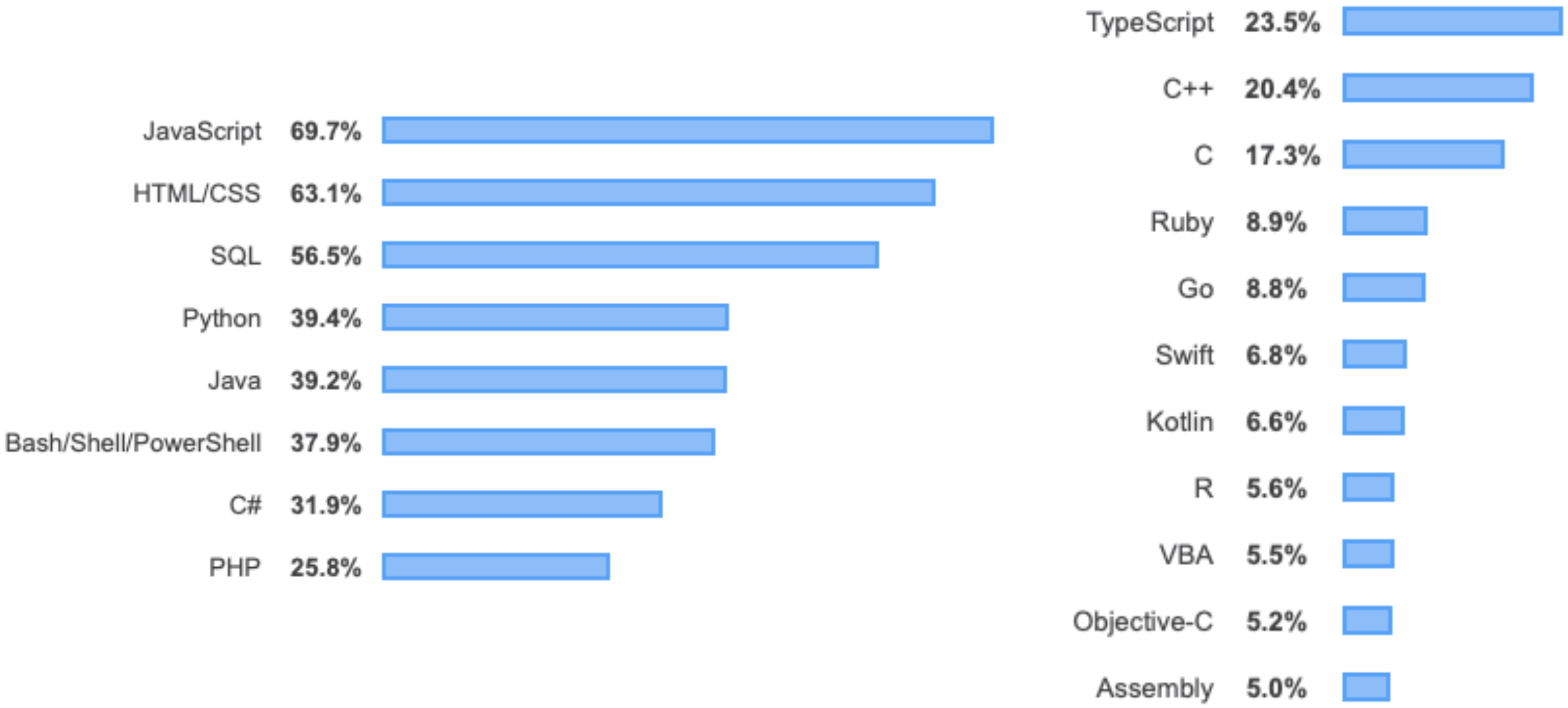
Why C++?

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 2019 Stack Overflow Survey (Professional Developers)



Why C++?

Pereira, R. et al. (2017) ‘Energy efficiency across programming languages: how do energy, time, and memory relate’. doi:[10.1145/3136014.3136031](10.1145/3136014.3136031)

	Energy
(c) C	1.00
(c) Rust	1.03
(c) C++	1.34
(c) Ada	1.70
(v) Java	1.98
(c) Pascal	2.14
(c) Chapel	2.18
(v) Lisp	2.27
(c) Ocaml	2.40
(c) Fortran	2.52
(c) Swift	2.79
(c) Haskell	3.10
(v) C#	3.14
(c) Go	3.23
(i) Dart	3.83
(v) F#	4.13
(i) JavaScript	4.45
(v) Racket	7.91
(i) TypeScript	21.50
(i) Hack	24.02
(i) PHP	29.30
(v) Erlang	42.23
(i) Lua	45.98
(i) Jruby	46.54
(i) Ruby	69.91
(i) Python	75.88
(i) Perl	79.58

