Integer Types in C

1 Integer Variables

Variables in C have two parts: a memory allocation and a type. The memory allocation is a sequence of 8-bit bytes which represent the current value of the variable. The encoding tells how those bytes are to be interpreted. The type of a variable specifies both the size of the memory allocation and the encoding (and also other information not discussed here).

1.1 Memory Allocation

C integer types come in various lengths. The shortest is `char`, which is generally one byte long. The longest is `long long int` which is eight bytes (64-bits) long on many machines. The C standard does not specify the actual memory sizes of its types; it only stipulates an ordering of types such that "shorter" types must not have longer memory allocations than "longer" types. We have, in increasing order: `char`, `short int`, `int`, `long int`, `long long int`. On the 64-bit Zoo machines, `char` is 1 byte, `short int` is 2 bytes, `int` is 4 bytes, and `long int` and `long long int` are both 8 bytes.

The `sizeof` operator returns the number of bytes in the memory allocation for its argument, which may be either an expression or a type name in parentheses. Thus, if `x` is a variable of type `int` on a machine in which `int` is 4 bytes long, then `sizeof(int)` and `sizeof x` both return 4.[1]

1.2 Encodings

C supports two different encodings for integers: `signed` and `unsigned`. If neither keyword is present, `signed` is assumed (except in the case of unadorned `char`, which is implementation dependent and may be either `signed` or `unsigned`).

The encoding determines which range of values the variable can represent. For types with n bits in their memory allocation, the signed encoding can represent any of the 2^n integers in the range $[-2^{n-1},\ 2^{n-1}-1]$, and the unsigned encoding can represent any of the 2^n integers in the range $[0,\ 2^n-1]$.

For example, a 2-byte `short int` has a 16-bit memory allocation, so it can represent integers in the range $[-32768,\ 32767]$. A `short unsigned int` has the same sized memory allocation, but it can represent integers in the range $[0,\ 65535]$.

1.3 Representations

For most purposes, how C represents integers internally is of no concern to the programmer, but there are constructs in C that expose the internal representation to view. What I describe here is the representation that is used on most modern machines.

[1] The parentheses are required when `sizeof` is applied to a type name but not when it is applied to an expression.

Unsigned numbers An unsigned number u is represented by a sequence of n bits $\mathbf{b} = (b_{n-1}, \ldots, b_0)$, with b_0 being the low-order (least significant) bit and b_{n-1} the most significant, such that

$$u = \sum_{i=0}^{n-1} b_i \cdot 2^i. \tag{1}$$

This is just ordinary base-2 radix ("binary") notation.

Signed numbers A signed number v is also represented by a sequence of n bits $\mathbf{b} = (b_{n-1}, \ldots, b_0)$, but the high-order bit b_{n-1} is the sign bit s, and bits $b_{n-2}, \ldots, b_0$ describe an unsigned number u in the range $[0, \ldots, 2^{n-1} - 1]$. If $s = 0$ (meaning positive), then $v = u$. If $s = 1$ (meaning negative), then $v = u - 2^{n-1}$. In terms of formulas, we have

$$v = \left(\sum_{i=0}^{n-2} b_i \cdot 2^i \right) - b_{n-1} \cdot 2^{n-1}. \tag{2}$$

This representation of a signed number is called *twos-complement representation*.

For example, if $n = 4$ and $\mathbf{b} = (1, 1, 0, 1)$, then equation 2 gives $v = 1 + 4 - 8 = -3$. Now if one uses ordinary binary addition to add 3 to **b**, one gets

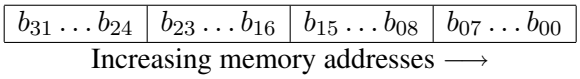
	1	1	0	1
+	0	0	1	1
1	0	0	0	0

Discarding the carry out of the high-order position results in the answer $(-3) + 3 = 0$ as one would expect.

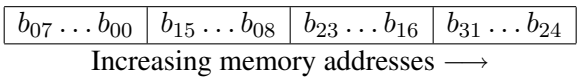
The pleasant fact about twos-complement representation (from the hardware designer's point of view) is that numbers can be added and subtracted without regard to whether they are signed or unsigned – the correct answer is always obtained as long as the result is within the range that can be represented.

Byte order One other property of number representations that is sometimes important is the order in which the bytes are laid out in memory. Two byte orderings are in common use. To illustrate this, let $(b_{31}, \ldots, b_0)$ be the 32 bits in a 4-byte integer.

