Getting Started in C++

COSC1076
Semester 1 2019
Week 01

Why C++?

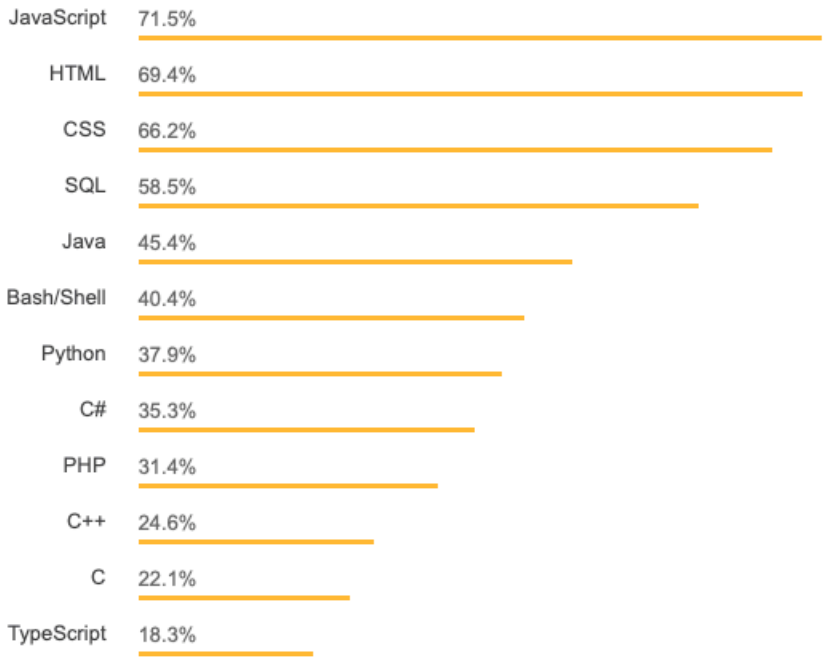
RMIT UNIVERSITY

Why C++?

- Primary reason: Learning Programming Skills & Techniques
  - Dynamic Memory Management
  - More explicit program control
  - Supported language feature set
- Secondary reason: Learn a foundational & common language family
  - C++ is used for:
    - Speed
    - Optimisation
    - Efficiency
  - GPU Programming

Why C++?

From 2018 Stack Overflow Survey (Professional Developers)



C, C++, C++11, or C++14 ?

- C++ is originally an extension to C
  - C is a legal subset of C++
  - Biggest introduction are Classes, Generics & the STL (standard template library)
  - This course works with C++, but many concepts are perfectly fine in C
- C++ has seen many ***standards***, that require standard compliant compilers to consistently handle
  - C++11 (2011), was a major overhaul to the language
  - C++14 (2014), additional language feature, consistency updates, bug fixes,
    - *This is the version we are using*
  - C++17 (2017), latest standard, we won't use this

Learning a new Language Is a Skill

RMIT UNIVERSITY

Java/C++ diff

RMIT UNIVERSITY

C++ Program Structure

- Header Includes
- Defines
- Namespace uses
- Function Declarations
- Main Function
- Function Definitions

```
#include <cstdio>
#include <iostream>
```

```
#define EXIT_SUCCESS    0
```

```
using std::cout;
using std::cin;
```

```
double foo(int x, float y, char z);
void bar(int x, float y, char z);
```

```
int main (void) {
   int i;
   float f;
   char c;
   double d;

   cin >> i;
   cin >> f;
   cin >> c;
   d = foo(i, f, c);

   cout << "foo:\t" << d << "+" << f << "*" << c << "=" << d << std::endl;
   printf("foo:\t%d + %.2f * %c = %.2lf\n", i, f, c, d);

   bar(i, f, c);
   printf("bar:\t%d + %.2f * %c = %.2lf\n", i, f, c, d);
   cout << "bar:\t" << d << "+" << f << "*" << c << "=" << d << std::endl;

   return EXIT_SUCCESS;
}
```

```
double foo(int x, float y, char z) {
   return x + y * z;
}

void bar(int x, float y, char z) {
   x = y;
}
```

