Introduction to C++

Philip Blakely

Laboratory for Scientific Computing, University of Cambridge

Part I

Getting Started

Outline

1 Preliminaries

2 Computer basics

3 First steps in C++

Suggested Reading

There are few books available that teach C++ well, and are not too advanced

- C++ Primer, Lippman, Lajoie, Moo, Addison Wesley (Introductory)
- Programming: Principles and Practice Using C++, Bjarne Stroustrup. (Introductory to Advanced)
- The C++ Programming Language 3rd edition, Bjarne Stroustrup, Addison Wesley (Advanced / Steep learning curve)

Reference manuals

- `http://en.cppreference.com` - maintained by multiple C++ experts.
- `www.cplusplus.com/reference`
- `http://www.parashift.com/c++-faq-lite/` (More an FAQ than a reference)

Comments on course content

- Some experience of programming (in any language) will help (I will try to not to assume anything, though)
- Experience of Linux/UNIX-based OS helpful
- This course will mostly teach the C++-language, not programming style or good practice.
- All compilation commands will assume Linux, using `gcc`, but should be transferable to any other compiler/OS with minimal man-page reading
- Twin-hemisphere electro-colloidal brain with cognitive functions necessary

Overview of C

- C was developed in the early 1970s as a replacement for assembly languages
- (Fairly) simple correspondance between C statements and assembler instructions
- C is a low-level language, used mostly for systems programming
- Very easy to make hard-to-find errors; e.g. no bounds-checking on arrays
- However, its relative simplicity can lead to efficient programs - highly dependent on compiler optimizations

Overview of C++

- Developed as a fusion of C and high-level features from Simula, starting in 1979 by Bjarne Stroustrup
- The aim was a language with the performance of C but with object-oriented features (classes) and meta-programming (templates)
- Intended to make C++ useful for advanced algorithms with complex designs but still suited to HPC.
- C++ includes many advanced, generic, algorithms for operations on data-structures
- C++ should be treated as a separate language from C; prior knowledge of C is not necessary (and may be a detriment).
- C++ is platform-independent; what works in Linux will work in Windows (and vice-versa)

More C++

- The first C++ standard emerged in 1998
- Corrections and additions were made to the standard in 2003
- Major updates occured in 2011, 2014, and 2017.
- C++ is an evolving language
- C++ is becoming more and more complex; possibly too complex
- This course will not introduce anything from C++17; all examples should work with a C++14-compliant compiler.

Advantages of C++

- Can be optimized fairly well by modern compilers
- With object-oriented features and operator overloading, allows programmers to create abstractions and easy-to-read code
- Complex operations and expressions can be written on a single (short) line
- C++ standard is such that compilers are allowed to "see through" function calls and generate optimized code equivalent to hand-coded C.
- If you avoid advanced features (STL, templates, inheritance), it's fairly user-friendly

Disadvantages of C++

- C++ is a complex language and takes a long time to learn
- C++ enables you to shoot yourself in the foot in myriad ways
- It is still relatively easy to make hard-to-find bugs
- Syntax errors can be hard to find, even with compiler help
- Can be easier to write efficient code in Fortran (latest version)

Alternative Languages

There are alternatives to C++ for scientific computing

- FORTRAN/Fortran - fairly widely used, easier to learn, harder to make undetectable mistakes
  far better at efficient array operations than C/C++
- Python - Easier to use - at the expense of some performance
- Matlab (or similar package) - Only suitable for some tasks - not sufficiently general

Types of language

- Computer langauges range from low-level to high-level, depending on how far removed the language instructions are from what happens on the CPU
  - Assembler / machine-code is low-level
  - C and old FORTRAN are fairly low-level
  - C++ and modern Fortran are high-level
  - Java, MatLab, Python, etc. are high-level
- Lower-level languages are often harder to use correctly, although higher-level ones can have too many features and be confusing
- Even high-level languages use routines written in low-level languages for important computations

Outline

1 Preliminaries

2 Computer basics

3 First steps in C++

What is a computer?

- For our purposes, a computer consists of a processor (CPU) and storage
- The storage consists of memory (both on CPU and RAM) and hard-disk
- The processor executes instructions that are stored in memory using data that is in the memory
- The results are output in some fashion (to screen, disk, etc.)

