

Introduction to Programming using C++

Lecture One: Getting Started

Carl Gwilliam

gwilliam@hep.ph.liv.ac.uk

<http://hep.ph.liv.ac.uk/~gwilliam/cppcourse>

Course Prerequisites

- What you should already know about C++

NOTHING!!

- I have assumed:
 - You have never encountered C++ before.
 - You have limited to no programming experience in any language.

By the end of the course

- You should:
 - Have a working knowledge of all of the common C++ syntax.
 - Know how to avoid common coding pitfalls.
 - Be able to use and create your own functions and classes.
 - Recognise the advantages of using pointers and references.
 - Be able to understand the fundamental ideas of object oriented (OO) design.
- Be aware!
 - This is not the comprehensive course for C++!
 - Some advanced topics will not be covered in this course.
 - Project specific C++ courses are worth attending if you have time.
- You are learning a sophisticated language! It will take some time and a fair amount of hands-on experience to become familiar with C++.

Course format

- Lecture 1: Getting Started
 - Statements, variables, types, operators, I/O, conditional statements
- Lecture 2: Further Syntax
 - For and while loops, increment and logical operators, sorting algorithms
- Lecture 3: Functions
 - Library and user functions, declarations, arguments, overloading
- Lecture 4: Pointers and References
 - References, pointers, passing by reference, pointers and arrays, consts
- Lecture 5: Introducing Classes
 - Declarations, member variables and functions, accessors, overloading

Course Format

- Lecture 6: Classes in Practice
 - Constructors and destructors, constant functions, memory management
- Lecture 7: Designing Classes
 - Passing by const reference, copy constructors, overloading operators
- Lecture 8: Towards OO Design
 - Inheritance, virtual functions, multiple inheritance, abstract classes
- Lecture 9: Templates & the STL
 - Function & Class Templates, Specialisation, String, Vector, Map
- Lecture 10: File I/O and Streams
 - Reading, writing, formatting, strings as streams

Why do you need to learn how to write code?

"I am not a computer scientist!"

- Software development skills are usually required in every element of your project - from extracting data to getting your results published.
- It will be very useful to learn good programming techniques now so you can save a lot of time later.

Why Learn C++?

**"I already know how to program in FORTRAN/C/ASSEMBLER
so why do I have to bother with C++?"**

- In recent times there has been a shift from procedural languages such as FORTRAN to OO-style languages such as C++ and Java.
- The language of choice, for now, is C++.
- C++ is ideally suited for projects where the active developers are not located in the same place.
- Our survey says: almost all of the current HEP Ph.D. students write and execute C++ code on a **daily** basis.

C++ is not **the** answer, it is a reasonable solution..

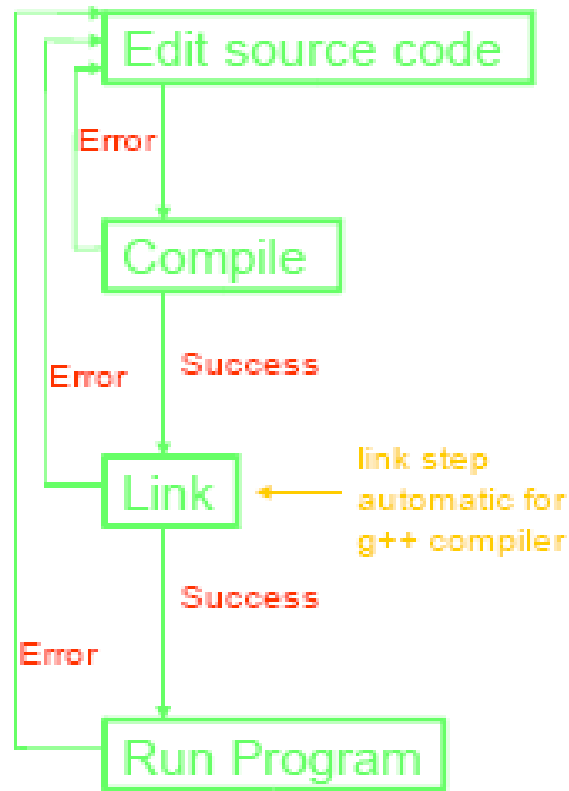
Hello World!