Compiling and Running C++ Programs

- Before being executed, C++ programs must be compiled into ***Machine Code***
  - Similar, but different from Java
  - Machine code is CPU (processor) specific
- Use GCC (g++) compiler

```
g++ -Wall -Werror -O -o <executable> <codefile.cpp> <codefile.cpp> …
```

- Compiler options
  - `-Wall` enable all error checking
  - `-Werror` convert warnings into errors, stopping compilation
  - `-O` turn on optimiser
  - `-o <filename>` output filename of executable

The Basics - What is the same

- Comments
- Some Types
  - bool
  - int
  - float/double
  - char
- Operators
  - Arithmetic
  - Comparison
- Selection
  - if / elseif / else
- Iteration
  - While
  - For

Differences

- Standard I/O
  - C++: cout / cin
  - C: printf / scanf
- Types
  - Strings
  - Extended types
  - Implicit casting
- Arrays
- Declarations
- Functions
  - Parameter Passing
- #defines
- Global Variables
- Namespaces
- Declare & Initialise?

Standard I/O - C++ STL (cout)

- For output, use the `cout` object
  - Contained in the `<iostream>` header
  - Within the `std` namespace
- Uses the output operator (`<<`)

```
<output location> << <what to output>
```

  - Uses default formatting for output
  - Returns a value - the output location
  - Allows operators to be chained
- Example

```
std::cout << 7 << 'a' << 4.567 << std::endl
```

Standard I/O - C++ STL (endl)

- Operating System independent newline character:
  - `std::endl`
  - Equivalent to using '`\n`' character.
- These are the same:

```
std::cout << 7 << std::endl
    std::cout << 7 << "\n"
```

Standard I/O - C++ STL (cin)

- For input, use the `cin` object
  - Contained in the `<iostream>` header
  - Within the `std` namespace
- Uses the input operator (`>>`)

```
<input location> >> <variable>
```

  - This is context sensitive!
  - Uses the type of the input variable to determine what to read from input
  - (Can also be chained)
- Example

```
int x
std::cin >> x
```

Standard I/O - C++ STL (cin)

- What about:
  - End of input?
  - Input error or failure?
- `cin` is an object - you should be familiar with these from Java
  - Has functions to check for these things
    - `eof()` - check for end of file
    - `fail()` - check for read error
  - (More on classes and objects next week)

Standard I/O - C++ STL

- Other functions for reading that could be used:
  - `std::getline()`
  - `std::read()`
  - More on these later in the course, since we haven't seen how to use their argument yet (need c-style strings)

Standard I/O - C functions (printf)

- C functions provide alternative I/O
- For output, use the `printf` function
  - Contained in the `<cstdio>` header
  - Function, not an object!
- Provides “formatted printing”
  - Takes two ‘sets’ of parameters
  - A format string, and an ordered list of variables
  - The format string contain ‘%’ terms to denote:
    - Where the value of variables should be placed
    - What type & format to use for displaying variables
    - Order of ‘%’ and parameter variables matters

Standard I/O - C functions

- Common format (%) codes are:

`d`	Decimal, can specify number of significant digits: `%<x>d`
`f`	Float, can specify number of digits: `%<x>.<y>f`
`lf`	Double (long float), uses same format as for float
`e`	Decimal in Scientific notation
`c`	Single ASCII Character
`s`	String (c-style)

Standard I/O - C functions (scanf)

- C functions provide alternative I/O, which may be
- For output, use the `scanf` function
  - Contained in the `cstdio` header
- Similar to `scanf`
  - Takes two ‘sets’ of parameters
  - A format string *to read*, and an ordered list of variables
    - When providing variable, must prepend `&` (ampersand) operator
    - Explanation for why you need this next week
  - Returns:
    - Number of items read
    - `EOF` - on end of file

