Introduction to C++ - lectures 1-2

Martin Robinson

2019

A few introductory remarks

- C++ is a common programming language for high performance scientific computing applications, but it is not the only choice. You should know what languages/platforms are available to you.
- It is generally useful to know at least one low-level language (e.g. C++) and one high-level language (e.g. Python, Matlab) for your work.
- C++ is useful for:
  - High performance applications. Concurrency, parallelism and compilation to efficient, optimised binaries is easier in C++
  - Software libraries. C++ is designed for zero-cost abstraction, so library authors can provide their functionality with minimal overhead.
  - Access to a vast array of third-party scientific libraries

Resources

- There are lots of resources for C++, particularly online.
  - Many books, e.g. Bjarne Stroustrup, "Programming: Principles and Practice Using C++". Make sure to use editions published well after 2011.
  - Reference material: http://www.cplusplus.com and http://cppreference.com
  - Q&A sites, http://stackoverflow.com etc.
- Integrated Development Environments (IDEs) exist—e.g. CLion or Visual Studio. These are useful for beginners as well as managing large code development. A command-line editor such as `vim` can also be highly customised for C++ development

Programming paradigms in C++

C++ is a multi-paradigm language with the goal of allowing powerful abstractions with *zero* run-time cost

- *Statically typed*: Variables have a constant type determined at compile-time, rather than determined on-the-fly at run-type (e.g. MATLAB)
- *Procedural*: The C-style basis of C++, algorithm consists of procedures (i.e. functions or subroutines) containing a series of computational steps
- *Object-orientated*: break computational task into objects (i.e. classes) that expose behaviours (methods) and data (members) through a public interface

Programming paradigms in C++

Recent updates (e.g. C++11, 14, 17) focus on generic and functional paradigms

- *Generic*: algorithms are written with generic types or variables to be determined later (at compile-time). This is achieved in C++ through *templates*
- *Functional*: Not a *pure* functional language, but functional elements exist in C++, for example the Standard Template Library (STL) and lambda functions

General structure of a basic C++ program

```cpp
#include <header1>
#include <header2>
int main() {
  line of code;
  // this is a comment, ignored by the compiler
  more code; // this is a comment as well
  /*
   * Multi-line
   * comment
   */
  return 0;
}
```

Things to note

- Header files are listed first. These are files that contain the functions needed for operations such as input, output and mathematical calculations
- There is a section of code that starts `int main()`
  - or `int main(int argc, char* argv[])` when command-line input is important
- This section of code is followed by more code enclosed between curly brackets, { and }
- Comments may easily be inserted into the code
- Lines of code that "do something" end with a semicolon `;`
- Just before the closing curly bracket at the end of the code is a statement `return 0;`

A first C++ program

```cpp
#include <iostream>
int main()
{
  std::cout << "Hello World\n";
  return 0;
}
```

[< compiler explorer >]

This program prints the text “Hello World” to the screen

- `iostream` is a header file that is needed when using input and output
- `std::cout` is a command that sends output to the console, i.e. the screen
- `'\n'` is a formatting command that starts a new line
- All statements (lines of the program) inside the curly brackets end with a semicolon `;`

Compiling the code

- A key difference between Matlab and C++ is that before the code can be executed it must be compiled
- When using software such as the Compiler Explorer or CLion, this is often done automatically
- When using the Gnu compiler of Unix/Cygwin this code can be compiled by saving the code and typing

  ```
  g++ -o hello_world hello_world.cpp
  ```

  followed by return.
- This produces an executable called `hello_world` that can be executed by typing

  ```
  ./hello_world
  ```

Makefiles

- Make is a general-purpose Unix tool that generates *targets* from their *depedencies* according to a set of *rules*, or *commands*, that you supply. These targets, dependencies and rules are all specified in a file called `Makefile`. For example:

```
hello_world: hello_world.cpp
  g++ -o hello_world hello_world.cpp
```

- Type `make` in a directory containing a `Makefile`, which by default generates the first target in the `Makefile`.

CMake

- CMake is a higher-level build system, which takes project definitions written in its own syntax, and generates `Makefiles` (or other, similar tools) that will compile your project, for example:

```
cmake_minimum_required(VERSION 3.10)
project(hello_world)
set(CMAKE_CXX_STANDARD 14)
add_executable(hello_world hello_world.cpp)
```

- Run `cmake` to generate the Makefile, then `make` to do the compilation

```
$ cmake <path-to-CMakeLists.txt>
$ make
```

Clion

- CLion is a C++ Interactive Development Environment (IDE) that, among other things, can handle compilation of (simple) projects automatically
- It uses CMake to do this, creating a `CMakeLists.txt` within each project
- You can edit the `CMakeLists.txt` directly to specify custom build behaviour
- CLion will be the recommended IDE to use during this course, although you are free to use whatever tool you are most familiar with.

