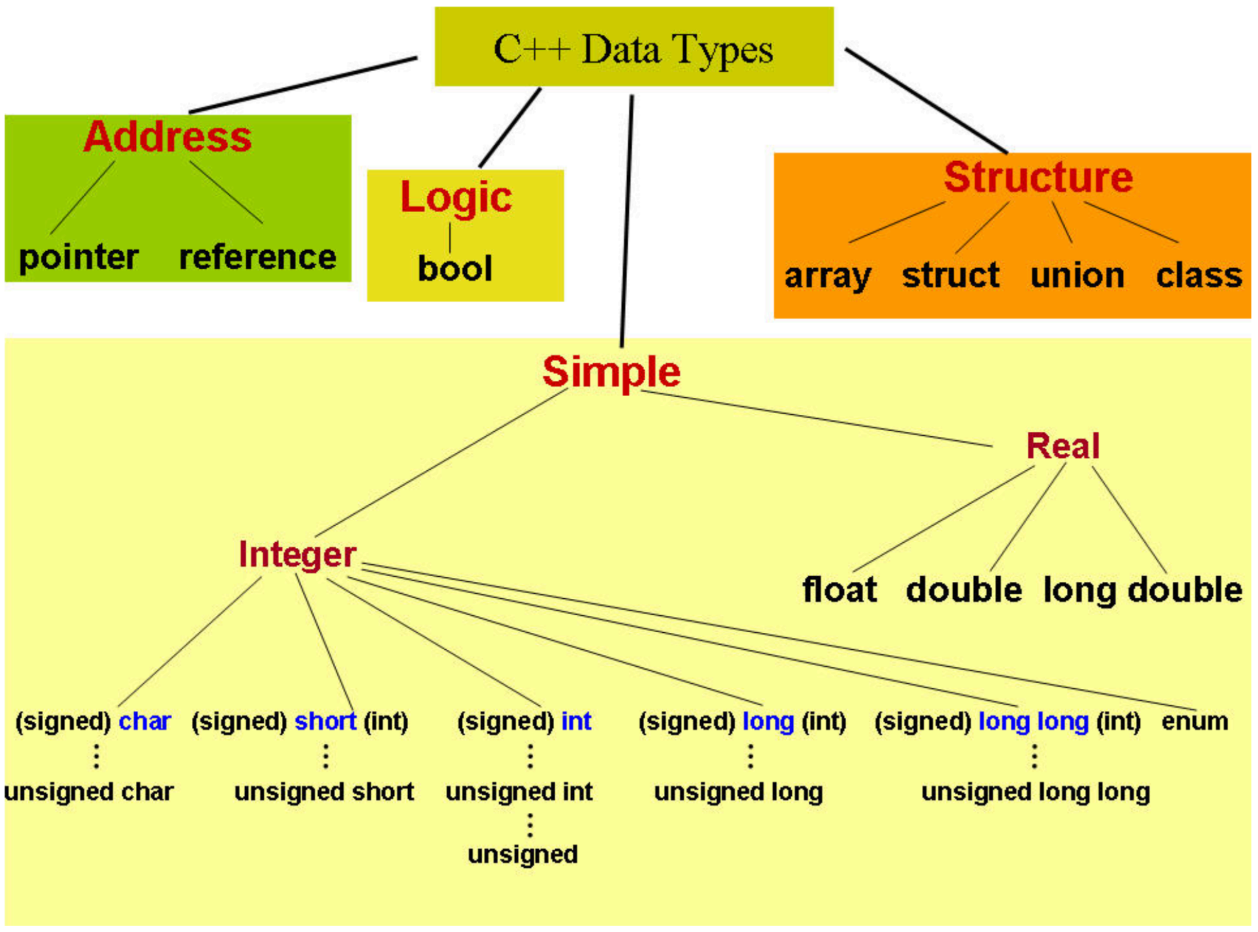
Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Lecture 03



What are Types?

