

CS319: Scientific Computing (with C++)

<http://www.maths.nuigalway.ie/~niall/CS319/>

ints and float; input & output; flow; loops; functions

Week 2: **9am and 4pm**, 18 Jan 2017



Source: [xkcd \(292\)](#)

Today:

1 What we learned about C++ last week

2 `int`: a closer look

3 `float`: a closer look

- `double`

4 Output Manipulators

- `endl`

- `setw`

5 Input

6 Flow of control: `if`-blocks

7 Loops

- `for` loops

8 Functions

- E.g, Prime or Composite?

- `void` functions

What we learned about C++ last week

- A “header file” is used to provide an interface to standard libraries. Every program that we will write will include the line:

```
#include <iostream>
```

- The heart of the program is the `main()` function – every program needs one. `void` is the default argument list and can be omitted.

- The C++ language is case-sensitive.

- “Curly brackets” are used to delimit a program block.

- Every (logical) line is terminated by a semicolon; `i=10; i++;`

- Two forward-slashes `//` indicate a comment – everything after them is ignored until an end-of-line is reached.

C-style: `/* ignore this */`

What we learned about C++ last week

- Use `cout` to send a output to (for example) a terminal window:
`std::cout << "Howya World." << "\n";`
- All variables must be declared before begin used.
- Each variable has a **type** that determines what sort of value will be stored in it.
- The most basic data types are
 - ▶ `int` used to store integers
 - ▶ `floats` – numbers that have a fractional (decimal) part.
 - ▶ `char` – stores a character (i.e., symbol)
- We can declare **arrays** as follows:
`int Fib[10];`
Arrays are indexed from 0.
- A *string* is used to store a sequence of characters. (Not a fundamental data-type — it is actually an object).

What we learned about C++ last week

- The objects defined when we ~~include~~ `#include <iostream>` are not truly global in scope, but belong to the `std` namespace. A **namespace** is a declarative region that localises the names of identifiers, etc., to avoid name collision. These are then accessed as `namespace::identifier`.
So, to use the `cout` object provided by `iostream`, and which belongs to the `std` namespace, refer to it as `std::cout`
- If one wants to avoid having to refer to the namespace, you can use the `using` key-word. Example: if your code includes `using namespace std;` then referring directly to `cout` is the same as `std::cout`.

.....
The use of “using” is not regarded as good programming practice..

int: a closer look

"bit" = "binary digit"

It is important for a course in scientific computing that we understand how numbers are stored and represented on a computer.

Your computer stores numbers in binary. That is, in base 2. The easiest examples to consider are integers.

Examples:

$_{10} = 111_2 = 2^0 + 2^1 + 2^2$
= 8 bits

8 bits

If we use a single byte to store an integer, then we can represent:

Smallest: 0000 0000

Smallest: 0000 0000
largest: 1111 1111 = 255 = $2^8 - 1$.

In reality, 4 bytes are used to store each integer. One of these is used for the sign. Therefore the largest integer we can store is $2^{31} - 1$...

4 bytes \Rightarrow 32 bit, with remaining 81 bits store number.

We'll return to related types (`unsigned int`, `short int`, and `long int`) later.

always non-negative.

float: a closer look

So if $sign = 0$, $x \geq 0$
if $sign = 1$, $x < 0$

C++ (and just about every language you can think of) uses IEEE Standard Floating Point Arithmetic to approximate the real numbers. This short outline, based on Chapter 1 of O'Leary "Scientific Computing with Case Studies".

The format of a float is $x = (-1)^{Sign} \times (Significant) \times 2^{Exponent}$ where

- *Sign* is a single bit that determines if the float is positive or negative;
- the *Significant* (also called the "**mantissa**") is the "fractional" part, and determines the precision;
- the *Exponent* determines how large or small the number is, and ~~usually~~ ^{has on} involves an offset (See below). (rather than a sign bit).

A **float** is a so-called "single-precision" number, and it is stored using 4 bytes (= 32 bits). These 32 bits are allocated as:

- 1 bit for the *Sign*;
- 23 bits for the *Significant* (as well as an leading implied bit); and
- 8 bits for the *Exponent*, which has an offset of $e = -127$.

float: a closer look

So this means that we write x as

$$x = \underbrace{(-1)^{\text{Sign}}}_{1 \text{ bit}} \times 1. \underbrace{\text{abcdefghijklmnopqrstuvw}}_{23 \text{ bits}} \times \underbrace{2^{-127 + \text{Exponent}}}_{8 \text{ bits}}$$

Since the *Significant* starts with the implied bit, which is always 1, it can never be zero. We need a way to represent zero, so that is done by setting all 32 bits to zero.

