C++ Programming Bootcamp 2

COSC2802

TOPIC 5

C++ Program Structure

Program Structure

C/C++ Preprocessor

Function *Declaration*

Main Function

Variable *Definitions and Declarations*

Input/Output

Function *Definitions*

```
#include <iostream>

#define SIZE 10

int func(int x);

int main() {
    int i = 0;
    char c;

    std::cin >> c;
    std::cout << func(i) << " " << c << std::endl;

    return EXIT_SUCCESS;
}

int func(int x) {
    return x * 2;
}
```

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 / #ifndef / #endif statements: Controls compilation visibility (cover later)

#Include

- #includes are used to import code segments from other sources
- There are two main versions:

#include <package_name>: looks for files in the STL

#include "package_name": looks for files in the programmer's workspace

This will be covered in more detail when we look at multi-file programs.

#Define

- #defines are used to define constant values

#define DEFINE_NAME value

- Convention dictates using uppercase names, so other programmers know what is a mutable variable vs a constant
- They are literal “find-and-replace” parameters, so anything you put after the name of the define will be pasted in place.
  - So be careful of brackets or a ‘;’ at the end, which will paste the ‘;’ in your code, and probably break things.

Namespaces

Something you've seen before: std::cout, is a part of the std namespace.

They define a new scope: like packages in Java.

Useful for organizing large codebases.

namespace myNamespace { … }

Any code within the brackets become part of the namespace and can be accessed by using: *namespace:: Example();*.

Usually used to organize functions, classes, and variables.

They can also be nested!

namespace Namespace1 { namespace Namespace2 { … } }

Namespace1::Namespace2::Example();

Namespaces

You can also import namespaces so you don't have to keep referring to it.

using std::cout

Using std::endl;

Lets you call *cout* without the *std::.*

You can also import an entire namespace:

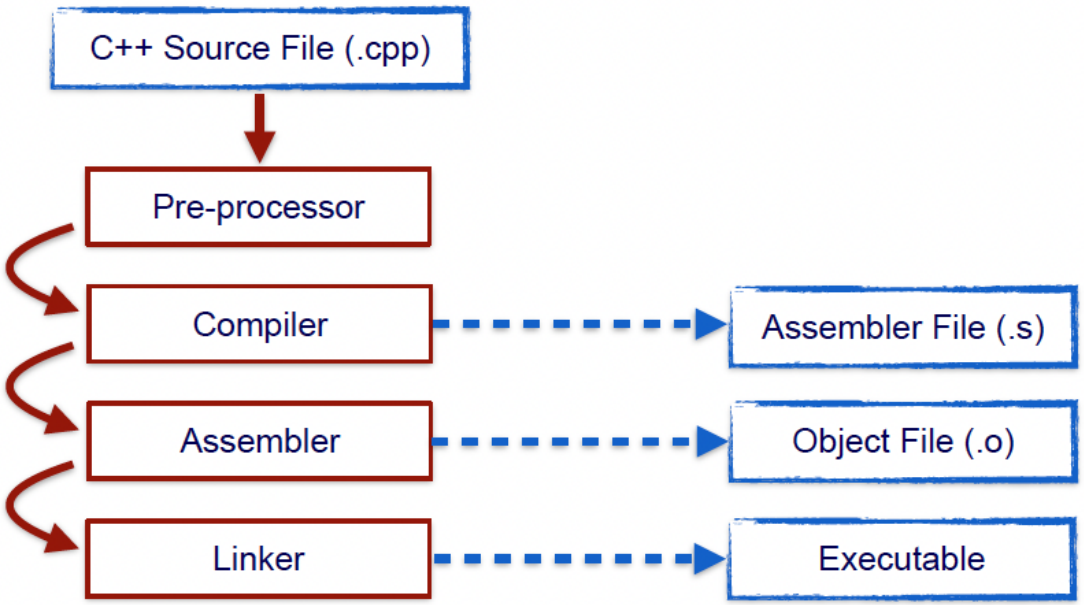
using namespace std

But this is bad practice and is banned in basically every course at RMIT.