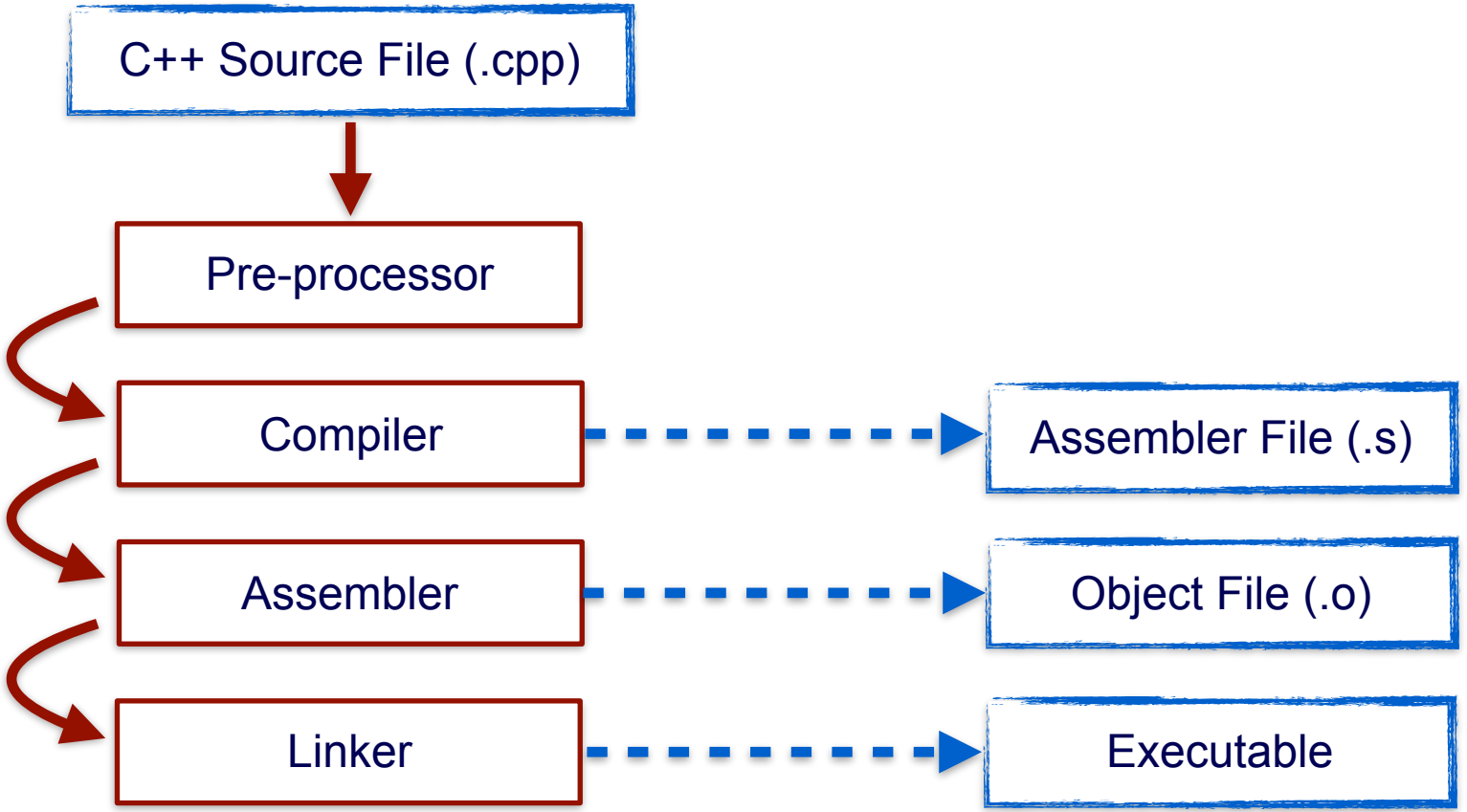
#define's

- `#define` statements allow constants to be defined in the program
  - Syntax

```
#define DEFINE_NAME <value>
```

  - By convention, always use uppercase
  - Placed at the top of the file (below headers)
- They act as a literal “find-and-replace”, so be careful about:
  - Brackets
  - ‘`;`’ for end-of-statement