	Time
(c) C	1.00
(c) Rust	1.04
(c) C++	1.56
(c) Ada	1.85
(v) Java	1.89
(c) Chapel	2.14
(c) Go	2.83
(c) Pascal	3.02
(c) Ocaml	3.09
(v) C#	3.14
(v) Lisp	3.40
(c) Haskell	3.55
(c) Swift	4.20
(c) Fortran	4.20
(v) F#	6.30
(i) JavaScript	6.52
(i) Dart	6.67
(v) Racket	11.27
(i) Hack	26.99
(i) PHP	27.64
(v) Erlang	36.71
(i) Jruby	43.44
(i) TypeScript	46.20
(i) Ruby	59.34
(i) Perl	65.79
(i) Python	71.90
(i) Lua	82.91

	Mb
(c) Pascal	1.00
(c) Go	1.05
(c) C	1.17
(c) Fortran	1.24
(c) C++	1.34
(c) Ada	1.47
(c) Rust	1.54
(v) Lisp	1.92
(c) Haskell	2.45
(i) PHP	2.57
(c) Swift	2.71
(i) Python	2.80
(c) Ocaml	2.82
(v) C#	2.85
(i) Hack	3.34
(v) Racket	3.52
(i) Ruby	3.97
(c) Chapel	4.00
(v) F#	4.25
(i) JavaScript	4.59
(i) TypeScript	4.69
(v) Java	6.01
(i) Perl	6.62
(i) Lua	6.72
(v) Erlang	7.20
(i) Dart	8.64
(i) Jruby	19.84

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,
  - C++17 (2017), We will be using this
  - C++20 (2020), latest standard, we won't use this

Development Cycle

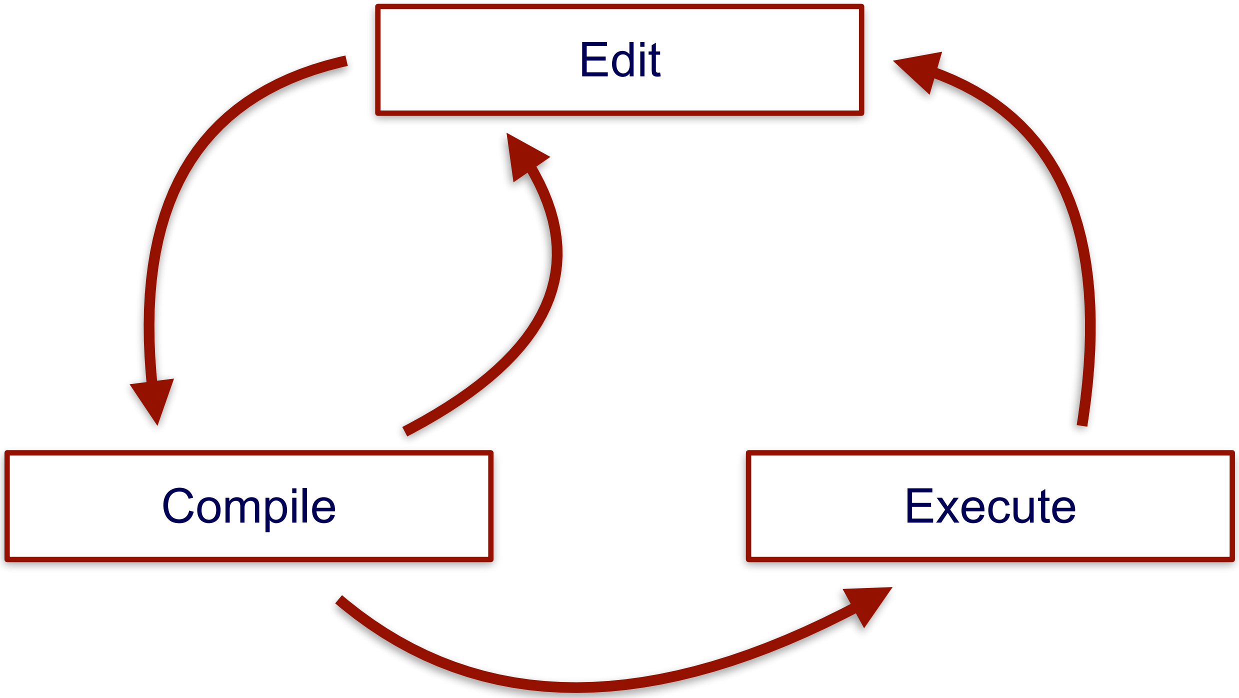
RMIT UNIVERSITY

C++ Program Structure

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

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

#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;

