Managing and Debugging C++ Programs

COSC1076

Semester 1 2019

Week 04

RMIT UNIVERSITY

Admin

- Assignment 1
  - Due, end of week 5
  - Search the forum to see if your questions have already been asked
- Extra-Help Session
  - More running this week
  - 1-hour sessions, BOYD
  - Times, announced on course forum
  - Register interest
- Week 5
  - I am away at RMIT Vietnam
  - Dale (tutor) will be taking the lecture
  - I will still answer questions on the forum

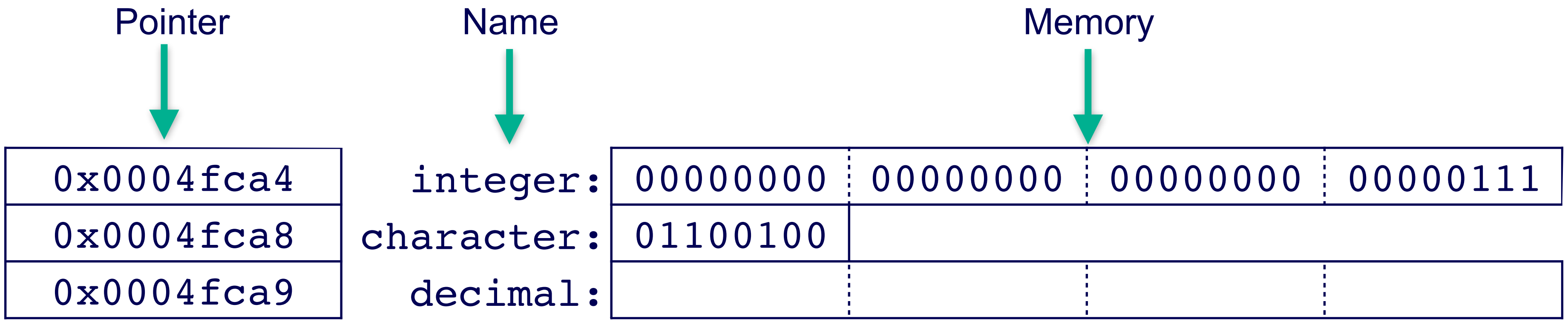
Review:
Pointers & References

The most important topic in the course!

RMIT UNIVERSITY

Computer Memory

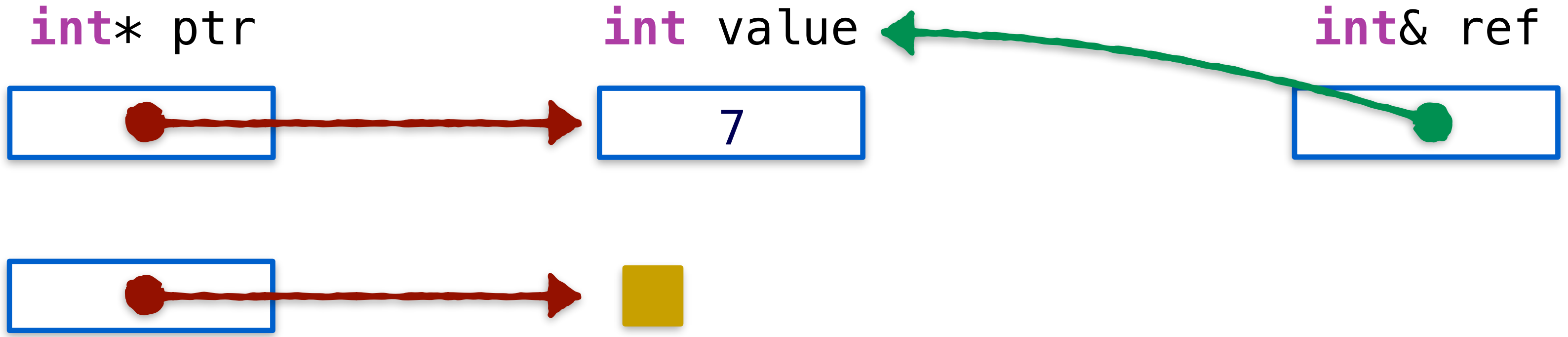
- Each ***byte*** has a unique ***address***
  - This is how a computer can find a piece of memory
  - Addresses are stored in hex, and adjacent memory locations are sequential



Pointers & References

- A reference type is denoted in syntax using a ‘&’

