CHAPTER 11

Variables and Data Storage

One of the C language's strengths is its flexibility in defining data storage. There are two aspects that can be controlled in C: scope and lifetime. Scope refers to the places in the code from which the variable can be accessed. Lifetime refers to the points in time at which the variable can be accessed.

Three scopes are available to the programmer:

`extern` This is the default for variables declared outside any function. The scope of variables with `extern` scope is all the code in the entire program.

`static` The scope of a variable declared `static` outside any function is the rest of the code in that source file. The scope of a variable declared `static` inside a function is the rest of the local block.

`auto` This is the default for variables declared inside a function. The scope of an `auto` variable is the rest of the local block.

Three lifetimes are available to the programmer. They do not have predefined keywords for names as scopes do. The first is the lifetime of `extern` and `static` variables, whose lifetime is from before `main()` is called until the program exits. The second is the lifetime of function arguments and automatics, which is from the time the function is called until it returns. The third lifetime is that of dynamically allocated data. It starts when the program calls `malloc()` or `calloc()` to allocate space for the data and ends when the program calls `free()` or when it exits, whichever comes first.

II.1: Where in memory are my variables stored?

Answer:

Variables can be stored in several places in memory, depending on their lifetime. Variables that are defined outside any function (whether of global or file `static` scope), and variables that are defined inside a function as `static` variables, exist for the lifetime of the program's execution. These variables are stored in the "data segment." The data segment is a fixed-size area in memory set aside for these variables. The data segment is subdivided into two parts, one for initialized variables and another for uninitialized variables.

Variables that are defined inside a function as `auto` variables (that are not defined with the keyword `static`) come into existence when the program begins executing the block of code (delimited by curly braces `{}`) containing them, and they cease to exist when the program leaves that block of code. Variables that are the arguments to functions exist only during the call to that function. These variables are stored on the "stack." The stack is an area of memory that starts out small and grows automatically up to some predefined limit. In DOS and other systems without virtual memory, the limit is set either when the program is compiled or when it begins executing. In UNIX and other systems with virtual memory, the limit is set by the system, and it is usually so large that it can be ignored by the programmer. For a discussion on what virtual memory is, see FAQ II.3.

The third and final area doesn't actually store variables but can be used to store data pointed to by variables. Pointer variables that are assigned to the result of a call to the `malloc()` function contain the address of a dynamically allocated area of memory. This memory is in an area called the "heap." The heap is another area that starts out small and grows, but it grows only when the programmer explicitly calls `malloc()` or other memory allocation functions, such as `calloc()`. The heap can share a memory segment with either the data segment or the stack, or it can have its own segment. It all depends on the compiler options and operating system. The heap, like the stack, has a limit on how much it can grow, and the same rules apply as to how that limit is determined.

Cross Reference:

I.1: What is a local block?

II.2: Do variables need to be initialized?

II.3: What is page thrashing?

VII.20: What is the stack?

VII.21: What is the heap?

II.2: Do variables need to be initialized?

Answer:

No. All variables should be given a value before they are used, and a good compiler will help you find variables that are used before they are set to a value. Variables need not be initialized, however. Variables defined outside a function or defined inside a function with the `static` keyword (those defined in the data segment discussed in the preceding FAQ) are already initialized to 0 for you if you do not explicitly initialize them.

Automatic variables are variables defined inside a function or block of code without the `static` keyword. These variables have undefined values if you don't explicitly initialize them. If you don't initialize an automatic variable, you must make sure you assign to it before using the value.

Space on the heap allocated by calling `malloc()` contains undefined data as well and must be set to a known value before being used. Space allocated by calling `calloc()` is set to 0 for you when it is allocated.

Cross Reference:

I.1: What is a local block?

VII.20: What is the stack?

VII.21: What is the heap?

II.3: What is page thrashing?

Answer:

Some operating systems (such as UNIX or Windows in enhanced mode) use virtual memory. Virtual memory is a technique for making a machine behave as if it had more memory than it really has, by using disk space to simulate RAM (random-access memory). In the 80386 and higher Intel CPU chips, and in most other modern microprocessors (such as the Motorola 68030, Sparc, and Power PC), exists a piece of hardware called the Memory Management Unit, or MMU.