Numerical variables

Before a variable is used the type of variable must be declared. For example if the variables i and j are integers and a is a double precision floating point number the statements

```
int i, j;
double a;
```

must be included in the program before these variables are used.

It is advisable to use `double` rather than `float` in scientific computing applications.

Some names, such as `int`, `for`, `return` may not be used as variable names because they are used by the language.

- These words are known as reserved words or keywords

The following code adds two integers and print the answer to screen:

```
int integer1 = 5;
int integer2 = 10;
int answer = integer1 + integer2;

std::cout << "The sum of " << integer1 << " and "
          << integer2 << " is " << answer << '\n';
```

The details of the `std::cout` statement will be explained later

More on built-in types

A variable may be initialised when defining the variable type, for example

```
int i = 5;
```

The value of more than one variable may be assigned in each statement:

```
int i, j;
i = j = 3;
```

This will cause confusion because it means

```
i = ( j = 3 );
```

and not

```
( i = j ) = 3;
```

[< compiler explorer >]

When assigning values to floating point variables it is good programming practice to write numbers with decimal points, i.e.

```
double x = 5.0;
float y = 7.0f;
```

rather than

```
double x = 5;
float y = 7;
```

If a quantity is a constant throughout the program it may be declared as such

```
const double density = 45.621;
```

Fundamental types

- Boolean type
  - `true` or `false`
- Integer types
  - `signed` or `unsigned`
  - `short`, `long`, `long long`
  - e.g. `short int`; `int`; `long int`; `unsigned short`; `unsigned long long`
- Floating point types
  - `float`
  - `double`
  - `long double`

Representable values

Some variable types and ranges are given below. Note the these are operating system dependent:

```
Variable type     C++ name       Range
-------------     ------------   -----------------
integer           int            -2^31 to 2^31 - 1
integer           long int       -2^63 to 2^63 - 1
unsigned integer  unsigned int    0 to 2^32 - 1
floating point    float          -3.4 x 10^38 to 3.4 x 10^38
floating point    double         -1.8 x 10^308 to 1.8 x 10^308
floating point    long double    -1.2 x 10^4932 to 1.2 x 10^4932
```

The values on your specific system can be found in `std::numeric_limits`:

`[< compiler explorer >]`

Mathematical operations

There is a shorthand for some mathematical operations

```
 Longhand     Shorthand
----------   ---------
a = a + b;    a += b;
a = a - b;    a -= b;
a = a * b;    a *= b;
a = a / b;    a /= b;
a = a + 1;    a++; if a is an integer
a = a - 1;    a--; if a is an integer
```

Some example lines of code

```
float a, b;
double d, e;
a = 3.0;
b = ( a * std::pow(a, 3) ) / 2.0;
d = 4.0;
e = 2.0 * std::sqrt(d);
```

`std::pow(x,y)` gives the value of x^y. The `std::` indicates the standard namespace.

`std::sqrt(d)` gives the square root of the variable `d`

When using mathematical functions such as `std::pow`, you need to

```
#include <cmath>
```

Division of an integer by another integer will almost certainly cause problems

An example is given in the following piece of code

```
int i = 5, j = 2, k;
k = i / j;
std::cout << k << '\n';
```

The variable `k` is an integer and so cannot store the true value, 2.5

Instead, it will store the value 2

Suppose an integer is divided by a variable of type `double` - or vice versa - and that the result returned is stored in a variable of type `int`, as shown in the code below.

The variable `k` in this code is unable to store the mathematically correct answer

```
double i = 5.0;
int j = 2, k;
k = i / j;
std::cout << k << '\n';
```

Only construct mathematical operations that are on elements of the same type

An integer can be converted, or cast, to a different data type – see the next slide

Variables can be converted from one type to another, for example:

```
double i = 5.0;
double k;
int j = 2;
k = i / static_cast<double>(j);
std::cout << k << '\n';
```

In this example, `static_cast<double>(j)` makes the variable `j` behave as if it were a `double` variable.

