of the terms *scope*, *linkage*, and *storage duration*. After that, we'll return to specific storage classes.

Scope

Scope describes the region or regions of a program that can access an identifier. A C variable has one of the following scopes: block scope, function scope, function prototype scope, or file scope. The program examples to date have used block scope almost exclusively for variables. A *block*, as you'll recall, is a region of code contained within an opening brace and the matching closing brace. For instance, the entire body of a function is a block. Any compound statement within a function also is a block. A variable defined inside a block has *block scope*, and it is visible from the point it is defined until the end of the block containing the definition. Also, formal function parameters, even though they occur before the opening brace of a function, have block scope and belong to the block containing the function body. So the local variables we've used to date, including formal function parameters, have block scope. Therefore, the variables `cleo` and `patrick` in the following code both have block scope extending to the closing brace:

```
double blocky(double cleo)
{
    double patrick = 0.0;
    ...
    return patrick;
}
```

Variables declared in an inner block have scope restricted just to that block:

```
double blocky(double cleo)
{
    double patrick = 0.0;
    int i;
    for (i = 0; i < 10; i++)
    {
        double q = cleo * i; // start of scope for q
        ...
        patrick *= q;
    }                        // end of scope for q
    ...
```

```
     return patrick;
}
```

In this example, the scope of `q` is limited to the inner block, and only code within that block can access `q`.

Traditionally, variables with block scope had to be declared at the beginning of a block. C99 relaxed that rule, allowing you to declare variables anywhere in a block. One new possibility is in the control section of a `for` loop. That is, you now can do this:

```
for (int i = 0; i < 10; i++)
    printf("A C99 feature: i = %d", i);
```

As part of this new feature, C99 expanded the concept of a block to include the code controlled by a `for` loop, `while` loop, `do while` loop, or `if` statement, even if no brackets are used. So in the previous `for` loop, the variable `i` is considered to be part of the `for` loop block. Therefore, its scope is limited to the `for` loop. After execution leaves the `for` loop, the program will no longer see that `i`.

Function scope applies just to labels used with `goto` statements. This means that even if a label first appears inside an inner block in a function, its scope extends to the whole function. It would be confusing if you could use the same label inside two separate blocks, and function scope for labels prevents this from happening.

Function prototype scope applies to variable names used in function prototypes, as in the following:

```
int mighty(int mouse, double large);
```

Function prototype scope runs from the point the variable is defined to the end of the prototype declaration. What this means is that all the compiler cares about when handling a function prototype argument is the types; the names you use, if any, normally don't matter, and they needn't match the names you use in the function definition. One case in which the names matter a little is with variable-length array parameters:

```
void use_a_VLA(int n, int m, ar[n][m]);
```

If you use names in the brackets, they have to be names declared earlier in the prototype.

A variable with its definition placed outside of any function has *file scope*. A variable with file scope is visible from the point it is defined to the end of the file containing the definition. Take a look at this example:

```
#include <stdio.h>
int units = 0;         /* a variable with file scope */
void critic(void);
int main(void)
{
   ...
}
```

```
void critic(void)
{
   ...
}
```

Here, the variable `units` has file scope, and it can be used in both `main()` and `critic()`. (More exactly, `units` has file scope with external linkage, a distinction we'll cover in the next section.) Because they can be used in more than one function, file scope variables are also called *global variables*.

> **Note Translation Units and Files**
>
> What you view as several files may appear to the compiler as a single file. For example, suppose that, as often is the case, you include one or more header files (`.h` extension) in a source code file (`.c` sextension). A header file, in turn, may include other header files. So several separate physical files may be involved. However, C preprocessing essentially replaces an `#include` directive with the contents of the header file. Thus the compiler sees a single file containing information from your source code file and all the header files. This single file is called a *translation unit*. When we describe a variable as having file scope, it's actually visible to the whole translation unit. If your program consists of several source code files, then it will consist of several translation units, with each translation unit corresponding to a source code file and its included files.

Linkage

Next, let's look at linkage. A C variable has one of the following linkages: external linkage, internal linkage, or no linkage. Variables with block scope, function scope, or function prototype scope have no linkage. That means they are private to the block, function, or prototype in which they are defined. A variable with file scope can have either internal or external linkage. A variable with external linkage can be used anywhere in a multifile program. A variable with internal linkage can be used anywhere in a single translation unit.

> **Note Formal and Informal Terms**
>
> The C Standard uses "file scope with internal linkage" to describe scope limited to one translation unit (a source code file plus its included header files) and "file scope with external linkage" to describe scope that, at least potentially, extends to other translation units. But programmers don't always have the time or patience to use those terms. Some common short cuts are to use "file scope" for "file scope with internal linkage" and "global scope" or "program scope" for "file scope with external linkage."

So how can you tell whether a file scope variable has internal or external linkage? You look to see if the storage class specifier `static` is used in the external definition:

```
int giants = 5;          // file scope, external linkage
static int dodgers = 3;  // file scope, internal linkage
int main()
{
    ...
}
...
```

The variable `giants` can be used by other files that are part of the same program. The `dodgers` variable is private to this particular file, but can be used by any function in the file.

Storage Duration

Scope and linkage describe the visibility of identifiers. Storage duration describes the persistence of the objects accessed by these identifiers. A C object has one of the following four storage durations: static storage duration, thread storage duration, automatic storage duration, or allocated storage duration.

If an object has static storage duration, it exists throughout program execution. Variables with file scope have static storage duration. Note that for file scope variables, the keyword `static` indicates the linkage type, not the storage duration. A file scope variable declared using `static` has internal linkage, but all file scope variables, using internal linkage or external linkage, have static storage duration.

Thread storage duration comes into play in concurrent programming, in which program execution can be divided into multiple threads. An object with thread storage duration exists from when it's declared until the thread terminates. Such an object is created when a declaration that would otherwise create a file scope object is modified with the keyword `_Thread_local`. When a variable is declared with this specifier, each thread gets its own private copy of that variable.

Variables with block scope normally have automatic storage duration. These variables have memory allocated for them when the program enters the block in which they are defined, and the memory is freed when the block is exited. The idea is that memory used for automatic variables is a workspace or scratch pad that can be reused. For example, after a function call terminates, the memory it used for its variables can be used to hold variables for the next function that is called.

Variable-length arrays provide a slight exception in that they exist from the point of declaration to the end of the block rather than from the beginning of the block to the end.

The local variables we've used so far fall into the automatic category. For example, in the following code, the variables `number` and `index` come into being each time the `bore()` function is called and pass away each time the function completes:

```
void bore(int number)
{
     int index;
     for (index = 0; index < number; index++)
          puts("They don't make them the way they used to.\n");
     return 0;
}
```

It is possible, however, for a variable to have block scope but static storage duration. To create such a variable, declare it inside a block and add the keyword `static` to the declaration:

```
void more(int number)
{
     int index;
     static int ct = 0;
     ...
     return 0;
}
```

Here the variable `ct` is stored in static memory; it exists from the time the program is loaded until the program terminates. But its scope is confined to the `more()` function block. Only while this function executes can the program use `ct` to access the object it designates. (However, one can allow indirect access by enabling the function to provide the address of the storage to other functions, for example, by a pointer parameter or return value.)

C uses scope, linkage, and storage duration to define several storage schemes for variables. This book doesn't cover concurrent programming, so we won't go into that aspect. And we'll discuss allocated storage later in this chapter. That leaves five storage classes: automatic, register, static with block scope, static with external linkage, and static with internal linkage. Table 12.1 lists the combinations. Now that we've covered scope, linkage, and storage duration, we can discuss these storage classes in more detail.

Table 12.1 **Five Storage Classes**

Storage Class	Duration	Scope	Linkage	How Declared
automatic	Automatic	Block	None	In a block
register	Automatic	Block	None	In a block with the keyword `register`
static with external linkage	Static	File	External	Outside of all functions
static with internal linkage	Static	File	Internal	Outside of all functions with the keyword `static`
static with no linkage	Static	Block	None	In a block with the keyword `static`

Automatic Variables

A variable belonging to the automatic storage class has automatic storage duration, block scope, and no linkage. By default, any variable declared in a block or function header belongs to the automatic storage class. You can, however, make your intentions perfectly clear by explicitly using the keyword `auto`, as shown here:

```
int main(void)
{
  auto int plox;
```

You might do this, for example, to document that you are intentionally overriding an external variable definition or that it is important not to change the variable to another storage class. The keyword `auto` is termed a *storage-class specifier*. C++ has repurposed the `auto` keyword for a quite different use, so simply not using `auto` as a storage-class specifier is better for C/C++ compatibility.

Block scope and no linkage imply that only the block in which the variable is defined can access that variable by name. (Of course, arguments can be used to communicate the variable's value and address to another function, but that is indirect knowledge.) Another function can use a variable with the same name, but it will be an independent variable stored in a different memory location.

Recall that automatic storage duration means that the variable comes into existence when the program enters the block that contains the variable declaration. When the program exits the block, the automatic variable disappears. Its memory location now can be used for something else, although not necessarily.

Let's look more closely at nested blocks. A variable is known only to the block in which it is declared and to any block inside that block:

```
int loop(int n)
{
     int m;          // m in scope
     scanf("%d", &m);
     {
          int i;    // both m and i in scope
          for (i = m; i < n; i++)
               puts("i is local to a sub-block\n");
     }
     return m;     // m in scope, i gone
}
```

In this code, `i` is visible only within the inner braces. You'd get a compiler error if you tried to use it before or after the inner block. Normally, you wouldn't use this feature when designing a program. Sometimes, however, it is useful to define a variable in a sub-block if it is not used elsewhere. In that way, you can document the meaning of a variable close to where it is used. Also, the variable doesn't sit unused, occupying memory when it is no longer needed. The variables `n` and `m`, being defined in the function head and in the outer block, are in scope for the whole function and exist until the function terminates.

What if you declare a variable in an inner block that has the same name as one in the outer block? Then the name defined inside the block is the variable used inside the block. We say it *hides* the outer definition. However, when execution exits the inner block, the outer variable comes back into scope. Listing 12.1 illustrates these points and more.

Listing 12.1 **The `hiding.c` Program**

```
// hiding.c -- variables in blocks
#include <stdio.h>
int main()
{
    int x = 30;      // original x

    printf("x in outer block: %d at %p\n", x, &x);
    {
        int x = 77;  // new x, hides first x
        printf("x in inner block: %d at %p\n", x, &x);
    }
    printf("x in outer block: %d at %p\n", x, &x);
    while (x++ < 33) // original x
    {
        int x = 100; // new x, hides first x
        x++;
        printf("x in while loop: %d at %p\n", x, &x);
    }
    printf("x in outer block: %d at %p\n", x, &x);

    return 0;
}
```

Here's the output:

```
x in outer block: 30 at 0x7fff5fbff8c8
x in inner block: 77 at 0x7fff5fbff8c4
x in outer block: 30 at 0x7fff5fbff8c8
x in while loop: 101 at 0x7fff5fbff8c0
x in while loop: 101 at 0x7fff5fbff8c0
x in while loop: 101 at 0x7fff5fbff8c0
x in outer block: 34 at 0x7fff5fbff8c8
```

First, the program creates an `x` variable with the value `30`, as the first `printf()` statement shows. Then it defines a new `x` variable with the value `77`, as the second `printf()` statement shows. That it is a new variable hiding the first `x` is shown by the address and also by the third `printf()` statement. It is located after the first inner block, and it displays the original `x` value, showing that the original `x` variable never went away and never got changed.