Thus, the smallest the *Significant* can be is $1.\underbrace{000000000000000000000000}_{22 \text{ zeros}}1 \approx 1.$

The largest it can be is $1.\underbrace{111111111111111111111111}_{23 \text{ ones}} = 2 - 2^{-23} \approx 2.$

Here it helps to remember that the binary fraction 1.1 means (in decimal) $1 + \frac{1}{2}$, 1.11 means $1 + \frac{1}{2} + \frac{1}{4}$, etc.

The *Exponent* has 8 bits, but since they can't all be zero (as mentioned above), the smallest it can be is $-127 + 1 = -126$. That means the smallest positive float one can represent is

$$x = (-1)^0 \times 1.000 \cdots 1 \times 2^{-126} \approx 2^{-126} \approx 1.1755e - 38.$$

float: a closer look

We also need a way to represent ∞ or “Not a number” (NaN). That is done by setting all 32 bits to 1. So the largest *Exponent* can be is $-127 + 254 = 127$.

That means the largest positive float one can represent is

$$x = (-1)^0 \times 1.111 \dots 1 \times 2^{127} \approx 2 \times 2^{127} \approx 2^{128} \approx 3.4028e + 38.$$

As well as working out how small or large a **float** can be, one should also consider how **precise** it can be. That often referred to as the **machine epsilon**, can be thought of as *eps*, where $1 - \text{eps}$ is the largest number that is less than 1 (i.e., $1 - \text{eps}/2$ would get rounded to 1. The value of *eps* is determined by the *Significant* For a **float**, this is $x = 2^{-23} \approx 1.192 \times 10^{-7}$.

For a `double` in C++, 64 bits are used to store numbers:

- 1 bit for the *Sign*;
- 52 bits for the *Significant* (as well as an leading implied bit); and
- 11 bits for the *Exponent*, which has an offset of $e = -1023$.

The smallest positive double that can stored is $2^{-1022} \approx 2.2251e - 308$, and the largest is

$$1.111111 \dots 111 \times 2^{2046-1023} = \left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots\right) \times 2^{2046-1023} \\ \approx 2 \times 2^{1023} \approx 1.7977e + 308.$$

(One might think that, since 11 bits are devoted to the exponent, the largest would be $2^{2048-1023}$. However, that would require all bits to be set to 1, which is reserved for NaN).

For a `double`, machine epsilon is $2^{-53} \approx 1.1102 \times 10^{-16}$.

Finished here 10 am

An important example:

Week02/01Rounding.cpp ← link!

```
int i, max;  
float x, increment;
```

```
cout << "Enter a (natural) number, n: ";
```

```
cin >> max;
```

```
x=0.0;
```

```
increment = 1/( (float) max);
```

start ↗

```
for (i=0; i<max; i++)
```

```
    x+=increment;
```

```
std::cout << "Difference between x and 1 is " << x-1 << "\n";
```

↙ Cost "more" from int to float.

↙ Increment $i++ \Leftrightarrow i = i + 1$

- If we input $n = 8$, we get: 0, because $1 - 8(1/8) = 1 - 1 = 0$.
- If we input $n = 10$, we get: 1.19209×10^{-7}

As well as passing variable names and strings to the output stream, we can also pass manipulators to change how variable values are displayed. Some manipulators (e.g., `setw`) require that `iomanip` is included.

- `endl` print a new line (and flush)

02Manipulators.cpp

```
10 #include <iomanip>
12 int main()
13 {
14     int i, fib[16];
15     fib[0]=1; fib[1]=1;

16     cout << "\n\nWithout the setw manipulator" << endl;
17     for (i=0; i<=12; i++)
18     {
19         if( i >= 2)    fib[i] = fib[i-1] + fib[i-2];
20         cout << "The " << i << "th " <<
21             "Fibonacci Number is " << fib[i] << endl;
22     }
}
```

- `setw(n)` will the width of a field to n . Useful for tabulating data.