ASCII characters

ASCII characters are numbers, uppercase letters, lowercase letters and some other symbols

These characters may be represented using the data type `char`

```
#include <iostream>
int main() {
  char letter;
  letter = 'a'; // note the single quotation marks

  std::cout << "The character is " << letter << '\n';

  return 0;
}
```

Boolean variables

These variables take the values `true` or `false`, and are of use when using `if` conditionals and `while` loops

They are used as follows:

```
bool flag = true;
```

Strings

A character is one letter or number, a string is an ordered collection of characters

For example, "C++" is a string consisting of the ordered list of characters "C", "+", and "+"

To use strings in C++ requires an extra header, as shown below:

```cpp
#include <iostream>
#include <string>

int main() {
  std::string city = "Oxford"; // note the std:: and the " marks
  std::cout << city << '\n';
}
```

The if statement

Suppose you want to execute some code only if the condition `p > q` is met

This is achieved using the following code:

```
if (p > q)
{
  statement1;
  statement2;
}
```

Note the indentation. This makes it clear which statements are executed in the body of the `if` statement.

If only one statement is to be executed curly brackets aren't strictly necessary.

For example, the following code will execute statement1 if the condition `p > q` is met

```
if (p > q)
  statement1;
```

but this is considered poor software engineering practice, and instead you should write this code as

```
if (p > q)
{
  statement1;
}
```

The use of curly brackets makes it clear which statement(s) are to be executed.

Third example – more than one condition

```
if (p == 0) {
  statement1;
} else if (p < 0) {
  statement2;
  statement3;
} else {
  // p > 0
  statement4;
}
```

Fourth example – nested `if` statements

```
if (p < q) {
  if (x >= y) {
    statement1;
  }
}
```

Fifth example – more than one condition

```
if (p < q || x < y) {
  statement1;
}
```

`statement1` is executed only if one or both of `p < q` and `x < y` is true - i.e. `||` is the logical `OR` operator.

Relational and logical operators

```
   relation                      operator
---------------          ---------------
equal to                 == (note: not "=" which is assignment)
not equal to             !=
greater than             >
less than                <
greater than or equal to >=
less than or equal to    <=
```

logical condition	operator
AND	&&
OR	||
NOT	!

Boolean variables may be used in `if` statements as follows

```
bool flag1 = true, flag2 = false;
if (flag1)
{
  std::cout << "Does print something" << '\n';
}
if (flag2)
{
  std::cout << "Doesn't print anything" << '\n';
}
if (!flag2)
{
  std::cout << "Does print something" << '\n';
}
```

The while loop

```
while ( x < 100.0 && i < 10 ) {
  x += x;
  i++;
}
```

The condition `x < 100.0 && i < 10` is tested only at the beginning of the statements in the loop, and not after every statement.

For example if the loop is entered when $x = 99.0$ and $i = 1$, the loop will be executed completely once.

Loop won't be entered when $x \geq 100$: `x` and `i` will be unchanged.

If you need a loop to execute at least once, with a test at the end, use:

- `do { ... } while (condition)`

The for loop

The following loop executes the statements inside the loop 10 times.

```
for (int i=0; i<10; i++)
{
  statement1;
  statement2;
}
```

Note that `i` can be previously declared, or declared in the loop statement.

`for` loops can be nested and run over variable indices:

```
for (int i=0; i<5; i++)
{
  for (int j=0; j<10; j++)
  {
    std::cout << "i = " << i << " j = " << j << '\n';
  }
}
```

Random Number generation

You can generate random numbers using the standard library

```
#include <iostream>
#include <random>
int main() {

    std::default_random_engine generator;
    std::uniform_real_distribution<double> dist(0,1);

    double my_rand = dist(generator);
    return 0;
}
```

[< compiler explorer >]; [< cpp reference >]

Fixed sized arrays

If the size of the array is known in advance (i.e. at compile-time) then it is better to use the fixed size array `std::array<T,N>`, where `T` is the type that the array will hold (e.g. `int`, `double`), and `N` is the length of the array.

```
#include <array>
std::array<int, 4> x; // x holds 4 int
std::array<double, 5> y; // y holds 5 double

// a 5 by 5 array
std::array<std::array<double,5>,5> z;
```

In contrast to Matlab and FORTRAN the indices of an array of length `n` start at 0 and end at `n-1`

Elements of the array are accessed by placing the indices in separate square brackets, for example

```
x[0] = 1;
z[1][2] = 3.0;
```

Arrays can be initialised when they are declared, for example

```
std::array<double,3> array1 = {0.0, 1.0, 2.0};
std::array<std::array<double,3>,2> array2 =
                                {{{0.0, 1.0, 2.0},
                                  {3.0, 4.0, 5.0}}};
```

Note that the values of arrays may only be set using the curly bracket notation when they are declared - for example the code

```
std::array<int,3> array = {0, 1, 2};
```

is correct, but the code

```
std::array<int,3> array;
array = {0, 1, 2};
```

is not correct.