Binary

- These days, all computers work in binary
- Everything is represented as 0s and 1s
- A single *b*inary dig*it* is a *bit*
- Eight bits make up a byte.
- All data, text, files, images, programs, arithmetic are represented as a series of bits
- The interpretation of these is up to the CPU or the program or operating system running on the CPU.

Representation of integers in binary

- Positive integers can easily be represented in binary:
  - $4 \mapsto 100$
  - $30 \mapsto 11110$
- Negative integers can also be represented in binary.
- The most common method is two's-complement, where a negative integer $-m$ is represented in binary by the positive integer $2^n - m$ for some value of n which is independent of m.
- (Although n will depend on the range of integers that is required to be represented.)
- For example, if $n = 8$, then:
  - $-15 \mapsto 256 - 15 = 241 \mapsto 11110001$
  - $-14 \mapsto 256 - 14 = 242 \mapsto 11110010$
  - $-127 \mapsto 256 - 127 = 129 \mapsto 10000001$
  - $-128 \mapsto 256 - 128 = 128 \mapsto 10000000$
- It should therefore be clear that the range of integers that can be contained in 8 bits is $[-128, 127]$.

Real numbers in binary

- Representing real numbers in binary is non-trivial
- Roughly, they are represented as an expansion in the inverses of powers of two

$$a = \sum_{i=1}^{N} \frac{a_i}{2^i}$$

  where $a \in [0, 1)$ and $a_i = 0, 1$, and N is fixed
- Given a finite number of bits, we can only represent real numbers to a particular precision
- Details and their consequences are too complicated to cover in this course.
- For more details see "What every computer-scientist should know about floating-point numbers" (available online)

How does a CPU work?

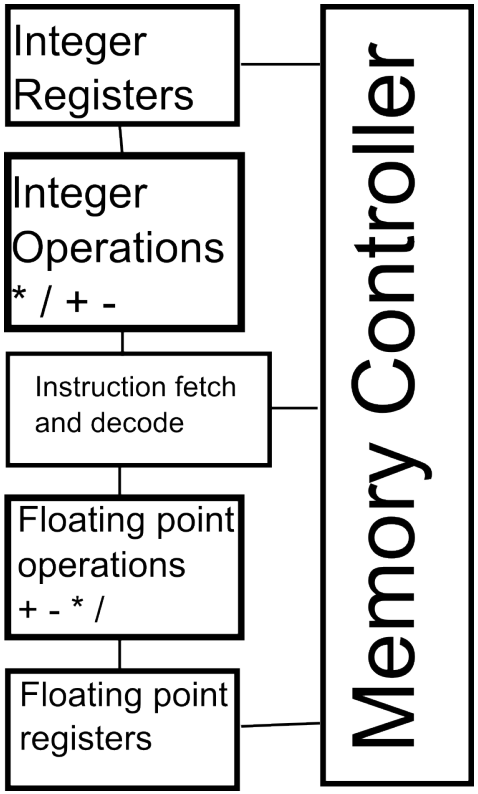
- Computers execute a series of simple instructions such as:
  - Read the data from memory location 12
  - Read the data from memory location 13
  - Add the two values
  - Assign the result to memory location 14
- These instructions are processor-specific (different for Intel and AMD processors, even between different Intel processors)
- The instructions are expressed in binary, e.g.
  - 00001101100
  - 00001101101
  - 01001101001
  - 00001111110
- One (bad) way of expressing the power of a chip is in (millions of) instructions per second
- The way that instructions are executed is more complicated than this...

More on the CPU

- When a processor needs to act on user-supplied data, it requests this from the main memory
- The memory is separate from the CPU, and communciation between them is governed by the motherboard circuitry
- Fetching data from the main memory takes a relatively long time.
- As a rough guide, a processor may be able to execute 10-100 instructions in the time it takes to read a byte of data from memory.
- This overhead (latency) can be partially overcome by a series of techniques used in most modern CPUs