   bar(i, f, c);
   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;
}
```

Development Cycle



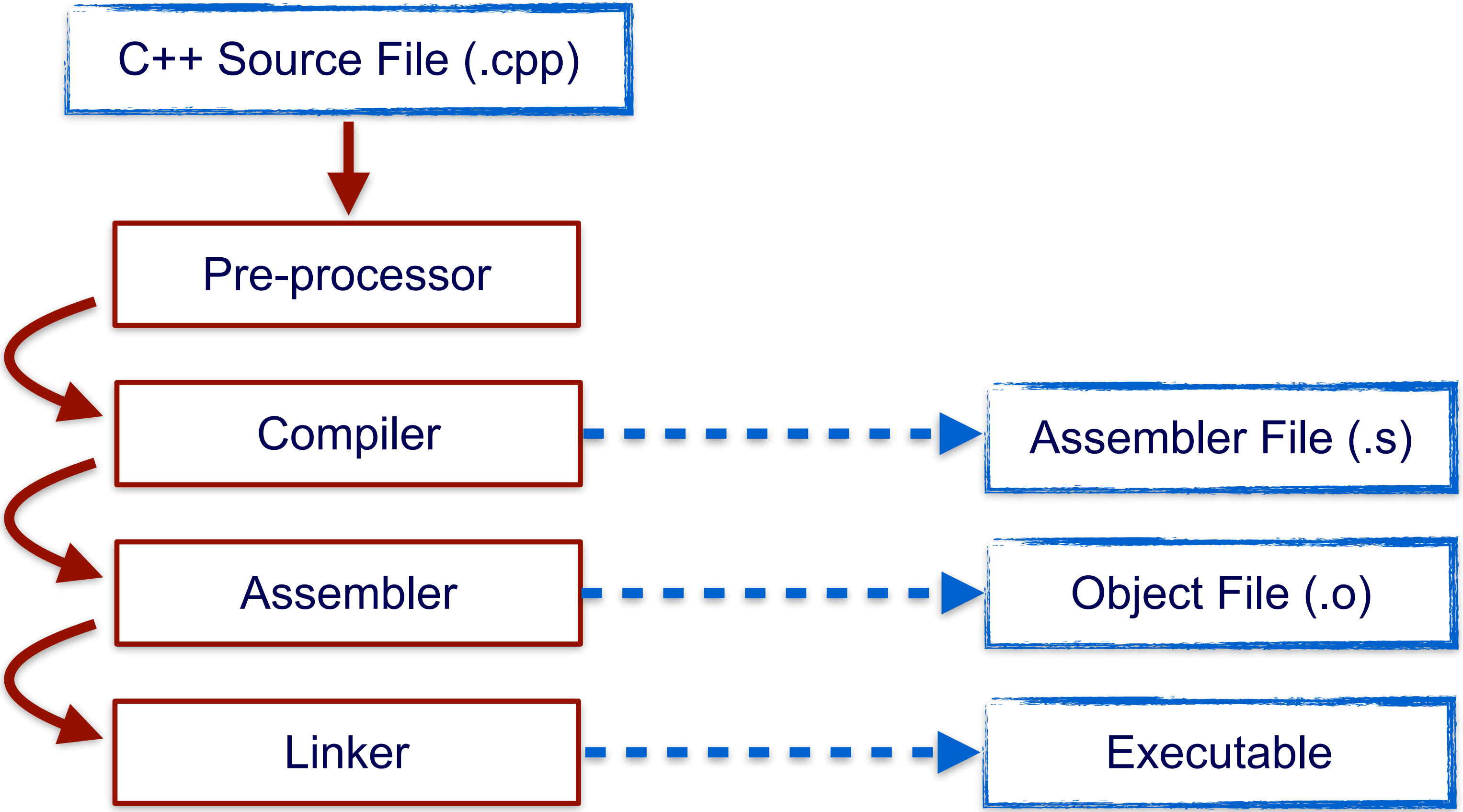
Compiling and Running C++ Programs

- 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 std=c++14 –O -o <executable> <codefile.cpp> ...
```

- Compiler options
  - `-Wall` enable all error checking
  - `-Werror` convert warnings into errors, stopping compilation
  - `-std=c++14` enable all c++14 features
  - `-O` turn on optimiser
  - `-o <filename>` output filename of executable

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

Multi-File Programs

RMIT UNIVERSITY

Header Files

- Header files contain definitions, such as
  - Function prototypes
  - Class descriptions
  - Typedef’s
  - Common #define’s
- Header files do not contain any code implementation
- The purpose of the header files is to describe to source files all of the information that is required in order to implement some part of the code

Source Files

- Code files have source code definitions / implementations
  - In a single combined program, every function or class method may only have a single implementation
- To successfully provide implementations, the code must be given all necessary declarations to fully describe all types and classes that are being used
  - Definitions in header files are included in the code file

```
#include “header.h”
```

- For local header files, use double-quotes
- Use *relative-path* to the header file from the code file

Multiple Includes

- What happens if a header file in included multiple times?
  - Can create naming errors re-declaration errors.
- Two solutions
  1. Be careful about what files are included, but this quickly becomes infeasible with long include chains
  2. Use pre-processor commands

#ifdef / #ifndef / #endif

- The pre-processor can be used to see if a name as been #define'd
  - #ifdef - check if a definition exists
  - #ifndef - check if a definition does not exist
  - #endif - Close a #if check
- Any code between a #if check will only be included if the check passes
  - If the check does not pass, the code is essentially ignored
- Typical pattern for a header file:

```
#ifndef TERM_FOR_HEADER_FILE
#define TERM_FOR_HEADER_FILE

/* Header file implementation */

#endif // TERM_FOR_HEADER_FILE
```

Standard I/O

RMIT UNIVERSITY

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
- Example

```
int x
std::cin >> x
double y
std::cin >> y
```

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)

Types

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

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
```

Functions

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?

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

Declaration (Prototype)