A data type denotes a set of elements that obey certain properties. These properties are defined mathematically, since computing is based on mathematics and logic. The simplest data type is **`bool`**, which has only two possible values, **`true`**, or **`false`**. These two values are the possible outcomes of a logical expression. There are three logical operations defined on the **`bool`** data type: **`!`** (not), **`&&`** (and), and **`||`** (or). These operations are defined by the following truth table

bool X	bool Y	!X	X && Y	X||Y
false	false	true	false	false
false	true	true	false	true
true	false	false	false	true
true	true	false	true	true

The operator **`!`** is classifies as a *unary* operator since it only requires one argument. In C++ an operator is a shorthand form of a function. A longer, more cumbersome, but equivalent form is the following function. The function takes an argument, **`x`**, and returns a **`bool`** value

```
bool not(bool x) {
  return !x;
}
```
Clearly, the operator form is much more convenient. Similarly, the **&&** and **||** functions are classified as *binary* operators, since they operate on two arguments. For example, the following function is equivalent to the **&&** operation

```
bool and(bool x, bool y) {
  return x&&y;
}
```

In general, a data type attains a data structure, which can only attain certain values. The data types illustrated in the figure above are called *primitive* or *built-in* data types of the C++ language. C++ is very flexible and allows new data types to be defined in terms of existing data types. It is also possible to define a new name for a type that has more significance for a given application. For example, if one is programming a simulator for digital logic gates, it is meaningful to define a data type for the logic signals at the input and outputs of gates. A reasonable name for this type could be **signal_t**. When a programmer sees a variable with this type he knows that it is defining a digital logic signal. C++ supports this custom type naming using the C++ *keyword* **typedef**. For the logical signal example

```
typedef bool signal_t;
```

The symbol **signal_t** is now a full-fledged data type and can be used to declare new variables throughout the program, just as if it were a built-in type. For example, the following code

```
typedef bool signal_t;
signal_t and_gate(signal_t x, signal_t y) {
  return x&&y;
}
```

implements a function that acts as a digital logic and gate with two inputs signals and one output signal. The use of a primitive **bool** type as the definition of the new **signal_t** data type is really an implementation choice. There is no particular reason for this choice. For example, the data type **signal_t** could be defined as a **char** variable, which is a sequence of 8 binary digits (8 bits). In this case the and_gate function would be implemented as follows

```
typedef char signal_t;
signal_t and_gate(signal_t x, signal_t y) {
  return x!=0 && y!=0;
}
```

where the new operator **!=** tests whether or not a **char** variable is different from zero, and the symbol **0** here means one of the possible values that a **char** variable is allowed to take. The result of this operation is a **bool** value, **true** or **false**. To be precise, the function *signature* of this operator is

```
bool not_equal(char x, char y);
```

Note that the signature of the two implementations of the **and_gate** function is the same in both cases

```
signal_t and_gate(signal_t x, signal_t y);
```

and the two implementations produce the same results when the inputs are the same, independently of how the **signal_t** data type is defined. In fact, since addressing a single bit in memory is not practical in modern computer architectures, many C++ compiles use the **char** data type to represent **bool** variables.

It is good programming practice to use **typedef** when variable types are likely to change as computer architectures evolve, because in that case the change of variable type is limited to one line of code. For example, the dimension of a vector of values, or a block of memory, should be defined as **size_t**, instead of unsigned int, since the size of a vector can have a different maximum size for 64 bits architectures than for 32 bits architectures. By using **typedef**, the programmer can control the capacity of the size variable.

The downside of overusing **typedef** is that unusual names may not suggest the proper semantics for the type, and lead to code that is difficult to understand. The best policy is to only use **typedef** when a type is widely used throughout the code and is likely to need to be redefined as the design evolves.

Integer Types

Integer types Integer types are variations on the mathematical concept of *integer*. There are three categories of integers: strictly positive, zero, and negative. The full set of integers is usually defined by the symbol $\boldsymbol{Z}$. The strictly positive integers, $\boldsymbol{Z}^+$,can be defined by a variant of Peano's axioms

- 1 is a member of the set $\boldsymbol{Z}^+$.
- If n is a member of $\boldsymbol{Z}^+$, then n+1 belongs to $\boldsymbol{Z}^+$ (where n+1 is the successor of n).
- 1 is not the successor of any element in $\boldsymbol{Z}^+$.

The zero integer can be defined as,

- 0 is in the set $\boldsymbol{Z}$.
- 1 is the successor of 0 in $\boldsymbol{Z}$.

Thus the nature of integers is fully expressed by the properties of the *successor* function. Intuitively, any positive integer, except 1, is the successor of the next smaller integer.