C/C++ Compilation Process



C/C++ Preprocessor

- Prepare source code files for actual compilation
- Process '#' pre-preprocessor directives
  - Process `#include` statements
    - locates and includes header files
  - Process `#define` statements
    - find-and-replace
  - Process `#ifdef` statements
    - will see later
  - Process `#pragma` statements
    - compiler specific directive, not used in this course

Types may not be what they seem

- Numbers represented true and false
  - 0 is false
  - Any non-zero value is true
- A `bool` is implemented as a number
  - `false` is always 0.
  - But `true` is not necessarily 1.
- A `char` is a signed 8-bit number.
  - You can ‘add’ and ‘subtract’ characters, which does have uses

STL Strings

- Like Java, the STL provides a string object
  - Contained in `<string>` header
  - Within the `std` namespace
- Supported operations include:
  - Assignment with “ ” - style syntax
  - Concatenation with ‘+’ operator (of string objects!)
- Has methods/functions that can be called
  - `c_str()` - talk more about this next week
  - `substr()` - substring
  - `find()` - find substring

Types

- The values a type can hold are dependent on the ‘size’ of the type:
- C++ has extended the following data types:
  - `{signed | unsigned} {long | short} int`
  - `{signed | unsigned} char`
  - `{long} double`
- By convention, the sizes are:

`int`	32 bits
`long`	64 bits
`short`	16 bits
`float`	32 bits
`double`	64 bits
`long double`	80 bits
`char`	8 bits

Type Casting

- C++ use implicit type casting to convert between compatible types
  - Typically this applies to numeric types
  - Be careful!
    - Implicit type conversion only happens *when absolutely necessary*
- Explicit type casting is done using bracket notation

```
(new type) value
```

```
(int) 7.4f
```

Declaration vs Definition vs Initialisation

- Declaration
  - Introduce a name (variable, class, function) into a scope
  - Fully specify all associate type information
- Definition
  - Fully specify (or describe) the name/entity
  - All definitions are declarations, but not vice versa
- Initialisation
  - Assign a value to a variable for the first time

What happens if you define a variable without initialising it?

Arrays

- Similar to Java Arrays
  - Largely syntactic difference when declaring
  - No need to “new” the array

```
int a[LENGTH];
```

  - Can be initialised when declared

```
int a[LENGTH] = {1};
```

- BUT, not automatic bounds checking!

  - Cells “before” and “after” and start/end of the array can be accessed!
  - It is the programmer’s responsibility to ensure that a program does not access outside an array’s limits.

Arrays

- Multi-dimensional arrays
  - Again, similar to Java

```
int a[DIM1][DIM2];
```

  - Inline initialisation is trickier

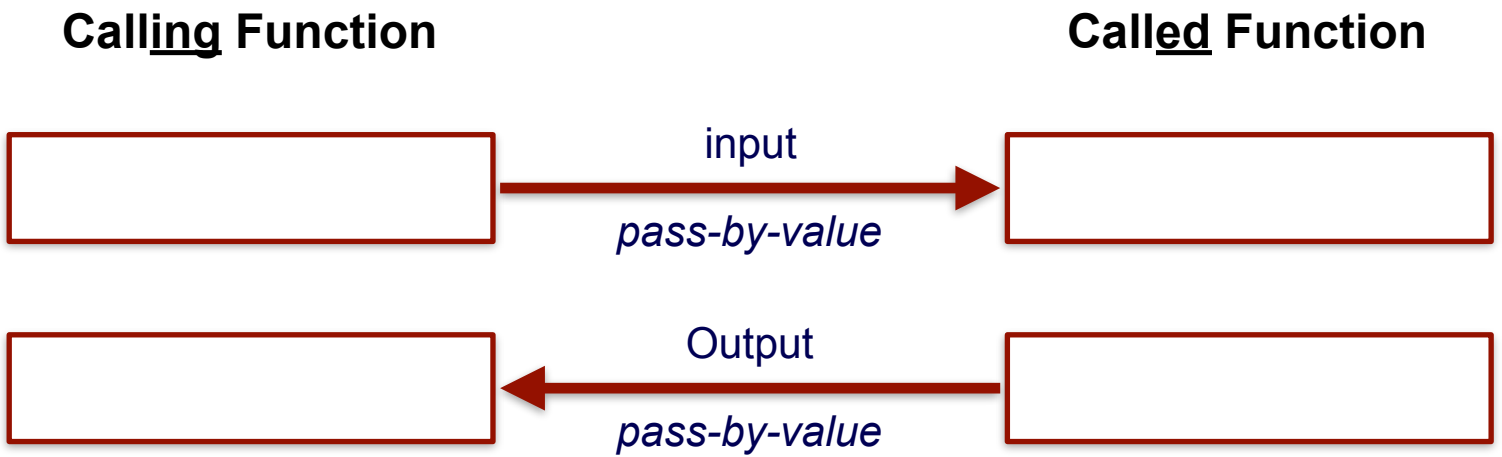
```
int a[DIM1][DIM2] = { {1,2,3}, {4,5,6}, …};
```