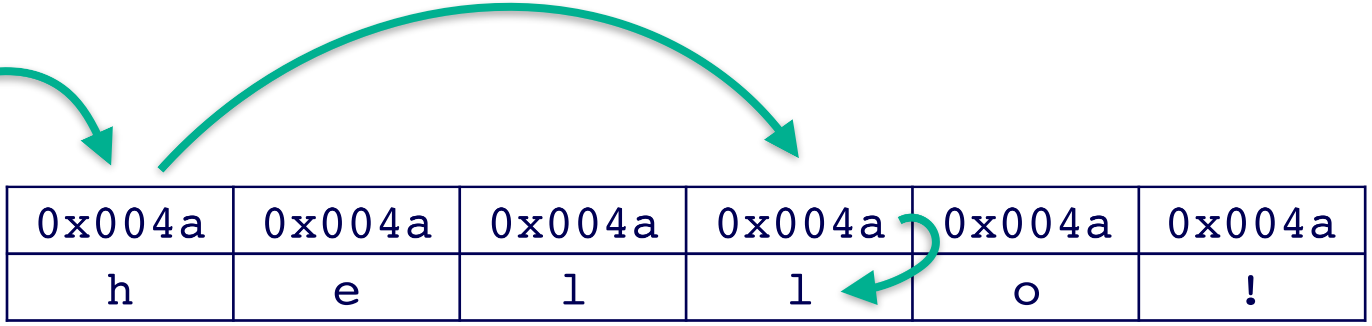
```
int  value = 7;
int* ptr = &value;
int& ref = value;
ptr = NULL;
```



Arrays as Pointers

- An array is actually an ***abstraction for using pointers***!
- The actual array "variable" is a pointer to the first element of the array
- The square-bracket lookup notation is short-hand for:
  - Go to the memory location of the array
  - Go to the 'ith' memory location from this
  - Dereference that memory address

```
array[3]
```



0x004a	0x004a	0x004a	0x004a	0x004a	0x004a
h	e	l	l	o	!

Arrays as Pointers

- In C/C++, a pointer type is another way to represent an array

```
int  array[10]
int* array;
```

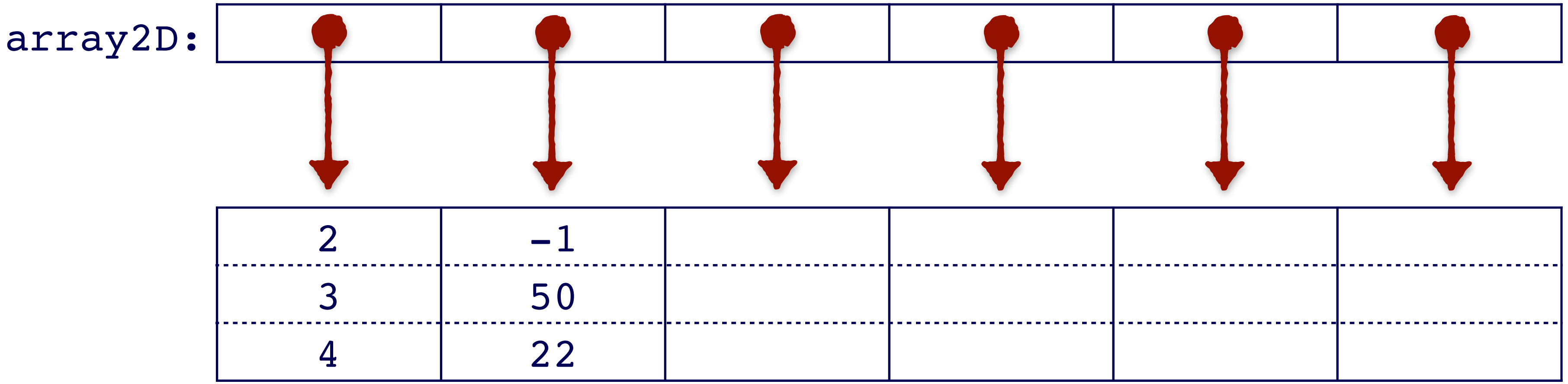
  - BUT!
  - The first version actually ***set's aside the memory*** for the array
  - The second ***can be interpreted*** as an array, but only sets aside memory for a single pointer