The negative integers arise because the solution for equations involving the operation of + (addition) must exist regardless of the values being added. For example, the expression $x + 1 = 0$ cannot have a solution in the set $\boldsymbol{Z}^+ \cup \{\boldsymbol{0}\}$. Thus the negative integers $\boldsymbol{Z}^-$ is the set of solutions x, of $x + y = 0$, for all $y \in \boldsymbol{Z}^+$. In C++, the set of non-negative integer numbers $\boldsymbol{Z}^+ \cup \{\boldsymbol{0}\}$ correspond to the data types called **unsigned**, and the full set of integer numbers $\boldsymbol{Z} = \boldsymbol{Z}^+ \cup \{0\} \cup \boldsymbol{Z}^-$ correspond to the data types called **signed**. Since it is far more common to need the full set of integers in computation, the **signed** designation is assumed by default. So for example, the data type **signed int** is the same as **int**.

In addition to the operator +, the arithmetic operators that apply to integers are:

-	(subtraction) ;
*	(multiplication);
/	(division ;
%	(remainder).

A problem arises with division since the ratio of two integers is not in general an integer. For example the value of 1/3 cannot be expressed as an integer. One solution is to define a new data type, called **rational_number**, which keeps the numerator and denominator as separate quantities. The set of integer numbers is a subset of this rational numbers, where the denominator is equal to 1. Since the ratio of two rational numbers is always a rational number, the problem would be solved. For example 1/3 = (1/1) / (3/1).

However, in programming it is often the case that this pure mathematical approach is perverted to gain efficiency in storage of variables. Until just recently, arithmetic instructions in computers operating on real (floating point) numbers were very slow compared to operations on integers. Thus there was a huge driving force to carry out all computations on integers (fixed point). The problem of division was side-stepped by keeping large enough integers to hold the equivalent of fractional results. This *scaling* problem, is then handled by the programmer. For example, if it is known that the divisors are never bigger than 100, then all values can be scaled by 100 to avoid the need for fractions. In the case of 1/3, the scaled result becomes 100/3 or 33 to the nearest integer. At the end of a long computational chain, the result can be un-scaled back to a real number with little overall overhead. More precision can be gained by using large scale factors.

To illustrate the operations of division and remainder consider the following program fragment.

```
int idiv, irem;
idiv = 100/3; // after operation idev = 33
irem = 100%3; // after operation irem = 1
```

The first line of the fragment is called a declaration, which tells the C++ compiler what type of variable is associated with the variable symbol. In this case the variables **idiv** and **irem** are being defined as signed integers. The remainder is 1 since 3*33 = 99.

Integer Precision

So far it has been assumed that the integer type can represent the full set of integers $\boldsymbol{Z}$. Clearly, no such representation is possible because the set is infinite in size. Instead, C++ offers a range of integer types corresponding to the size of the set of valid integers for that type. The ranges of the integer types are shown in the table below for a 32 bit CPU architecture

Integer type	Number of bytes	range
`char`	1	-128 to 127
`unsigned char`	1	0 to 255
`short`	2	-32,768 to 32,767
`unsigned short`	2	0 to 65,535
`int`	4	-2,147,483,648 to 2,147,483,647
`unsigned int`	4	0 to 4,294,967,295
`long`	4	-2,147,483,648 to 2,147,483,647
`unsigned long`	4	0 to 4,294,967,295
`long long`	8	9,223,372,036,854,775,807 to 9,223,372,036,854,775,808
`unsigned long long`	8	0 to 18,446,744,073,709,551,615

These ranges can vary according to the architecture of the CPU and the operating system. For example, for Unix on a 64 bit architecture, the size of a **long int** is 8 bytes. The actual integer range for your machine and architecture can be determined by the function **sizeof(type)**. For example, **sizeof(int)** will return 4. Note that **sizeof()** works for variables as well, as illustrated by the following fragment.

```
long my_long_variable = 20L; //declaration with assignment
unsigned my_size = sizeof(my_long_variable); //my_size equals 4
```

The value of **my_long_variable** is being declared to be of type long. The function returns the value 4 as expected. Note that a variable can be declared and assigned a value at the same time. There can be ambiguity as to the type of integer constants in an assignment statement. In the example, the constant 20 could be any of the integer types. To make it clear that the constant is of type long, the value ends with either a lower or uppercase "L." There are two other integer constant suffixes: lower or uppercase "U" that designates the constant as **unsigned**; and lower or uppercase "LL" that designates the constant as **long long**.