The MMU treats memory as if it were composed of a series of "pages." A page of memory is a block of contiguous bytes of a certain size, usually 4096 or 8192 bytes. The operating system sets up and maintains a table for each running program called the Process Memory Map, or PMM. This is a table of all the pages of memory that program can access and where each is really located.

Every time your program accesses any portion of memory, the address (called a "virtual address") is processed by the MMU. The MMU looks in the PMM to find out where the memory is really located (called the "physical address"). The physical address can be any location in memory or on disk that the operating system has assigned for it. If the location the program wants to access is on disk, the page containing it must be read from disk into memory, and the PMM must be updated to reflect this action (this is called a "page fault").

Hope you're still with me, because here's the tricky part. Because accessing the disk is so much slower than accessing RAM, the operating system tries to keep as much of the virtual memory as possible in RAM. If you're running a large enough program (or several small programs at once), there might not be enough RAM to hold all the memory used by the programs, so some of it must be moved out of RAM and onto disk (this action is called "paging out").

The operating system tries to guess which areas of memory aren't likely to be used for a while (usually based on how the memory has been used in the past). If it guesses wrong, or if your programs are accessing lots of memory in lots of places, many page faults will occur in order to read in the pages that were paged out. Because all of RAM is being used, for each page read in to be accessed, another page must be paged out. This can lead to more page faults, because now a different page of memory has been moved to disk. The problem of many page faults occurring in a short time, called "page thrashing," can drastically cut the performance of a system.

Programs that frequently access many widely separated locations in memory are more likely to cause page thrashing on a system. So is running many small programs that all continue to run even when you are not actively using them. To reduce page thrashing, you can run fewer programs simultaneously. Or you can try changing the way a large program works to maximize the capability of the operating system to guess which pages won't be needed. You can achieve this effect by caching values or changing lookup algorithms in large data structures, or sometimes by changing to a memory allocation library which provides an implementation of `malloc()` that allocates memory more efficiently. Finally, you might consider adding more RAM to the system to reduce the need to page out.

Cross Reference:

VII.17: How do you declare an array that will hold more than 64KB of data?

VII.21: What is the heap?

XVIII.14: How can I get more than 640KB of memory available to my DOS program?

XXI.31: How is memory organized in Windows?

II.4: What is a *const* pointer?

Answer:

The access modifier keyword `const` is a promise the programmer makes to the compiler that the value of a variable will not be changed after it is initialized. The compiler will enforce that promise as best it can by not enabling the programmer to write code which modifies a variable that has been declared `const`.

A "`const` pointer," or more correctly, a "pointer to `const`," is a pointer which points to data that is `const` (constant, or unchanging). A pointer to `const` is declared by putting the word `const` at the beginning of the pointer declaration. This declares a pointer which points to data that can't be modified. The pointer itself can be modified. The following example illustrates some legal and illegal uses of a `const` pointer:

```
const char  *str = "hello";
char  c = *str    /* legal */
str++;            /* legal */
*str = 'a';       /* illegal */
str[1] = 'b';     /* illegal */
```

The first two statements here are legal because they do not modify the data that `str` points to. The next two statements are illegal because they modify the data pointed to by `str`.

Pointers to `const` are most often used in declaring function parameters. For instance, a function that counted the number of characters in a string would not need to change the contents of the string, and it might be written this way:

```
my_strlen(const char *str)
{
        int count = 0;
        while (*str++)
        {
```

```
            count++;
        }
        return count;
}
```

Note that non-`const` pointers are implicitly converted to `const` pointers when needed, but `const` pointers are not converted to non-`const` pointers. This means that `my_strlen()` could be called with either a `const` or a non-`const` character pointer.

Cross Reference:

II.7: Can a variable be both `const` and `volatile`?

II.8: When should the `const` modifier be used?

II.14: When should a type cast not be used?

II.18: What is the benefit of using `const` for declaring constants?

II.5: When should the register modifier be used? Does it really help?

Answer:

The register modifier hints to the compiler that the variable will be heavily used and should be kept in the CPU's registers, if possible, so that it can be accessed faster. There are several restrictions on the use of the register modifier.