- The "pointer form" of an array is often used in functions and methods

Multi-Dimensional Arrays

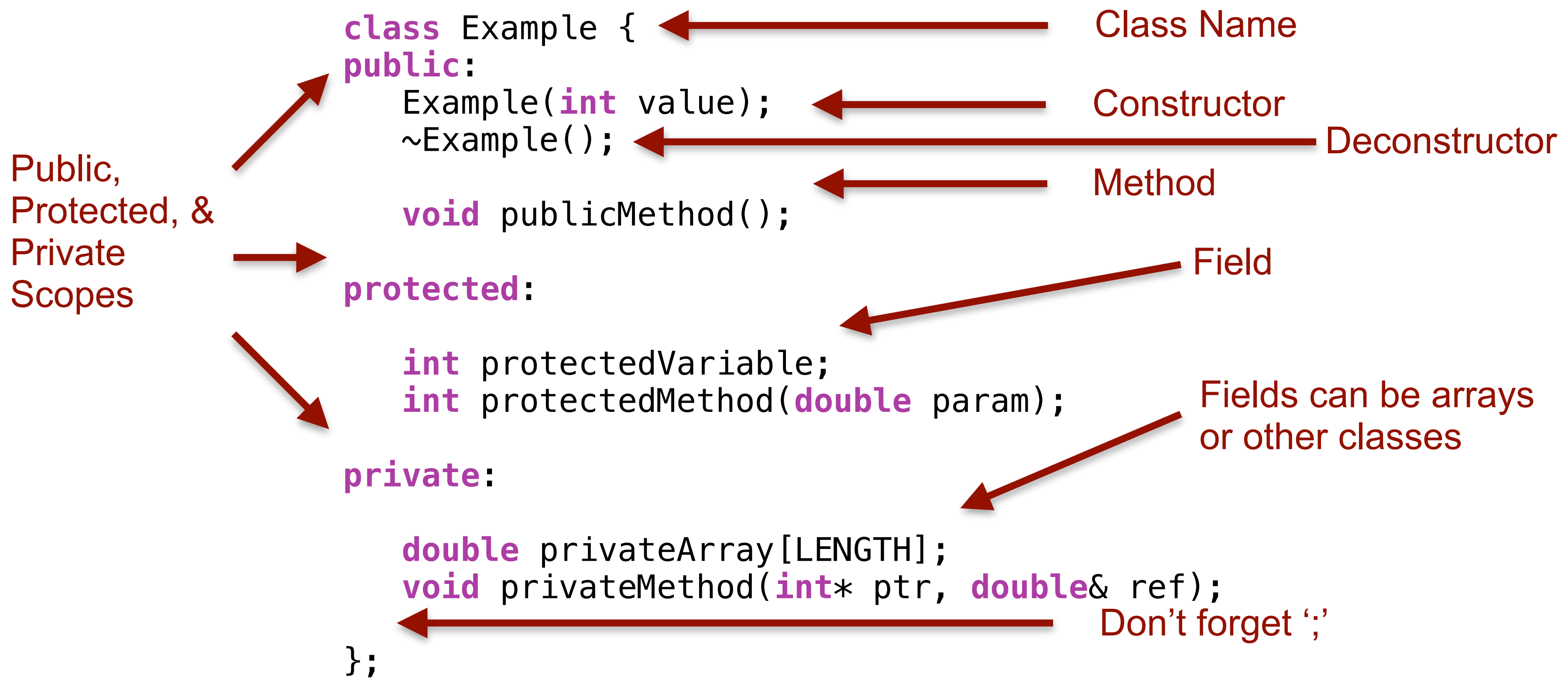
- The best way to think of them, is as an ***"array-of-arrays"***
  - At the first layer, is an array
  - Each element of this is another array

```
int** array2D;
```



Review: Classes

Classes Declaration


Class Name
Constructor
Deconstructor
Method
Public, Protected, & Private Scopes
Field
Fields can be arrays or other classes
Don’t forget ‘;’

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

   void publicMethod();

protected:

   int protectedVariable;
   int protectedMethod(double param);

private:

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

};
```

"this" Pointer

- Like in Java, C++ Classes have a special keyword **this**
  - It gives a ***pointer*** to the current object of the class
  - Using the keyword, all methods and fields of the class can be accessed
  - Works in constructors and methods

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

int Example::protectedMethod(double param) {
   this->protectedVariable = param;
   return 0;
}
```

Classes - allocating

- Allocating memory for a Class ***creates an object*** of that class

```
Example* ex = new Example(10);
```

