2.2. Primitive Data Types

The primitive data types are provided by the C++ compiler itself. The descriptions that follow give the typical range for each type. Although you need to be careful when targeting widely different platforms, the following ranges should be taken as reliable for all 32-bit systems.

Integer types cannot hold fractional portions. Some integer types are unsigned: Types cannot hold negative values but compensate by being able to hold a larger range of positive values.

▶ char

One-byte integer: This is a data type wide enough to hold a single printable character. Typical range is -128 to 127. The **char** type is special in that if a value is printed as a **char**, a character is displayed rather than its numeric value.

For example, the value 97 is the ASCII character code for “a”. If you send the value 97 to the console as a **char** value, the character “a” is printed. If you promote the data to **int** type before displaying it, then “97” is printed.

▶ unsigned char

Unsigned version of **char**. Unlike **char**, an **unsigned char** is always treated as its numeric value, even when printed. Range is 0 to 255.

▶ short

Short integer, usually two bytes wide. Range is usually -32768 to 32767.

▶ unsigned short

Unsigned short integer, usually two bytes wide. Range is usually 0 to 65535.

▶ int

“Natural” size integer, reflecting processor size. On a 32-bit system, this type is 32 bits (four bytes) wide, making **int** equivalent to **long**. Typical range is approximately ±2 billion.

▶ unsigned int

Same as **int**, except that it is unsigned. Typical range is approximately 0 to 4 billion.

▶ long

Long integer, usually four bytes wide. Range is approximately ±2 billion.

▶ unsigned long

Unsigned version of **long**. Range is approximately 0 to 4 billion.

▶ long long

Extra-long integer, usually eight bytes (64 bits) wide. Range is approximately ±9 times 10 to the 18th. This type is mandated by the C++11 spec and is not supported by all older compilers; however, some compilers (such as Microsoft) have supported it for years.

▶ unsigned long long

Same as **long long**, except that it is unsigned. Range is approximately 0 to 1.8 times 10 to the 19th. This type is mandated by the C++11 spec and is not supported by all older compilers.

▶ float

Single precision floating point (four bytes wide). Seven digits of precision. Range is ±3.4 times 10 to the 38th. This type is not the default floating-point type; **double** should generally be used instead. (See guidelines in Section 2.2.2, “Floating-Point Types: Guidelines.”)

This type can store 0.0 precisely; otherwise, values can get as close to 0 as 1.175 times 10 to the -38th.

▶ double

Double precision floating point (eight bytes wide). Fifteen digits of precision. Range is ±1.8 times 10 to the 308th. This is the natural floating type and is preferred except when storage space is at a premium. All floating-point values are converted to **double** when they appear in expressions.

This type can store 0.0 precisely; otherwise, values can get as close to 0 as 2.225074 times 10 to the -308th.

▶ long double

At least as wide as **double**, but compilers may use it to support a larger range and precision. See your vendor’s documentation for more information.

▶ bool

Boolean: This is an integer value that holds the value **true** or **false**; every non-negative value assigned to a **bool** variable is converted to **true**. Zero equates to **false**.

▶ w_char

Wide character field, for holding character data for extended character sets.

2.2.1. Integer Types: Guidelines

The following guidelines apply to integer types.

■ Integer data types store values precisely. For example, if you add 1 to a high value of a **long long** integer, the new value will always be distinct from the old. No rounding errors can occur. The limitation on integers is that they cannot hold fractional quantities. But they are both more accurate and more efficient for any data that doesn't involve fractions.

■ The typical ranges listed earlier are nearly universal on the vast majority of computers in use today—especially personal computers. The most notable exception is the **int** type, which (unlike **short** and **long**) is the least fixed of C++ types. This type is the "natural integer" type; it matches the register size of the target platform and is therefore 16 bits wide on a 16-bit system, 32 bits wide on a 32-bit system, and so on. The advantage of this type is that it is presumably the most efficient integer size for a platform to work with (although even that is not a safe assumption in all cases).

■ The **int** type has a drawback. Because its size can potentially vary, code that runs on one platform may break when ported to another. For this reason, **int** should be avoided if you intend to compile your program for a variety of different architectures; **short int** and **long int** are to be preferred where multiple platforms are involved. (These type names, by the way, are equivalent to the more abbreviated names **short** and **long**, respectively.)

■ The C++ specification does not set absolute sizes even for other types—although the sizes given earlier are reliable for 32-bit systems. If your code will be ported to many platforms, the safest approach is to use header files (see Chapter 8, "Preprocessor Directives") to define precise sizes: For example, you might first define INT32 as a **long** when compiling for platforms that implement **long** as 32 bits, and then use INT32 to declare variables. (See the **#define** directive in Chapter 8 and the **typedef** keyword in Section 2.9.1, "The typedef Keyword.")

2.2.2. Floating-Point Types: Guidelines

The following guidelines apply to floating-point types (**float** and **double**).

■ Floating-point types can be subject to rounding errors, because they store values as a combination of mantissa and exponent. Therefore, always use an integer type when you have a choice, as long as its range is sufficient and no fractional amounts are involved.

■ During program execution, all floating-point data is promoted to type **double** (the larger of the two formats) before being evaluated in arithmetic expressions.

■ For this reason, it may seem pointless to use the **float** type at all, because it is promoted to **double** when being evaluated in an expression. But when storage space is at a premium and you are using many numbers (storing them in a file or creating large arrays) you might consider using **float**; otherwise, **double** is always preferable.

■ Note that in addition to storing fractional portions, floating-point types have bigger ranges than integer types. Therefore, floating-point data is often used just for its greater range, especially in scientific applications. (However, note that the new **long long int** type also provides an exceptionally large range while also preserving the absolute precision provided by an integer type.)