CPU Schematic

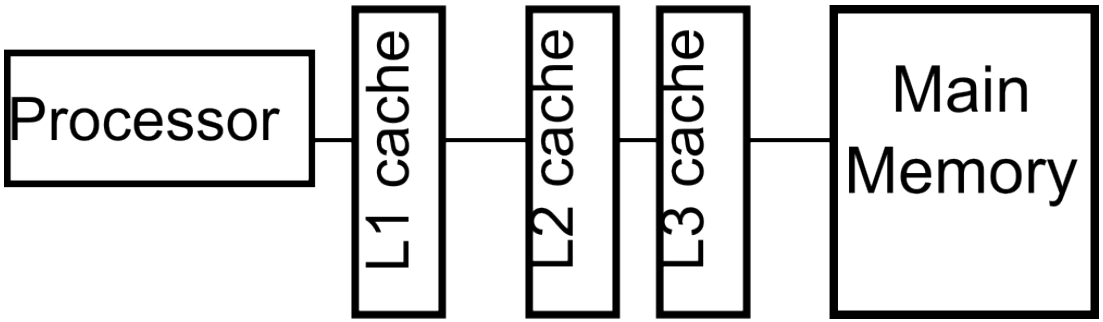
- Floating point and integer operations done separately for historical reasons
- Integer/FP operations can only act on values in registers
- There are of the order of 100 registers in total
- Processor can access these very quickly
- Modern processors can perform more than one instruction per clock cycle



Much simplified and not to scale!

CPU Cache Schematic

- Solution to memory latency is caching
- Store copy of recently used data in a fast-access cache so it can be retrieved again quickly.
- Utility of this relies on programmer using data in a sensible fashion
- Arranging for cache-reuse can be very important for speed
- Not strictly under programmer's control, but "locality of reference" is usually a safe bet.



The point of programming

- Given the preceeding, how can we program computers efficiently?
- Answer is to have a standard, high-level, language
- So, we might express the previous binary as:
  `totalBill = foodCost + serviceCharge;`
  and allow a "compiler" to translate this into the binary instructions above
- We then delegate the responsibility of keeping up with different processors to the compiler-writers.
- We are left with the "simple" task of writing human-readable instructions.

Fundamentals of Programming

- Even with high-level languages, it is still hard to write efficient and correct programs
- Source code should be human-readable first, and efficient second
- It is useless having a program that runs fast if you can't remember how it works (either to implement a new feature or to describe it to someone else)
- On very rare occasions, efficiency may be more important than human-readablity
- A good programmer is someone who can write correct, readable, maintainable, and efficient code (in decreasing order of importance)

How to write a program

- Switch off your computer
- Determine what it is you want your program to do
- This will be at a fairly high-level:
  "The program should solve any given ODE approximately"
- Determine the required features of your algorithm:
  "Assuming we use an integration step Δx, the accuracy of the final answer should converge as Δx"
- Find a suitable algorithm to perform this process: "Use 1st-order forward Euler"
- An algorithm is the sort of description of a process that you find in a text-book or academic paper.
- It will include sentences, equations, and maybe an example, and even some validated results
- You should first convince yourself that the algorithm is correct either by doing a simple version with pen and paper or by looking at other people's results using the same algorithm.

How to write a program ctd.

- Now start writing your program in pseudo code
- Pseudo-code is human-readable instructions written as comments:
  - // Read in initial state x, final time T, and dt from user
  - // Start integration at x
  - // Loop until t > T
  - // Use Euler to determine updated x
  - // Update and output x
  - // Add dt to current time
  - // End loop
- Now we can start programming!
- Below each line of pseudo code, put the appropriate language-constructs for that instruction.
- Each line of pseudo-code should give only a few lines (~ 5) of code

Expanding the pseudocode

```
// Read in initial state x, final time T, and dt from user
   std::cin >> startX >> T >> dt;
// Start integration at x
   double x = startX;
// Loop until t > T
   while( t < T ){
// Use Euler to determine updated x
     x = x + dt * df(x, t);
// Update and output x
     std::cout << x << std::endl;
// Add dt to current time
     t = t + dt;
// End loop
   }
```

Nomenclature

Definitions of various programming terms:

- Variable: A label for a place in memory where a particular value or object is stored
- Type: The kind of information stored in a variable
- Array: An ordered set of values of the same type.
- Boolean type: Choice of two values (true/false or 1/0).
- String: A string of characters, e.g. “Hello World!”
- Seg(mentation) fault: Usually caused by out-of-bounds memory access.