- **Big endian** reads the numbers left-to-right and places them into memory in the natural way, so the first byte contains bits $(b_{31}, \ldots, b_{24})$, the second byte contains $(b_{23}, \ldots, b_{16})$, the third byte contains $(b_{15}, \ldots, b_8)$, and the last byte contains $(b_7, \ldots, b_0)$. If one pictures memory as being a sequence of boxes, each capable of holding 8 bits, one gets the picture:



  Little endian takes the same four bytes but places them in memory in the opposite order. The little endian representation of this same number is illustrated below:



  Notice that the order of the *bits* within a byte are the same for big and little endian; the difference is only in the order in which the bytes are placed into memory.

Normally, the hardware takes care of fetching the bytes from memory in the right order, so by the time the computer operates on them, they are already in a register in the correct order. The programmer only needs to be concerned about byte order when reinterpreting the storage occupied by an integer as a sequence of bytes, for example, for the purpose of sending those bytes across a network.

1.4 Conversions

C permits conversions from one integer type to another, subject to certain rules. There are two ways to understand these rules: according to the semantics of the types, and according to what actually takes place on the bits. I will describe both here.

Consider an assignment statement `b=a`, where `a` and `b` are of possibly different integer types. We say `a` is the *source* variable and `b` the *target* variable. A conversion is necessary when `a` and `b` differ in length, encoding, or both.

1.4.1 Semantic rules

Two general rules cover the conversions that are guaranteed by the standard:

1. If the integer value of `a` is included in the allowable range of `b`, then `b` assumes that value.
2. If `b` is unsigned, then `b` assumes the unique value in the range $[0, 2^n - 1]$ that is congruent to `a`'s integer value modulo 2^n, where n is the length of `b` in bits.

By rule 1, if `b` is signed, then `a`'s value is preserved if it will fit into `b`, even if `a` is of unsigned type or is longer than `b`. If it won't fit, then overflow is said to have occurred and the standard does not specify what `b`'s new value is.

By rule 2, if `b` is unsigned, then the result of the assignment is always defined, regardless of the length or encoding of `a`. If the value of `a` can be represented by `b`, then both rules 1 and 2 assert that `b` assumes that value. If `a`'s value cannot be represented, then multiples of 2^n are added or subtracted from `a`'s value in order to get a value that *can* be represented, where n is the length of `b` in bits. For example, if $a = -3$ is a signed type and b is an unsigned short (of length 16 bits), then -3 is not in `b`'s range of $[0, 65535]$, but $-3 + 2^{16} = -3 + 65536$ is in range, so the number stored in `b` will be 65533.

1.4.2 Representation-based rules

The other way to understand C conversions is to look at how the representations are manipulated.

If `a` and `b` have the same length, then the bits representing `a` are simply copied unchanged into `b`. When `a` and `b` differ in signedness, then the meaning of the high-order bit changes. Comparing equations 1 and 2, one sees that the values of `a` and `b` will be the same when the high-order bit is 0, and the unsigned value will be 2^n larger than the signed value when the high-order bit is 1.

If `a` is longer than `b`, then the rightmost bits of `a` are copied into `b`, and the rest of the bits of `a` are discarded. This turns out to be consistent with rules 1 and 2, but there are several cases to verify. For example, if both `a` and `b` are unsigned and the discarded bits of `a` are all zero, then indeed `a` and `b` have the same value after the conversion. Similarly, if both are signed and if the discarded bits of `a` all equal the high-order (sign) bit of `b` after the copy took place, then the values of `a` and `b` are equal, even if `a` is negative.

Finally, if `a` is shorter than `b`, then the bits of `a` are copied into the right hand part of `b` and the remaining bits of `b` are filled with either all 0's or all 1's. If `a` is signed, then the sign of `a` is used as

the filler. If `a` is unsigned, then 0 is used as the filler. This give a result consistent with our semantic rules regardless of whether `b` is signed or unsigned.

Suppose `b` is an unsigned int and `a` is a short signed int. The result is the same as if `a` were first converted to a signed int (by sign extension) and then converted to an unsigned int. For example, if `a` = -3, then the value after conversion to a signed int is still -3, and conversion to an unsigned int gives $-3 + 2^{32} = 4294967293$. Note that this is *not* equivalent to first converting `a` to an unsigned short int and then converting that to a long int. In the latter case, the result would be $-3 + 2^{16} = 65533$.

1.5 Summary

To summarize: Rules 1 and 2 describe those conversions for which the C99 standard prescribes the result. These are the only conversions that your program should rely on.

To understand what likely will happen when those rules do not apply, think about the two-complement representation of the numbers involved and use the fact that the conversions are implemented by copying as many low-order bits as will fit into the target, remembering that sign-extension is used when increasing the length of signed source values.