```
int foo(int x, double y);
```

Return type

Name

Parameters

Definition

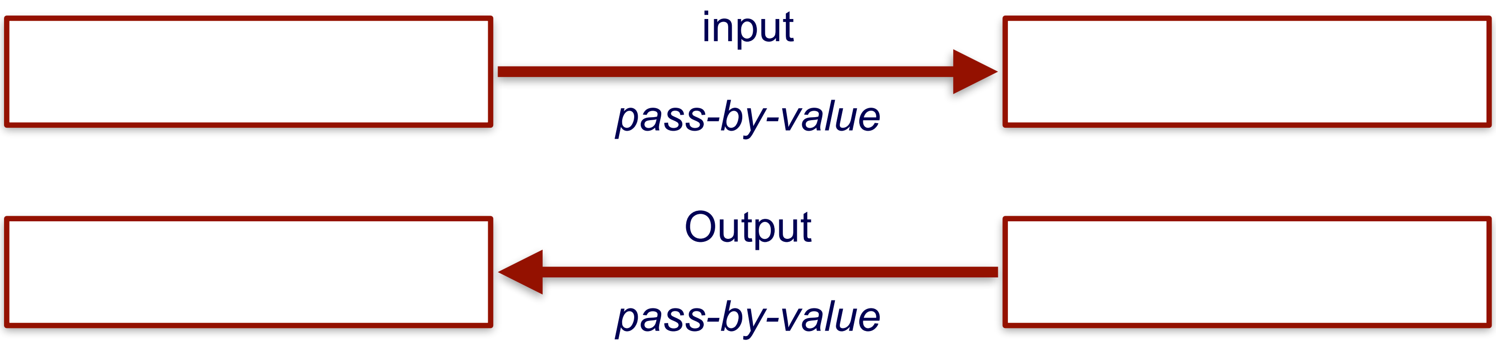
```
int foo(int x, double y) {
    // Do stuff
    return x;
}
```

Return Value

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

Calling Function | **Called Function**



Arrays

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

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

  - Can be initialised when declared

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

- BUT, not automatic bounds checking!

a[-1]	a[0]	a[1]	a[2]	a[3]	a[4]

  - 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

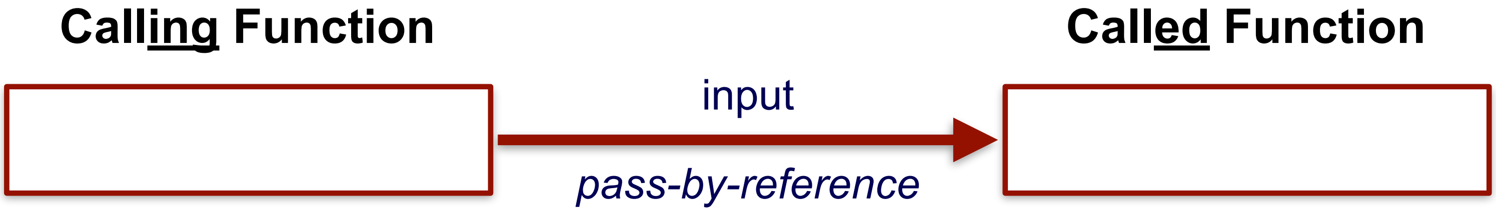
```
int a[DIM1][DIM2];
```

  - Inline initialisation is trickier

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

Functions

- 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 & global Variables

RMIT UNIVERSITY

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

#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

Multi-File Programs

File Types

- C/C++ has two types of files:
  - Header files (.h / .hpp)
  - Source files (c.pp)
-

Header Files

- Header files contain definitions, such as
  - Function prototypes
  - Class descriptions
  - Typedef's
  - Common #define's
- Header files do not contain any code implementation
- The purpose of the header files is to describe to source files all of the information that is required in order to implement some part of the code

Source Files

- Code files have source code definitions / implementations
  - In a single combined program, every function or class method may only have a single implementation
- To successfully provide implementations, the code must be given all necessary declarations to fully describe all types and classes that are being used
  - Definitions in header files are included in the code file