Type Conversion

Since there are a number of choices for the representation of an integer, the question of conversion between types arises. The C++ language supports *implicit* and *explicit* type conversion, also called a *cast*. Implicit type conversion occurs during arithmetic operations. Consider the following fragment.

```
unsigned short sy;
int ix = 10;
sy = ix*20;
```

What is the type of the constant **20** in the last statement? Since the type of **ix** is **int** then the type of the constant **20** will also be **int**. The product takes on the value of 200 as an **int** type. The full set of conditions for implicit type conversion for binary arithmetic operations are:

- If either operand is of type **unsigned long**, the other operand is converted to type **unsigned long**.
- If preceding condition not met, and if either operand is of type **long** and the other of type **unsigned int**, both operands are converted to type **unsigned long**.
- If the preceding two conditions are not met, and if either operand is of type **long**, the other operand is converted to type **long**.
- If the preceding three conditions are not met, and if either operand is of type **unsigned int**, the other operand is converted to type **unsigned int**.
- If none of the preceding conditions are met, both operands are converted to type **int**.

In the final line of the previous code fragment, the equal sign performs an implicit conversion to a **short** type with the value **200**. What happens if the value of the right hand side exceeds the range of an **unsigned short**? For example the constant **20** in the program could be changed to **100000**. The result is undefined and such "overflow" operations should be avoided.

Explicit type casting is accomplished using the expression,

```
static_cast<T>(variable)
```

The symbol **T** stands for the type to which the variable is to be converted. This syntax is expressing a C++ *templated* function, but discussion of templates will be deferred until later. It is good coding practice to use explicit type casting so that the result is what is intended and this style will minimize the number of compiler warnings about casting conflicts.

As an example, the following code fragment gives some examples of casting between the types that have been covered so far.

```
bool b1 = true, b2;
int i1 = 20, i2;
unsigned short us;
int si = -1;
i2 = static_cast<int>(b1);
b2 = static_cast<bool>(i1);
```

```
us = static_cast<unsigned short>(si);
```

The first cast statement is converting a bool to an integer. The result will depend on the particular compiler representation for bool. In the case of Visual C++, the result is that b1(true) is cast to integer 1. In the second cast the integer 20 is cast to true. Indeed any non-zero integer will cast to true, and zero will cast to false. The third cast is problematic in that there is a cast between incompatible types, i.e. between a signed integer and unsigned integer. The result in this case is that us takes on the value 65535, the maximum unsigned short value.

The char Data Type

The **char** data type is typically used to represent string characters and thus the name. Discussion of the representation of strings will be taken up later after the concept of an array has been introduced. A single character can be assigned to a **char**. The syntax of the assignment uses single quotes, as follows,

```
char s = ‘a’;
```

The characters are expressed internally as integer values defined by the American Standard Code for Information Interchange (ASCII). The numerical definition of the ASCII characters is given by http://en.wikipedia.org/wiki/ASCII. For example ‘a’ has the integer value 97.

Comparison Operators

While still focused on integers, it is a good time to discuss the operations for comparing quantities. From a formal point of view, a binary comparison operator is a function on two variables of the same type into the bool type. The binary C++ comparison operators are:

==	(equal);
!=	(not equal);
<	(less than);
<=	(less than or equal) ;
>	(greater than);
>=	(greater than or equal) .

The unary logical **!** operator also behaves as a comparison operator. When a variable is tested, if its value is exactly zero, then the **!** operator returns **true**, otherwise it returns **false**.

To be more specific, the signatures for the comparison operators for **int** data type variables are

```
        bool operator==(int x, int y);
        bool operator!=(int x, int y);
        bool operator<(int x, int y);
        bool operator<=(int x, int y);
        bool operator>(int x, int y);
        bool operator>=(int x, int y);
        bool operator!(int x);
```

The following code fragment indicates several examples of comparisons of integers.

```
int i1 = 30;
unsigned i2 = 6;
unsigned short us = 31;
short sus = 31;
long l2 = 30L;
char c = 5;
unsigned zero = 0;
bool int_equal_long = (i1 == l2); // result is true
bool short_neq_long = (us != l2); // result is true
bool int_gt_char = (i2 > c); // result is true
bool us_gteq_i1 = (us >= i1); // result is true
bool us_eq_sus = (us == sus); // result is true
bool not_zero = !zero; // result is true
```

The comment lines indicate the results of the tests. The C++ compiler can generate warnings if there are inconsistencies in the comparison. For example, a common warning is to compare a signed integer with an unsigned integer. In the example, **`us == sus`**. The problem is that the binary representation of a signed integer is indistinguishable from an unsigned integer.