Functions

- Similar in concept to Java Methods
- Functions are not associated with a class, and sit in the “Global" scope
- Usage:
  - Functions must be *declared* before they can be used (called)
    - A function declaration is also called a ***function prototype***
  - Functions must only be *defined* once
    - This can be after it is called
    - It doesn’t not even have to be in the same cpp file! (more on this later)
- Pass-by-value
  - Pass-by-reference later (next week)
  - Array passing (next week, more detail)

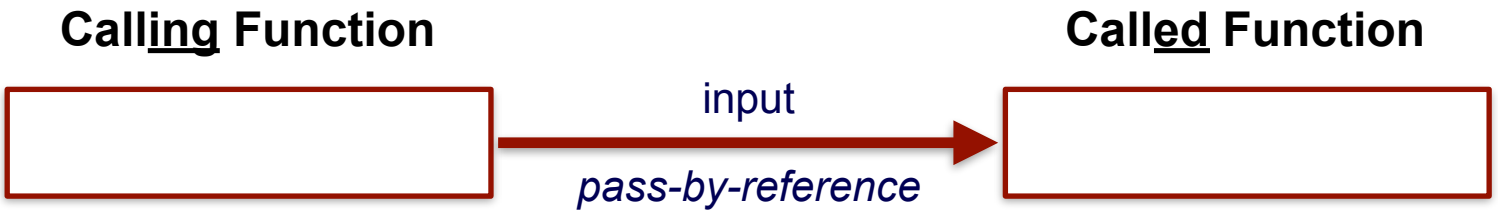
Functions

- Function calls operate through an approach called ***pass-by-value***
  - The value of the parameter is ***copied*** when it is given to the function
  - Changing the parameter within a function does not modify the value from the calling entity
  - This is similar to primitive types in Java



Functions

- Arrays are different** (sort-of)
  - Arrays (as parameters) operate through ***pass-by-reference***
  - The actual array is passed.
    - Changing a value in the array within the called function modifies the value from the calling function



** As we will see next week:

- Under-the-hood an array is implemented using a *pointer*
- The pointer is copied (pass-by-value)
- The high-level effect to the programmer is pass-by-reference

Namespaces

- Define a new scope
  - Similar to packages in Java
  - Useful for organising large codebases

```
namespace myNamespace { … }
```

- Function, Class, Variables, etc labels can be enclosed within a namespace
  - The namespace must be referenced to access the entity, using ::

```
<namespace>::<label>
```

  - Namespaces can be nested

```
namespace namespace1 { namespace namespace2 { … } }
<namespace1>::<namespace2>::<label>
```

Namespaces

- Namespace entities can be exported

```
using std::cout
```

- Everything in a namespace can be exported

```
using namespace std
```

  - This is banned within this course
- The `std` namespace
  - Most STL entities we will use exist within the std namespace

Global Variables

- So far, all variables have been *defined* within the *scope* of a function.
  - The variable only exists within that function
  - The variable cannot be referenced from elsewhere
- A variable defined *outside* of any function is global
  - Can be used within any function, so long as the definition appears before the variable is used
  - These are incredibly bad design and style

Global variables are banned in this course

C++ Style Guide

RMIT UNIVERSITY

RMIT
UNIVERSITY