

# Introduction to C++ Programming Language

## Session 1

**Rishad Shafik (EEE)**

rishad.shafik@ncl.ac.uk

Slides by

**Jonas Rylund Glesaaen**

jonas@glesaaen.com

2017

# Introduction

Dr Rishad Shafik

# Who am I?

- Rishad Shafik
- "A microsystems designer"
- I teach computer systems
- Someone who believes **everyone** should know computer programming

# Course material



Combination of lectures, exercises and demonstrations

# What will we learn?

- **Basic C++ syntax**
- **Control structures**
- **Functions**
- **Structs and classes**
- **Templates and STL**
- **Exceptions**

# Session 1 Topics

- 1 Introduction
- 2 Syntax and structure
- 3 Types and variables
- 4 Control Structures
- 5 Crash Introduction to IO
- 6 Coding Environments
- 7 Programming Practices
- 8 Recap

# What is C?

**A relic from the 70s, 80s and 90s that has had a huge influence on most modern programming languages.**

# What is C?

## Notable features

- It is a procedural language
- It is statically typed
- It has low-level access to memory
- **Readable syntax** (in my opinion)



# What is C++?

**Anything you can do, I can do better.  
I can do anything better than you.**

Annie Get Your Gun

# What is C++?

**Anything you can do, I can do better.  
I can do anything better than you.**

Annie Get Your Gun

**C++ is a language built on top of the C programming language**

# What is C++?

## Additional features

- **Classes and inheritance**
- **Templates**
- **Exceptions**
- **A huge standard library**
- **... and it is in active development**

# Versions of C++

**C++ is constantly evolving, hence there are many standards**

- **C++98**
- **C++03**
- **C++11**
- **C++14**
- **C++17**

# Versions of C++

**C++ is constantly evolving, hence there are many standards**

- **C++98**
- **C++03** ← **Current standard of many compilers**
- **C++11**
- **C++14**
- **C++17**

# Versions of C++

**C++ is constantly evolving, hence there are many standards**

■ **C++98**

■ **C++03**

■ **C++11**

■ **C++14**

■ **C++17**

} **We will use these**

# The slides

Things that are bad practice will be marked

**Bad Practice**

Things new to C++11 or C++14 will be marked

**{C++11}**  
**{C++14}**

# Syntax and structure



# Hello World in C++

```
#include<iostream>

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

# Hello World in C++

Include external libraries

```
#include<iostream>
```

```
int main() ← The main function
```

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

A string literal

Built in terminal stream object

# A program in C++

In essence all C++ programs consist of two things

**1** Sentences

**2** Blocks

# A program in C++

## 1 Sentences

```
#include<iostream>

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

**A complete instruction ending with a ;**

# A program in C++

## 2 Blocks

```
#include<iostream>

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

A group of instructions inside of a pair of **{ }**

# Types and variables

# What is a variable?

**A variable is simply a named location in memory**

```
int main()  
{  
    int n = 5;  
    std::cout << &n << std::endl; //0x7fff27ea5464  
}
```

↑  
**name of location  
in memory**

# What is a variable?

**Its data type tells the compiler two important things**

- 1 How much memory the variable needs**
- 2 The allowed operations on the variable**



# Variable initialisation

```
double rate_of_decay = 0.75;
```

# Variable initialisation

```
double rate_of_decay = 0.75;
```

 Data type of the variable

Tells the compiler:

- `rate_of_decay` is a `double`
- It needs 8 bytes of memory {usually}

# Variable initialisation

**Name of the variable**

**Your hook to the newly allocated memory**



```
double rate_of_decay = 0.75;
```

# Variable initialisation

```
double rate_of_decay = 0.75;
```

A **double** literal which  
will in this case be placed in  
the allocated memory slot



# Variable initialisation

## Assignment Style

```
double rate_of_decay = 0.75;
```

# Variable initialisation

## C++03 Constructor Style

```
double rate_of_decay (0.75);
```

# Variable initialisation

## C++11 Constructor Style

```
double rate_of_decay {0.75};
```

# Variable initialisation

## Undefined Declaration Style