Perhaps the most intriguing part of the program is the `while` loop. The `while` loop test uses the original `x`:

```
while(x++ < 33)
```

Inside the loop, however, the program sees a third `x` variable, one defined just inside the `while` loop block. So when the code uses `x++` in the body of the loop, it is the new `x` that is incremented to `101` and then displayed. When each loop cycle is completed, that new `x` disappears. Then the loop test condition uses and increments the original `x`, the loop block is entered again, and the new `x` is created again. In this example, that `x` is created and destroyed three times. Note that, to terminate, this loop had to increment `x` in the test condition because incrementing `x` in the body increments a different `x` than the one used for the test.

This particular compiler didn't reuse the inner block memory location of `x` for the `while` loop version of `x`, but some compilers do.

The intent of this example is not to encourage you to write code like this. Rather, it is to illustrate what happens when you define variables inside a block. (Given the variety of names available via C's naming rules, it shouldn't be too difficult to come up with names other than `x`.)

Blocks Without Braces

A C99 feature, mentioned earlier, is that statements that are part of a loop or `if` statement qualify as a block even if braces (that is, `{ }`) aren't used. More completely, an entire loop is a sub-block to the block containing it, and the loop body is a sub-block to the entire loop block. Similarly, an `if` statement is a block, and its associated sub-statement is a sub-block to the `if` statement. These rules affect where you can declare a variable and the scope of that variable. Listing 12.2 shows how this works in a `for` loop.

Listing 12.2 **The `forc99.c` Program**

```
// forc99.c -- new C99 block rules
#include <stdio.h>
int main()
{
    int n = 8;

    printf("   Initially, n = %d at %p\n", n, &n);
    for (int n = 1; n < 3; n++)
        printf("      loop 1: n = %d at %p\n", n, &n);
    printf("After loop 1, n = %d at %p\n", n, &n);
    for (int n = 1; n < 3; n++)
    {
        printf(" loop 2 index n = %d at %p\n", n, &n);
        int n = 6;
        printf("      loop 2: n = %d at %p\n", n, &n);
        n++;
    }
    printf("After loop 2, n = %d at %p\n", n, &n);

    return 0;
}
```

Here is the output, assuming the compiler supports this modern C feature:

```
   Initially, n = 8 at 0x7fff5fbff8c8
      loop 1: n = 1 at 0x7fff5fbff8c4
      loop 1: n = 2 at 0x7fff5fbff8c4
After loop 1, n = 8 at 0x7fff5fbff8c8
 loop 2 index n = 1 at 0x7fff5fbff8c0
      loop 2: n = 6 at 0x7fff5fbff8bc
 loop 2 index n = 2 at 0x7fff5fbff8c0
      loop 2: n = 6 at 0x7fff5fbff8bc
After loop 2, n = 8 at 0x7fff5fbff8c8
```

Note C99 and C11 Support

Some compilers may not support these C99/C11 scope rules. (At this time Microsoft Visual Studio 2012 is one of those compilers.) Others may provide an option for activating these rules. For example, at the time of this writing, GCC supports many C99 features by default but requires using the `–std=c99` option to activate the features used in Listing 12.2:

```
gcc –std=c99 forc99.c
```

Similarly, versions of GCC or Clang may require using the `–std=c1x` or `-std=c11` options to recognize C11 features.

The `n` declared in the control section of the first `for` loop is in scope to the end of the loop and hides the initial `n`. But after execution leaves the loop, the original `n` comes into scope.

In the second `for` loop, the `n` declared as a loop index hides the initial `n`. Then, the `n` declared inside the loop body hides the loop index `n`. When the program finishes executing the body, the `n` declared in the body disappears, and the loop test uses the index `n`. When the entire loop terminates, the original `n` comes back into scope. Again, there's no need to keep reusing the same variable name, but this is what happens if you do.

Initialization of Automatic Variables

Automatic variables are not initialized unless you do so explicitly. Consider the following declarations:

```
int main(void)
{
  int repid;
  int tents = 5;
```

The `tents` variable is initialized to `5`, but the `repid` variable ends up with whatever value happened to previously occupy the space assigned to `repid`. You cannot rely on this value being `0`. You can initialize an automatic variable with a non-constant expression, provided any variables used have been defined previously:

```
int main(void)
{
  int ruth = 1;
  int rance = 5 * ruth;   // use previously defined variable
```

Register Variables

Variables are normally stored in computer memory. With luck, register variables are stored in the CPU registers or, more generally, in the fastest memory available, where they can be accessed and manipulated more rapidly than regular variables. Because a register variable may be in a register rather than in memory, you can't take the address of a register variable. In most other respects, register variables are the same as automatic variables. That is, they have block scope, no linkage, and automatic storage duration. A variable is declared by using the storage class specifier `register`:

```
int main(void)
{
   register int quick;
```

We say "with luck" because declaring a variable as a `register` class is more a request than a direct order. The compiler has to weigh your demands against the number of registers or amount of fast memory available, or it can simply ignore the request, so you might not get your wish. In that case, the variable becomes an ordinary automatic variable; however, you still can't use the address operator with it.

You can request that formal parameters be register variables. Just use the keyword in the function heading:

```
void macho(register int n)
```

The types that can be declared `register` may be restricted. For example, the registers in a processor might not be large enough to hold type `double`.

Static Variables with Block Scope

The name *static variable* sounds like a contradiction, like a variable that can't vary. Actually, *static* means that the variable stays put in memory, not necessarily in value. Variables with file scope automatically (and necessarily) have static storage duration. As mentioned earlier, you also can create local variables having block scope but static duration. These variables have the same scope as automatic variables, but they don't vanish when the containing function ends its job. That is, such variables have block scope, no linkage, but static storage duration. The computer remembers their values from one function call to the next—such variables are created by declaring them in a block (which provides the block scope and lack of linkage) with the storage-class specifier `static` (which provides the static storage duration). The example in Listing 12.3 illustrates this technique.

Listing 12.3 **The `loc_stat.c` Program**

```
/* loc_stat.c -- using a local static variable */
#include <stdio.h>
void trystat(void);

int main(void)
{
    int count;

    for (count = 1; count <= 3; count++)
    {
        printf("Here comes iteration %d:\n", count);
        trystat();
    }

    return 0;
}

void trystat(void)
{
    int fade = 1;
    static int stay = 1;

    printf("fade = %d and stay = %d\n", fade++, stay++);
}
```

Note that `trystat()` increments each variable after printing its value. Running the program returns this output:

```
Here comes iteration 1:
fade = 1 and stay = 1
Here comes iteration 2:
fade = 1 and stay = 2
Here comes iteration 3:
fade = 1 and stay = 3
```

The static variable `stay` remembers that its value was increased by 1, but the `fade` variable starts anew each time. This points out a difference in initialization: `fade` is initialized each time `trystat()` is called, but `stay` is initialized just once, when `trystat()` is compiled. Static variables are initialized to zero if you don't explicitly initialize them to some other value.

The two declarations look similar:

```
int fade = 1;
static int stay = 1;
```

However, the first statement is really part of the `trystat()` function and is executed each time the function is called. It is a runtime action. The second statement isn't actually part of the `trystat()` function. If you use a debugger to execute the program step-by-step, you'll see that the program seems to skip that step. That's because static variables and external variables are already in place after a program is loaded into memory. Placing the statement in the `trystat()` function tells the compiler that only the `trystat()` function is allowed to see the variable; it's not a statement that's executed during runtime.

You can't use `static` for function parameters:

```
int wontwork(static int flu);   // not allowed
```

Another term for a static variable with block scope is a "local static variable." Also, if you read some of the older C literature, you'll find this storage class referred to as the *internal static storage class*. However, the word *internal* was used to indicate internal to a function, not internal linkage.

Static Variables with External Linkage

A static variable with external linkage has file scope, external linkage, and static storage duration. This class is sometimes termed the *external storage class*, and variables of this type are called *external variables*. You create an external variable by placing a defining declaration outside of any function. As a matter of documentation, an external variable can additionally be declared inside a function that uses it by using the `extern` keyword. If a particular external variable is defined in one source code file and is used in a second source code file, declaring the variable in the second file with `extern` is mandatory. Declarations look like this:

```
int Errupt;           /* externally defined variable    */
double Up[100];       /* externally defined array       */
extern char Coal;     /* mandatory declaration if       */
                      /* Coal defined in another file   */
void next(void);
int main(void)
{
  extern int Errupt;  /* optional declaration           */

  extern double Up[]; /* optional declaration           */
  ...
}
void next(void)
```

```
{
  ...
}
```

Note that you don't have to give the array size in the optional declaration of `double Up`. That's because the original declaration already supplied that information. The group of `extern` declarations inside `main()` can be omitted entirely because external variables have file scope, so they are known from the point of declaration to the end of the file. They do serve, however, to document your intention that `main()` use these variables.

If only `extern` is omitted from the declaration inside a function, a separate automatic variable is set up. That is, replacing

```
extern int Errupt;
```

with

```
int Errupt;
```

in `main()` causes the compiler to create an automatic variable named `Errupt`. It would be a separate, local variable, distinct from the original `Errupt`. The local variable would be in scope while the program executes `main()`, but the external `Errupt` would be in scope for other functions, such as `next()`, in the same file. In short, a variable in block scope "hides" a variable of the same name in file scope while the program executes statements in the block. If, for some improbable reason, you actually need to use a local variable with the same name as a global variable, you might opt to use the `auto` storage-specifier in the local declaration to document your choice.

External variables have static storage duration. Therefore, the array `Up` maintains its existence and values regardless of whether the program is executing `main()`, `next()`, or some other function.

