Understanding Memory Allocation and Lifetime

Every instance of a value—be it a literal, an intrinsic data type, or a complex data type—exists in memory. Here, we will explore various ways in which memory is allocated. The different mechanisms for memory allocation are called **storage classes**. In this chapter, we will review the storage class we've been using thus far, that of *automatic* storage, as well as introduce the *static* storage class. We will also explore the lifetime of each storage class, as well as introduce the scope of a storage class—internal versus external storage.

After exploring automatic and static storage classes, this chapter paves the way for a special and extremely flexible storage class—that of dynamic memory allocation. Dynamic memory allocation is so powerful and flexible that it will be introduced in *Chapter 18, Using Dynamic Memory Allocation*, with the creation and manipulation of a dynamic data structure called a *linked list*.

Each storage class also has a specific scope or visibility to other parts of the program; the scope of both variables and functions will be explored in *Chapter 25, Understanding Scope*.

The following topics will be covered in this chapter:

- Defining storage classes
- Understanding automatic versus dynamic storage classes
- Understanding internal versus external storage classes
- Exploring the `static` storage class
- Exploring the lifetime of each storage class

Technical requirements

As detailed in the *Technical requirements* section of *Chapter 1, Running Hello, World!*, continue to use the tools you have chosen.

The source code for this chapter can be found at https://github.com/PacktPublishing/Learn-C-Programming.

Defining storage classes

C provides a number of storage classes. These fall into the following two general categories:

- **Fixed storage allocation**: Fixed storage allocation means that memory is allocated in the location where it is declared. All fixed storage is named; we have called these variable as identifiers, or just variables. Fixed storage includes both the *automatic* storage class and

the *static* storage class. We have been using automatic storage for every variable thus far. When you declare a variable and—optionally—initialize it, you are using automatic storage. We will introduce static storage later in this chapter.

- **Dynamic storage allocation**: Dynamic storage allocation means that memory is allocated upon demand and is only referenced via a pointer. The pointer may be a fixed, named pointer variable, or it may be a part of another dynamic structure.

Two properties of storage classes are their visibility—or scope—within a program or statement block, and their lifetime, or how long that memory exists as the program runs.

Within the general category of fixed storage, there are the following two sub-categories:

- **Internal storage allocation**: Internal storage is storage that is declared within the context of a function block or compound statement; in other words, declared between `{` and `}`. Internal storage has both limited scope and a limited lifetime.
- **External storage allocation**: External storage is storage that is declared outside of any function block. It has a much broader scope and lifetime than that of internal memory.

We address each of these categories in turn.

Understanding automatic versus dynamic storage classes

In all the preceding chapters, we have been using a fixed or named storage allocation. That is, whenever we declared a variable or a structure, we gave that memory location a data type and a name. This was fixed in position in our program's main routine and functions. Once that named memory was created, we could access it directly via its name, or indirectly with a pointer to that named location. In this chapter, we will specifically explore the fixed storage classes in greater detail.

Likewise, whenever we declare a literal value—say, 52 or 13—the compiler interprets these values and puts them directly into the code, fixing them in the place where they have been declared. The memory that they occupy is part of the program itself.

In contrast, dynamic storage allocation is unnamed; it can only be accessed via pointers.
Dynamic memory allocation will be introduced and explored in the next chapter.

Automatic storage

Automatic storage means that memory is allocated by the compiler at precisely the point when a literal value, variable, array, or structure is declared. A less obvious but well-defined point is when a formal parameter to a function is declared. That memory is automatically deallocated at specific and other well-known points within the program.

In all cases except literal values, when this storage class is allocated, it is given a name—its variable name—along with its data type. Even a pointer to another, already allocated memory location is given a name. When that memory is an element of an array, it is the array name and its offset in the array.

Dynamic storage

In comparison to fixed storage, dynamic storage is a memory that is unnamed but is accessed solely indirectly via pointers. There are special library functions to allocate and deallocate dynamic memory. As we will see in the next chapter, we must take extra care to keep track of the unnamed allocated memory.