```
double rate_of_decay;
```



# Naming variables

`( [_a-zA-Z] ) [_a-zA-Z0-9]*`

# Naming variables

$([_a-zA-Z])([_a-zA-Z0-9])^*$

## Exceptions

- Keywords defined by the language
- Names starting with `_` or `__` are reserved

**Keywords:** `int`, `float`, `while`, `const`, `false`, ...

# Naming variables

One should find a system and stick to it

E.g. mixed style

**Variables:** `snake_case`

**Functions:** `mixedCase`

**Classes:** `CamelCase`

# Naming variables

**One should find a system and stick to it**

**E.g. Stroustrup style**

**Variables:** `snake_case`

**Functions:** `Mixed_case`

**Classes:** `Mixed_case`

# Built in data types

Basically four built in data types in C++

<b>Boolean:</b>	<code>bool</code>	<code>true, false</code>
<b>Character:</b>	<code>char</code>	<code>'c', '#', '7', ...</code>
<b>Integer:</b>	<code>int</code>	<code>0, 12, -42, ...</code>
<b>Floating point:</b>	<code>float</code>	<code>0.0, 1.33, -4.11, ...</code>

# Type qualifiers

**Type qualifiers manipulate the built in types**

# Type qualifiers

## Manipulate memory size

short long

## Manipulate value range

signed unsigned

# Type qualifiers

Type	Size* (minimum)
short int	<b>2 byte</b>
int	<b>2 byte</b>
long int	<b>4 byte</b>
long long int	<b>8 byte</b>
float	<b>4 byte</b>
double	<b>8 byte</b>
long double	<b>10 byte</b>



# Type qualifiers

Type	Value range
int	-32,768 to 32,767
long int	-2,147,483,648 to 2,147,483,647
unsigned int	0 to 65,535
float	$\pm 1.175,494,3 \cdot 10^{-38}$ to $\pm 3.402,823,4 \cdot 10^{38}$
double	$\pm 2.225,073,858,507,201,4 \cdot 10^{-308}$ to $\pm 1.797,693,134,862,315,7 \cdot 10^{308}$

# Literals

Explicit values whose type are syntax dependent

- Integers are just numbers: 5
- Floats have a decimal point: 4.5
- Characters are surrounded by `'`: `'c'`
- Booleans are either `true` or `false`
- C strings are surrounded by `"`: `"Hello"`
- Function literals: `[] (int) { /* ... */ };` `{C++11}`

# Literals

One can add qualifiers to literals as well

Literal	Type
42u	unsigned int
167l	long
5.62	double
1.0e-2	double
4.12f	float

# Operators

Operators in programming are much the same as operators in mathematics

- Arithmetic operators: `+` `-` `*` `/` `%`
- Logic operators: `and` `&&` `or` `||` `!`
- Comparison operators: `==` `<` `!=` `>` `<=` `>=`
- Combined operators: `+=` `-=` `*=` `/=`
- Others: `<<` `>>` `=` `++` `--` `?` `&` `::` `->`

# Operators

**Operators have different precedence levels**

■  $5+12*7/4-2$

■  $4>3$  and  $7==8$  or  $16<=72$

# Operators

**Operators have different precedence levels**

■  $5+12*7/4-2$  ← **24**

■  $4>3$  and  $7==8$  or  $16<=72$

# Operators

Operators have different precedence levels

■  $5+12*7/4-2$  ← **24**

■  $4>3$  and  $7==8$  or  $16<=72$  ← **true**

Whitespace does nothing in C++

# Operators

Expressions can be grouped with ( )

- $(5+12)*7/4-2$

- $(5+12)*7/(4-2)$



# Operators

Expressions can be grouped with ( )

■  $(5+12)*7/4-2 \leftarrow 27$

■  $(5+12)*7/(4-2)$

# Operators

Expressions can be grouped with ( )

■  $(5+12)*7/4-2 \longleftarrow 27$

■  $(5+12)*7/(4-2) \longleftarrow 59$

# Type casting

C++ is **statically** typed, but not **strongly** typed

One can change between the types using casting

```
int n = 5;  
double cn = (double)n; ← C style cast  
double cppn = static_cast<double>(n); ← C++ style cast
```

But you can never change the type of the variable **n**

# Variable qualifiers: **const**

Constants are declared with **const**

