

# C++ Continued

## CS 2ME3/SE 2AA4

Steven Palmer

Department of Computing and Software  
McMaster University

# Outline

- 1 The `main()` Function
- 2 Building C++ Code
  - The Build Process
  - Preprocessing
  - Compilation
  - Linking
  - Makefiles
- 3 Example
- 4 Doxygen with C++
- 5 Unit Testing in C++

# The main() Function

- To build a C++ program, a main function is required
- This is the entry point to your program – this function will be called when your program begins
- The main method in C++ has the following definition:

```
int main(){  
  
    // do stuff here  
  
    return 0;  
}
```

## `main()` Return Type

- If you have experience with Java, you might have noticed that the return type of `main()` in C++ is `int` and not `void`
- In C++, `main` returns an `int` to signify success or failure of the program:
  - a return value of 0 signifies that the program ended normally
  - any other value signifies an error
- The return value can be accessed via the terminal/console to check for failures

# C++ Build Steps

Building from source in C++ is a three step process:

## 1 Preprocessing:

- replaces preprocessor directives (`#define`, `#include`, etc.) with C++ code

## 2 Compilation:

- once preprocessing is done, the resulting pure C++ code can be compiled
- compile translates the C++ code into machine code “object” (.o) files

## 3 Linking:

- object files are linked into binaries or libraries

# The Preprocessor

- The preprocessor is really just a text substitution tool
- Preprocessing must occur first in the build process:  
preprocessor directives are not code and need to be replaced
- Preprocessing is usually not carried out on its own – we can have the compiler do this for us automatically when we do the compilation to object files
- We can choose to run the preprocessor, however, if we would like to see the resulting code before compilation
- Command: `g++ -E -P source.cpp`

# Exercise 1: Run the Preprocessor

## Exercise 1

There is some example code in `examples/compilation/withguards` that we can use to test the preprocessor.

Try running the preprocessor on each source file and see what it does to the code in each file:

```
g++ -E -P A.cpp
```

```
g++ -E -P B.cpp
```

```
g++ -E -P main.cpp
```

## Exercise 1: Run the Preprocessor

### Exercise 1: Notes

Notice that the preprocessor replaces `#include` directives by simply substituting the contents of the header file. Recall from last week that we (generally) only put declarations in header files. Declaring something is a promise to the compiler that whatever was declared will eventually be defined. Definitions for everything that is declared must exist somewhere, but so long as every variable, function, class, etc. that is used in a source is declared, we are able to compile that source.



# Exercise 1: Run the Preprocessor

## Exercise 1: Notes

You might have also noticed that the header files used some other preprocessor directives:

```
#ifndef NAMEOFFILE_H  
#define NAMEOFFILE_H  
...  
#endif
```

We will discuss what this does on the next slides.

# Header Guards

- The pattern of preprocessor directives in the previous slide is called a header guard
- This prevents headers from being substituted more than once while preprocessing a source file
- This is extremely important: if a header is substituted more than once, we will end up with repeated declarations and the source will fail to compile

# How it Works

- `#ifndef NAMEOFFILE_H:`
  - this means “if not defined”
  - the preprocessor checks if `NAMEOFFILE_H` has previously been defined
  - if `NAMEOFFILE_H` has not been defined the preprocessor continues to the next line
  - if `NAMEOFFILE_H` has been defined, the preprocessor skips to `#endif`
- `#define NAMEOFFILE_H:`
  - adds `NAMEOFFILE_H` to the preprocessor's set of definitions
- `#endif:`
  - ends an if block (in this case the block started by `#ifndef NAMEOFFILE_H`)

## Exercise 2: Run the Preprocessor Without Guards

### Exercise 2

The code in `examples/compilation/withguards` contains the same code as the last example with the header guards removed.

Look at `main.cpp`. Notice that `main.cpp` includes `B.h`, and **`B.h` includes `A.h`**. Notice that **`main.cpp` also includes `A.h`**.

Try running

```
g++ -E -P main.cpp
```

and note the difference from before.