```
#include “header.h”
```

  - For local header files, use double-quotes
  - Use *relative-path* to the header file from the code file

Multiple Includes

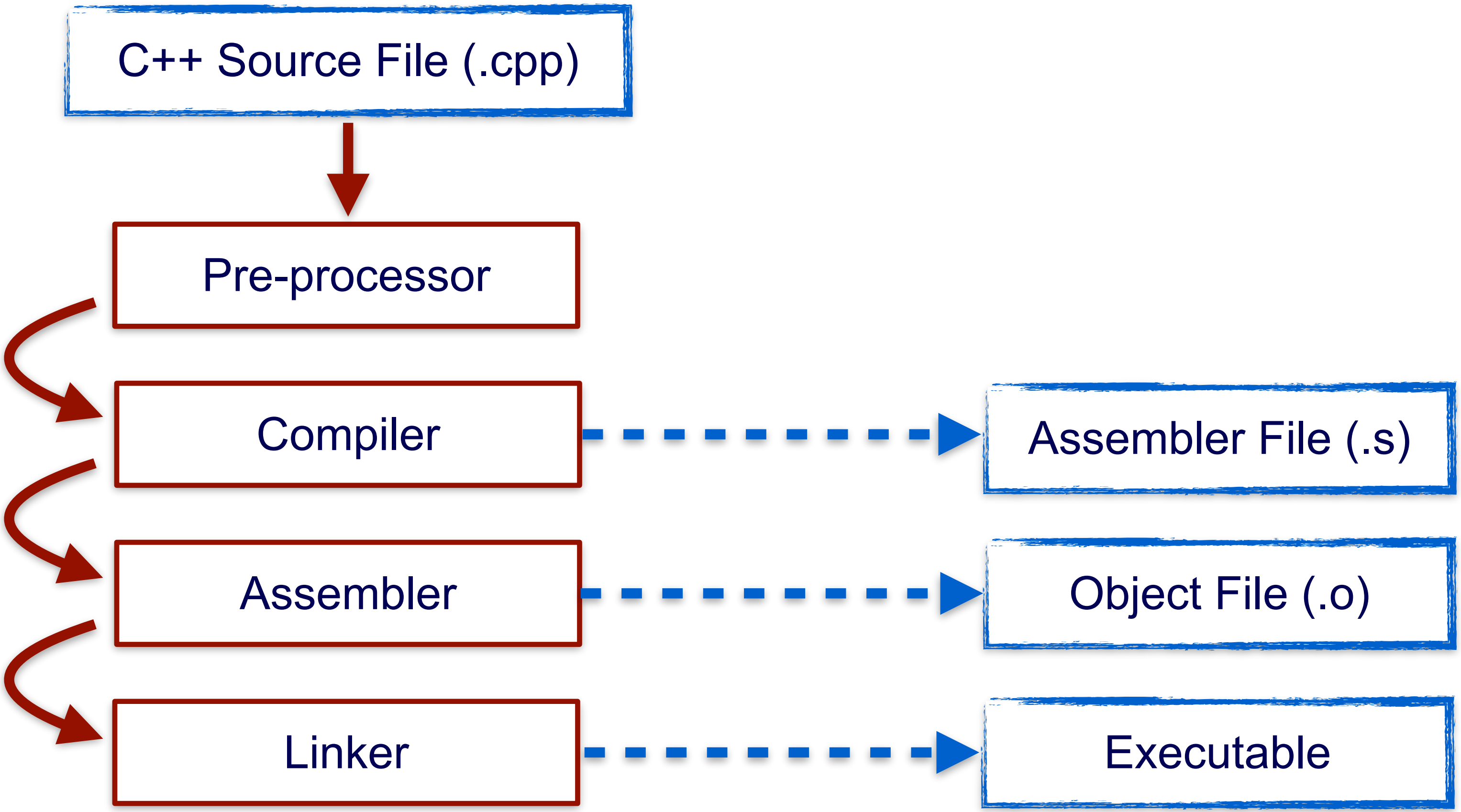
- What happens if a header file in included multiple times?
  - Can create naming errors re-declaration errors.
- Two solutions
  1. Be careful about what files are included, but this quickly becomes infeasible with long include chains
  2. Use pre-processor commands

#ifdef / #ifndef / #endif

- The pre-processor can be used to see if a name as been `#define`'d
  - #ifdef - check if a definition exists
  - #ifndef - check if a definition does not exist
  - #endif - Close a #if check
- Any code between a #if check will only be included if the check passes
  - If the check does not pass, the code is essentially ignored
- Typical pattern for a header file:

```
#ifndef TERM_FOR_HEADER_FILE
#define TERM_FOR_HEADER_FILE

/* Header file implementation */

#endif // TERM_FOR_HEADER_FILE
```

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

Classes

RMIT UNIVERSITY

Classes

- C++ Classes are similar to Java Classes
- Divide creating a class into ***declaration*** and ***definition***
  - A declaration is like a Java interface
    - Describes the components of the class
  - The definition is like a Java class file,
    - Provides the implementation of the class methods.

Classes Declaration

- C++ Class ***Declaration***

```
class Example {
public:
    Example(int value);

    void publicMethod();

protected:

    int protectedVariable;
    int protectedMethod(double param);

private:

    double privateArray[LENGTH];
    void privateMethod(int* ptr, double& ref);

};
```

Class Name

Constructor

Method

Field

Fields can be arrays or other classes

Don’t forget ‘;’

Public, Protected, & Private Scopes

Class Method Definitions

- C++ Class method definitions provide the implementation of each method
  - Definitions provided individually
  - Scope is not relevant to the definition
  - The Class name creates a namespace!

Return Type

Class Name

Method Name

Parameters

```
int Example::protectedMethod(double param) {
    return 0;
}
```

Namespace separator

Class Initialisation

- Objects (variables of a given class) can be created like any other variable
  - Does not need to be “new’ed”
- The constructor is called when defining the variable
  - Use bracket notation to provide the parameters to a class object

No return type

Constructor

Parameter

```
Example::Example(int value) {
   protectedVariable = value;
}
```

Object definition