Understanding internal versus external storage classes

In the storage class of a fixed or named memory, C has explicit mechanisms to allocate that memory. These correlate to the following four C keywords:

- `auto`
- `static`
- `register`
- `extern`

Note that the `auto` keyword represents the automatic storage class, and the static keyword specifies the `static` storage class. We are currently interested in only the first two of these mechanisms. These keywords precede a variable specification, as follows:

```
<storage class> [const] <data type> <name> [= <initial value>];
```

In this specification, the following applies:

- `<storage class>` is one of the preceding four keywords.
- `[const]` is an optional keyword to indicate whether the named memory can be changed after initialization. If `const` is present, an initial value must be supplied.
- `<data type>` is an intrinsic or custom data type.
- `<name>` is the variable or constant name for the value and data type.
- `[= <initial value>]` is an optional initial value or values to be assigned to the named memory location. If `const` is present, the value in that memory cannot be changed; otherwise, it can be reassigned another value.

When `<storage class>` is omitted, the `auto` keyword is assumed. So, all of our programs up to this point have been using `auto` memory variables by default. Function parameters are also `auto` memory variables and have all the same properties as those we explicitly declare in the body of functions or in compound statements.

The `register` keyword was used in older versions of C to signal to the compiler to store a value in one of the registers of the **central processing unit** (**CPU**) for very quick access to that value. Compilers have become so much more sophisticated that this keyword is ignored, except in some very specialized C compilers.

The `extern` keyword has to do with the scope of external variables declared in other files. We will return to the use of this keyword in *Chapter 25, Understanding Scope*.

Internal or local storage classes

Not only have we been using automatic, fixed storage in all the preceding chapters, we have also been using the sub-class of internal storage. Internal storage is a memory that is allocated either with a compound statement (between `{` and `}`) or as a function parameter.

Internal memory includes loop variables that are allocated when the loop is entered and deallocated when the loop is exited or completes.

Internal memory variables are only accessible within the compound statement where they've been declared, and any sub-compound statement declared within that compound statement. Their scope is limited to their enclosing `{` and `}`. They are not accessible from any other function or any function that calls them. Therefore, they are often referred to as a local memory because they are strictly local to the code block within which they are declared.

Consider the following function:

```
double doSomething( double aReal, int aNumber ) {
  double d1 = aReal;
  double d2 = 0.0 ;
  int    n1 = aNumber;
  int    n2 = aNumber * 10 ;

  for( int i = 1; i < n1 , i++ ) {
    for( int j = 1; j < n2 ; j++ {
        d1 = i / j;
        d2 + = d1;
    }
  }
  return d2;
```

This function consists of two function parameters and a return value. It also contains within its function body four local variables and two looping variables. This function might be called with the following statement:

```
double aSum = doSomething( 2.25 , 10 );
```

When the function is called, the `aReal` and `aNumber` automatic local variables are allocated and assigned (copied) the values of `2.25` and `10`, respectively. These variables can then be used throughout the function body. Within the function body, the `d1`, `d2`, `n1`, and `n2` variables are automatic local variables. They, too, can be used throughout the function body.

Lastly, we create the loop with the `i` loop-local variable, where `i` is only accessible within its loop block. Within that block is another loop with the `j` loop-local variable, where both `j` and `i`, and all other function-local variables, are accessible. Finally, the function returns the value of `d2`.

In the calling statement, the function assigns the value of the `d2` function-local variable to the `aSum` automatic variable. At the completion of `doSomething()`, all of the memory allocated by that function is no longer accessible.

External or global storage classes

External storage is memory that is declared outside of any function body, including `main()`. Such variables can potentially be accessed from any part of the program. These are more often called **global variables** because they are globally accessible from within the file where they are declared.

Historical Note

In versions of C before C90, a global variable in one file (compilation unit) was accessible to all other files. This is no longer true. And so, the global storage class, without any other directives in other files, can only be accessed from the file where they are declared. Yet they can be accessed by any function within that file.

One advantage of global variables is their ease of accessibility. However, this is also their disadvantage. When a variable can be accessed from anywhere, it becomes increasingly difficult as a program grows in size and complexity to know when that variable changed and what changed it. Global variables should be used sparingly and with great care.

The lifetime of automatic storage

When we consider the various storage classes, not only do we consider when they are created and accessed, but we must also consider when they are deallocated or destroyed. This is their lifetime—from creation to destruction.