First, the variable must be of a type that can be held in the CPU's register. This usually means a single value of a size less than or equal to the size of an integer. Some machines have registers that can hold floating-point numbers as well.

Second, because the variable might not be stored in memory, its address cannot be taken with the unary `&` operator. An attempt to do so is flagged as an error by the compiler.

Some additional rules affect how useful the register modifier is. Because the number of registers is limited, and because some registers can hold only certain types of data (such as pointers or floating-point numbers), the number and types of register modifiers that will actually have any effect are dependent on what machine the program will run on. Any additional register modifiers are silently ignored by the compiler.

Also, in some cases, it might actually be slower to keep a variable in a register because that register then becomes unavailable for other purposes or because the variable isn't used enough to justify the overhead of loading and storing it.

So when should the register modifier be used? The answer is never, with most modern compilers. Early C compilers did not keep any variables in registers unless directed to do so, and the register modifier was a valuable addition to the language. C compiler design has advanced to the point, however, where the compiler will usually make better decisions than the programmer about which variables should be stored in registers. In fact, many compilers actually ignore the register modifier, which is perfectly legal, because it is only a hint and not a directive.

In the rare event that a program is too slow, and you know that the problem is due to a variable being stored in memory, you might try adding the register modifier as a last resort, but don't be surprised if this action doesn't change the speed of the program.

Cross Reference:

II.6: When should the `volatile` modifier be used?

II.6: When should the *volatile* modifier be used?

Answer:

The `volatile` modifier is a directive to the compiler's optimizer that operations involving this variable should not be optimized in certain ways. There are two special cases in which use of the `volatile` modifier is desirable. The first case involves memory-mapped hardware (a device such as a graphics adaptor that appears to the computer's hardware as if it were part of the computer's memory), and the second involves shared memory (memory used by two or more programs running simultaneously).

Most computers have a set of registers that can be accessed faster than the computer's main memory. A good compiler will perform a kind of optimization called "redundant load and store removal." The compiler looks for places in the code where it can either remove an instruction to load data from memory because the value is already in a register, or remove an instruction to store data to memory because the value can stay in a register until it is changed again anyway.

If a variable is a pointer to something other than normal memory, such as memory-mapped ports on a peripheral, redundant load and store optimizations might be detrimental. For instance, here's a piece of code that might be used to time some operation:

```
time_t time_addition(volatile const struct timer *t, int a)
{
        int     n;
        int     x;
        time_t  then;
        x = 0;
        then = t->value;
        for (n = 0; n < 1000; n++)
        {
                x = x + a;
        }

       return t->value - then;
}
```

In this code, the variable `t->value` is actually a hardware counter that is being incremented as time passes. The function adds the value of `a` to `x` 1000 times, and it returns the amount the timer was incremented by while the 1000 additions were being performed.

Without the `volatile` modifier, a clever optimizer might assume that the value of `t` does not change during the execution of the function, because there is no statement that explicitly changes it. In that case, there's no need to read it from memory a second time and subtract it, because the answer will always be 0. The compiler might therefore "optimize" the function by making it always return 0.

If a variable points to data in shared memory, you also don't want the compiler to perform redundant load and store optimizations. Shared memory is normally used to enable two programs to communicate with each other by having one program store data in the shared portion of memory and the other program read the same portion of memory. If the compiler optimizes away a load or store of shared memory, communication between the two programs will be affected.

Cross Reference:

II.7: Can a variable be both `const` and `volatile`?

II.14: When should a type cast not be used?

II.7: Can a variable be both *const* and *volatile*?

Answer:

Yes. The `const` modifier means that this code cannot change the value of the variable, but that does not mean that the value cannot be changed by means outside this code. For instance, in the example in FAQ II.6, the timer structure was accessed through a `volatile const` pointer. The function itself did not change the value of the timer, so it was declared `const`. However, the value was changed by hardware on the computer, so it was declared `volatile`. If a variable is both `const` and `volatile`, the two modifiers can appear in either order.

Cross Reference:

II.6: When should the `volatile` modifier be used?

II.8: When should the `const` modifier be used?

II.14: When should a type cast not be used?

II.8: When should the *const* modifier be used?