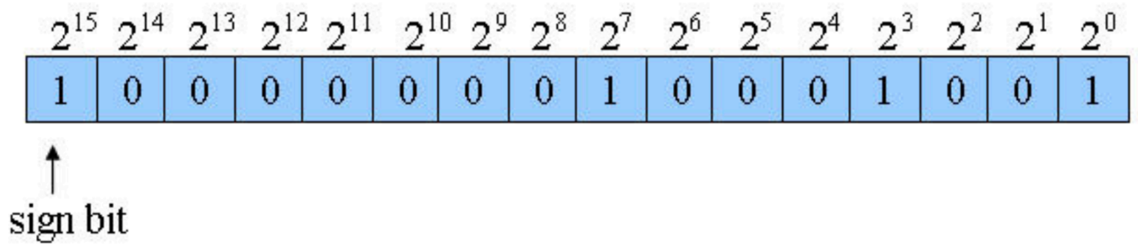


Figure 2 A representation of a signed short integer.

A typical representation for a short integer is shown in Figure 2. The short is stored in two bytes. The leftmost bit is called the sign bit. If the short is negative then the sign bit is set to "1." Otherwise the sign bit is zero. Thus, there are 15 binary digits to represent the values of the short integer range, -32,768 to 32,767. Note that 2^{15} = 32,768. To avoid having two representations for 0, 2's complement is used to represent negative numbers. Negative Unfortunately, the C++ compiler has a difficult problem in interpreting the sign bit when comparing with an unsigned short. The unsigned short has a maximum value of 65,535, which is incomparable with a maximum signed value of 32,767. The compiler's only recourse is to issue a warning and hope that the programmer realizes that there is a problem. Sadly, the standard in C++ compilers for how warnings are triggered is far from uniform.

Real Types

For many mathematical operations it is necessary to represent real numbers. Despite both being infinite, the set of real numbers is a very different from the integer numbers. For example, no matter how close two real numbers are to each other, there are as many real numbers between them as the total number of real numbers. In a loose sense, real numbers are doubly infinite.

The need for real numbers arises because not all results of mathematical operations on rational numbers are in the set of rational numbers. For example, it is easy to prove that $\sqrt{2}$ cannot be a rational number. Thus, in order to have a complete arithmetic computing system it is necessary to represent real numbers and operations on the real numbers. This task is significantly more difficult than representing integers. The form of a real number is written as,

$\sqrt{2}$ = 1.41421356237309504880168872420969807856967187537694807317667973799

where the number of digits in the number is called the precision. In this example the precision is 66. The true value of $\sqrt{2}$ requires infinite precision, so any real number representation of an irrational number, or repeating fraction such as 1/3, will have *round off* error. This inaccuracy is the bane of numerical computing. Since computations cannot be carried out exactly, the effect of such errors on computational results have to be studied in great detail, in order to avoid nonsensical answers. For example, suppose that a computation only maintains three digits of precision. Consider the real number, 0.01, which is clearly not zero. However if this number is squared, the result is 0.00. Taking the square root of this result leads to the contradiction that 0.01 = 0.00. This result is an example of *underflow*.

A standard computer representation of real numbers is the IEEE 754 format. The representation is shown in Figure 3. A real number, represented as a *floating point* data type comprises the product of three components: f = sign × $2^{exponent}$ × fraction.

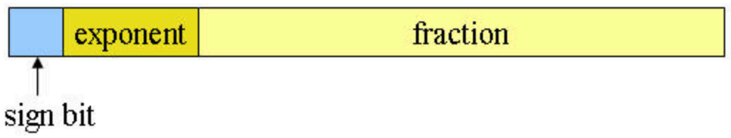


Figure 3: The IEEE standard representation of real numbers (floating point).

The fraction component is stored as a binary number of the form 1.XXX. Since the leading digit is always "1" it is not explicitly stored.

In order to avoid manipulating negative numbers in the exponent field, the field value is biased so that it always represents a zero or positive integer. The bias is accounted for when the number is evaluated.

An example of the IEEE 754 format for a 32 bit **float** real data type is shown in Figure 4. The real number being represented is 0.1.