Automatic, internal variables are created when the variable is declared either in the body of a compound statement or in a function's formal parameter list. Internal variables are destroyed and no longer accessible when that compound statement or function is exited.

Consider the `doSomething()` function. The `aReal`, `aNumber`, `d1`, `d2`, `n1`, and `n2` variables are created when the function is called. All of them are destroyed after the function returns its `d2` value. The `i` loop variable is created when we enter the loop and is destroyed when we exit that outer loop. The `j` variable is created at each iteration of the outer loop controlled by `i` and destroyed at the completion of the inner loop controlled by `j`.

Local variables have a lifetime that is only as long as the compound statement in which they are declared.

Automatic, external variables are created when the program is loaded into memory. They exist for the lifetime of the program. When the program exits (the `main()` function block returns), they are destroyed.

Exploring the static storage class

Sometimes, it is desirable to allocate memory in such a way that it can hold a value beyond the lifetime of automatic memory variables. An example of this might be a routine that could be called from anywhere within a program that returns a continuously increasing value each time it is called, such as a page number or a unique record identifier. Furthermore, we might want to give such a function a starting value and increment the sequence of numbers from that point. We will see how to do each of these.

Neither of these can be achieved easily with automatic storage classes. For this, there is the `static` storage class. As with the `automatic` storage class, it can exist as both internal and external storage.

Internal static storage

When a variable is declared within a function block with the `static` keyword, that variable is accessible only from within that function block when the function is called. The initial value of the static value is assigned at compile time and is not re-evaluated at runtime. Therefore, the value assigned to the static variable must be known at compile time and cannot be an expression or variable.

Consider the following program:

```
#include <stdio.h>

void printPage( const char* aHeading );

int main( void )  {
  printPage( "Title Page" );
  printPage ( "Chapter 1 " );
  printPage ( "           " );
  printPage ( "           " );
  printPage ( "Chapter 2 " );
  printPage ( "           " );
  printPage ( "Conclusion" );
}

void printPage( const char* aHeading )  {
  static int pageNo = 1;
  printf( "--------------------\n"
          "| %10s         |\n" , aHeading );

  printf( "|                    |\n"
          "|    Page Content    |\n"
          "|     Goes Here      |\n"
          "|                    |\n");

  printf( "|             Page %1d |\n"
          "--------------------\n" , pageNo );
```

```
    pageNo++;
}
```

The `printPage()` function contains the `pageNo` static variable. `pageNo` has `1` as its initial value when the program is started. When `printPage()` is called, the given heading string is printed along with the current page number value. `pageNo` is then incremented in preparation for the next call to it.

This program also demonstrates the ability for `printf()` to automatically concatenate multiple strings together into a single string. This is useful for printing multiple lines of text such that the text of each can be easily aligned.

Create a file called `pageCounter.c` and enter the preceding program. Compile and run this program. You should see the following output:

```
> cc pageCounter.c -o pageCounter -Wall -Werror -std=c17
> ./pageCounter
--------------------
| Title Page       |
|                  |
|    Page Content  |
|     Goes Here    |
|                  |
|           Page 1 |
--------------------
--------------------
| Chapter 1        |
|                  |
|    Page Content  |
|     Goes Here    |
|                  |
|           Page 2 |
--------------------
--------------------
|                  |
|                  |
|    Page Content  |
|     Goes Here    |
|                  |
|           Page 3 |
--------------------
--------------------
| Chapter 2        |
|                  |
|    Page Content  |
|     Goes Here    |
|                  |
|           Page 4 |
--------------------
--------------------
| Conclusion       |
|                  |
|    Page Content  |
|     Goes Here    |
|                  |
|           Page 5 |
--------------------
```

Figure 17.1 - Screenshot of pageCount.c using static variable.

The value of the static memory is incremented and preserved even after the function exits.