```
Example ex(10);
```

Class is a type

variable/object name

Constructor Parameters

Access Class Members

- Class members (variables and methods) are accessed using dot '.' Syntax

```
Example ex(10);
ex.publicMethod();
```

- For pointers to object, arrow syntax '->' is a shortcut for dereferencing

```
Example* ptrEx = &ex;
(*ex).publicMethod();
ex->publicMethod();
```

- Class members can only be accessed from the correct scope
  - Public members are always accessible
  - Private members are only accessible only from within the class
  - Protected members can be accessed from this class and all children

Class & Functions

- Pass classes to functions either by
  - Pointer
  - Reference
- Passing the class directly:
  - Is possible
  - BUT!
    - Requires a special constructor (called a copy constructor)
    - We will cover this in future week(s)

Make

- `make` is a tool for automated compilation that dates back to 1976
- Make is a simple tool to for automated build
  - It is language independent, though typically used for C/C++
  - Make specified automated build through a series of rules.
  - A rule contains:
    - A target
    - Dependencies
    - A command

Make

- Make rules are *always* placed in a file called ‘`Makefile`’
  - A rule of a makefile are executed using the “make” utility/command

```
make <target>
```

  - If no target is given, the “`default`” target is run

Makefile

```
.default: all
```

Default Rule

Target Name

```
all: unit_tests
```

Dependencies

```
clean:
	rm -f unit_tests *.o
```

Rule

```
unit_tests: Particle.o ParticleList.o ParticleFilter.o
unit_tests.o
	g++ -Wall -Werror -std=c++14 -O -o $@ $^
```

Compilation Command

```
%.o: %.cpp
	g++ -Wall -Werror -std=c++14 -O -c $^
```

Pattern Rule

Beyond Make

- Make is a simple but effective tool
  - However, for complex projects it quickly becomes annoying to manually write makefiles
  - Additional automated build tools provide layers of abstract to further automate different aspects of building programs
  - Interestingly, many of these tools eventually generate and use makefiles!
- Common tools include
  - automake
  - CMake

IDEs

- IDEs (Integrated Development Environments) often provide automated build processes
  - Under-the-hood many use existing tools, such as make
  - IDE builds are only as good as their configuration
- If you are only working with simple programs, the “default” of an IDE will be sufficient, however:
  - For complex programs, use of external libraries, multiple sub-packages, etc:
  - The IDE build process will need to be configured
  - This requires a knowledge of what the IDE is actually doing

COSC2804 automated builds

- For this course we will use only use `make`
  - The focus of this course is understanding what is happening, not trying to hide stuff from you
  - It is good to practice these more “low-level” primitive skills, because you will need them, even when using IDEs!

Automated Build

Make

- `make` is a tool for automated compilation that dates back to 1976
- Make is a simple tool to for automated build
  - It is language independent, though typically used for C/C++
  - Make specified automated build through a series of rules.
  - A rule contains:
    - A target
    - Dependencies
    - A command

Make

- Make rules are *always* placed in a file called ‘`Makefile`’
  - A rule of a makefile are executed using the “make” utility/command

```
make <target>
```

  - If no target is given, the “`default`” target is run

Makefile

```
.default: all

all: unit_tests

clean:
	rm -f unit_tests *.o

unit_tests: Particle.o ParticleList.o ParticleFilter.o
unit_tests.o
	g++ -Wall -Werror -std=c++14 -O -o $@ $^