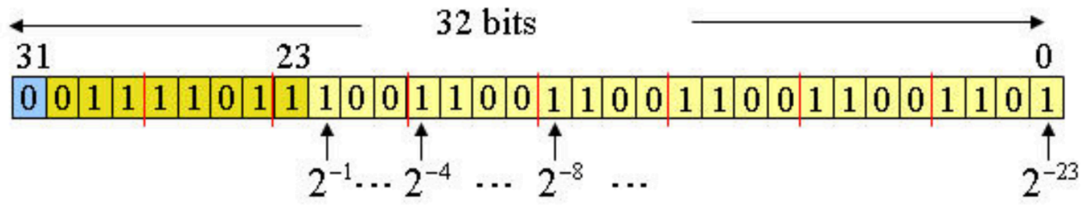


Figure 4: The float data type represented by the IEEE 754 standard.

The red marks indicate so called "nibble" boundaries. A nibble, 4 bits, is 1⁄2 of a byte. A useful encoding of digital bit sequences is the *hexadecimal*, or hex, notation. The hex digits are **{0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}**. For example a hex **f** is a nibble with all binary "1" digits. In C++ notation, the hexadecimal value of the 32 bit number in Figure 4 is **0x3dcccccd**. The prefix **0x** indicates that a value is hexadecimal.

The number is seen to be positive since the sign bit is zero. The exponent field has the hex value **0x7b**, or decimal value 123. The bias value is 127, so the exponent is actually -4. The fractional part is an approximation of the value 1.6. This follows because $0.1 = \frac{1.6}{16}$, and $16^{-1} = 2^{-4}$. The fraction bits thus represent an approximation of 0.6. That is $0.6 = \frac{1}{2} + \frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} + \cdots + \frac{1}{2^{23}}$. The value 0.6 is a repeating binary fraction and cannot be exactly represented as a finite series, so the last term is rounded up to achieve as close a value to 0.6 as possible. The actual value of the IEEE representation in Figure 4 is **0.10000000149011612**. The format of a 64 bit real number (double) is shown in

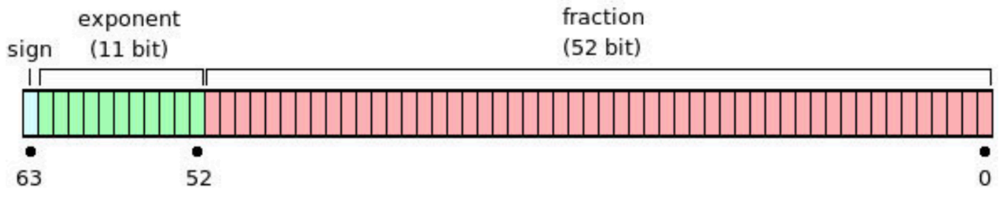


Figure 5: The format of a 64 bit floating point number.

C++ also supports a form of scientific notation for real numbers. The syntax is **x.xxxxxesnnn** where **x.xxxxx** is a real number with fraction **e** is the letter e, **s** is either **+** or **–** and **nnn** is an integer exponent. The ranges of real number types in C++ are shown in the following table.

real type number of bytes largest magnitude smallest magnitude float 4 double 8 long double 8 same

same

Real type	Number of bytes	Largest magnitude	Smallest magnitude
`float`	4	3.4028235e+038	1.1754944e-038
`double`	8	1.7976931348623157e+308	2.2250738585072014e-308
`long double`	8	1.7976931348623157e+308	2.2250738585072014e-308

The full range is given by plus and minus times the largest magnitude. Note that **`double`** and **`long double`** are identical in range. However in the future as CPU word lengths grow longer, **`long double`** might be extended to, 12 or even 16 bytes.

As before, there can be ambiguity in the type of a real constant. For example,

```
float fval = 0.3174f;
double dval = 1.414;
float cfval = static_cast<float>(dval);
```

In the first line, the C++ compiler assumes any number with a decimal point is of type **`double`**. The lower or upper case "f" suffix indicates to the compiler that the number is to interpreted as a **`float`** type. Without the "f" the compiler might generate a warning that the precision of the number is being degraded during the implicit cast by the assignment to **`fval`**. The second line is fine, since the real constant is interpreted as a double.

The cast operation also holds for real numbers, as shown by the third line of code above. There, the compiler is being told that the programmer knows there could be some loss of precision, but that is what is intended.

The finite precision of computer representation can cause significant errors if allowed to accumulate over a large number of operations. For example, suppose the task is to integrate a function, such as $c = \int_0^{1000} x^3 dx$. One approach is to convert the integral into a sum $c = \Delta \sum_0^{1000/\Delta} x^3$. The exact answer is $\frac{x^4}{4}|_0^{1000} = 2.5e + 011$. The table below shows the integral computed by the sum approximation with various values of Δ.