02Manipulators.cpp

```
24  cout << "\n\nWith the setw  manipulator" << endl;  
    for (i=0; i<=12; i++)  
26  {  
        if( i >= 2)    fib[i] = fib[i-1] + fib[i-2];  
28  cout << "The " << setw(2) << i << "th " <<  
        "Fibonacci Number is " << setw(3) << fib[i] << endl;  
30  }  
    return(0);  
32 }
```

.....
Other useful manipulators:

- `setfill`
- `setprecision`
- `fixed` and `scientific`
- `dec`, `hex`, `oct`

Input

In C++, the object `cin` is used to take input from the standard input stream (usually, this is the keyboard). It is a name for the **C**onsole **I**Nput.

In conjunction with the operator `>>` (called the **get from** or **extraction** operator), it assigns data from input stream to the named variable.

(Later we will see that `cin` is an **object**, with more sophisticated uses/methods than is going to be shown here. However, we will defer this discussion until we have studied something of **objects** and **classes**).

03Input.cpp

```
6 #include <iostream>
  #include <iomanip> // needed for setprecision

  using namespace std;

  int main()
12 {
14     const double StirlingToEuro=1.15735; // Correct 17/01/17
16     double Stirling;
18     cout << "Input amount in Stirling: ";
20     cin >> Stirling;
    cout << "That is worth " << Stirling*StirlingToEuro << " Euros\n";
    cout << "That is worth " << fixed << setprecision(2) <<
        "\u20AC" << Stirling*StirlingToEuro << endl;
    return(0);
}
```

value of this variable cannot change.

Euro symbol.

Flow of control: if-blocks

if statements are used to conditionally execute part of your code.

Structure (i):

```
if( exprn )  
{  
    only logical expression, eg a < 5  
    statements to execute if exprn evaluates as  
    non-zero  
}
```

```
else  
{  
    statements if exprn evaluates as 0  
}
```

Optional.

If { & } are not used, the next line (only) after if is used, eg

```
if (a < 5)  
    cout << "a is small";  
else
```


Flow of control: if-blocks

The argument to `if()` is a **logical expression**.

Example

- `x == 8` , but never `if (x = 8)`
- `m == '5'` ← sets x to 8 & evals as 'true'.
- `y <= 1` ← character 5.
- `y != x`
- `y > 0`

More complicated examples can be constructed using **AND** `&&` and **OR** `||`.

`y <= 1` "y ≤ 1"
`y != x` $\nRightarrow y \neq x$.
Some as `!(y == x)`

Don't use `&` or `!`,
these are bit-wise operators.

Flow of control: if-blocks

04EvenOdd.cpp

```
#include <iostream>

12 int main(void)
    int Number;

    std::cout << "Please enter an integer: ";
16    std::cin >> Number;

    if ( (Number%2) == 0)
18        std::cout << "That is an even number." << std::endl;
20    else
        std::cout << "That number is odd." << std::endl;

    return(0);
24 }
```

remainder on dividing by 2.

Flow of control: if-blocks

More complicated examples are possible:

Structure (ii):

```
if( exp1 )
{
    statements to execute if exp1 is "true"
}
else if ( exp2 )
{
    statements run if exp1 is "false" but exp2 is "true"
}
else
{
    "catch all" statements if neither exp1 or exp2 true.
}
```

Flow of control: if-blocks

05Grades.cpp

```
10 int main(void)
11 {
12     int NumberGrade;
13     char LetterGrade;

14     std::cout << "Please enter the grade (percentage): ";
15     std::cin >> NumberGrade;

16     if ( NumberGrade >= 70 )
17         LetterGrade = 'A';
18     else if ( NumberGrade >= 60 )
19         LetterGrade = 'B';
20     else if ( NumberGrade >= 50 )
21         LetterGrade = 'C';
22     else if ( NumberGrade >= 40 )
23         LetterGrade = 'D';
24     else
25         LetterGrade = 'E';

26     std::cout << "A score of " << NumberGrade << "% cooresponds to a "
27         << LetterGrade << "." << std::endl;

28     return(0);
29 }
```

Flow of control: `if`-blocks

The other main flow-of-control structures are the

- `switch ... case` structures
- the use of the `?` and `:` operators.

