Programming Studio 2 – C++

Week 5 Class 1

Dr Ruwan Tennakoon
Dr Steven Korevaar

RMIT UNIVERSITY

Programming Studio 2 – C++ Stream

Purpose of this course:

Learning to implement high level concepts in low level programming languages.

- Started with assembly / machine code
- Moving into C++

Why C++?

Offers much more premade functionality while allowing advanced users to control the program extensively.

- Much easier to use than assembly
- Compilation process allows for portability of code to different CPU architectures.
- High level of control over computation and memory wise.

Class Structure:

Basically, the same as the previous 4 weeks.

First two classes each week have a brief live coding session at the start, then free working time on extra problems and the assignment.

Final class each week will be assignment checkpoint demonstrations (more information in the assignment specifications)

RMIT UNIVERSITY

Bootcamp 2 Recap

In Bootcamp 2 you learnt the basic theory of C++:

- Program control: if statements and loops,
- Basic data types: ints, chars, etc
- C style arrays
- Strings
- Functions (Operator overloading)
- Classes and Objects (Inheritance and Polymorphism)
- Pointers
- Memory management

In Studio 2 we will focus on more highlevel details of C++ programming:

- Black box testing
- Data structures (memory management)
- Program efficiency
- Smart pointers
- Recursion
- Advanced topics
  - Inheritance
  - Polymorphism
  - Generics/Template Classes

RMIT UNIVERSITY

Programming Studio 2 – C++ Stream

Content Overview:

Week 5: Basic C++ revision

- 5.1 – C++ programming revision
- 5.2 – Testing programs
- 5.3 – Checkpoint 1

Week 6: Data structures

- 6.1 – Data structures (memory management)
- 6.2 – Program efficiency
- 6.3 – Checkpoint 2

Week 7: Pre-built functionality

- 7.1 – Smart pointers
- 7.2 – STL data structures
- 7.3 – Checkpoint 3

Week 8: Advanced topics

- 8.1 + 8.2 – Advanced topics
- 8.3 – Checkpoint 4

RMIT UNIVERSITY

Assignment 3

Assignment!

MINECRAFT!

MAZES!

BUILDING, SOLVING, COOL FUN

Both group work and individual components

Weekly Checkpoints

Demonstrate progress, and ensure teams are working well together.

Deliverables:

Milestone 1 – Main menu and testing

Milestone 2 – Reading a maze from input and maze solving

Milestone 3 – Generating mazes and cleaning up terrain

Milestone 4 – Enhancements

- 4.1: Create a Maze Without Flattening the Terrain
- 4.2: Find the Shortest Path to Exit
- 4.3: Validate the user input maze

Video demonstration

RMIT UNIVERSITY

Assignment 3 - Group component (30 marks):

Base Implementation & Testing

Milestone 1 – Main menu and testing
Milestone 2 – Reading a maze from input and maze solving
Milestone 3 – Generating mazes and cleaning up terrain
Video demonstration

Groups of 3: must be registered on Canvas by Monday of Week 6.

All students who are not registered to a group on Monday Week 6 will be allocated a group randomly within their assigned workshop.

Late group registrations will not be accepted!

Checkpoints

Weekly check in with the team to make sure everything is progressing

Not-marked! But important for assessing team dynamics and ensuring all members are contributing

The assignment will run on GitHub classrooms.

Your submission will be your final repository.

Students must assign themselves the tasks they will complete for the assignment during the checkpoints. Your contributions will be assessed via GitHub commits and during the checkpoints.

RMIT UNIVERSITY

Assignment 3 – Individual component (15 marks):

Each student will choose 1 "enhancement" to do each.

Milestone 4 – Enhancements (1 enhancement each)
4.1: Create a Maze Without Flattening the Terrain
4.2: Find the Shortest Path to Exit
4.3: Validate the user input maze

Students should create their own fork of their group's repository, so as not to share code accidentally.

RMIT UNIVERSITY

Assignment 3 – Demonstration

RMIT UNIVERSITY

Writing Complex Programs

In the real-world, the programs you need to write are often significantly larger and more complex than what you've been working on in Bootcamp 2.

This course focuses on developing good programming practises and code management to allow you to work in complex environments.

- How do we know our programs are correct? Testing
- How do we go about writing good code? Programming Practises
- How do we know our programs are efficient? Data structures and code analysis

Today we'll talk about compiling multi-file programs, the first step to working on large projects.

RMIT UNIVERSITY

Writing Complex Programs

Large programs are often broken up into multiple files: why??

What type of files are there?

Header Files (*.h)

- Function prototypes
- Class descriptions
- Typedef's
- Common #define's

Usually, Header files do not contain any code implementation

Source Files (*.cpp)

- Contains the actual code/implementation

We can include code from other files with

`#include "filename.h"`

Be careful of multiple includes! Need to use include guards to prevent this.

#ifdef : include the following if a definition exists

#ifndef : include the following if a definition does not exist

#endif : Close an if(n)def block

RMIT UNIVERSITY

Compilation Basics

Single file programs can be compiled very simply:

```
g++ <filename>.cpp
```

For multi-file programs you must add on all .cpp files to the compilation command

In this class we have additional flags:

1. “-Wall” : Which makes the compiler show all possible warnings.
2. “-Werror”: Which makes the compiler treat all warnings as errors and stops compilation.
3. “-std=c++17”: Uses the C++17 standard
4. “-g”: Debugging flag (sets the compiler to allow the use of debugging tools)
5. “-O”: Tells the compiler to optimise the code with minimal additional compilation time
6. “-o <output name>”: Sets the output of the compilation step to <output name>, usually the name of the executable.
7. “-lmcpp”: Tells the compiler we want to use the lmcpp library.

All this is put into one long command:

```
g++ -Wall -Werror -std=c++17 -g -O –o <output name> <all CPP files in the project> -lmcpp
```

RMIT UNIVERSITY

Automated Compilation with Makefiles!

Structure of a Makefile:

```
<ruleName>: <prerequisites>
      commmand
```

```
.default: all

all: mazeRunner

clean:
    rm -f mazeRunner *.o

mazeRunner: Maze.o MazeBuilder.o ModifiedBlock.o Agent.o assign3.o
    g++ -Wall -Werror -std=c++17 -g -O -o $@ $^ -lmcpp

%.o: %.cpp
    g++ -Wall -Werror -std=c++17 -g -O -c $^
```

Running “make” calls “.default”.

“.default” calls “all”.

“all” calls “mazeRunner”.

“mazeRunner” builds all the object files for each CPP file in the project.

For each CPP file, compile using “%.o”, into an object file: <filename>.o

Once all the object files are created: compile them into the final executable: “mazeRunner”

$@: replace symbol with name of the rule (mazeRunner, %.o, etc)

$^: replace symbol with the requirements for the current tag (Maze.o, MazeBuilder.o, etc)

RMIT UNIVERSITY

What to do today:

Canvas -> Modules -> Week 5 -> Studio Class 1: Writing Complex Programs

1. Go through the class notes pdf (not the solutions)
2. Have a go at the C++ task!
3. Socialise and form a group (make sure to register by Monday Week 6!)

// Build in Minecraft!

Set up connection:

```
mcpp::MinecraftConnection mc;
```

Place a block:

```
mcpp::Coordinate startCoord(build_x, build_y, build_z);
mc.setBlock(startCoord, mcpp::Blocks::BRICKS);
```

RMIT UNIVERSITY