- The first complete C++ program in this course:

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!";
    return 0;
}
```

- Every C++ program must contain one (and only one) **main** function.
- When the program is executed "**Hello World**" is displayed on the terminal screen.

Compiling Hello World



program output `> Hello.exe`
Hello World!

- Once the code is written it has to be compiled:

`g++ -Wall -o Hello.exe
HelloWorld.cpp`
- If the code compiles successfully an **object** file (**HelloWorld.o**) is created.
- This object file is then linked with other object files and libraries required to create the exe.
- The executable can be run at the command prompt, just like any other UNIX command.

A simple example

```
int a; float b; float c;  
float d = 10.2;  
float result;  
// get two numbers  
cout << "Enter two numbers" << endl;  
cin >> a; cin >> b;  
c = 2.1;  
result = b * a + c;  
result = (result + 2)/d;  
// display the answer  
cout << "Result of calculation: " << result << endl;
```

Statements

- Statements are used to control the sequence of execution, evaluate an expression (or even do nothing).
- ALL statements are terminated with a semi colon ;

```
cout << "Enter two numbers" << endl;  
result = b * a + c;
```

- One statement can trail over several lines.
- Code is easier to read with a sensible use of **whitespace**.

```
result=      b* a  
+c          ;  
//is the same as  
result = b * a + c;
```

Comments

- Comments are useful for you and other developers!

```
// get two numbers  
// display the answer
```

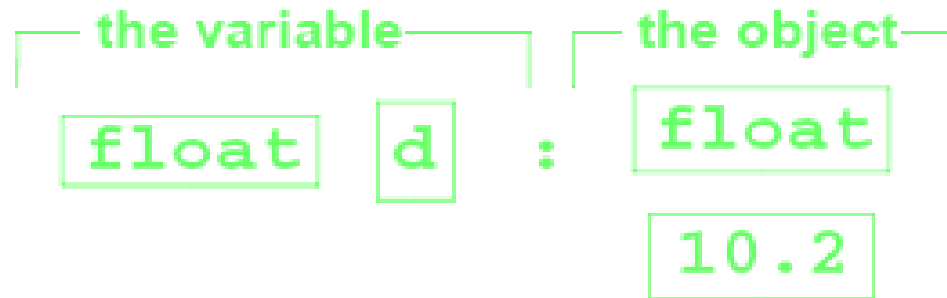
- A comment begins with `//` and ends with a line break.
- Comments indicate the purpose of the surrounding code.
- They can be placed anywhere in the program, except within the body of a statement:

```
// this is a valid comment  
result = b * a; // this is also valid  
result = // this is NOT valid (result  
+ 2)/d;  
/* You may also use this 'C' style  
comment to span multiple lines */
```

Variables

- A variable is a name associated with a location in computer memory used to store a value.
- In order to use a variable in C++, we must first **declare** it by specifying which data type we want it to be.

```
int a;  
float d = 10.2;
```



- Variable 'a' has a **type integer** associated with some object of **type integer**.
- Variable 'd' has a **type float** associated with some object of **type float** with a value of 10.2

Variable Types

- What does it mean to say a variable has a **type**?
- The C++ built-in types are:

int a;	//integer
float b;	//real
double c;	//real
bool d;	//boolean (0 or 1)
char e;	//character (ASCII value)

- The compiler needs to know how much **memory** to set aside for the object associated with the variable.
- Each type requires a specific amount of space in memory.

The Built-in Types

Type	Size (bytes)	Range
bool	1	true/false
char	1	256 chars
int	2 or 4	(see below)
short int	2	$\pm 32,768$
long int	4	$\pm 2.1 \cdot 10^9$
float	4	$\pm 3.4 \cdot 10^{38}$
double	8	$\pm 1.8 \cdot 10^{308}$

1 byte = 8 bits = 2^8
possible combos.

Size determined
by the compiler.

Range can be
doubled by using
unsigned int

- That is all? How does C++ deal with complex numbers?
- C++ has the capacity to create **user defined** types. This is a fundamental concept of the language.

Initialisation

- When declaring a variable, it's value is by default undetermined.
- We can give the variable a value at the same moment it is declared
- There are two ways to do this in C++.
 - The more used c-like method:

```
// type identifier = initial_value  
int a = 0;
```

- The constructor method:

```
// type identifier (initial_value)  
int a(0);
```

- It is good practice to **always** initialise a variable to a value

Assignment

- A variable can be assigned (or reassigned) to a particular value with the **assignment operator (=)**.