The following three examples show four possible combinations of external and automatic variables. Example 1 contains one external variable: `Hocus`. It is known to both `main()` and `magic()`.

```
/* Example 1 */
int Hocus;
int magic();
int main(void)
{
   extern int Hocus;  // Hocus declared external
   ...
}
int magic()
{
   extern int Hocus;  // same Hocus as above
   ...
}
```

Example 2 has one external variable, `Hocus`, known to both functions. This time, `magic()` knows it by default.

```
/* Example 2 */
int Hocus;
int magic();
int main(void)
{
   extern int Hocus;  // Hocus declared external
   ...
}
int magic()
{
                      // Hocus not declared but is known
   ...
}
```

In Example 3, four separate variables are created. The `Hocus` variable in `main()` is automatic by default and is local to `main`. The `Hocus` variable in `magic()` is automatic explicitly and is known only to `magic()`. The external `Hocus` variable is not known to `main()` or `magic()` but would be known to any other function in the file that did not have its own local `Hocus`. Finally, `Pocus` is an external variable known to `magic()` but not to `main()` because `Pocus` follows `main()`.

```
/* Example 3 */
int Hocus;
int magic();
int main(void)
{
  int Hocus;        // Hocus declared, is auto by default
   ...
}
int Pocus;
int magic()
{
   auto int Hocus;  // local Hocus declared automatic
   ...
}
```

These examples illustrate the scope of external variables: from the point of declaration to the end of the file. They also illustrate the lifetimes of variables. The external `Hocus` and `Pocus` variables persist as long as the program runs, and, because they aren't confined to any one function, they don't fade away when a particular function returns.

Initializing External Variables

Like automatic variables, external variables can be initialized explicitly. Unlike automatic variables, external variables are initialized automatically to zero if you don't initialize them. This rule applies to elements of an externally defined array, too. Unlike the case for automatic variables, you can use only constant expressions to initialize file scope variables:

```
int x = 10;              // ok, 10 is constant
int y = 3 + 20;          // ok, a constant expression
size_t z = sizeof(int);  // ok, a constant expression
int x2 = 2 * x;          // not ok, x is a variable
```

(As long as the type is not a variable array, a `sizeof` expression is considered a constant expression.)

Using an External Variable

Let's look at a simple example that involves an external variable. Specifically, suppose you want two functions, call them `main()` and `critic()`, to have access to the variable `units`. You can do this by declaring `units` outside of and above the two functions, as shown in Listing 12.4. (Note: The intent of this example is to show how an external variable works, not to show a typical use.)

Listing 12.4 **The `global.c` Program**

```
/* global.c  -- uses an external variable */
#include <stdio.h>
int units = 0;         /* an external variable      */
void critic(void);
int main(void)
{
    extern int units;  /* an optional redeclaration */

    printf("How many pounds to a firkin of butter?\n");
    scanf("%d", &units);
    while ( units != 56)
        critic();
    printf("You must have looked it up!\n");

    return 0;
}

void critic(void)
{
    /* optional redeclaration omitted */
    printf("No luck, my friend. Try again.\n");
    scanf("%d", &units);
}
```

Here is some sample output:

```
How many pounds to a firkin of butter?
14
No luck, my friend. Try again.
56
You must have looked it up!
```

(We did.)

Note how the second value for `units` was read by the `critic()` function, yet `main()` also knew the new value when it finished the `while` loop. So both the `main()` function and the `critic()` function use the identifier `units` to access the same variable. In C terminology, we say that `units` has file scope, external linkage, and static storage duration.

We made `units` an external variable by defining it outside of (that is, external to) any function definition. That's all you need to do to make `units` available to all the subsequent functions in the file.

Let's look at some of the details. First, declaring `units` where it is declared makes it available to the functions below it without any further action taken. Therefore, `critics()` uses the `units` variable.

Similarly, nothing needed to be done to give `main()` access to `units`. However, `main()` does have the following declaration in it:

```
extern int units;
```

In the example, this declaration is mainly a matter of documentation. The storage class specifier `extern` tells the compiler that any mention of `units` in this particular function refers to a variable defined outside the function, perhaps even outside the file. Again, both `main()` and `critic()` use the externally defined `units`.

External Names

The C99 and C11 standards require compilers to recognize the first 63 characters for local identifiers and the first 31 characters for external identifiers. This revises the previous requirement of recognizing the first 31 characters for local identifiers and the first six characters for external identifiers. It's possible that you may be working with the old rules. The reason the rules for names of external variables are more restrictive than for local variables is that external names need to comply with the rules of the local environment, which may be more limiting.

Definitions and Declarations

Let's take a longer look at the difference between defining a variable and declaring it. Consider the following example:

```
int tern = 1;            /* tern defined                */
main()
{
     external int tern;  /* use a tern defined elsewhere */
```

Here, `tern` is declared twice. The first declaration causes storage to be set aside for the variable. It constitutes a definition of the variable. The second declaration merely tells the compiler to use the `tern` variable that has been created previously, so it is not a definition. The first declaration is called a *defining declaration*, and the second is called a *referencing declaration.* The keyword `extern` indicates that a declaration is not a definition because it instructs the compiler to look elsewhere.

Suppose you do this:

```
extern int tern;
int main(void)
{
```

The compiler will assume that the actual definition of `tern` is somewhere else in your program, perhaps in another file. This declaration does not cause space to be allocated. Therefore, don't use the keyword `extern` to create an external definition; use it only to *refer* to an existing external definition.

An external variable can be initialized only once, and that must occur when the variable is defined. Suppose you have this:

```
// file one.c
char permis = 'N';
...
// file two.c
extern char permis = 'Y';   /* error */
```

This is an error because the defining declaration in `file_one.c` already has created and initialized `permis`.

Static Variables with Internal Linkage

Variables of this storage class have static storage duration, file scope, and internal linkage. You create one by defining it outside of any function (just as with an external variable) with the storage class specifier `static`:

```
static int svil = 1;  // static variable, internal linkage
int main(void)
{
```

Such variables were once termed *external static* variables, but that's a bit confusing because they have internal linkage. Unfortunately, no new compact term has taken the place of *external static*, so we're left with *static variable with internal linkage*. The ordinary external variable can be used by functions in any file that's part of the program, but the static variable with internal linkage can be used only by functions in the same file. You can redeclare any file scope variable within a function by using the storage class specifier `extern`. Such a declaration doesn't change the linkage. Consider the following code:

```
int traveler = 1;        // external linkage
static int stayhome = 1; // internal linkage
int main()
{
    extern int traveler;  // use global traveler
    extern int stayhome;  // use global stayhome
    ...
```

Both `traveler` and `stayhome` are global for this particular translation unit, but only `traveler` can be used by code in other translation units. The two declarations using `extern` document that `main()` is using the two global variables, but `stayhome` continues to have internal linkage.

Multiple Files

The difference between internal linkage and external linkage is important only when you have a program built from multiple translation units, so let's take a quick look at that topic.

Complex C programs often use several separate files of source code. Sometimes these files might need to share an external variable. The C way to do this is to have a defining declaration in one file and referencing declarations in the other files. That is, all but one declaration (the defining declaration) should use the `extern` keyword, and only the defining declaration should be used to initialize the variable.

Note that an external variable defined in one file is not available to a second file unless it is also declared (by using `extern`) in the second file. An external declaration by itself only makes a variable potentially available to other files.

Historically, however, many compilers have followed different rules in this regard. Many Unix systems, for example, enable you to declare a variable in several files without using the `extern` keyword, provided that no more than one declaration includes an initialization. If there is a declaration with an initialization, it is taken to be the definition.

Storage-Class Specifier Roundup

You may have noticed that the meaning of the keywords `static` and `extern` depends on the context. The C language has six keywords that are grouped together as storage-class specifiers. They are `auto`, `register`, `static`, `extern`, `_Thread_local`, and `typedef`. The `typedef` keyword doesn't say anything about memory storage, but it is thrown in for syntax reasons. In particular, in most cases you can use no more than one storage-class specifier in a declaration, so that means you can't use one of the other storage-class specifiers as part of a `typedef`. The one exception is that `_Thread_local` may be used together with `static` and `extern`.

The `auto` specifier indicates a variable with automatic storage duration. It can be used only in declarations of variables with block scope, which already have automatic storage duration, so its main use is documenting intent.

The `register` specifier also can be used only with variables of block scope. It puts a variable into the register storage class, which amounts to a request to minimize the access time for that variable. It also prevents you from taking the address of the variable.

The `static` specifier creates an object with static duration, one that's created when the program is loaded and ends when the program terminates. If `static` is used with a file scope declaration, scope is limited to that one file. If `static` is used with a block scope declaration, scope is limited to that block. Thus, the object exists and retains its value as long as the program is running, but it can be accessed by the identifier only when code within the block is being executed. A static variable with block scope has no linkage. A static variable with file scope has internal linkage.

The `extern` specifier indicates that you are declaring a variable that has been defined elsewhere. If the declaration containing `extern` has file scope, the variable referred to must have external linkage. If the declaration containing `extern` has block scope, the referred-to variable can have either external linkage or internal linkage, depending on the defining declaration for that variable.

Summary: Storage Classes

Automatic variables have block scope, no linking, and automatic storage duration. They are local and private to the block (typically a function) in which they are defined. Register variables have the same properties as automatic variables, but the compiler may use faster memory or a register to store them. You can't take the address of a register variable.

Variables with static storage duration can have external linkage, internal linkage, or no linkage. When a variable is declared external to any function in a file, it's an external variable and has file scope, external linkage, and static storage duration. If you add the keyword `static` to such a declaration, you get a variable with static storage duration, file scope, and internal linkage. If you declare a variable inside a function and use the keyword `static`, the variable has static storage duration, block scope, and no linkage.

Memory for a variable with automatic storage duration is allocated when program execution enters the block containing the variable declaration and is freed when the block is exited. If uninitialized, such a variable has a garbage value. Memory for a variable with static storage duration is allocated at compile time and lasts as long as the program runs. If uninitialized, such a variable is set to 0.

A variable with block scope is local to the block containing the declaration. A variable with file scope is known to all functions in a file (or translation unit) following its declaration. If a file scope variable has external linkage, it can be used by other translation units in the program. If a file scope variable has internal linkage, it can be used just within the file in which it is declared.

Here's a short program that uses all five storage classes. It's spread over two files (Listing 12.5 and Listing 12.6), so you will have to do a multiple-file compile. (See Chapter 9, "Functions," or your compiler manual for guidance.) Its main goal is to use all five storage types, not to offer a design model; a better design wouldn't need the file-scope variables.