Δ	c with float computation	c with double computation
1.0	2.4950021e+011	249500250000.00000
0.1	2.4984992e+011	249950002500.00006
0.01	2.4999505e+011	249995000025.00012
0.001	2.4999002e+011	249999500000.24460
0.0001	2.5130222e+011	249999949999.99191
0.00001	1.8014398e+011	249999984999.99167
0.000001	1.8014398e+010	249999999500.05374

For larger values of Δ , the sum is inaccurate because the integration approximation is too crude. For smaller values of Δ the result is inaccurate because of the finite limit of the real number precision. Clearly, **`double`** precision is more effective than **`float`** precision.

The implicit conversion of real types during binary operations following these rules:

1. If either operand is of type **`long double`**, the other operand is converted to type **`long double`**.
2. If the above condition is not met and either operand is of type **`double`**, the other operand is

converted to type **double**.

3. If the above two conditions are not met and either operand is of type **float**, the other operand is converted to type **float**.

Conversion from real to integer

It is often necessary to convert from a real number to an integer. The key issue is how the fractional part of the real number is to be treated. There are three possible treatments:

1. round down to the nearest smaller integer – called a **floor** operation
2. round up to the nearest larger integer – called a **ceil** operation
3. round down if the fraction is less than 0.5 and round up if the fraction is >= 0.5 – called a **round** operation.

The default behavior by the C++ compiler is to simply remove the fraction part, which is the same as a floor operation for positive numbers and the ceil operation for negative numbers. The round operation can be achieved simply by adding 0.5 to the real value before casting to an integer. For example,

```
float x = 1.45, y = 1.82;
int ix = x+0.5, iy = y+0.5; // ix = 1, iy = 2
```

Not a Number

Consider following program

```
int xzero = 0;
int y = 1;
cout << y/xzero << endl;
```

This result is not representable as an integer in C++ and therefore will return an error condition during run time.

For floating point calculations the IEEE standard requires that calculations should always be carried out even if the result is not representable or meaningless. The following program produces a number too large for a double to represent.

```
double yd = 1.0, xdzero= 1.0e-155;
cout << yd/xdzero << ' ' << yd/(xdzero*xdzero)<< endl;
```

The output of this program is,

```
1e+155 1.#INF
```

The output of the second expression indicates that a number too large to be represented by a double type has been encountered and is given the value shown. This condition can be tested by the function (in visual C++)

```
#include <float.h>
int _finite(double value)
```

The leading underscore indicates that the function is part of the Windows C++ environment. In the example, **_finite(yd/(xdzero*xdzero))** returns **0** indicating a **false** bool result.

It is also possible to generate indeterminant results. For example,

```
double inf = yd/(xdzero*xdzero);
cout << 0*inf << endl;
```

prints out the result,

```
-1.#IND
```

The bit structure of #INF and #IND are shown in Figure 6 for 32 bit real numbers (float). It is seen that the difference is that the most significant bit of the fraction field is set in the case of an indeterminate result.

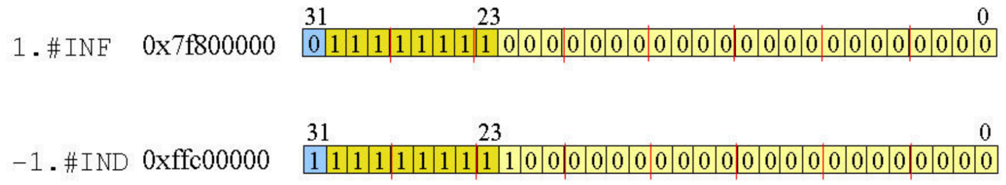


Figure 6: The format of infinite and indeterminate real 32 bit numbers.

A bad real number is represented by all "1" digits in the exponent field. The fraction field can be used to represent a large number of classifications of "bad." Some compilers use the symbol **`NaN`** to indicate that the result of the computation is not valid. Some also distinguish two types of **`NaN`**s, a "quiet NaN" , **`QNaN`**, indicating an indeterminate result, and a "signaling NaN", **`SNaN`**, indicating an exceptional or error condition. For example, for Visual C++ the float value of **`0x7fffffff`** has the value, **`1.#QNAN00`** indicating a quiet **`NaN`**. It cannot be expected that the behavior of these classifications will be standard across compilers.