Console output

Console output may be achieved by using `std::cout`

We have already seen that the statement

```
std::cout << "Hello World\n";
```

prints the text “Hello World” to the screen, followed by a newline.

The statements

```
int x = 1, y = 2;
std::cout << "x = " << x << " and y = " << y << '\n';
```

give the following output:

```
x = 1 and y = 2
```

Note that any spaces required in the output must be included within quotation marks.

Some useful formatting commands are:

Command	Symbol
newline	\n
tab	\t
'	\'
"	\"
(bell)	\a

Sometimes, for example if the computer is busy doing a large volume of computation, the program may not print the output to the screen immediately. If immediate output is desirable then use `std::cout.flush()` after any `std::cout` commands

```
std::cout << "Hello World\n";
std::cout.flush();
```

or use `std::endl`, which combines a newline with a flush:

```
std::cout << "Hello World" << std::endl;
```

Keyboard input

Keyboard input for numbers and characters is achieved using `std::cin`

The following statements prompts someone to enter their PIN

```
int pin;
std::cout << "Enter your PIN, then hit RETURN\n";
std::cin >> pin;
```

`cin` may be used to ask for more than one input at a time

```
int accountno, pin;
std::cout << "Enter your account number then hit RETURN,\n";
std::cout << "and then your PIN followed by RETURN\n";
std::cin >> accountno >> pin;
```

Keyboard input for strings is slightly different. An example is given below

```cpp
#include <iostream>
#include <string>
int main()
{
  std::string name;
  std::cout << "Enter your name and then hit RETURN\n";
  std::getline(std::cin, name);
  std::cout << "Your name is " << name << '\n';
  return 0;
}
```

Writing to file

When writing to file, an additional header function `fstream` is needed.

```
#include <iostream>
#include <fstream>
#include <string>
```

The file `output.dat` may be opened using the statement

```
std::ofstream out("output.dat");
```

We can then write to this file in a similar manner as writing to the screen, with the exception that `cout` is replaced by `out`

There are a number of formatting options available. The following prints data in scientific format:

```cpp
#include <iostream>
#include <fstream>
int main() {
  double x = -1.0, y = 45.3275893627129, z = 0.00000001;
  std::ofstream out("output.dat");
  out.setf(std::ios::scientific|std::ios::showpos);
  out << x << " " << y << " " << z << '\n';
  out.close();
  return 0;
}
```

[< compiler explorer >]; [< cpp reference >]

Reading from file

Suppose the file `numbers.dat` has 3 columns of numbers. This file can be opened using the following code

```
#include <fstream>

double x, y, z;
std::string line;
std::ifstream input("numbers.dat");
assert(input.is_open());
```

`input.is_open()` returns `true` if the file was successfully opened

We can read the file like so:

```
#include <sstream>

while(std::getline(input, line)) {
    std::istringstream s(line);
    s >> x >> y >> z;
}
input.close();
```

`std::getline` gets the next line of the file and returns **`false`** if we are at the end of the file.

`std::istringstream` converts the `std::string line` to a stream (like `std::cin`) which can be used to separate out the three columns.

Tip - Use of assert statements for debugging

- `assert` statements can be used in code to confirm something you expect to be true
- For example, you may wish to confirm that a number you are about to take the square root of is non–negative
- If the condition is not met, the code aborts giving an error message that explains what went wrong
- To use `assert` statements you must `#include <cassert>`
- An example of the use of `assert` statements is given on the next slide

```cpp
#include <iostream>
#include <cassert>
#include <cmath>
int main()
{
  double a;
  std::cout << "Enter a non-negative number\n";
  std::cin >> a;
  assert(a >= 0.0);
  std::cout << "The square root of a is " << std::sqrt(a) << '\n';
  return 0;
}
```

If the code on the previous slide is compiled and run, and the user enters “-5” at the prompt, the code will be terminated and the following error message given:

```
a.out:: program.cpp:10: int main(): Assertion 'a >= 0.0' failed
```

This error message tells us to look at line 10 in `program.cpp`, where the assertion in quotes failed the test.