Answer:

There are several reasons to use `const` pointers. First, it allows the compiler to catch errors in which code accidentally changes the value of a variable, as in

```
while (*str = 0) /* programmer meant to write *str != 0 */
{
    /* some code here */
    str++;
}
```

in which the = sign is a typographical error. Without the `const` in the declaration of `str`, the program would compile but not run properly.

Another reason is efficiency. The compiler might be able to make certain optimizations to the code generated if it knows that a variable will not be changed.

Any function parameter which points to data that is not modified by the function or by any function it calls should declare the pointer a pointer to `const`. Function parameters that are passed by value (rather than through a pointer) can be declared `const` if neither the function nor any function it calls modifies the data. In practice, however, such parameters are usually declared `const` only if it might be more efficient for the compiler to access the data through a pointer than by copying it.

Cross Reference:

II.7: Can a variable be both `const` and `volatile`?

II.14: When should a type cast not be used?

II.18: What is the benefit of using `const` for declaring constants?

II.9: How reliable are floating-point comparisons?

Answer:

Floating-point numbers are the "black art" of computer programming. One reason why this is so is that there is no optimal way to represent an arbitrary number. The Institute of Electrical and Electronic Engineers (IEEE) has developed a standard for the representation of floating-point numbers, but you cannot guarantee that every machine you use will conform to the standard.

Even if your machine does conform to the standard, there are deeper issues. It can be shown mathematically that there are an infinite number of "real" numbers between any two numbers. For the computer to distinguish between two numbers, the bits that represent them must differ. To represent an infinite number of different bit patterns would take an infinite number of bits. Because the computer must represent a large range of numbers in a small number of bits (usually 32 to 64 bits), it has to make approximate representations of most numbers.

Because floating-point numbers are so tricky to deal with, it's generally bad practice to compare a floating-point number for equality with anything. Inequalities are much safer. If, for instance, you want to step through a range of numbers in small increments, you might write this:

```
#include <stdio.h>
const float first = 0.0;
const float last = 70.0;
const float small = 0.007;
main()
{
        float   f;
        for (f = first; f != last && f < last + 1.0; f += small)
                ;
        printf("f is now %g\n", f);
}
```

However, rounding errors and small differences in the representation of the variable `small` might cause `f` to never be equal to `last` (it might go from being just under it to being just over it). Thus, the loop would go past the value `last`. The inequality `f < last + 1.0` has been added to prevent the program from running on for a very long time if this happens. If you run this program and the value printed for `f` is 71 or more, this is what has happened.

A safer way to write this loop is to use the inequality `f < last` to test for the loop ending, as in this example:

```
float   f;
for (f = first; f < last; f += small)
        ;
```

You could even precompute the number of times the loop should be executed and use an integer to count iterations of the loop, as in this example:

```
float   f;
int     count = (last - first) / small;
for (f = first; count-- > 0; f += small)
        ;
```

Cross Reference:

II.11: Are there any problems with performing mathematical operations on different variable types?

II.10: How can you determine the maximum value that a numeric variable can hold?

Answer:

The easiest way to find out how large or small a number that a particular type can hold is to use the values defined in the ANSI standard header file limits.h. This file contains many useful constants defining the values that can be held by various types, including these:

Value	*Description*
`CHAR_BIT`	Number of bits in a `char`
`CHAR_MAX`	Maximum decimal integer value of a `char`
`CHAR_MIN`	Minimum decimal integer value of a `char`
`MB_LEN_MAX`	Maximum number of bytes in a multibyte character
`INT_MAX`	Maximum decimal value of an `int`
`INT_MIN`	Minimum decimal value of an `int`
`LONG_MAX`	Maximum decimal value of a `long`
`LONG_MIN`	Minimum decimal value of a `long`
`SCHAR_MAX`	Maximum decimal integer value of a `signed char`
`SCHAR_MIN`	Minimum decimal integer value of a `signed char`
`SHRT_MAX`	Maximum decimal value of a `short`
`SHRT_MIN`	Minimum decimal value of a `short`
`UCHAR_MAX`	Maximum decimal integer value of `unsigned char`

continues

Value	*Description*
UINT_MAX	Maximum decimal value of an unsigned integer
ULONG_MAX	Maximum decimal value of an unsigned long int
USHRT_MAX	Maximum decimal value of an unsigned short int

For integral types, on a machine that uses two's complement arithmetic (which is just about any machine you're likely to use), a signed type can hold numbers from $-2^{(\text{number of bits}-1)}$ to $+2^{(\text{number of bits}-1)}-1$. An unsigned type can hold values from 0 to $+2^{(\text{number of bits})}-1$. For instance, a 16-bit signed integer can hold numbers from -2^{15} (–32768) to $+2^{15}-1$ (32767).

