Publications
The C Book
Preface
Introduction
Variables & arithmetic
Fundamentals
The C alphabet
Textual program structure
Keywords & identifiers
Declaring variables
Reals
Integers
Expressions & arithmetic
Constants
Summary
Exercises
Control flow
Functions
Arrays & pointers
Structures
Preprocessor
Specialized areas
Libraries
Complete Programs
Answers
Copyright

Contact Us

2.7. Integral types

The real types were the easy ones. The rules for the integral types are more complicated, but still tolerable, and these rules really should be learnt. Fortunately, the only types used in C for routine data storage are the real and integer types, or *structures* and *arrays* built up from them. C doesn't have special types for character manipulation or the handling of logical (boolean) quantities, but uses the integral types instead. Once you know the rules for the reals and the integers you know them all.

We will start by looking at the various types and then the conversion rules.

2.7.1. Plain integers

There are two types (often called 'flavours') of integer variables. Other types can be built from these, as we'll see, but the plain undecorated `ints` are the base. The most obvious of the pair is the 'signed' `int`, the less obvious is its close relative, the `unsigned int`. These variables are supposed to be stored in whatever is the most convenient unit for the machine running your program. The `int` is the natural choice for undemanding requirements when you just need a simple integral variable, say as a counter in a short loop. There isn't any guarantee about the number of bits that an int can hold, except that it will **always** be 16 or more. The standard header file `<limits.h>` details the actual number of bits available in a given implementation.

Curiously, Old C had no guarantee whatsoever about the length of an `int`, but consensus and common practice has always assumed at least 16 bits.

Actually, `<limits.h>` doesn't quite specify a number of bits, but gives maximum and minimum values for an `int` instead. The values it gives are 32767 and -32767 which implies 16 bits or more, whether ones or twos complement arithmetic is used. Of course there is nothing to stop a given implementation from providing a greater range in either direction.

The range specified in the Standard for an `unsigned int` is 0 to at least 65535, meaning that it cannot be negative. More about these shortly.

If you aren't used to thinking about the number of bits in a given variable, and are beginning to get worried about the portability implications of this apparently machine-dependent concern for the number of bits, then you're doing the right thing. C takes portability seriously and actually bothers to tell you what values and ranges are guaranteed to be safe. The bitwise operators encourage you to think about the number of bits in a variable too, because they give direct access to the bits, which you manipulate one by one or in groups. Almost paradoxically, the overall result is that C programmers have a healthy awareness of portability issues which leads to more portable programs. This is **not** to say that you can't write C programs that are horribly non-portable!

2.7.2. Character variables

A bit less obvious than int is the other of the plain integral types, the `char`. It's basically just another sort of `int`, but has a different application. Because so many C programs do a lot of character handling, it's a good idea to provide a special type to help, especially if the range provided by an `int` uses up much more storage than is needed by characters. The limits file tells us that three things are guaranteed about `char` variables: they have at least 8 bits, they can store a value of at least +127, and the minimum value of a `char` is zero or lower. This means that the only guaranteed range is 0–127. Whether or not `char` variables behave as `signed` or `unsigned` types is implementation defined.

In short, a character variable will probably take less storage than an `int` and will most likely be used for character manipulation. It's still an integer type though, and can be used for arithmetic, as this example shows.

```
include <limits.h>
include <stdio.h>
include <stdlib.h>

main(){
      char c;

      c = CHAR_MIN;
      while(c != CHAR_MAX){
              printf("%d\n", c);
              c = c+1;
      }

      exit(EXIT_SUCCESS);
}
```

Example 2.3

Running that program is left as an exercise for the easily amused. If you are bothered about where `CHAR_MIN` and `CHAR_MAX` come from, find `limits.h` and read it.

Here's a more enlightening example. It uses character constants, which are formed by placing a character in single quotes:

```
'x'
```

Because of the rules of arithmetic, the type of this sort of constant turns out to be `int`, but that doesn't matter since their value is always small enough to assign them to `char` variables without any loss of precision. (Unfortunately, there is a related version where that guarantee does not hold. Ignore it for the moment.) When a character variable is printed using the `%c` format with `printf`, the appropriate character is output. You can use `%d`, if you like, to see what integer value is used to represent the character. Why `%d`? Because a `char` is just another integral type.

It's also useful to be able to read characters into a program. The library function `getchar` is used for the job. It reads characters from the program's *standard input* and returns an `int` value suitable for storing into a `char`. The int value is for one reason only: not only does getchar return all possible character values, but it also returns an **extra** value to indicate that end-of-input has been seen. The range of a `char` might not be enough to hold this extra value, so the int has to be used.

The following program reads its input and counts the number of commas and full stops that it sees. On end-of-input, it prints the totals.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int this_char, comma_count, stop_count;

      comma_count = stop_count = 0;
      this_char = getchar();
      while(this_char != EOF){
              if(this_char == '.')
                      stop_count = stop_count+1;
              if(this_char == ',')
                      comma_count = comma_count+1;
              this_char = getchar();
      }
      printf("%d commas, %d stops\n", comma_count,
                      stop_count);
      exit(EXIT_SUCCESS);
}
```

Example 2.4

The two features of note in that example were the multiple assignment to the two counters and the use of the defined constant `EOF`. `EOF` is the value returned by `getchar` on end of input (it stands for End Of File), and is defined in `<stdio.h>`. The multiple assignment is a fairly common feature of C programs.

Another example, perhaps. This will either print out the whole lower case alphabet, if your implementation has its characters stored consecutively, or something even more interesting if they aren't. C doesn't make many guarantees about the ordering of characters in internal form, so this program produces **non-portable** results!