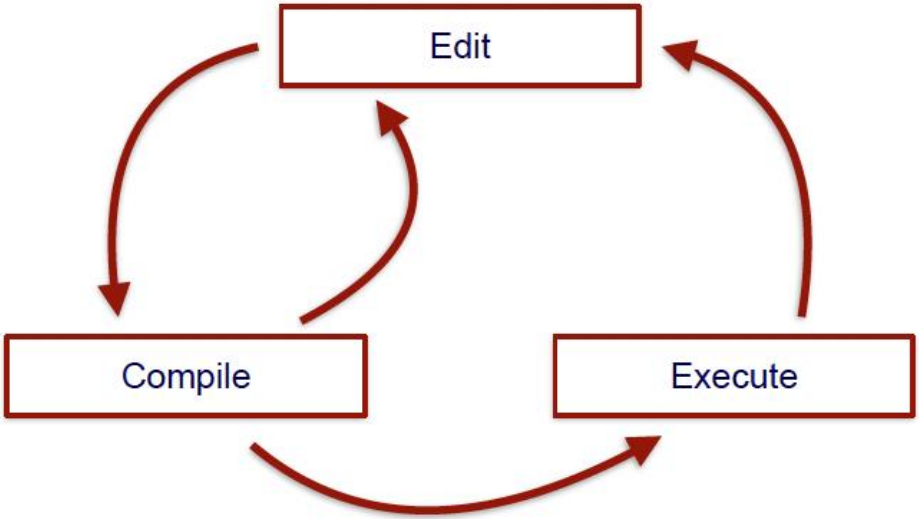
The std namespace is massive, and you will end up with naming conflicts. Is it always going to fail? No, but you may encounter very strange behaviour.

Compiling & Running C++ Programs

C/C++ Compilation Process



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++17 -O -o <executable> <codefile.cpp> ...
```

- Compiler options
  - `-Wall` enable all error checking
  - `-Werror` convert warnings into errors, stopping compilation
  - `-std=c++17` enable all `-std=c++17` features
  - `-O` turn on optimiser
  - `-o <filename>` output filename of executable

What occurs in zyBooks?

Compiling and Running | zybooks

Compilation

	Flag	Description
☑	-Wall	Turns on most warnings
☑	-Werror	Treats all warnings as errors
☐	-Wextra	Enables some extra warning flags that are not enabled by -Wall
☐	-Wuninitialized	Warns about the use of uninitialized variables
☐	-pedantic-errors	Issue errors when code doesnt conform to ANSI standard
☐	-Wconversion	Warn if implicit type conversions can alter values (and potentially cause overflows, underflows, and loss of precision)
☐	-fopenmp	Enables parallel computing

Additional flags

Provide any additional flags you'd like

Compile command

```
g++ main.cpp task.cpp -Wall -Werror -o a.out
```

Used to compile student's code

Separate files

File Types: C/C++ has two types of files

As discussed previously: *#include* can be used to import code from other files.

Header files (.h)

- 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 (.cpp)

- 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

Multiple Includes

As we know, functions can only be defined once per program. So, what happens if we include the same file twice? (From multiple sources)

It breaks!

But we can use: **Header File Guards** to ensure files are only included once

- *#ifndef filename_h*
- *#endif*

```
#ifndef FILENAME_H
#define FILENAME_H

// Header file contents

#endif
```

- instructs preprocessor to process code between them only *if filename_h not defined*
- Good practice is to guard every header file

Compiling and Running | Separate Files

g++ -Wall -Werror -std=c++17 -O –o a.out main.cpp file2.cpp

main.cpp

```cpp
#include <iostream>
#include "file2.h"

int main() {
    std::cout << "Hello from main!" << std::endl;
    say_hello();
    return 0;
}
```

file2.h

```cpp
#ifndef FILE2
#define FILE2

#include <iostream>

void say_hello();

#endif
```

file2.cpp

```cpp
#include "file2.h"

void say_hello() {
    std::cout << "Hello from File 2!" << std::endl;
}
```

Compiling and Running | Separate Files

Example: zybooks “LAB: Exact change: functions, separate files”

- 3 separate files:
  main.cpp, task.h and task.cpp

- Compiled using:
  g++ -Wall -Werror -std=c++17 -O -o testfile main.cpp task.cpp

Separate Files - Guidelines

.h files contain your declarations

- for much the same reasons we declare functions before they are defined
- Should contain as little code as possible
- Typically: we have separate .h/.cpp files for new classes.

.cpp files contain your implementations

- Never include .cpp files, only include .h files, .cpp files are included via the compilation step.

You should use include guards in any and all files you plan on #include-ing.

```
#ifndef FILENAME_H
#define FILENAME_H

// Header file contents

#endif
```