```
c = 2.1;           //assignment operator  
result = b * a + c;
```

- Do not mistake the above statements for algebra!
- Value of the expression to the right of the assignment operator (**rvalue**) is assigned to the variable on the left (the **lvalue**). Remember assignment is right-to-left:

```
c = 2.1; //c assigned the value 2.1  
2.1 = c; //incorrect!
```

Arithmetic Operators

- There are **five** primary mathematical operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Modulus	%

the modulus is the remainder from integer division (e.g. **33%6** is equal to **3**).

- Be careful applying the division operator when assigning a variable with a different type:

```
int a = 8; int b = 5;  
float c = a / b;           // c is 1  
c = (float)a / (float)b;  // c is 1.6
```

Casting an integer to a float ('c'-style)

Multiple Operators and Precedence

```
result = b * a + c;  
result = (result + 2)/d;
```

- There are no restrictions on the amount of operators in an expression.

```
float a = 6 * 4 + 4 * 2;           // answer is 32  
float b = 6 * (4+4) * 2;           // answer is 96  
// expression is incomprehensible  
float c = b+c * 2/d + 3 + b/c + d - b *c/d;  
// still unclear!  
float d = b + ((c*2)/(d+3)+(b/(c+d))-(b*c/d));
```

*
/
%
+
-
↑
PRECEDENCE

- Do not assume that an expression is evaluated from left to right by the compiler. In C++, an operator has a **precedence** value.

Input and Output

- **cin** reads information from the keyboard and **cout** prints information to the terminal (they're defined in the header `iostream`).

```
cin >> a;  
cout << "Enter two numbers" << endl;
```

- For now, you can use these statements without fully appreciating the syntax.

```
float a = 13.3;  
cout << "This is a string" << endl;  
cout << "Value of a: " << a << endl;  
cout << "a: " << (int) a << endl;  
cout << "a =\t" << a << "\n";
```

New line	\n
Horizontal tab	\t
Backspace	\b
Alert	\a
Backslash	\\

```
This is a string  
Value of a: 13.3  
a: 13  
a =      13.3
```

Equality example

```
float a, b, c = 0;
cout << "Enter two numbers" << endl;
cin >> a; cin >> b;
c = a - b;
if (a == b) cout << "Two numbers are equal\n";
if (!c) {
    cout << "Two numbers are equal\n"; return 0;
}
if (a > b) {
    cout << "The first number is greater\n";
} else {
    cout << "The second number is greater\n";
}
return 0;
```

Conditional Statements

```
if (expression) statement;
```

- Apply the condition in the expression to determine **if** the statement will be executed.
- The capability of altering the program flow depending on a outcome of an expression is a powerful tool in programming.
- The expression in a conditional statement is always evaluated as **false (0)** or **true (non-0)**.

Conditional Statements

- A statement will be included in the program flow if the condition is true.
- The greater than symbol $>$ and the equivalence symbol $==$ are relational operators.

```
if (a > b) statement;  
if (a == b) statement;
```

- Remember the difference between the assignment ($=$) and equivalence ($==$) operators!

Condition
is always
true

```
int x = 4; int y = 3;  
if (x = 4) cout << "x equals 4\n";  
if (y = 4) cout << "y equals 4\n";
```

x equals 4
y equals 4

Relational Operators

- Relational operators are used to evaluate if an expression is true or false.

$x < y$	Less than
$x \leq y$	Less than or equal to
$x > y$	Greater than
$x \geq y$	Greater than or equal to
$x == y$	equal
$x != y$	Not equal

- The negation operator ! inverts the logic of the result.

```
if (!(x > y)) cout << "x less than y";
```


Boolean Logic

if (!c)

- Conditional statements use **boolean** logic to determine whether the expression is true or false.
- A **non-zero** value is true, a **zero** value is false.

```
int x = 2; int y = 0;  
if (x) cout << "x equals 2" << endl;  
if (!y) cout << "y equals 0" << endl;
```

x equals 2

y equals 0

Boolean logic is used frequently in C++ so be prepared!

Compound Statements

- Compound statements (or **blocks**) act as a single statement.
- They enable multiple statements to be attached to one condition.
- Blocks are started and ended with **braces { }**
- A **;** is not placed at the end of a block

```
{  
    statement 1;  
    :  
    statement n;  
}
```

If-else Statements

- A true condition in the **if** expression will result in **block a** being executed otherwise **block b** will be executed.
- The **if** statement can be used for multiple conditions and nested conditions:

```
if (expression) {  
    block a;  
} else {  
    block b;  
}
```

```
if (a > b) {  
    :  
} else if (a < b) {  
    :  
} else {  
    :  
}
```

```
if (a > b) {  
    if (a >= c) {  
        :  
    } else {  
        :  
    }  
}
```

- Note, else block is only executed if none of above is true