Cross Reference:

X.1: What is the most efficient way to store flag values?

X.2: What is meant by "bit masking"?

X.6: How are 16- and 32-bit numbers stored?

II.11: Are there any problems with performing mathematical operations on different variable types?

Answer:

C has three categories of built-in data types: pointer types, integral types, and floating-point types.

Pointer types are the most restrictive in terms of the operations that can be performed on them. They are limited to

- subtraction of two pointers, valid only when both pointers point to elements in the same array. The result is the same as subtracting the integer subscripts corresponding to the two pointers.
- \+ addition of a pointer and an integral type. The result is a pointer that points to the element which would be selected by that integer.

Floating-point types consist of the built-in types float, double, and long double. Integral types consist of char, unsigned char, short, unsigned short, int, unsigned int, long, and unsigned long. All of these types can have the following arithmetic operations performed on them:

+	Addition
-	Subtraction
*	Multiplication
/	Division

Integral types also can have those four operations performed on them, as well as the following operations:

%	Modulo or remainder of division
<<	Shift left
>>	Shift right
&	Bitwise AND operation
|	Bitwise OR operation

^ Bitwise exclusive OR operation
! Logical negative operation
~ Bitwise "one's complement" operation

Although C permits "mixed mode" expressions (an arithmetic expression involving different types), it actually converts the types to be the same type before performing the operations (except for the case of pointer arithmetic described previously). The process of automatic type conversion is called "operator promotion." Operator promotion is explained in FAQ II.12.

Cross Reference:

II.12: What is operator promotion?

II.12: What is operator promotion?

Answer:

If an operation is specified with operands of two different types, they are converted to the smallest type that can hold both values. The result has the same type as the two operands wind up having. To interpret the rules, read the following table from the top down, and stop at the first rule that applies.

If Either Operand Is	*And the Other Is*	*Change Them To*
`long double`	any other type	`long double`
`double`	any smaller type	`double`
`float`	any smaller type	`float`
`unsigned long`	any integral type	`unsigned long`
`long`	`unsigned` > `LONG_MAX`	`unsigned long`
`long`	any smaller type	`long`
`unsigned`	any signed type	`unsigned`

The following example code illustrates some cases of operator promotion. The variable `f1` is set to `3 / 4`. Because both `3` and `4` are integers, integer division is performed, and the result is the integer `0`. The variable `f2` is set to `3 / 4.0`. Because `4.0` is a `float`, the number `3` is converted to a `float` as well, and the result is the `float 0.75`.

```
#include <stdio.h>
main()
{
        float f1 = 3 / 4;
        float f2 = 3 / 4.0;
        printf("3 / 4 == %g or %g depending on the type used.\n",
                f1, f2);
}
```

Cross Reference:

II.11: Are there any problems with performing mathematical operations on different variable types?

II.13: When should a type cast be used?

II.13: When should a type cast be used?

Answer:

There are two situations in which to use a type cast. The first use is to change the type of an operand to an arithmetic operation so that the operation will be performed properly. If you have read FAQ II.12, the following listing should look familiar. The variable `f1` is set to the result of dividing the integer `i` by the integer `j`. The result is `0`, because integer division is used. The variable `f2` is set to the result of dividing `i` by `j` as well. However, the `(float)` type cast causes `i` to be converted to a `float`. That in turn causes floating-point division to be used (see FAQ II.11) and gives the result `0.75`.

```
#include <stdio.h>
main()
{
        int     i = 3;
        int     j = 4;
        float f1 = i / j;
        float f2 = (float) i / j;
        printf("3 / 4 == %g or %g depending on the type used.\n",
                f1, f2);
}
```

The second case is to cast pointer types to and from `void *` in order to interface with functions that expect or return `void` pointers. For example, the following line type casts the return value of the call to `malloc()` to be a pointer to a `foo` structure.

```
struct foo      *p = (struct foo *) malloc(sizeof(struct foo));
```

Cross Reference:

II.6: When should the `volatile` modifier be used?