Listing 12.5 **The `parta.c` File**

```
// parta.c --- various storage classes
// compile with partb.c
#include <stdio.h>
void report_count();
void accumulate(int k);
int count = 0;       // file scope, external linkage

int main(void)
{
    int value;       // automatic variable
    register int i;  // register variable

    printf("Enter a positive integer (0 to quit): ");
    while (scanf("%d", &value) == 1 && value > 0)
    {
        ++count;     // use file scope variable
        for (i = value; i >= 0; i--)
            accumulate(i);
        printf("Enter a positive integer (0 to quit): ");
    }
    report_count();

    return 0;
}

void report_count()
{
    printf("Loop executed %d times\n", count);
}
```

Listing 12.6 **The `partb.c` File**

```
// partb.c -- rest of the program
// compile with parta.c
#include <stdio.h>

extern int count;       // reference declaration, external linkage

static int total = 0;   // static definition, internal linkage
void accumulate(int k); // prototype


void accumulate(int k)  // k has block scope, no linkage
{
    static int subtotal = 0;  // static, no linkage
    if (k <= 0)
    {
        printf("loop cycle: %d\n", count);
        printf("subtotal: %d; total: %d\n", subtotal, total);
        subtotal = 0;
    }
    else
    {
        subtotal += k;
        total += k;
    }
}
```

In this program, the block scope static variable `subtotal` keeps a running subtotal of the values passed to the `accumulate()` function, and the file scope, internal linkage variable `total` keeps a running total. The `accumulate()` function reports `total` and `subtotal` whenever a nonpositive value is passed to it; when the function reports, it resets `subtotal` to 0. The `accumulate()` prototype in `parta.c` is mandatory because the file contains an `accumulate()` function call. For `partb.c`, the prototype is optional because the function is defined, but not called in that file. The function also uses the external variable `count` to keep track of how many times the `while` loop in `main()` has been executed. (Incidentally, this is a good example of how not to use an external variable, because it unnecessarily intertwines the code of `parta.c` with the code of `partb.c`.) In `parta.c`, `main()` and `report_count()` share access to `count`.

Here's a sample run:

```
Enter a positive integer (0 to quit): 5
loop cycle: 1
subtotal: 15; total: 15
Enter a positive integer (0 to quit): 10
loop cycle: 2
subtotal: 55; total: 70
Enter a positive integer (0 to quit): 2
loop cycle: 3
subtotal: 3; total: 73
Enter a positive integer (0 to quit): 0
Loop executed 3 times
```

Storage Classes and Functions

Functions, too, have storage classes. A function can be either external (the default) or static. (C99 adds a third possibility, the inline function, discussed in Chapter 16, "The C Preprocessor and the C Library.") An external function can be accessed by functions in other files, but a static function can be used only within the defining file. Consider, for example, a file containing these function prototypes:

```
double gamma(double);         /* external by default */
static double beta(int, int);
extern double delta(double, int);
```

The functions `gamma()` and `delta()` can be used by functions in other files that are part of the program, but `beta()` cannot. Because this `beta()` is restricted to one file, you can use a different function having the same name in the other files. One reason to use the `static` storage class is to create functions that are private to a particular module, thereby avoiding the possibility of name conflicts.

The usual practice is to use the `extern` keyword when declaring functions defined in other files. This practice is mostly a matter of clarity because a function declaration is assumed to be `extern` unless the keyword `static` is used.

Which Storage Class?

The answer to the question "Which storage class?" is most often "automatic." After all, why else was automatic selected as the default? Yes, we know that at first glance external storage is quite alluring. Just make all your variables external, and you never have to worry about using arguments and pointers to communicate between functions. There is a subtle pitfall, however. You will have to worry about function `A()` sneakily altering the variables used in function `B()`, despite your intentions to the contrary. The unquestionable evidence of untold years of collective computer experience is that this one subtle danger far outweighs the superficial attraction of using external storage indiscriminately.

One common exception are `const` data. Because they can't be altered, you don't have to worry about inadvertent alterations:

```
const int DAYS = 7;
const char * MSGS[3] = {"Yes", "No", Maybe"};
```

One of the golden rules of protective programming is the "need to know" principle. Keep the inner workings of each function as private to that function as possible, sharing only those variables that need to be shared. The other classes are useful, and they are available. Before using one, though, ask yourself whether it is necessary.

A Random-Number Function and a Static Variable

Now that you have some background on the different storage classes, let's look at a couple programs that use some of them. First, let's look at a function that makes use of a static variable with internal linkage: a random-number function. The ANSI C library provides the `rand()` function to generate random numbers. There are a variety of algorithms for generating random numbers, and ANSI C enables implementations to use the best algorithm for a particular machine. However, the ANSI C standard also supplies a standard, portable algorithm that produces the same random numbers on different systems. Actually, `rand()` is a "pseudorandom number generator," meaning that the actual sequence of numbers is predictable (computers are not known for their spontaneity), but the numbers are spread pretty uniformly over the possible range of values.

Instead of using your compiler's built-in `rand()` function, we'll use the portable ANSI version so that you can see what goes on inside. The scheme starts with a number called the "seed." The function uses the seed to produce a new number, which becomes the new seed. Then the new seed can be used to produce a newer seed, and so on. For this scheme to work, the random-number function must remember the seed it used the last time it was called. Aha! This calls for a static variable. Listing 12.7 is version 0. (Yes, version 1 comes soon.)

Listing 12.7 **The `rand0.c` Function File**

```
/* rand0.c -- produces random numbers           */
/*               uses ANSI C portable algorithm  */
static unsigned long int next = 1;  /* the seed  */

int rand0(void)
{
/* magic formula to generate pseudorandom number */
     next = next * 1103515245 + 12345;
     return (unsigned int) (next/65536) % 32768;
}
```

In Listing 12.7, the static variable `next` starts with the value `1` and is altered by the magic formula each time the function is called. The result is a return value somewhere in the range of `0` to `32767`. Note that `next` is static with internal linkage, rather than merely static with no linkage. That's because the example will be expanded later so that `next` is shared between two functions in the same file.

Let's try the `rand0()` function with the simple driver shown in Listing 12.8.

Listing 12.8 **The `r_drive0.c` Driver**

```
/* r_drive0.c -- test the rand0() function */
/* compile with rand0.c                    */
#include <stdio.h>
extern int rand0(void);

int main(void)
{
    int count;

    for (count = 0; count < 5; count++)
        printf("%d\n", rand0());

    return 0;
}
```

Here's another chance to practice using multiple files. Use one file for Listing 12.7 and one for Listing 12.8. The `extern` keyword reminds you that `rand0()` is defined in a separate file, but it's not required.

The output is this:

```
16838
5758
10113
17515
31051
```

The output looks random, but let's run it again. This time the result is as follows:

```
16838
5758
10113
17515
31051
```

Hmmm, that looks familiar; this is the "pseudo" aspect. Each time the main program is run, you start with the same seed of 1. You can get around this problem by introducing a second function called `srand1()` that enables you to reset the seed. The trick is to make `next` a static variable with internal linkage known only to `rand1()` and `srand1()`. (The C library equivalent to `srand1()` is called `srand()`.) Add `srand1()` to the file containing `rand1()`. Listing 12.9 is the modification.

Listing 12.9 **The `s_and_r.c` Program**

```
/* s_and_r.c -- file for rand1() and srand1()    */
/*                uses ANSI C portable algorithm */
static unsigned long int next = 1;  /* the seed  */

int rand1(void)
{
/* magic formula to generate pseudorandom number */
    next = next * 1103515245 + 12345;
    return (unsigned int) (next/65536) % 32768;
}

void srand1(unsigned int seed)
{
    next = seed;
}
```

Notice that `next` is a file-scope static variable with internal linkage. That means it can be used by both `rand1()` and `srand1()`, but not by functions in other files. To test these functions, use the driver in Listing 12.10.

Listing 12.10 **The `r_drive1.c` Program**

```
/* r_drive1.c -- test rand1() and srand1() */
/* compile with s_and_r.c                  */
#include <stdio.h>
extern void srand1(unsigned int x);
extern int rand1(void);

int main(void)
{
    int count;
    unsigned seed;

    printf("Please enter your choice for seed.\n");
    while (scanf("%u", &seed) == 1)
    {
        srand1(seed);    /* reset seed */
        for (count = 0; count < 5; count++)
            printf("%d\n", rand1());
        printf("Please enter next seed (q to quit):\n");
    }
    printf("Done\n");

    return 0;
}
```

Again, use two files, and run the program.

```
Please enter your choice for seed.
1
16838
5758
10113
17515
31051
Please enter next seed (q to quit):
513
20067
23475
8955
20841
15324
```

```
Please enter next seed (q to quit):
q
Done
```

Using a value of `1` for `seed` yields the same values as before, but a `seed` value of `3` gives new results.

Note Automated Reseeding

If your C implementation gives you access to some changing quantity, such as the system clock, you can use that value (possibly truncated) to initialize the seed value. For instance, ANSI C has a `time()` function that returns the system time. The time units are system dependent, but what matters here is that the return value is an arithmetic type and that its value changes with time. The exact type is system dependent and is given the label `time_t`, but you can use a type cast. Here's the basic setup:

```
#include <time.h>   /* ANSI prototype for time() */
    srand1((unsigned int) time(0));   /* initialize seed */
```

In general, `time()` takes an argument that is the address of a type `time_t` object. In that case, the time value is also stored at that address. However, you can pass the null pointer (`0`) as an argument, in which case the value is supplied only through the return value mechanism.

You can use the same technique with the standard ANSI C functions `srand()` and `rand()`. If you do use these functions, include the `stdlib.h` header file. In fact, now that you've seen how `srand1()` and `rand1()` use a static variable with internal linkage, you might as well use the versions your compiler supplies. We'll do that for the next example.

Roll 'Em

We are going to simulate that very popular random activity, dice-rolling. The most popular form of dice-rolling uses two six-sided dice, but there are other possibilities. Many adventure-fantasy games use all of the five geometrically possible dice: 4, 6, 8, 12, and 20 sides. Those clever ancient Greeks proved that there are but five regular solids having all faces the same shape and size, and these solids are the basis for the dice varieties. You could make dice with other numbers of sides, but the faces would not all be the same, so they wouldn't all necessarily have equal odds of turning up.

Computer calculations aren't limited by these geometric considerations, so we can devise an electronic die that has any number of sides. Let's start with six sides and then generalize.

We want a random number from 1 to 6. However, `rand()` produces an integer in the range 0 to `RAND_MAX`; `RAND_MAX` is defined in `stdlib.h`. It is typically `INT_MAX`. Therefore, we have some adjustments to make. Here's one approach:

1. Take the random number modulus 6. It produces an integer in the range 0 through 5.

2. Add 1. The new number is in the range 1 through 6.
3. To generalize, just replace the number 6 in step 1 by the number of sides.

The following code implements these ideas:

```
#include <stdlib.h>   /* for rand() */
int rollem(int sides)
{
    int roll;

    roll = rand() % sides + 1;
    return roll;
}
```

Let's get a bit more ambitious and ask for a function that lets you roll an arbitrary number of dice and returns the total count. Listing 12.11 does this.

Listing 12.11 **The `diceroll.c` File**

```
/* diceroll.c -- dice role simulation */
/* compile with mandydice.c           */
 #include "diceroll.h"
#include <stdio.h>
#include <stdlib.h>           /* for library rand()   */

int roll_count  = 0;          /* external linkage     */

static int rollem(int sides)  /* private to this file */
{
    int roll;

    roll = rand() % sides + 1;
    ++roll_count;             /* count function calls */

    return roll;
}

int roll_n_dice(int dice, int sides)
{
    int d;
    int total = 0;
    if (sides < 2)
    {
        printf("Need at least 2 sides.\n");
        return -2;
    }
    if (dice < 1)
    {
        printf("Need at least 1 die.\n");
        return -1;
    }

    for (d = 0; d < dice; d++)
        total += rollem(sides);

    return total;
}
```

This file adds some wrinkles. First, it turns `rollem()` into a function private to this file. It's there as a helper function for `roll_n_dice()`. Second, to illustrate how external linkage works, the file declares an external variable called `roll_count`. This variable keeps track of how many times the `rollem()` function is called. The example is a little contrived, but it shows how the external variable feature works.