Now, consider what would happen if the `static` keyword was removed. Do that—remove the `static` keyword. Compile and run the program. You should see the following output:

```
> cc pageCounter.c -o pageCounter -Wall -Werror -std=c17
> ./pageCounter
---------------------
| Title Page        |
|                   |
|    Page Content   |
|     Goes Here     |
|                   |
|            Page 1 |
---------------------
---------------------
| Chapter 1         |
|                   |
|    Page Content   |
|     Goes Here     |
|                   |
|            Page 1 |
---------------------
---------------------
|                   |
|                   |
|    Page Content   |
|     Goes Here     |
|                   |
|            Page 1 |
---------------------
---------------------
| Chapter 2         |
|                   |
|    Page Content   |
|     Goes Here     |
|                   |
|            Page 1 |
---------------------
---------------------
| Conclusion        |
|                   |
|    Page Content   |
|     Goes Here     |
|                   |
|            Page 1 |
---------------------
```

Figure 17.2 - Screenshot of pageCount.c using auto variable

In this case, the automatic variable is initialized each time the function is called, and we never see the incremented value because it is destroyed when the function exits.

External static storage

Because an internal static variable can only be initialized by the compiler, we need another mechanism to safely store a value that we might want to initialize, or seed, ourselves. For this, we can use an external static variable.

External static variables can only be accessible by any other variable or code block, including function blocks, within the file where it is declared. The `static` attribute prevents them from having external visibility outside of file where they are declared. Ideally, then, the code for the external static variable and the function that accesses it should be in a single, separate `.c` file, as follows:

```
  // seriesGenerator.c

static int seriesNumber = 100; // default seed value
```

```
void seriesStart( int seed )  {
   seriesNumber = seed;
}

int series( void )  {
   return seriesNumber++;
}
```

To use these functions, we would need to include a header file with function prototypes for it, as follows:

```
  // seriesGenerator.h

void seriesStart( int seed );
int  series( void );
```

We would create the `seriesGenerator.c` and `seriesGenerator.h` files with these functions and prototypes. We would also have to add `#include <seriesGenerator.h>` to any file that calls these functions. We would then compile these files, along with our other source files, into a single executable file. We did this briefly in *Chapter 10*, *Creating Custom Data Types with typedef*; we will explore this more fully in *Chapter 24*, *Working with Multi-File Programs*.

This series generator, when compiled into the main program, would be initialized, or seeded, with a call to `seriesStart()` with some known integer value. After that, each call to `series()` would generate the next number in the series.

This pattern may seem familiar from the last chapter, *Chapter 16*, *Creating and Using More Complex Structures*. There, we used `srand()` to seed our **pseudorandom number generator** (**PRNG**), and then subsequently called `rand()` to get the next number in the random sequence. You can now more clearly imagine how `srand()` and `rand()` would be implemented using static external memory allocation.

The lifetime of static storage

The lifetimes for both internal and external static memory are the same. Static memory is allocated when the program is loaded before any statements are executed. Static memory is only destroyed when the program completes or exits. Therefore, the lifetime of static memory is the same as the lifetime of the program.

Summary

In this chapter, we explored various storage classes, and, how memory is allocated. In particular, we clarified automatic memory allocation, or fixed and named memory—the method we've been using exclusively in all chapters prior to this chapter. In addition to automatic memory allocation, we explored static memory allocation. With both of those approaches, we distinguished between internal memory allocation—variables declared within a compound statement or function parameters—and external memory allocation—variables declared outside of any function. For each of these storage classes

(automatic internal, automatic external, static internal, and static external memory allocation), we considered the lifetime of the memory—when that memory is destroyed and no longer accessible.

We are now ready to explore in the next chapter the much more flexible storage class, dynamic memory, which is unnamed and can only be accessed via pointer variables. Dynamic memory techniques will bring us to the threshold of very powerful dynamic data structures.

Questions

1. We have seen that variables have 3 characteristics: an identifier, a data type, and a value. What 3 additional characteristics to they have?
2. What are the 4 storage classes?
3. What is the lifetime of the static and external storage classes?
4. What is the visibility of a static variable?
5. What is the visibility of an external variable?
6. What happens to automatic variables when execution leaves the block where they are declared?

Answers

1. In addition to an identifier, a data type and a value, variables also have a storage class, a lifetime, and visibility (or scope).
2. The 4 storage classes are auto, static, extern (global), and register.
3. The lifetime of static and external storage classes are created when the program loads and destroyed when the program ends.
4. A static variable is only visible within the block or compilation unit (file) where it is declared.
5. An external (global) variable is visible to every function and block within the compiliation unit (file) where it is declared.
6. Automatic variables are destroyed/no longer accessible after execution leaves the block where they are declared.