II.8: When should the `const` modifier be used?

II.11: Are there any problems with performing mathematical operations on different variable types?

II.12: What is operator promotion?

II.14: When should a type cast not be used?

VII.5: What is a `void` pointer?

VII.6: When is a `void` pointer used?

VII.21: What is the heap?

VII.27: Can math operations be performed on a `void` pointer?

II.14: When should a type cast not be used?

Answer:

A type cast should not be used to override a `const` or `volatile` declaration. Overriding these type modifiers can cause the program to fail to run correctly.

A type cast should not be used to turn a pointer to one type of structure or data type into another. In the rare events in which this action is beneficial, using a union to hold the values makes the programmer's intentions clearer.

Cross Reference:

II.6: When should the `volatile` modifier be used?

II.8: When should the `const` modifier be used?

II.15: Is it acceptable to declare/define a variable in a C header?

Answer:

A global variable that must be accessed from more than one file can and should be declared in a header file. In addition, such a variable must be defined in one source file. Variables should not be defined in header files, because the header file can be included in multiple source files, which would cause multiple definitions of the variable. The ANSI C standard will allow multiple external definitions, provided that there is only one initialization. But because there's really no advantage to using this feature, it's probably best to avoid it and maintain a higher level of portability.

NOTE

Don't confuse declaring and defining variables. FAQ II.16 states the differences between these two actions.

"Global" variables that do not have to be accessed from more than one file should be declared `static` and should not appear in a header file.

Cross Reference:

II.16: What is the difference between declaring a variable and defining a variable?

II.17: Can `static` variables be declared in a header file?

II.16: What is the difference between declaring a variable and defining a variable?

Answer:

Declaring a variable means describing its type to the compiler but not allocating any space for it. Defining a variable means declaring it and also allocating space to hold the variable. You can also initialize a variable at the time it is defined. Here is a declaration of a variable and a structure, and two variable definitions, one with initialization:

```
extern int      decl1;  /* this is a declaration */
struct decl2 {
        int member;
}; /* this just declares the type--no variable mentioned */
int     def1 = 8;       /* this is a definition */
int     def2;           /* this is a definition */
```

To put it another way, a declaration says to the compiler, "Somewhere in my program will be a variable with this name, and this is what type it is." A definition says, "Right here is this variable with this name and this type."

NOTE

One way to remember what each term means is to remember that the Declaration of Independence didn't actually make the United States independent (the Revolutionary War did that); it just stated that it was independent.

A variable can be declared many times, but it must be defined exactly once. For this reason, definitions do not belong in header files, where they might get `#include`d into more than one place in your program.

Cross Reference:

II.17: Can `static` variables be declared in a header file?

II.17: Can *static* variables be declared in a header file?

Answer:

You can't declare a `static` variable without defining it as well (this is because the storage class modifiers `static` and `extern` are mutually exclusive). A `static` variable can be defined in a header file, but this would cause each source file that included the header file to have its own private copy of the variable, which is probably not what was intended.

Cross Reference:

II.16: What is the difference between declaring a variable and defining a variable?

II.18: What is the benefit of using *const* for declaring constants?

Answer:

The benefit of using the `const` keyword is that the compiler might be able to make optimizations based on the knowledge that the value of the variable will not change. In addition, the compiler will try to ensure that the values won't be changed inadvertently.

Of course, the same benefits apply to `#define`d constants. The reason to use `const` rather than `#define` to define a constant is that a `const` variable can be of any type (such as a `struct`, which can't be represented by a `#define`d constant). Also, because a `const` variable is a real variable, it has an address that can be used, if needed, and it resides in only one place in memory (some compilers make a new copy of a `#define`d character string each time it is used—see FAQ IX.9).

Cross Reference:

II.7: Can a variable be both `const` and `volatile`?

II.8: When should the `const` modifier be used?

II.14: When should a type cast not be used?

IX.9: What is the difference between a string and an array?