Third, the file contains the following statement:

```
#include "diceroll.h"
```

When you use standard library functions, such as `rand()`, you include the standard header file (`stdlib.h` for `rand()`) instead of declaring the function. That's because the header file already contains the correct declaration. We'll emulate that approach by providing a `diceroll.h` header file to be used with the `roll_n_dice()` function. Enclosing the filename in double quotation marks instead of in angle brackets instructs the compiler to look locally for the file instead of in the standard locations the compiler uses for the standard header files. The meaning of "look locally" depends on the implementation. Some common interpretations are placing the header file in the same directory or folder as the source code files or in the same directory or folder as the project file (if your compiler uses them). Listing 12.12 shows the contents of the header file.

Listing 12.12 **The `diceroll.h` File**

```
//diceroll.h
extern int roll_count;

int roll_n_dice(int dice, int sides);
```

This header file contains function prototypes and an `extern` declaration. Because the `diceroll.c` file includes this header, `diceroll.c` actually contains two declarations of `roll_count`:

```
extern int roll_count;  // from header file
int roll_count = 0;     // from source code file
```

This is fine. You can have only one defining declaration of a variable. But the declaration with `extern` is a reference declaration, and you can have as many of those as you want.

The program using `roll_n_dice()` should also include this header file. Not only does this provide the prototype for `roll_n_dice()`, it also makes `roll_count` available to that program. Listing 12.13 illustrates these points.

Listing 12.13 **The `manydice.c` File**

```
/* manydice.c -- multiple dice rolls                    */
/* compile with diceroll.c                              */
#include <stdio.h>
#include <stdlib.h>              /* for library srand() */
#include <time.h>                /* for time()          */
#include "diceroll.h"            /* for roll_n_dice()   */
/* and for roll_count  */
int main(void)
{
    int dice,roll;
    int sides;

    srand((unsigned int) time(0)); /* randomize seed      */
    printf("Enter the number of sides per die, 0 to stop.\n");
    while (scanf("%d", &sides) == 1 && sides > 0 )
    {
        printf("How many dice?\n");
        if ((status =scanf("%d", &dice)) != 1)
        {
            if (status == EOF)
                break;             /* exit loop           */
            else
            {
                printf("You should have entered an integer.");
                printf(" Let's begin again.\n");
                while (getchar() != '\n')
                    continue;     /* dispose of bad input */
                printf("How many sides? Enter 0 to stop.\n");
                continue;         /* new loop cycle       */
             }
        }
       roll = roll_n_dice(dice, sides);
        printf("You have rolled a %d using %d %d-sided dice.\n",
               roll, dice, sides);
        printf("How many sides? Enter 0 to stop.\n");
    }
    printf("The rollem() function was called %d times.\n",
           roll_count);           /* use extern variable */
    printf("GOOD FORTUNE TO YOU!\n");

    return 0;
}
```

Compile Listing 12.13 with the file containing Listing 12.11. To simplify matters, have Listings 12.11, 12.12, and 12.13 all in the same folder or directory. Run the resulting program. The output should look something like this:

```
Enter the number of sides per die, 0 to stop.
6
How many dice?
2
You have rolled a 12 using 2 6-sided dice.
How many sides? Enter 0 to stop.
6
How many dice?
2
You have rolled a 4 using 2 6-sided dice.
How many sides? Enter 0 to stop.
6
How many dice?
2
You have rolled a 5 using 2 6-sided dice.
How many sides? Enter 0 to stop.
0
The rollem() function was called 6 times.
GOOD FORTUNE TO YOU!
```

Because the program uses `srand()` to randomize the random-number seed, you most likely won't get the same output even with the same input. Note that `main()` in `manydice.c` does have access to the `roll_count` variable defined in `diceroll.c`.

The outer `while` loop can terminate for three reasons: `sides` is less than 1, there is a type mismatch for input (`scanf()` return value is `0`), or end-of-file is encountered (return value is `EOF`). For reading the number of dice, the program handles end-of-file differently from how it handles a type mismatch; it exits the `while` loop in the former case and initiates a new loop cycle in the latter case.

You can use `roll_n_dice()` in many ways. With `sides` equal to 2, the program simulates a coin toss with "heads" being 2 and "tails" being 1 (or vice versa, if you really prefer it). You can easily modify the program to show the individual results as well as the total, or you can construct a craps simulator. If you require a large number of rolls, as in some role-playing games, you can easily modify the program to produce output like this:

```
Enter the number of sets; enter q to stop.
18
```

```
How many sides and how many dice?
6 3
Here are 18 sets of 3 6-sided throws.
  12  10   6   9   8  14   8  15   9  14  12  17  11   7  10
  13   8  14
How many sets? Enter q to stop.
q
```

Another use for `rand1()` or `rand()` (but not of `rollem()`) is creating a number-guessing program so that the computer chooses and you guess. You can try that yourself.

Allocated Memory: `malloc()` and `free()`

The storage classes we discussed have one thing in common. After you decide which storage class to use, the decisions about scope and storage duration follow automatically. Your choices obey the prepackaged memory management rules. There is, however, one more choice, one that gives you more flexibility. That choice is using library functions to allocate and manage memory.

First, let's review some facts about memory allocation. All programs have to set aside enough memory to store the data they use. Some of this memory allocation is done automatically. For example, you can declare

```
float x;
char place[] = "Dancing Oxen Creek";
```

and enough memory to store that `float` or `string` is set aside, or you can be more explicit and ask for a certain amount of memory:

```
int plates[100];
```

This declaration sets aside 100 memory locations, each fit to store an `int` value. In all these cases, the declaration also provides an identifier for the memory, so you can use `x` or `place` to identify data. Static data, recall, is allocated when the program is loaded into memory, and automatic data is allocated when program execution enters a block and deallocated when execution leaves the block.

C goes beyond this. You can allocate more memory as a program runs. The main tool is the `malloc()` function, which takes one argument: the number of bytes of memory you want. Then `malloc()` finds a suitable block of free memory. The memory is anonymous; that is, `malloc()` allocates memory but it doesn't assign a name to it. However, it does return the address of the first byte of that block. Therefore, you can assign that address to a pointer variable and use the pointer to access the memory. Because `char` represents a byte, `malloc()` has traditionally been defined as type pointer-to-`char`. Since the ANSI C standard, however, C uses a new type: pointer-to-`void`. This type is intended to be a "generic pointer." The `malloc()` function can be used to return pointers to arrays, structures, and so forth, so normally the return value is typecast to the proper value. Under ANSI C, you should still typecast for clarity, but assigning a pointer-to-`void` value to a pointer of another type is not considered a type clash. If `malloc()` fails to find the required space, it returns the null pointer.

Let's apply `malloc()` to the task of creating an array. You can use `malloc()` to request a block of storage as the program is running. You also need a pointer to keep track of where the block is in memory. For example, consider this code:

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

This code requests space for 30 type `double` values and sets `ptd` to point to the location. Note that `ptd` is declared as a pointer to a single `double` and not to a block of 30 `double` values. Remember that the name of an array is the address of its first element. Therefore, if you make `ptd` point to the first element of the block, you can use it just like an array name. That is, you can use the expression `ptd[0]` to access the first element of the block, `ptd[1]` to access the second element, and so on. As you've learned earlier, you can use pointer notation with array names, and you can use array notation with pointers.

You now have three ways to create an array:

- Declare an array using constant expressions for the array dimensions and use the array name to access elements. Such an array can be created using either static or automatic memory.
- Declare a variable-length array using variable expressions for the array dimensions and use the array name to access elements. (Recall that this is a C99 feature.) This feature is available only for automatic memory.
- Declare a pointer, call `malloc()`, assign the return value to the pointer, and use the pointer to access elements. The pointer can be either static or automatic.

You can use the second and third methods to do something you can't do with an ordinary declared array—create a *dynamic array*, one that's allocated while the program runs and that you can choose a size for while the program runs. Suppose, for example, that `n` is an integer variable. Prior to C99, you couldn't do the following:

```
double item[n];    /* pre C99: not allowed if n is a variable */
```

However, you can do the following, even with a pre-C99 compiler:

```
ptd = (double *) malloc(n * sizeof(double)); /* okay */
```

This works, and, as you'll see, it's a bit more flexible than the variable-length array.

Normally, you should balance each use of `malloc()` with a use of `free()`. The `free()` function takes as its argument an address returned earlier by `malloc()` and frees up the memory that had been allocated. Thus, the duration of allocated memory is from when `malloc()` is called to allocate the memory until `free()` is called to free up the memory so that it can be reused. Think of `malloc()` and `free()` as managing a pool of memory. Each call to `malloc()` allocates memory for program use, and each call to `free()` restores memory to the pool so it can be reused. The argument to `free()` should be a pointer to a block of memory allocated by `malloc()`; you can't use `free()` to free memory allocated by other means, such as declaring an array. Both `malloc()` and `free()` have prototypes in the `stdlib.h` header file.

By using `malloc()`, then, a program can decide what size array is needed and create it while the program runs. Listing 12.14 illustrates this possibility. It assigns the address of the block of memory to the pointer `ptd`, and then it uses `ptd` in the same fashion you would use an array name. Also, the `exit()` function, prototyped in `stdlib.h`, is called to terminate the program if memory allocation fails. The value `EXIT_FAILURE` also is defined in that header file. The standard provides for two return values that are guaranteed to work with all operating systems: `EXIT_SUCCESS` (or, equivalently, the value `0`) to indicate normal program termination, and `EXIT_FAILURE` to indicate abnormal termination. Some operating systems, including Unix, Linux, and Windows, can accept additional integer values denoting particular forms of failure.