## Exercise 2: Run the Preprocessor Without Guards

### Exercise 2: Notes

Note that without the header guards, the code in A.h is now substituted into main.cpp twice upon preprocessing. This will result in compilation errors due to multiple declarations of the class A. (the same situation occurs with B.cpp)

**Whenever you write a header file, remember to always start by writing the header guards!**

## Compiling to Object Files

- After preprocessing, we have a pure C++ source that can be compiled
- **Note that compilation with `g++` will automatically run the preprocessor first, and then compile: we do not have to run the preprocessor manually**
- We can invoke C++ compilation via: `g++ -c source.cpp`
- This will create the object file `source.o`

# What are Object Files?

- Object files are machine code fragments of a larger program or library
- So long as every identifier used in a source has been declared, we can compile the source to an object (we don't need the definitions)
- This gives us the ability to separate code into different source files, which allows for the separate compilation of objects
- This means that when we change code in some source file, we only need to recompile that source file into a new object file – all of the other object files can remain unchanged
- In large projects, this can save minutes or hours of compilation time

## Exercise 3: Run the Compiler

### Exercise 3

Let's create object files for the source code in `examples/compilation/withguards`. Run the commands:

```
g++ -c A.cpp  
g++ -c B.cpp  
g++ -c main.cpp
```

You should now have three corresponding `.o` files.



# Linking Object Files

- Recall that object files are machine code fragments of an overall program/library
- Object files are essentially sets of symbols (names of variables, functions, classes, etc.) with associated definitions
- Linking combines object files into functioning programs or libraries
- Object files created from source files that have included headers of other modules will have symbols with no associated definition
- The linker tries to resolve all undefined symbols by finding their definitions in other objects and providing a memory reference to their machine code (you get a linking error if it is unsuccessful)

## Exercise 4: Run the Linker

### Exercise 4

Let's try linking the .o files you created in Exercise 3. Run the command:

```
g++ -o prog main.o A.o B.o
```

You should now have an executable called prog in your directory. You have successfully compiled a C++ program!

## Exercise 5: Linker Error

### Exercise 5

Now let's see what happens when we try to link with missing definitions. Try running the command:

```
g++ -o prog main.o
```

Notice all of the undefined reference errors: this is telling us that we are missing definitions. These definitions are in A.o and B.o (which came from A.cpp and B.cpp, respectively).

# Using Makefiles

- As projects get larger, manually compiling C++ via the command line becomes tedious and hard to do efficiently
- Makefiles are used to automate the compilation process
- You were given makefiles for A1 and A2 – these makefiles were trivial since Python is interpreted and doesn't require compilation
- You will be supplied a makefile for A3 so you don't have to compile by hand
- Knowing about the compilation process as described in the previous slides will still be useful: you will undoubtedly run into compilation errors and having an idea of how compilation works will be helpful

## Exercise 6: Sample Makefile

### Exercise 6

Look at `examples/prev_assignment/Makefile` for a sample makefile for a C++ project. We will go over it briefly so you can get an idea of how it works.

A complete understanding of this makefile is not necessary – you will be given the makefile for A3, and you can also use this makefile for A4. See the [GNU make documentation](#) if you want to learn more.

## C++ Example: Implementation of A4 2009

- We will now look at a larger example of a C++ implementation
- See `examples/prev_assignment`

## Exercise 7: Experimenting with A4 Implementation

### Exercise 7

Let's play with the code in the A4 implementation a bit.

The makefile includes a rule `make experiment`, which runs the main method defined in `experimentation/main.cpp` – let's write some code in main to experiment with the implementation.

## Doxygen with C++

- You will be using Doxygen again to build documentation for your C++ implementations
- Since you already know how to use doxygen, this will be easy!
- The only difference is a small change in syntax for the comments – commands remain the same
- **Note that the doxygen comments go in the header files and not the source files**



## Exercise 8: Doxygen

### Exercise 8

The easiest way to see the difference in syntax is by example: see `examples/doxygen`

Look at `Example.h` and notice the syntax difference between doxygen in C++ and what you have done in Python.

Run doxygen via `doxygen doxconfig`

# Unit Testing in C++

- You will use the Catch2 unit testing library to write tests for your code in A3 and A4
- Your experience with pytest will help you quickly learn to use Catch2 – same idea, different syntax
- We will again look at a code example to show the differences in syntax between Catch2 and pytest

## Exercise 9: Unit Testing with Catch2

### Exercise 9

Let's look at the code in

`examples/prev_assignment/test/tests.cpp`

Refer to the comments in the code for an explanation of how Catch2 is used.

Note that a main method is automatically generated with `testmain.cpp` – you don't need to write one.