Exercise 2.1

- Find out how `switch.. case` works. Rewrite the Even/Odd example above using `switch ... case`.
- What errors/bugs/problems are there with the Grades example? That is, how could you get it to break?
- Read up on the `switch / case` construct. Can it be used to write an improved version of the programme. (Hint: yes, but you need a recent C++ compiler...).

We meet a **for**-loop briefly in the Fibonacci example. The most commonly used loop structure is **for**

← all optional.

```
for(initial value; test condition; step)
{
    // code to execute inside loop
}
```

```
int i=0
for( ; i>=1; i--).
```

Example: 06CountDown.cpp

```
10 int main(void)
11 {
12     int i;
13
14     for (i=10; i>=1; i--)
15         std::cout << i << "... ";
16
17     std::cout << "Zero!\n";
18
19     return(0);
20 }
```

$i-- \Leftrightarrow i = i - 1;$

Similarly

$a++ \Leftrightarrow a = a + 1$

$a+=6 \Leftrightarrow a = a + 6$

$a*=2 \Leftrightarrow a = 2*a.$

The other two common forms of loop in C++ are

- `while` loops
- `do ... while` loops

Exercise 2.2

Rewrite the **count down** example above using a

- 1 `while` loop.
- 2 `do ... while` loop.

Functions

A good understanding of **functions**, and their uses, is of prime importance. Some functions return/compute a single value. However, many important functions return more than one value, or modify one of its own arguments. For that reason, we need to understand the difference between **call-by-value** and **call-by-reference** (← next week).

.....

Every C++ program has at least one function: `main()`

Example

```
#include <iostream>

int main(void )
{
    /* Stuff goes here */
    return(0);
}
```


Functions

Each function consists of two main parts:

- Function “header” or **prototype** which gives the function’s
 - ▶ return value data type, or `void` if there is none, and
 - ▶ parameter list data types or `void` if there are none.

The prototype is often given near the start of the file, before the **main()** section.

Important: The prototype should be written before the function—perhaps when the program is begin specified.

- **Function definition.** Begins with the function names, parameter list and return type, followed by the body of the function contained within curly brackets.

Format:

```
ReturnType FnName ( param1, param2, ... )  
{  
    statements  
}
```

- **ReturnType** is the data type of the data returned by the function.
- **FnName** the identifier by which the function is called.
- **Param1, ...** consists of
 - ▶ the data type of the parameter
 - ▶ the name of the parameter will have in the function. It acts within the function as a local variable.
- the statements that form the function's body, contained with braces **{...}**.

07IsComposite.cpp

```
30 int IsComposite(int i)
31 {
32     int k;
33
34     for (k=2; k<i; k++)
35         if ( (i%k) == 0)
36             return(k);
37
38     // If we get to here, then i has no divisors between 2 and i-1
39     return(0);
40 }
```

Finished here 5pm 15/1/17.

Calling the IsComposite function:

07IsComposite.cpp

```
12 int main(void )  
13 {  
14     int i;  
  
16     std::cout << "Enter a natural number: ";  
    std::cin >> i;  
  
    if (IsComposite(i))  
20         std::cout << i << " is a composite number." << std::endl;  
    else  
22         std::cout << i << " is a prime number." << std::endl;  
  
24     return(0);  
}
```

Most functions will return some value. In rare situations, they don't, and so have a `void` argument list.

08Kth.cpp

```
14 #include <iostream>
    void Kth(int i);
    int main(void )
14 {
    int i;

    18 std::cout << "Enter a natural number: ";
    std::cin >> i;

    20 std::cout << "That is the ";
    Kth(i);
    22 std::cout << " number." << std::endl;

    24 return(0);
    }
```

08Kth.cpp (continued)

```
26 // FUNCTION KTH
27 // ARGUMENT: single integer
28 // RETURN VALUE: void (does not return a value)
29 // WHAT: if input is 1, displays 1st, if input is 2, displays 2nd,
30 // etc.
31 void Kth(int i)
32 {
33     std::cout << i;
34     i = i%100;
35     if ( ((i%10) == 1) && (i != 11))
36         std::cout << "st";
37     else if ( ((i%10) == 2) && (i != 12))
38         std::cout << "nd";
39     else if ( ((i%10) == 3) && (i != 13))
40         std::cout << "rd";
41     else
42         std::cout << "th";
43 }
```