Listing 12.14 **The `dyn_arr.c` Program**

```
/* dyn_arr.c -- dynamically allocated array */
#include <stdio.h>
#include <stdlib.h> /* for malloc(), free() */

int main(void)
{
    double * ptd;
    int max = 0;
    int number;
    int i = 0;

    puts("What is the maximum number of type double entries?");
    if (scanf("%d", &max) != 1)
    {
        puts("Number not correctly entered -- bye.");
        exit(EXIT_FAILURE);
    }
    ptd = (double *) malloc(max * sizeof (double));
    if (ptd == NULL)
    {
        puts("Memory allocation failed. Goodbye.");
        exit(EXIT_FAILURE);
    }
    /* ptd now points to an array of max elements */
    puts("Enter the values (q to quit):");
    while (i < max && scanf("%lf", &ptd[i]) == 1)
        ++i;
    printf("Here are your %d entries:\n", number = i);
    for (i = 0; i < number; i++)
    {
        printf("%7.2f ", ptd[i]);
        if (i % 7 == 6)
            putchar('\n');
    }
    if (i % 7 != 0)
        putchar('\n');
    puts("Done.");
    free(ptd);

    return 0;
}
```

Here's a sample run. In it, we entered six numbers, but the program processes just five of them because we limited the array size to 5.

```
What is the maximum number of entries?
5
Enter the values (q to quit):
20 30 35 25 40 80
Here are your 5 entries:
  20.00   30.00   35.00   25.00   40.00
Done.
```

Let's look at the code. The program finds the desired array size with the following lines:

```
if (scanf("%d", &max) != 1)
{
    puts("Number not correctly entered -- bye.");
    exit(EXIT_FAILURE);
}
```

Next, the following line allocates enough space to hold the requested number of entries and then assigns the address of the block to the pointer `ptd`:

```
ptd = (double *) malloc(max * sizeof (double));
```

The typecast to `(double *)` is optional in C but required in C++, so using the typecast makes it simpler to move a program from C to C++.

It's possible that `malloc()` can fail to procure the desired amount of memory. In that case, the function returns the null pointer, and the program terminates:

```
if (ptd == NULL)
{
     puts("Memory allocation failed. Goodbye.");
     exit(EXIT_FAILURE);
}
```

If the program clears this hurdle, it can treat `ptd` as though it were the name of an array of `max` elements, and so it does.

Note the free() function near the end of the program. It frees memory allocated by malloc(). The free() function frees only the block of memory to which its argument points. Some operating systems will free allocated memory automatically when a program finishes, but others may not. So use free() and don't rely on the operating system to clean up for you.

What have you gained by using a dynamic array? In this case, you've gained program flexibility. Suppose you know that most of the time the program will need no more than 100 elements, but sometimes it will need 10,000 elements. If you declare an array, you would have to allow for the worst case and declare it with 10,000 elements. Most of the time, that program would be wasting memory. Then, the one time you need 10,001 elements, the program will fail. You can use a dynamic array to adjust the program to fit the circumstances.

The Importance of free()

The amount of static memory is fixed at compile time; it does not change while the program is running. The amount of memory used for automatic variables grows and shrinks automatically as the program executes. But the amount of memory used for allocated memory just grows unless you remember to use free(). For example, suppose you have a function that creates a temporary copy of an array as sketched in the following code:

```
...
int main()
{
double glad[2000];
int i;
...for (i = 0; i < 1000; i++)
    gobble(glad, 2000);
...}

void gobble(double ar[], int n)
{
    double * temp = (double *) malloc( n * sizeof(double));
...    /* free(temp);  // forgot to use free()  */
}
```

The first time gobble() is called, it creates the pointer temp, and it uses malloc() to allocate 16,000 bytes of memory (assuming double is 8 bytes). Suppose, as indicated, we don't use free(). When the function terminates, the pointer temp, being an automatic variable, disappears. But the 16,000 bytes of memory it pointed to still exists. It can't be accessed because we no longer have the address. It can't be reused because we didn't call free().

The second time gobble() is called, it creates temp again, and again it uses malloc() to allocate 16,000 bytes. The first block of 16,000 bytes is no longer available, so malloc() has to find a second block of 16,000 bytes. When the function terminates, this block of memory also becomes inaccessible and not reusable.

But the loop executes 1,000 times, so by the time the loop finishes, 16,000,000 bytes of memory have been removed from the memory pool. In fact, the program may have run out of memory before getting this far. This sort of problem is called a *memory leak*, and it could have been prevented by having a call to `free()` at the end of the function.

The `calloc()` Function

Another option for memory allotment is to use `calloc()`. A typical use looks like this:

```
long * newmem;

newmem = (long *)calloc(100, sizeof (long));
```

Like `malloc()`, `calloc()` returns a pointer-to-`char` in its pre-ANSI version and a pointer-to-`void` under ANSI. You should use the cast operator if you want to store a different type. This new function takes two arguments, both of which should be unsigned integers (type `size_t` since ANSI). The first argument is the number of memory cells you want. The second argument is the size of each cell in bytes. In our case, `long` uses 4 bytes, so this instruction sets up 100 4-byte units, using 400 bytes in all for storage.

Using `sizeof (long)` instead of `4` makes this coding more portable. It will work on those systems where `long` is some size other than 4.

The `calloc()` function throws in one more feature: It sets all the bits in the block to zero. (Note, however, that on some hardware systems, a floating-point value of `0` is not represented by all bits set to 0.)

The `free()` function can also be used to free memory allocated by `calloc()`.

Dynamic memory allocation is the key to many advanced programming techniques. We'll examine some in Chapter 17, "Advanced Data Representation." Your own C library probably offers several other memory-management functions—some portable, some not. You might want to take a moment to look them over.

Dynamic Memory Allocation and Variable-Length Arrays

There's some overlap in functionality between variable-length arrays (VLAs) and the use of `malloc()`. Both, for example, can be used to create an array whose size is determined during runtime:

```
int vlamal()
{
    int n;
    int * pi;
    scanf("%d", &n);
    pi = (int *) malloc (n * sizeof(int));
    int ar[n];   // vla
    pi[2] = ar[2] = -5;
...
}
```

One difference is that the VLA is automatic storage. One consequence of automatic storage is that the memory space used by the VLA is freed automatically when the execution leaves the defining block—in this case, when the `vlamal()` function terminates. Therefore, you don't have to worry about using `free()`. On the other hand, the array created using `malloc()` needn't have its access limited to one function. For example, one function could create an array and return the pointer, giving the calling function access. Then the calling function could call `free()` when it is finished. It's okay to use a different pointer variable with `free()` than with `malloc()`; what must agree are the addresses stored in the pointers. However, you should not try to free the same block of memory twice.

VLAs are more convenient for multidimensional arrays. You can create a two-dimensional array using `malloc()`, but the syntax is awkward. If a compiler doesn't support the VLA feature, one of the dimensions has to be fixed, just like in function calls:

```
int n = 5;
int m = 6;
int ar2[n][m];     // n x m VLA
int (* p2)[6];     // works pre-C99
int (* p3)[m];     // requires VLA support
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int));  // n * 6 array
p3 = (int (*)[m]) malloc(n * m * sizeof(int));  // n * m array
// above expression also requires VLA support
ar2[1][2] = p2[1][2] = 12;
```

It's worth reviewing the pointer declarations. The `malloc()` function returns a pointer, so `p2` has to be a pointer of a suitable type. The declaration

```
int (* p2)[6];    // works pre-C99
```

says that `p2` points to an array of six `int`s. This means that `p2[i]` would be interpreted as an element consisting of six `int`s and that `p2[i][j]` would be a single `int`.

The second pointer declaration uses a variable to specify the size of the array to which `p3` points. This means that `p3` is considered to be a pointer to a VLA, which is why the code won't work with the C90 standard.

Storage Classes and Dynamic Memory Allocation

You might be wondering about the connection between storage classes and dynamic memory allocation. Let's look at an idealized model. You can think of a program as dividing its available memory into three separate sections: one for static variables with external linkage, internal linkage, and no linkage; one for automatic variables; and one for dynamically allocated memory.

The amount of memory needed for the static duration storage classes is known at compile time, and the data stored in this section is available as long as the program runs. Each variable of these classes comes into being when the program starts and expires when the program ends.

An automatic variable, however, comes into existence when a program enters the block of code containing the variable's definition and expires when its block of code is exited. Therefore, as a program calls functions and as functions terminate, the amount of memory used by automatic variables grows and shrinks. This section of memory is typically handled as a stack. That means new variables are added sequentially in memory as they are created and then are removed in the opposite order as they pass away.

Dynamically allocated memory comes into existence when `malloc()` or a related function is called, and it's freed when `free()` is called. Memory persistence is controlled by the programmer, not by a set of rigid rules, so a memory block can be created in one function and disposed of in another function. Because of this, the section of memory used for dynamic memory allocation can end up fragmented—that is, unused chunks could be interspersed among active blocks of memory. Also, using dynamic memory tends to be a slower process than using stack memory.

Typically, a program uses different regions of memory for static objects, automatic objects, and dynamically allocated objects. Listing 12.15 illustrates this point.

Listing 12.15 **The `where.c` Program**

```
//  where.c  -- where's the memory?

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int static_store = 30;
const char * pcg = "String Literal";
int main()
{
    int auto_store = 40;
    char auto_string[] = "Auto char Array";
    int * pi;
    char * pcl;

    pi = (int *) malloc(sizeof(int));
    *pi = 35;
    pcl = (char *) malloc(strlen("Dynamic String") + 1);
    strcpy(pcl, "Dynamic String");

    printf("static_store: %d at %p\n", static_store, &static_store);
    printf("  auto_store: %d at %p\n", auto_store, &auto_store);
    printf("         *pi: %d at %p\n", *pi, pi);
    printf("  %s at %p\n", pcg, pcg);
    printf(" %s at %p\n", auto_string, auto_string);
    printf("  %s at %p\n", pcl, pcl);
    printf("   %s at %p\n", "Quoted String", "Quoted String");
    free(pi);
    free(pcl);

    return 0;
}
```

Here is the output for one system:

```
static_store: 30 at 00378000
  auto_store: 40 at 0049FB8C
         *pi: 35 at 008E9BA0
  String Literal at 00375858
 Auto char Array at 0049FB74
  Dynamic String at 008E9BD0
   Quoted String at 00375908
```

As you can see, static data, including string literals occupies one region, automatic data a second region, and dynamically allocated data a third region (often called a *memory heap* or *free store*).

ANSI C Type Qualifiers

You've seen that a variable is characterized by both its type and its storage class. C90 added two more properties: constancy and volatility. These properties are declared with the keywords `const` and `volatile`, which create *qualified types*. The C99 standard added a third qualifier, `restrict`, designed to facilitate compiler optimizations. And C11 adds a fourth, `_Atomic`. C11 provides an optional library, managed by `stdatomic.h`, to support concurrent programming, and `_Atomic` is part of that optional support.

C99 granted type qualifiers a new property—they now are idempotent! Although this sounds like a powerful claim, all it really means is that you can use the same qualifier more than once in a declaration, and the superfluous ones are ignored:

```
const const const int n = 6; // same as const int n = 6;
```

This makes it possible, for example, for the following sequence to be accepted:

```
typedef const int zip;
const zip q = 8;
```

The const Type Qualifier

Chapter 4, “Character Strings and Formatted Input/Output,” and Chapter 10, “Arrays and Pointers,” have already introduced const. To review, the const keyword in a declaration establishes a variable whose value cannot be modified by assignment or by incrementing or decrementing. On an ANSI-compliant compiler, the code

```
const int nochange;   /* qualifies m as being constant */
nochange = 12;        /* not allowed                   */
```

should produce an error message. You can, however, initialize a const variable. Therefore, the following code is fine:

```
const int nochange = 12;  /* ok */
```

The preceding declaration makes nochange a read-only variable. After it is initialized, it cannot be changed.