Outline

1 Preliminaries

2 Computer basics

3 First steps in C++

Hello World - simple C++ program

```
#include <iostream>

int main(void)
{
  std::cout<< "Hello World" << std::endl;
  return 0;
}
```

- Includes the *system header* file defining IO functions

Hello World - simple C++ program

```
#include <iostream>

int main(void)
{
  std::cout<< "Hello World" << std::endl;
  return 0;
}
```

- Includes the *system header* file defining IO functions
- The first function to be called within a C++ program is (usually) `main`

Hello World - simple C++ program

```
#include <iostream>

int main(void)
{
  std::cout<< "Hello World" << std::endl;
  return 0;
}
```

- Includes the *system header* file defining IO functions
- The first function to be called within a C++ program is (usually) `main`
- `std::cout` is a *stream*, which goes to `stdout`
  Equivalently, `std::cerr` goes to `stderr`

Hello World - simple C++ program

```
#include <iostream>

int main(void)
{
  std::cout<< "Hello World" << std::endl;
  return 0;
}
```

- Includes the *system header* file defining IO functions
- The first function to be called within a C++ program is (usually) `main`
- `std::cout` is a *stream*, which goes to `stdout`
  Equivalently, `std::cerr` goes to `stderr`
- We're sending the string "Hello World!" to the standard output stream, usually the terminal

Hello World - simple C++ program

```
#include <iostream>

int main(void)
{
  std::cout<< "Hello World" << std::endl;
  return 0;
}
```

- Includes the *system header* file defining IO functions
- The first function to be called within a C++ program is (usually) `main`
- `std::cout` is a *stream*, which goes to `stdout`
  Equivalently, `std::cerr` goes to `stderr`
- We're sending the string "Hello World!" to the standard output stream, usually the terminal
- Return an exit code zero to the OS

Compiling and running

- Before the program can be run, it needs to be compiled
- This translates the C++ code into machine code that runs directly on the computer
- A lot of work is done at compile-time
- This makes for more efficient code at run-time

To compile, type: `g++ helloWorld.C -o helloWorld`
To run, type `./helloWorld`

The resulting executable only needs C++ libraries present on the system to run.

Compiler Warnings

- The compiler will report any errors in C++ syntax
- Most compilers are also capable of suggesting where you may have made errors (even if it's valid C++)
- Use `g++ helloWorld.C -o helloWorld -Wall -Wextra -pedantic`
- Pay attention to all the warnings!
- Only ignore them if you know why they've occured in the first place.
- They will teach you more about the workings of C++.
- Consider using `-Werror`, which makes all warnings into errors and forces you to understand and fix them.

Compilers

- Throughout this course I shall assume the compiler is `g++`.
- This is open-source, and reasonably robust and standards compliant.
- I strongly recommend using `gcc 9.4.0` at least.
- This will compile using the C++14 standard by default; later versions of the compiler may change to a later default standard.
- Other compilers exist, e.g. `icpc` (Intel) and `clang++` and have equivalent options.
- It may be advisable to try compiling your code with multiple compilers and comparing behaviour (should not change, unless you or the compiler writers have made an error) and performance (more likely to change).

Undefined behaviour

- The C++ standard refers to some constructs as leading to "undefined behaviour"
- For example, assigning a too large value to a variable
- In theory, the result of this could be anything up to and including "delete all your files and OS"
- A better result would be to just abort the program
- Unfortunately, many compilers produce executables that will attempt to carry on, and produce essentially random results
- Even worse, this behaviour may change depending on what options you pass to the compiler (especially optimization)
- This can lead to many hours spent trying to track down the problem

Implementation-defined behaviour

- There are some C++ constructs that have "implementation-defined behaviour"
- For example, $(-5)/2$ may be -2 or -3
- This is guaranteed to be consistent independent of any compiler flags
- However, it is not necessarily consistent across different compilers or versions of the same compiler
- This may cause you problems if you switch from `gcc` to Microsoft's compiler, for example.

Optimization

- Optimization of C++ programs is not trivial. There are rules about the reordering of sub-expressions.
- Optimization may not alter the output of a program
- Arithmetic expressions can be reordered so long as the result would be unchanged, i.e. associativity and commutivity are assumed.
- (This makes some restrictions where signed overflow is concerned)
- Overloaded operators are not assumed to be associative or commutative

Hello Name

```
#include <iostream>
#include <string> // Definition of a string of chars

int main(void){
  std::string name;
  std::cout << "What is your name?" << std::flush;
  std::cin >> name; // Reads from stdin
  int num; // Define an integer called num
  std::cout << "What is your favourite integer?" << std::flush;
  std::cin >> num; // Read an integer from stdin
  std::cout << "Hello " << name
            << ", your favourite number is "
            << num << std::endl; // Output data
  return 0;
}
```

- `std::cout/cin` recognises and formats types automatically
- `std::endl` marks the end of a line
- `std::` is a *namespace* - just accept this for the moment.