%.o: %.cpp
	g++ -Wall -Werror -std=c++14 -O -c $^
```

Default Rule

Target Name

Dependencies

Rule

Compilation Command

Pattern Rule

Beyond Make

- Make is a simple but effective tool
  - However, for complex projects it quickly becomes annoying to manually write makefiles
  - Additional automated build tools provide layers of abstract to further automate different aspects of building programs
  - Interestingly, many of these tools eventually generate and use makefiles!
- Common tools include
  - automake
  - CMake

IDEs

- IDEs (Integrated Development Environments) often provide automated build processes
  - Under-the-hood many use existing tools, such as make
  - IDE builds are only as good as their configuration
- If you are only working with simple programs, the “default” of an IDE will be sufficient, however:
  - For complex programs, use of external libraries, multiple sub-packages, etc:
  - The IDE build process will need to be configured
  - This requires a knowledge of what the IDE is actually doing

COSC2804 automated builds

- For this course we will use only use `make`
  - The focus of this course is understanding what is happening, not trying to hide stuff from you
  - It is good to practice these more “low-level” primitive skills, because you will need them, even when using IDEs!

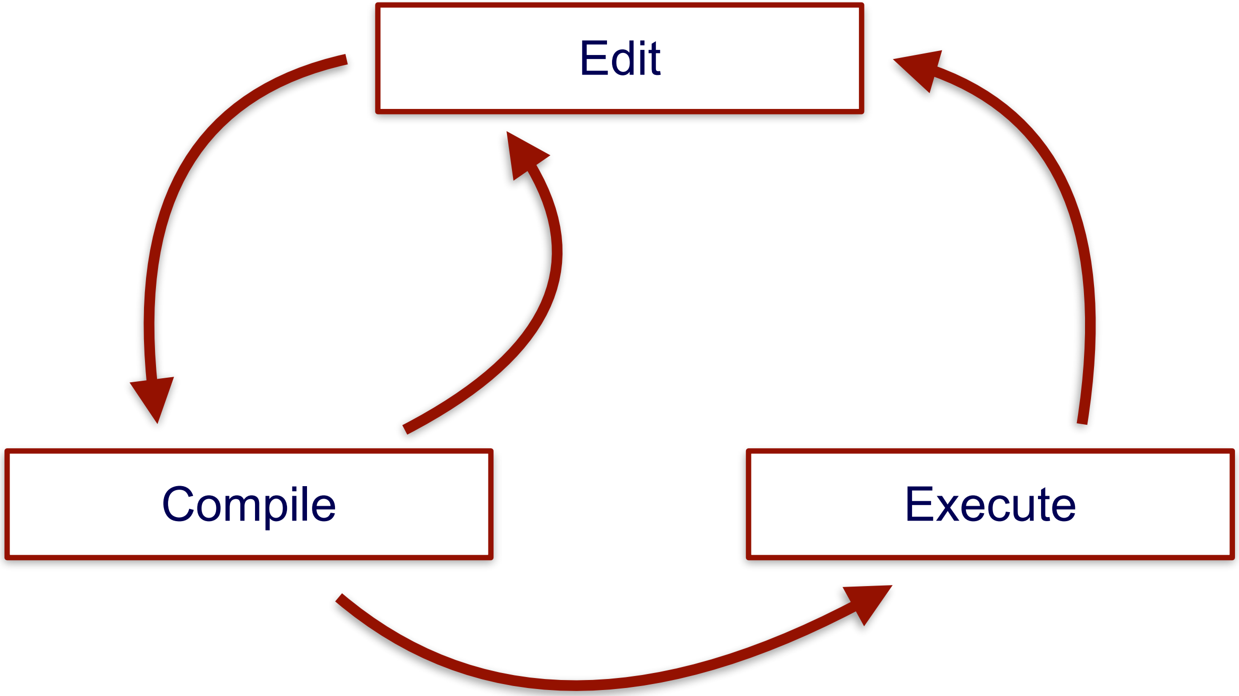
Partial Compilation

Executable Generation

- A full C++ program that can be run is termed ***executable***
- The full C++ build process* takes written code and converts it into an executable
- So far, we take all code and build it together, but during the development cycle this is inefficient for developers.
  - Often, only a small portion of the code is modified
  - Full rebuilds can be slow and are unnecessary

- ** Most programming languages follow a similar process to varying degrees. Thus, this discussion is not specific to C++!*

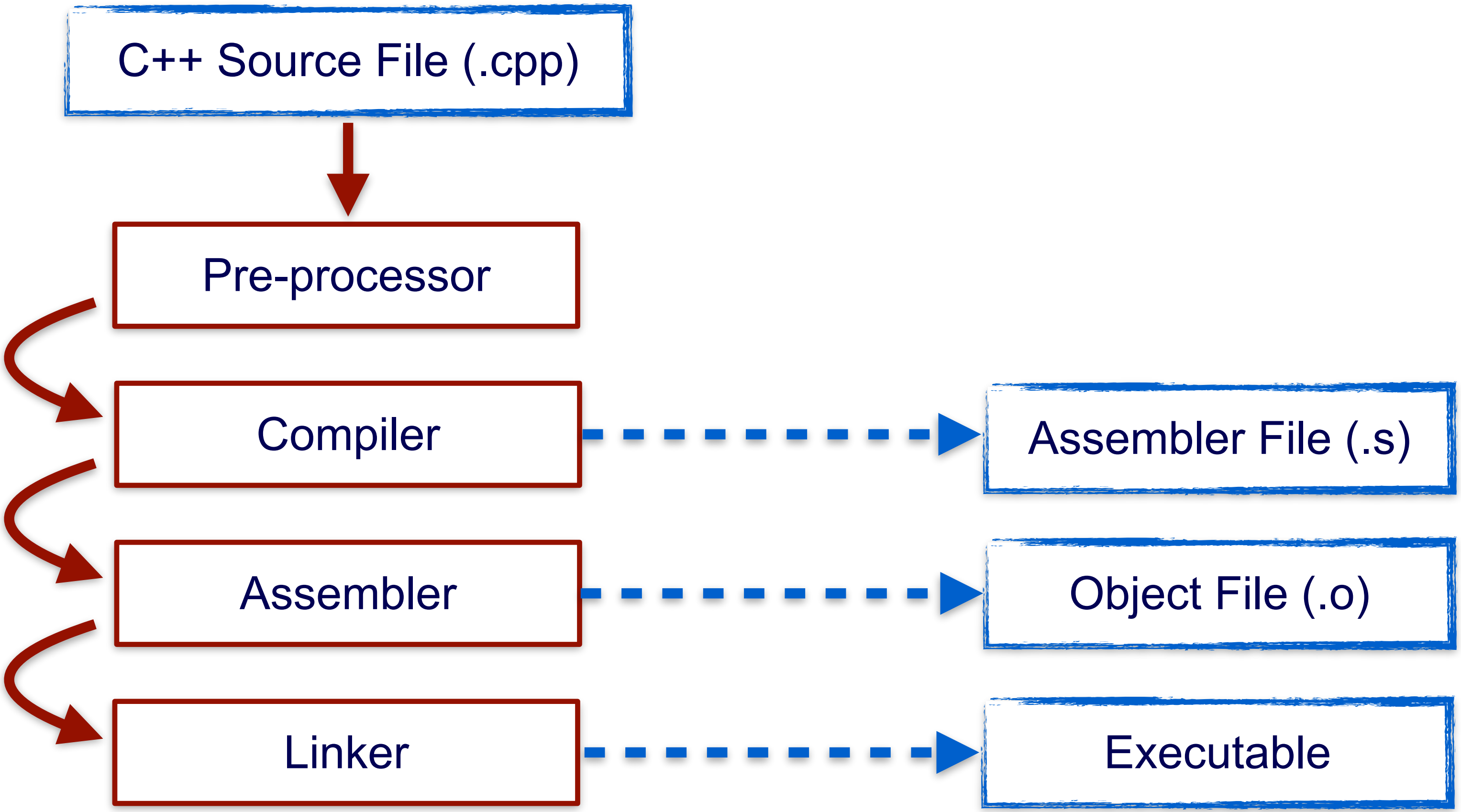
Development Cycle



Partial Compilation

- Partial compilation:
  - Compiles a subset of a codebase
  - Compiles the subset to as close to an executable format as possible without the "missing” code
  - Does not need to be rebuilt unless the originating subset of the code is modified
- Full compilation takes the partially compiled code and produces an executable

C/C++ Compilation Process



Preprocessor

- The ***pre-processor*** first runs and evaluates all '#' statements
  - `#include` - import header file
  - `#define` - define a new constant
  - `#ifdef` - check if a definition exists
  - `#ifndef` - check if a definition does not exist
  - `#endif` - Close a #if check
- This generates a temporary C++ code file that is given to the compiler
- The pre-processor is compiler dependent
  - Not all compilers support all '#' statements
  - There are very few '#' statements included in the C++14 language

#ifdef / #ifndef / #endif

- The pre-processor can be used to see if a name as been #define'd
  - #ifdef - check if a definition exists
  - #ifndef - check if a definition does not exist
  - #endif - Close a #if check
- Any code between a #if check will only be included if the check passes
  - If the check does not pass, the code is essentially ignored
- Typical pattern for a header file:

```
#ifndef TERM_FOR_HEADER_FILE
#define TERM_FOR_HEADER_FILE

/* Header file implementation */

#endif // TERM_FOR_HEADER_FILE
```

C++ Partial Compilation (Object Files)

- An individual code file (cpp file) can be partially compiled into an ***object file***
- An object file is
  - Machine code (1's and 0's)
  - Is missing "links" to variables, functions, and methods that the code file calls and depends on for it to actually run
  - Contains a ***symbol table*** for all of these "missing links"
  - The symbols denote what the code depends upon

C++ Partial Compilation (Object Files)

- Object files made with g++, with the -c flag
  - The -o flag is optional

```
g++ ... –c <file1.cpp>
```

- Object files are then compiled together just like compiling multiple code files

```
g++ ... –o <executable> <file1.o> <file2.o> ...
```

  - The -o flag is required

C++ Partial Compilation (Linker)

- The ***linker*** connects all of the “missing links” between the object files
  - This is done by matching up the symbols in each of the symbol tables of the object files
  - The symbols must exactly match for linking to occur
- After linking, objects files become a full executable

Libraries

- Libraries are code that have been developed by other and is shared
- So the question is:
  - Should code in libraries be compiled anew by every developer who uses the library?
  - Should libraries be pre-compiled?
- If a library is rebuilt:
  - Requires more time for the developer using the library
  - Easier to share
  - We’ve seen this with:
    - iostream
    - std::string
    - cstdio

Pre-compiled Libraries

- If a library is pre-compiled:
  - Must be pre-compiled for every different CPU architecture
  - Saves time for developers rebuilding the library, which can be significant for large libraries
  - The library developers have more control over compilation, which may be especially significant for optimisation of the code
- In C++, libraries must link against the pre-compiled library
  - Uses the “linker” flag

```
-l<library name>
```

  - Linker flag is supplied when building the full executable or object file
- A common C++ library is Math (cmath)

```
-lm
```

Executable Generation Pitfalls

- Partial compilation can lead to runtime errors
  - The most common mistake is that code is not rebuilt!
  - In large projects, or where libraries are used, conflicts may occur is different subsets of the codebase are compiled against different versions of the project, or against different versions of libraries.
- If in doubt, re-compile from scratch

- *Java 1.4 (which introduced generics) actually had a major problem that could occur through partial compilation which let to run-time errors!*

C++ Style Guide

RMIT UNIVERSITY