You can use the const keyword to, for example, create an array of data that the program can’t alter:

```
const int days1[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Using const with Pointers and Parameter Declarations

Using the const keyword when declaring a simple variable and an array is pretty easy. Pointers are more complicated because you have to distinguish between making the pointer itself const and making the value that is pointed to const. The declaration

```
const float * pf;  /* pf points to a constant float value */
```

establishes that pf points to a value that must remain constant. The value of pf itself can be changed. For example, it can be set to point at another const value. In contrast, the declaration

```
float * const pt;    /* pt is a const pointer */
```

says that the pointer pt itself cannot have its value changed. It must always point to the same address, but the pointed-to value can change. Finally, the declaration

```
const float * const ptr;
```

means both that ptr must always point to the same location and that the value stored at the location must not change.

There is a third location in which you can place const:

```
float const * pfc;   // same as const float * pfc;
```

As the comment indicates, placing const after the type name and before the * means that the pointer can’t be used to change the pointed-to value. In short, a const anywhere to the left of the * makes the data constant; and a const to the right of the * makes the pointer itself constant.

One common use for this new keyword is declaring pointers that serve as formal function parameters. For example, suppose you have a function called display() that displays the contents of an array. To use it, you would pass the name of the array as an actual argument, but the name of an array is an address. That would enable the function to alter data in the calling function. But the following prototype prevents this from happening:

```
void display(const int array[], int limit);
```

In a prototype and a function header, the parameter declaration const int array[] is the same as const int * array, so the declaration says that the data to which array points cannot be changed.

The ANSI C library follows this practice. If a pointer is used only to give a function access to values, the pointer is declared as a pointer to a const-qualified type. If the pointer is used to alter data in the calling function, the const keyword isn't used. For example, the ANSI C declaration for strcat() is this:

```
char *strcat(char * restrict s1, const char * restrict s2);
```

Recall that strcat() adds a copy of the second string to the end of the first string. This modifies the first string, but leaves the second string unchanged. The declaration reflects this. We'll get back to the role of restrict in a short while.

Using const with Global Data

Recall that using global variables is considered a risky approach because it exposes data to being mistakenly altered by any part of a program. That risk disappears if the data is constant, so it is perfectly reasonable to use global variables with the const qualifier. You can have const variables, const arrays, and const structures. (Structures are a compound data type discussed in the next chapter.)

One area that requires care, however, is sharing const data across files. There are two strategies you can use. The first is to follow the usual rules for external variables—use defining declarations in one file and reference declarations (using the keyword extern) in the other files:

```
/* file1.c -- defines some global constants */
const double PI = 3.14159;
const char * MONTHS[12] =
     {"January", "February", "March", "April", "May", "June", "July",
      "August", "September", "October", "November", "December"};

/* file2.c -- use global constants defined elsewhere */
extern const double PI;
extern const * MONTHS[];
```

The second approach is to place the constants in an `include` file. Here, you must take the additional step of using the static external storage class:

```
/* constant.h -- defines some global constants */
static const double PI = 3.14159;
static const char * MONTHS[12] =
     {"January", "February", "March", "April", "May", "June", "July",
      "August", "September", "October", "November", "December"};

/* file1.c -- use global constants defined elsewhere */
#include "constant.h"

/* file2.c -- use global constants defined elsewhere */
#include "constant.h"
```

If you don't use the keyword `static`, including `constant.h` in `file1.c` and in `file2.c` would result in each file having a defining declaration of the same identifier, which is not supported by the C standard. (Some compilers, however, do allow it.) By making each identifier static external, you actually give each file a separate copy of the data. That wouldn't work if the files are supposed to use the data to communicate with one another because each file would see only its own copy. Because the data is constant (by using the `const` keyword) and identical (by having both files include the same header file), however, that's not a problem.

The advantage of the header file approach is that you don't have to remember to use defining declarations in one file and reference declarations in the next; all files simply include the same header file. The disadvantage is that the data is duplicated. For the preceding examples, that's not a real problem, but it might be one if your constant data includes enormous arrays.

The `volatile` Type Qualifier

The `volatile` qualifier tells the compiler that a variable can have its value altered by agencies other than the program. It is typically used for hardware addresses and for data shared with other programs or threads running simultaneously. For example, an address might hold the current clock time. The value at that address changes as time changes, regardless of what your program is doing. Or an address could be used to receive information transmitted from, say, another computer.

The syntax is the same as for `const`:

```
volatile int loc1;   /* loc1 is a volatile location        */
volatile int * ploc; /* ploc points to a volatile location */
```

These statements declare `loc1` to be a `volatile` value and `ploc` to point to a `volatile` value.

You may think that `volatile` is an interesting concept, but you might be wondering why the ANSI committee felt it necessary to make `volatile` a keyword. The reason is that it facilitates compiler optimization. Suppose, for example, you have code like this:

```
val1 = x;
 /* some code not using x */
val2 = x;
```

A smart (optimizing) compiler might notice that you use `x` twice without changing its value. It would temporarily store the `x` value in a register. Then, when `x` is needed for `val2`, it can save time by reading the value from a register instead of from the original memory location. This procedure is called *caching*. Ordinarily, caching is a good optimization, but not if `x` is changed between the two statements by some other agency. If there were no `volatile` keyword, a compiler would have no way of knowing whether this might happen. Therefore, to be safe, the compiler couldn't cache. That was the pre-ANSI situation. Now, however, if the `volatile` keyword is not used in the declaration, the compiler can assume that a value hasn't changed between uses, and it can then attempt to optimize the code.

A value can be both `const` and `volatile`. For example, the hardware clock setting normally should not be changed by the program, making it `const`, but it is changed by an agency other than the program, making it `volatile`. Just use both qualifiers in the declaration, as shown here; the order doesn't matter:

```
volatile const int loc;
const volatile int * ploc;
```

The `restrict` Type Qualifier

The `restrict` keyword enhances computational support by giving the compiler permission to optimize certain kinds of code. It can be applied only to pointers, and it indicates that a pointer is the sole initial means of accessing a data object. To see why this is useful, we need to look at a few examples. Consider the following:

```
int ar[10];
int * restrict restar = (int *) malloc(10 * sizeof(int));
int * par = ar;
```

Here, the pointer `restar` is the sole initial means of access to the memory allocated by `malloc()`. Therefore, it can be qualified with the keyword `restrict`. The pointer `par`, however, is neither the initial nor the sole means of access to the data in the `ar` array, so it cannot be qualified as `restrict`.

Now consider the following rather artificial example, in which `n` is an `int`:

```
for (n = 0; n < 10; n++)
{
    par[n] += 5;
    restar[n] += 5;
    ar[n] *= 2;
    par[n] += 3;
    restar[n] += 3;
}
```

Knowing that `restar` is the sole initial means of access to the block of data it points to, the compiler can replace the two statements involving `restar` with a single statement having the same effect:

```
restar[n] += 8;   /* ok replacement */
```

It would be a computational error, however, to condense the two statements involving `par` into one:

```
par[n] += 8;    / * gives wrong answer */
```

The reason it gives the wrong answer is that the loop uses `ar` to change the value of the data between the two times `par` accesses the same data.

Without the `restrict` keyword, the compiler has to assume the worse case; namely, that some other identifier might have changed the data in between two uses of a pointer. With the `restrict` keyword used, the compiler is free to look for computational shortcuts.

You can use the `restrict` keyword as a qualifier for function parameters that are pointers. This means that the compiler can assume that no other identifiers modify the pointed-to data within the body of the function and that the compiler can try optimizations it might not otherwise use. For example, the C library has two functions for copying bytes from one location to another. Under C99, they have these prototypes:

```
void * memcpy(void * restrict s1, const void * restrict s2, size_t n);
void * memmove(void * s1, const void * s2, size_t n);
```

Each one copies `n` bytes from location `s2` to location `s1`. The `memcpy()` function requires that there be no overlap between the two locations, but `memmove()` doesn't have that requirement. Declaring `s1` and `s2` as `restrict` means each pointer is a sole means of access, so they can't access the same block of data. This matches the requirement that there be no overlap. The `memmove()` function, which does allow overlap, has to be more careful about copying data so that it doesn't overwrite data before it is used.

The keyword `restrict` has two audiences. One is the compiler, and it tells the compiler it is free to make certain assumptions concerning optimization. The other audience is the user, and it tells the user to use only arguments that satisfy the `restrict` requirements. In general, the compiler can't check whether you obey this restriction, but you flout it at your own risk.

The `_Atomic` Type Qualifier (C11)

Concurrent programming divides program execution into threads that may be executed in parallel. This creates several programming challenges, including how to manage different threads that access the same data. C11 provides, as an option and not a requirement, management methods set up by the optional header files `stdatomic.h` and `threads.h`. One aspect is the concept of an atomic type for which access is controlled by various macro functions. While a thread performs an atomic operation on an object of atomic type, other threads won't access that object. For instance, something like

```
int hogs;   // regular declaration
hogs = 12;   // regular assignment
```

could be replaced by the following:

```
_Atomic int hogs;         // hogs an atomic variable
atomic_store(&hogs, 12);  // macro from stdatomic.h
```

Here, the storing of the value `12` in `hogs` is an atomic process during which other threads won't access `hogs`.

At the time of this writing, compiler support for this feature is anticipated.

New Places for Old Keywords

C99 allows you to place the type qualifiers and the storage class qualifier `static` inside the initial brackets of a formal parameter in a function prototype and function header. In the case of the type qualifiers, this provides an alternative syntax for an existing capability. For example, here is a declaration with the older syntax:

```
void ofmouth(int * const a1, int * restrict a2, int n);  // older style
```

It says that `a1` is a `const` pointer to `int`, which, as you'll recall, means that the pointer is constant, not the data to which it points. It also indicates that `a2` is a restricted pointer, as described in the preceding section. The new and equivalent syntax is

```
void ofmouth(int a1[const], int a2[restrict], int n);   // allowed by C99
```

Basically, the new rule allows you to use these two qualifiers with either pointer or array notation in declaring function parameters.

The case for `static` is different because it introduces a new and unrelated use for this keyword. Instead of indicating the scope or linkage of a static storage variable, the new use is to tell the compiler how a formal parameter will be used. For example, consider this prototype:

```
double stick(double ar[static 20]);
```

This use of `static` indicates that the actual argument in a function call will be a pointer to the first element of an array having at least 20 elements. The purpose of this is to enable the compiler to use that information to optimize its coding of the function. Why use the keyword in such a different fashion? The C standards committee is reluctant to create a new keyword because that would invalidate old programs that use that word as an identifier, so if they can squeeze a new use out of an old keyword, they will.

As with `restrict`, the keyword `static` has two audiences. One is the compiler, and it tells the compiler it is free to make certain assumptions concerning optimization. The other audience is the user, and it tells the user to only provide arguments that satisfy the `static` requirements.

Key Concepts

C provides several models for managing memory. You should become familiar with the various choices. You also need to develop a sense of when to choose the various types. Most of the time, the automatic variable is the best choice. If you decide to use another type, you should have a good reason. For communicating between functions, it's usually better to use automatic variables, function parameters, and return values rather than global variables. On the other hand, global variables are particularly useful for constant data.

You should try to understand the properties of static memory, automatic memory, and allocated memory. In particular, be aware that the amount of static memory used is determined at compile time, and that static data is loaded into memory when the program is loaded into memory. Automatic variables are allocated and freed as the program runs, so the amount of memory used by automatic variables changes while a program executes. You can think of automatic memory as a rewriteable workspace. Allocated memory also grows and shrinks, but, in this case, the process is controlled by function calls rather than happening automatically.

Summary

The memory used to store data in a program can be characterized by storage duration, scope, and linkage. Storage duration can be static, automatic, or allocated. If static, memory is allocated at the start of program execution and persists as long as the program is running. If automatic, memory for a variable is allocated when program execution enters the block in which the variable is defined and is freed when the block is exited. If allocated, memory is allocated by calling `malloc()` (or a related function) and freed by calling the `free()` function.

Scope determines which parts of a program can access the data. A variable defined outside of any function has file scope and is visible to any function defined after the variable's declaration. A variable defined inside a block or as a function parameter has block scope and is visible just in that block and any blocks nested in it.

Linkage describes the extent to which a variable defined in one unit of a program can be linked to elsewhere. Variables with block scope, being local, have no linkage. Variables with file scope can have internal linkage or external linkage. Internal linkage means the variable can be used only in the file containing the definition. External linkage means the variable also can be used in other files.

The following are C's five storage classes (excluding thread concepts):

- **Automatic—**A variable declared in a block (or as a parameter in a function header) with no storage class modifier, or with the `auto` storage class modifier, belongs to the automatic storage class. It has automatic storage duration, block scope, and no linkage. Its value, if uninitialized, is not undetermined.
- **Register—**A variable declared in a block (or as a parameter in a function header) with the `register` storage class modifier belongs to the register storage class. It has automatic storage duration, block scope, and no linkage, and its address cannot be taken. Declaring

a variable as a register variable is a hint to the compiler to provide the fastest access possible. Its value, if uninitialized, is not undetermined.

- **Static, no linkage—**A variable declared in a block with the `static` storage class modifier belongs to the "static, no linkage" storage class. It has static storage duration, block scope, and no linkage. It is initialized just once, at compile time. If not initialized explicitly, its bytes are set to 0.
- **Static, external linkage—**A variable defined external to any function and without using the `static` storage class modifier belongs to the "static, external linkage" storage class. It has static storage duration, file scope, and external linkage. It is initialized just once, at compile time. If not initialized explicitly, its bytes are set to 0.
- **Static, internal linkage—**A variable defined external to any function and using the `static` storage class modifier belongs to the "static, internal linkage" storage class. It has static storage duration, file scope, and internal linkage. It is initialized just once, at compile time. If not initialized explicitly, its bytes are set to 0.

Allocated memory is provided by using the `malloc()` (or related) function, which returns a pointer to a block of memory having the requested number of bytes. This memory can be made available for reuse by calling the `free()` function, using the address as the argument.

The type qualifiers are `const`, `volatile`, and `restrict`. The `const` specifier qualifies data as being constant. When used with pointers, `const` can indicate that the pointer itself is constant or that the data it points to is constant, depending on the placement of `const` in the declaration. The `volatile` specifier indicates that data may be altered by processes other than the program. Its purpose is to warn the compiler to avoid optimizations that assume otherwise. The `restrict` specifier is also provided for reasons of optimization. A pointer qualified with `restrict` is identified as providing the only access to a block of data.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. Which storage classes create variables local to the function containing them?
2. Which storage classes create variables that persist for the duration of the containing program?
3. Which storage class creates variables that can be used across several files? Restricted to just one file?
4. What kind of linkage do block scope variables have?
5. What is the `extern` keyword used for?

6. Consider this code fragment:

```
int * p1 = (int *) malloc(100 * sizeof(int));
```

In terms of the final outcome, how does the following statement differ?

```
int * p1 = (int *) calloc(100, sizeof(int));
```

7. Which functions know each variable in the following? Are there any errors?

```
/* file 1 */
int daisy;
int main(void)
{
  int lily;
  ...;
}
int petal()
{
  extern int daisy, lily;
  ...;
}
/* file 2 */
extern int daisy;
static int lily;
int rose;
int stem()
{
  int rose;
  ...;
}
void root()
{
  ...;
}
```

8. What will the following program print?

```
#include <stdio.h>
char color= 'B';
void first(void);
void second(void);

int main(void)
{
  extern char color;

  printf("color in main() is %c\n", color);
  first();
  printf("color in main() is %c\n", color);
  second();
  printf("color in main() is %c\n", color);
  return 0;
}

void first(void)
{
  char color;

  color = 'R';
  printf("color in first() is %c\n", color);
}

void second(void)
{
  color = 'G';
  printf("color in second() is %c\n", color);
}
```

9. A file begins with the following declarations:

```
static int plink;
int value_ct(const int arr[], int value, int n);
```

   a. What do these declarations tell you about the programmer's intent?
   b. Will replacing `int value` and `int n` with `const int value` and `const int n` enhance the protection of values in the calling program?

Programming Exercises

1. Rewrite the program in Listing 12.4 so that it does not use global variables.

2. Gasoline consumption commonly is computed in miles per gallon in the U.S. and in liters per 100 kilometers in Europe. What follows is part of a program that asks the user to choose a mode (metric or U.S.) and then gathers data and computes fuel consumption:

```
// pe12-2b.c
// compile with pe12-2a.c
#include <stdio.h>
#include "pe12-2a.h"
int main(void)
{
  int mode;

  printf("Enter 0 for metric mode, 1 for US mode: ");
  scanf("%d", &mode);
  while (mode >= 0)
  {
     set_mode(mode);
     get_info();
     show_info();
     printf("Enter 0 for metric mode, 1 for US mode");
     printf(" (-1 to quit): ");
     scanf("%d", &mode);
  }
  printf("Done.\n");
  return 0;
}
```

Here is some sample output:

```
Enter 0 for metric mode, 1 for US mode: 0
Enter distance traveled in kilometers: 600
Enter fuel consumed in liters: 78.8
Fuel consumption is 13.13 liters per 100 km.
Enter 0 for metric mode, 1 for US mode (-1 to quit): 1
Enter distance traveled in miles: 434
Enter fuel consumed  in gallons: 12.7
Fuel consumption is 34.2 miles per gallon.
Enter 0 for metric mode, 1 for US mode (-1 to quit): 3
Invalid mode specified. Mode 1(US) used.
Enter distance traveled in miles: 388
Enter fuel consumed  in gallons: 15.3
Fuel consumption is 25.4 miles per gallon.
Enter 0 for metric mode, 1 for US mode (-1 to quit): -1
Done.
```

If the user enters an incorrect mode, the program comments on that and uses the most recent mode. Supply a `pe12-2a.h` header file and a `pe12-2a.c` source code file to make this work. The source code file should define three file-scope, internal-linkage variables. One represents the mode, one represents the distance, and one represents the fuel consumed. The `get_info()` function prompts for data according to the mode setting and stores the responses in the file-scope variables. The `show_info()` function calculates and displays the fuel consumption based on the mode setting. You can assume the user responds with numeric input.

3. Redesign the program described in Programming Exercise 2 so that it uses only automatic variables. Have the program offer the same user interface—that is, it should prompt the

user to enter a mode, and so on. You'll have to come up with a different set of function calls, however.

4. Write and test in a loop a function that returns the number of times it has been called.

5. Write a program that generates a list of 100 random numbers in the range 1–10 in sorted decreasing order. (You can adapt the sorting algorithm from Chapter 11, "Character Strings and String Functions," to type `int`. In this case, just sort the numbers themselves.)

6. Write a program that generates 1,000 random numbers in the range 1–10. Don't save or print the numbers, but do print how many times each number was produced. Have the program do this for 10 different seed values. Do the numbers appear in equal amounts? You can use the functions from this chapter or the ANSI C `rand()` and `srand()` functions, which follow the same format that our functions do. This is one way to examine the randomness of a particular random-number generator.

7. Write a program that behaves like the modification of Listing 12.13, which we discussed after showing the output of Listing 12.13. That is, have the program produce output like the following:

```
Enter the number of sets; enter q to stop: 18
How many sides and how many dice? 6 3
Here are 18 sets of 3 6-sided throws.
  12  10   6   9   8  14   8  15   9  14  12  17  11   7  10
  13   8  14
How many sets? Enter q to stop: q
```

8. Here's part of a program:

```
// pe12-8.c
#include <stdio.h>
int * make_array(int elem, int val);
void show_array(const int ar[], int n);
int main(void)
{
  int * pa;
  int size;
  int value;

  printf("Enter the number of elements: ");
  while (scanf("%d", &size) == 1 && size > 0)
  {
      printf("Enter the initialization value: ");
      scanf("%d", &value);
```

```
            pa = make_array(size, value);
            if (pa)
            {
                show_array(pa, size);
                free(pa);
            }
            printf("Enter the number of elements (<1 to quit): ");
        }
        printf("Done.\n");
        return 0;
    }
```

Complete the program by providing function definitions for `make_array()` and `show_array()`. The `make_array()` function takes two arguments. The first is the number of elements of an `int` array, and the second is a value that is to be assigned to each element. The function uses `malloc()` to create an array of a suitable size, sets each element to the indicated value, and returns a pointer to the array. The `show_array()` function displays the contents, eight numbers to a line.

9. Write a program with the following behavior. First, it asks you how many words you wish to enter. Then it has you enter the words, and then it displays the words. Use `malloc()` and the answer to the first question (the number of words) to create a dynamic array of the corresponding number of pointers-to-`char`. (Note that because each element in the array is a pointer-to-`char`, the pointer used to store the return value of `malloc()` should be a pointer-to-a-pointer-to-`char`.) When reading the string, the program should read the word into a temporary array of `char`, use `malloc()` to allocate enough storage to hold the word, and store the address in the array of `char` pointers. Then it should copy the word from the temporary array into the allocated storage. Thus, you wind up with an array of character pointers, each pointing to an object of the precise size needed to store the particular word. A sample run could look like this:

```
How many words do you wish to enter? 5
Enter 5 words now:
I enjoyed doing this exerise
Here are your words:
I
enjoyed
doing
this
exercise
```