```
#include <stdio.h>
#include <stdlib.h>
main(){
      char c;

      c = 'a';
      while(c <= 'z'){
```

```
                printf("value %d char %c\n", c, c);
                c = c+1;
        }

        exit(EXIT_SUCCESS);
}
```

Example 2.5

Yet again this example emphasizes that a `char` is only another form of integer variable and can be used just like any other form of variable. It is **not** a ‘special’ type with its own rules.

The space saving that a `char` offers when compared to an `int` only becomes worthwhile if a lot of them are being used. Most character-processing operations involve the use of not just one or two character variables, but large arrays of them. That's when the saving can become noticeable: imagine an array of 1024 `int`s. On a lot of common machines that would eat up 4096 8-bit bytes of storage, assuming the common length of 4 bytes per `int`. If the computer architecture allows it to be done in a reasonably efficient way, the C implementor will probably have arranged for `char` variables to be packed one per byte, so the array would only use 1024 bytes and the space saving would be 3072 bytes.

Sometimes it doesn't matter whether or not a program tries to save space; sometimes it does. At least C gives you the option of choosing an appropriate type.

2.7.3. More complicated types

The last two types were simple, in both their declaration and subsequent use. For serious systems programming they just aren't adequate in the precision of control over storage that they provide and the behaviour that they follow. To correct this problem, C provides extra forms of integral types, split into the categories of `signed` and `unsigned`. (Although both these terms are reserved words, they will also be used as adjectives.) The difference between the two types is obvious. Signed types are those that are capable of being negative, the unsigned types cannot be negative at any time. Unsigned types are usually used for one of two reasons: to get an extra bit of precision, or when the concept of being negative is simply not present in the data that is being represented. The latter is by far the better reason for choosing them.

Unsigned types also have the special property of never overflowing in arithmetic. Adding 1 to a signed variable that already contains the maximum possible positive number for its type will result in overflow, and the program's behaviour becomes undefined. That can never happen with unsigned types, because they are defined to work ‘modulo one greater than the maximum number that they can hold’. What this means is best illustrated by example:

```
#include <stdio.h>
#include <stdlib.h>
main(){
      unsigned int x;
      x = 0;
      while(x >= 0){
              printf("%u\n", x);
              x = x+1;
      }

      exit(EXIT_SUCCESS);
}
```

Example 2.6

Assuming that the variable `x` is stored in 16 bits, then its range of values will be 0–65535 and that sequence will be printed endlessly. The program can't terminate: the test

```
x >= 0
```

must always be true for an unsigned variable.

For both the signed and unsigned integral types there are three subtypes: `short`, ordinary and `long`. Taking those into account, here is a list of all of the possible integral types in C, except for the character types:

```
unsigned short int
unsigned int
unsigned long int
signed short int
```

```
   signed int
   signed long int
```

In the last three, the `signed` keyword is unnecessary because the `int` types are signed types anyway: you **have** to say `unsigned` to get anything different. It's also permissible, but not recommended, to drop the int keyword from any of those declarations provided that there is at least one other keyword present—the `int` will be ‘understood’ to be present. For example `long` is equivalent to `signed long int`. The long and short kinds give you more control over the amount of space used to store variables. Each has its own minimum range specified in `<limits.h>` which in practice means at least 16 bits in a `short` and an `int`, and at least 32 bits in a `long`, whether signed or unsigned. As always, an implementation can choose to give you more bits than the minimum if it wants to. The only restriction is that the limits must be equalled or bettered, and that you don't get more bits in a shorter type than a longer one (not an unreasonable rule).

The only character types are the `signed char` and the `unsigned char`. The difference between `char` and `int` variables is that, unless otherwise stated, all `int`s are signed. The same is not true for `char`s, which are signed or unsigned depending on the implementor's choice; the choice is presumably taken on efficiency grounds. You can of course explicitly force signed or unsignedness with the right keyword. The only time that it is likely to matter is if you are using character variables as extra short `short`s to save more space.

Summary of integral types

- The integral types are the `short`, `long`, `signed`, `unsigned` and plain `int`s.
- The commonest is the ordinary `int`, which is signed unless declared not to be.
- The `char` variables **can** be made signed or unsigned, as you prefer, but in the absence of indications to the contrary, they will be allocated the most efficient type.

2.7.4. Printing the integral types

Once again you can use `printf` to print these various types. Character variables work exactly the same way that the other integral variables do, so you can use the standard format letters to print their contents—although the actual numbers stored in them are not likely to be very interesting. To see their contents interpreted as characters, use `%c` as was done earlier. All of the integral types can be printed as if they were signed decimal numbers by using the `%d` format, or `%ld` for long types. Other useful formats are shown in Table 2.5; notice that in every case a letter ‘l’ is put in front of the normal format letter if a `long` is to be printed. That's not just there to get the right result printed: the behaviour of `printf` is undefined if the wrong format is given.

Format	Use with
`%c`	`char` (in character form)
`%d`	decimal `signed int`, `short`, `char`
`%u`	decimal `unsigned int`, `unsigned short`, `unsigned char`
`%x`	hexadecimal `int`, `short`, `char`
`%o`	octal `int`, `short`, `char`
`%ld`	decimal `signed long`
`%lu %lx %lo`	as above, but for `long`s

Table 2.5. More format codes

A full description of the format codes that you can use with printf is given in Chapter 9.

Previous section | Chapter contents | Next section