```
const int size_of_arrays = 100;
```

```
/* ... */
```

```
size_of_arrays *= 2; ← Compile error
```

# Variable qualifiers: **const**

**mutable**  **const**

# Variable qualifiers: `const`

`mutable`  `const`

In `C++` variables are mutable by default

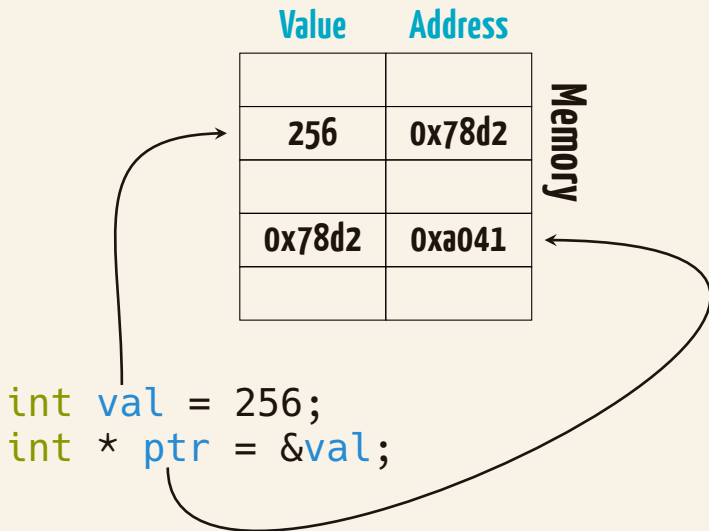
Different from languages like `rust`

# Pointers

**Variables are still just named locations in memory**

**We can work with these locations using pointers**

# Pointers





# Pointers

Value	Address
256	0x78d2
?	0x78d3
0x78d2	0xa041
0x78d3	0xb6d2

Memory

```
int * ptr = &val;  
int * ptrpp = ptr + 1;
```

# Pointers

You can access the value of the memory the pointer points to by **dereferencing** it



```
int * ptr = &val;  
int half_val = *ptr / 2;
```

**This is why the type of the pointer is important, so the program knows how many bits to read.**

# The null pointer

Sometimes it is useful to be able to say that the pointer doesn't point to anything

```
double * ptr = NULL; ← Old style  
double * ptr = nullptr; ← New style {C++11}
```

# References

A reference works as an alias for a variable

```
int value = 134;  
int & ref = value;  
int value_half = ref / 2;  
&value == &ref; ← true
```

In many languages assignment automatically creates references and not copies, e.g. **JavaScript**

# What's the deal with \* and & ?

In type specifications:

- \* declares a pointer
- & declares a reference

As operators

- \* converts pointer to reference
- & converts reference to pointer {sort of}

# The `auto` type

When assigning

`type of variable` = `type of expression` ;

These types are generally the same

`auto` = whatever the type on the right is

# The auto type

**auto** only picks up the base type

```
const char & char_ref = some_char;  
auto var = char_ref;
```



char

**But you can use type qualifiers on it**

# C Style Arrays

One can create a list of objects like this:

```
int array[10];
```

Places 10 integers consecutively in memory

Memory

a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
...



# C Style Arrays

As a type qualifier `[]` decides how much memory should be reserved

```
int fibonacci_numbers[10];
```

As an operator `[]` is used to access the various memory positions

```
fibonacci_numbers[5] = 8;  
fibonacci_numbers[6] = 13;
```

# C Style Arrays

The array is actually just a pointer in disguise

```
array[0] == *array
```

The `[]` operator is also just a shorthand

```
array[n] == *(array + n)
```

# C Style Arrays

One can initialise the array with an initialiser list

```
int lucky[] = {4, 12, 42, 7};
```



Size of the array inferred from context

# C Style Strings

A C style string is simply an array of characters

```
char message[] = "How are you all doing?";
```

The final entry is always the null character `'\0'`

(`message` has 23 elements)

# Control Structures

# Controlling program flow

**So far we have learned to write programs that execute "in a straight line"**



# Controlling program flow

So far we have learned to write programs that execute "in a straight line"

- What if we want to **branch**?



# Controlling program flow

So far we have learned to write programs that execute "in a straight line"

- What if we want to **branch**?
- What if we want to **repeat**?





# Controlling program flow

So far we have learned to write programs that execute "in a straight line"

- What if we want to **branch**?
- What if we want to **repeat**?

Then we need control structures



# Conditionals: if, else

```
if ( condition ) {  
← true  
} else {  
← false  
}
```

# Conditionals: if, else

Can also do multiple statements

```
if ( condition ) {  
  
} else if ( condition ) {  
  
} else {  
  
}
```

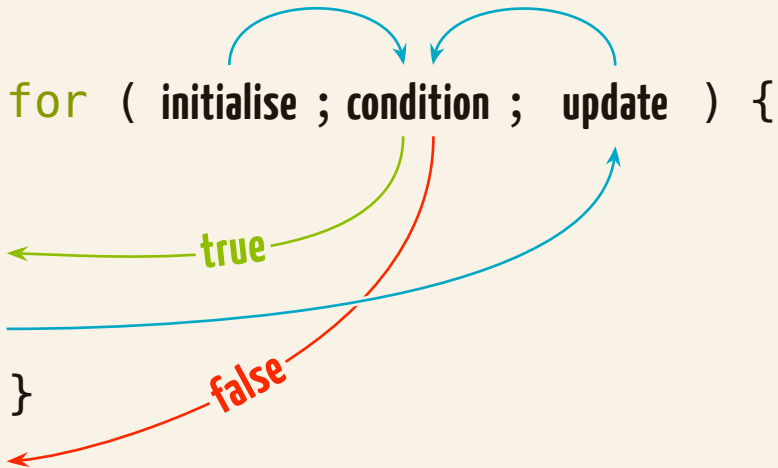
It chooses the **first** condition that matches

# Conditionals: switch