Hello Name

```
#include <iostream>
#include <string> // Definition of a string of chars

int main(void){
  std::string name;
  std::cout << "What is your name?" << std::flush;
  std::cin >> name; // Reads from stdin
  int num; // Define an integer called num
  std::cout << "What is your favourite integer?" << std::flush;
  std::cin >> num; // Read an integer from stdin
  std::cout << "Hello " << name
            << ", your favourite number is "
            << num << std::endl; // Output data
  return 0;
}
```

- `std::cout/cin` recognises and formats types automatically
- `std::endl` marks the end of a line
- `std::` is a *namespace* - just accept this for the moment.

Hello Name

```
#include <iostream>
#include <string> // Definition of a string of chars

int main(void){
  std::string name;
  std::cout << "What is your name?" << std::flush;
  std::cin >> name; // Reads from stdin
  int num; // Define an integer called num
  std::cout << "What is your favourite integer?" << std::flush;
  std::cin >> num; // Read an integer from stdin
  std::cout << "Hello " << name
            << ", your favourite number is "
            << num << std::endl; // Output data
  return 0;
}
```

- `std::cout/cin` recognises and formats types automatically
- `std::endl` marks the end of a line
- `std::` is a *namespace* - just accept this for the moment.

Hello Name

```
#include <iostream>
#include <string> // Definition of a string of chars

int main(void){
  std::string name;
  std::cout << "What is your name?" << std::flush;
  std::cin >> name; // Reads from stdin
  int num; // Define an integer called num
  std::cout << "What is your favourite integer?" << std::flush;
  std::cin >> num; // Read an integer from stdin
  std::cout << "Hello " << name
            << ", your favourite number is "
            << num << std::endl; // Output data
  return 0;
}
```

- `std::cout/cin` recognises and formats types automatically
- `std::endl` marks the end of a line
- `std::` is a *namespace* - just accept this for the moment.

Hello Name

```
#include <iostream>
#include <string> // Definition of a string of chars

int main(void){
  std::string name;
  std::cout << "What is your name?" << std::flush;
  std::cin >> name; // Reads from stdin
  int num; // Define an integer called num
  std::cout << "What is your favourite integer?" << std::flush;
  std::cin >> num; // Read an integer from stdin
  std::cout << "Hello " << name
            << ", your favourite number is "
            << num << std::endl; // Output data
  return 0;
}
```

- `std::cout/cin` recognises and formats types automatically
- `std::endl` marks the end of a line
- `std::` is a *namespace* - just accept this for the moment.

Comments

- When writing programs it is *very* useful to put comments in.
- Anything following `//` up to the end of the line is ignored:

```
// Compute root of quadratic
float x = ( -b + sqrt(b*b - 4*a*c) ) / (2*a);
```

- Anything between `/* */` is ignored:

```
/* Solve ODE using 1st order Euler method.
   For details see Unman, Wittering, and Zigo (42nd edition)
*/
...Code goes here...
```

- You *will* need reminding of what your code does when you return to it tomorrow, next week, or next year. Make useful comments!
- Comments such as

```
int a = 5; // Define integer a to be 5
```

are worse than useless.

Whitespace

- Spaces, tabs, and newlines (whitespace) are essentially irrelevant in C++
- So long as separate tokens (variable names, operators, keywords) are separated, there is no problem

```
std::
cout
<< "My text" <<
std
::
endl;
```

is equivalent to

```
std::cout << "My text" << std::endl;
```

(but is far less readable).

Variable names

- Variable names are very important in programming
- They should describe what they refer to in a succinct way
- In C++ variable names can consist of A-Z, a-z, 0-9, and _ (but cannot start with a digit)
- Good variable names include:
  `numVehicles`, `timestep`, `distToNextTown`
- Bad variable names include:
  `zz`, `numberOfObjectsInThisListOfVehicles`,
  `l1`, `dghRType`

Code formatting

- Use an editor with automatic code formatting and indentation.
- Use one statement per line.
- Braces should go on their own lines.
- Statements inside braces should be consistently indented by 2-4 spaces.
- Use blank lines liberally; they improve readability.
- My code examples largely stick to these suggestions, except where space saving to fit on one slide.
- None of these are requirements for a valid C++ program, but will make your code far more readable and easy to understand.