```
switch ( variable ) {  
  case value #1:  
  ← case 1  
  case value #2:  
  ← case 2  
  default:  
  ← no case  
}
```

A diagram illustrating the structure of a switch statement. The code is shown in a monospace font. The opening curly brace is on the right. Three cases are listed: 'case value #1:', 'case value #2:', and 'default:'. Each case is followed by a left-pointing arrow and a label: '← case 1' (green), '← case 2' (blue), and '← no case' (red). Three curved arrows originate from the right side of the code block: a green arrow from the first case label points to the first case, a blue arrow from the second case label points to the second case, and a red arrow from the 'no case' label points to the default case. A closing curly brace is on the left, aligned with the first case.

# Loops: for

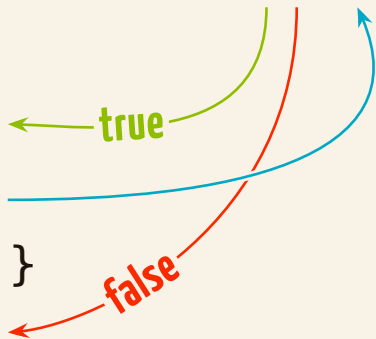


# Loops: for - example

```
int numbers[100];  
  
for (auto i = 0; i < 100; ++i) {  
    numbers[i] = i;  
}
```

# Loops: while

```
while ( condition ) {
```



# Loops: while - example

```
// Calculate 10!  
unsigned factorial = 1;  
unsigned counter = 10;  
  
while (counter > 1) {  
    factorial *= counter;  
    --counter;  
}  
  
std::cout << "10! = " << factorial;
```



# Exiting loops

There are two commands for altering loop flow

**break** and continue

```
while ( condition ) {
```

```
    break;
```

```
}
```


**break** exits the loop



# Exiting loops

There are two commands for altering loop flow  
**break** and **continue**

```
while ( condition ) {  
    continue;  
}
```



**continue** jumps back to the loop update

# Crash Introduction to IO

# Streams

In C++ IO is handled by something called streams

We use the shift operators to interact with the stream objects

Original meaning

**object1** << **object2** ← left shift operator

**object1** >> **object2** ← right shift operator

# Streams

In C++ IO is handled by something called streams

We use the shift operators to interact with the stream objects

Meaning adopted by stream objects



# Standard in and out

The `iostream` library includes two convenient stream objects

`std::cout` for writing to console

`std::cin` for reading from keyboard

# Standard in and out - example

```
#include<iostream>

int main()
{
    double input_from_user {0.};

    std::cin >> input_from_user;

    std::cout << "You wrote: \"" << input_from_user
        << "\"" << std::endl;
}
```

# Standard in and out - example

```
#include<iostream>
int main()
{
    double input_from_user {0.};

    std::cin >> input_from_user;

    std::cout << "You wrote: \"\" << input_from_user
        << "\"\" << std::endl;
}
```

To gain access to `std::cout` and `std::cin`

Character escape

Flush stream and create newline



# Coding Environments

# Live Example

# Programming Practices

Dr Rishad Shafik

# Good Programming Practices

- **Do not** use global variables (constants are OK)
- **Always** initialise built in functions with a default value
- **Always** use descriptive variable names (and stay away from magic numbers)
- Use **const** consistently

# Good Programming Practices

Stay away from C functionality that has been superseded

- **Macros, especially `#define`**
- **Pointers to `void`**
- **`NULL` for empty pointers**
- **`printf` and `scanf` for IO**

# Recap

# Recap Session 1

- A C++ program consists of **sentences** and **blocks**
- The type of the variable tells the compiler
  - How much memory the variable needs
  - What actions are allowed on the variable
- There are four basic types in C++
  - `bool`, `char`, `int`, `float`
- Types can be modified with qualifiers

# Recap Session 1

- Use **pointers** to examine memory locations
- **References** if you want to alias a variable
- Use **if** and **switch** to make branches in your code
- Use **for** and **while** to repeat stuff