- Creating an object calls a constructor of the class
  - A constructor must always be called
  - Even if that constructor is empty (takes no parameters)

    ```
    Example* ex = new Example();
    ```
  - If a class defines no constructors, C++ will generate a default empty constructor

Classes - deallocating

- When an object is deleted, a special method is called
  - This method is the ***deconstructor***
    - The deconstructor is denoted with a tilde (~)
    - It always takes no parameters

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

Example::~Example() {
    // cleanup
}
```

Deconstructor

Deconstructor Implementation

```
Example* ex(10);
delete ex;
```

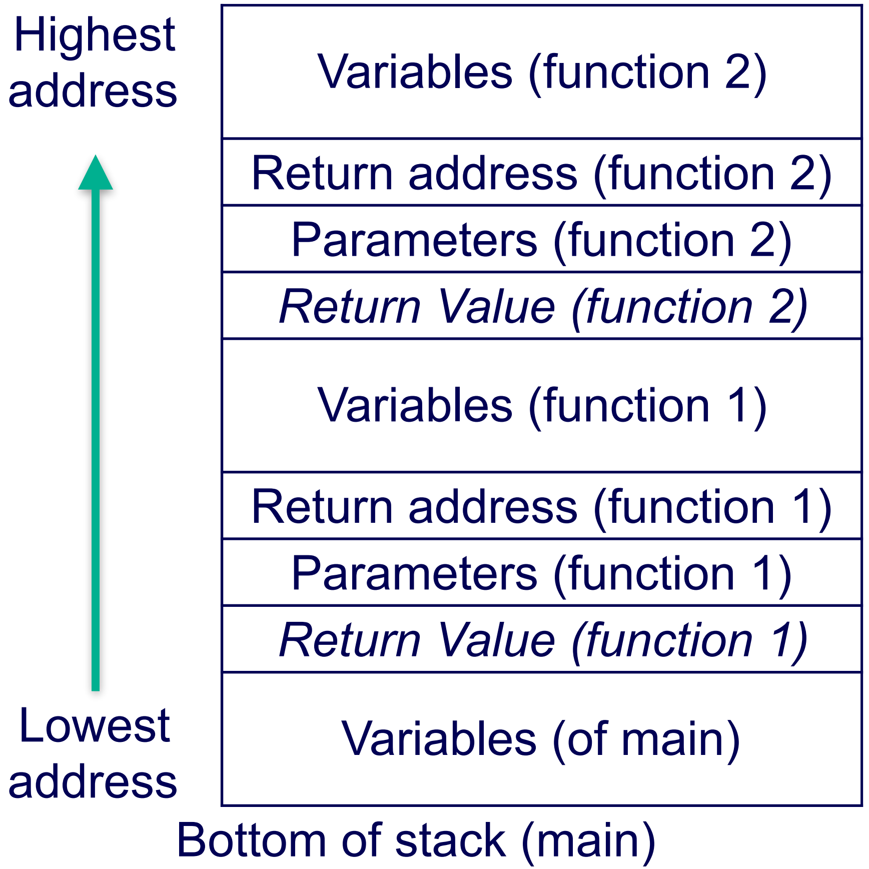
Calls deconstructor

Review:
Program Memory Management

RMIT UNIVERSITY

Program/Call Stack

- A stack is used, because of how memory is allocated
- Memory allocation has two components
  1. Variable allocation
  2. Function call



Heap (Free Store)

- The programmer manages the allocation and de-allocation of memory on the ***heap***
- In Java, objects are allocated on the heap

```
    Object obj = new Object();
```

  - The “new” keyword creates the object
  - The object is stored on the heap

Heap - De-allocation

- Any memory that is allocated, should be ***de-allocated***
  - Also called ***"freeing"*** or ***"deleting"*** memory
  - By de-allocating memory, a program can re-use it for another purpose
  - If memory is not "cleaned-up", the Operating System is not aware that memory is no longer needed
    - Any new memory that a program requires, must be allocated elsewhere
- Java has automated garbage collection, so the programmer does not have to be concerned with de-allocating the memory
  - In C++, ***You (the programmer) MUST*** de-allocate all memory
- Only when a program is terminated, will the operating system reclaim all memory that was not de-allocated

Review:
Multi-File Programs

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

Multi-File Programs (continued)

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
```

Software Problem Solving

RMIT
UNIVERSITY

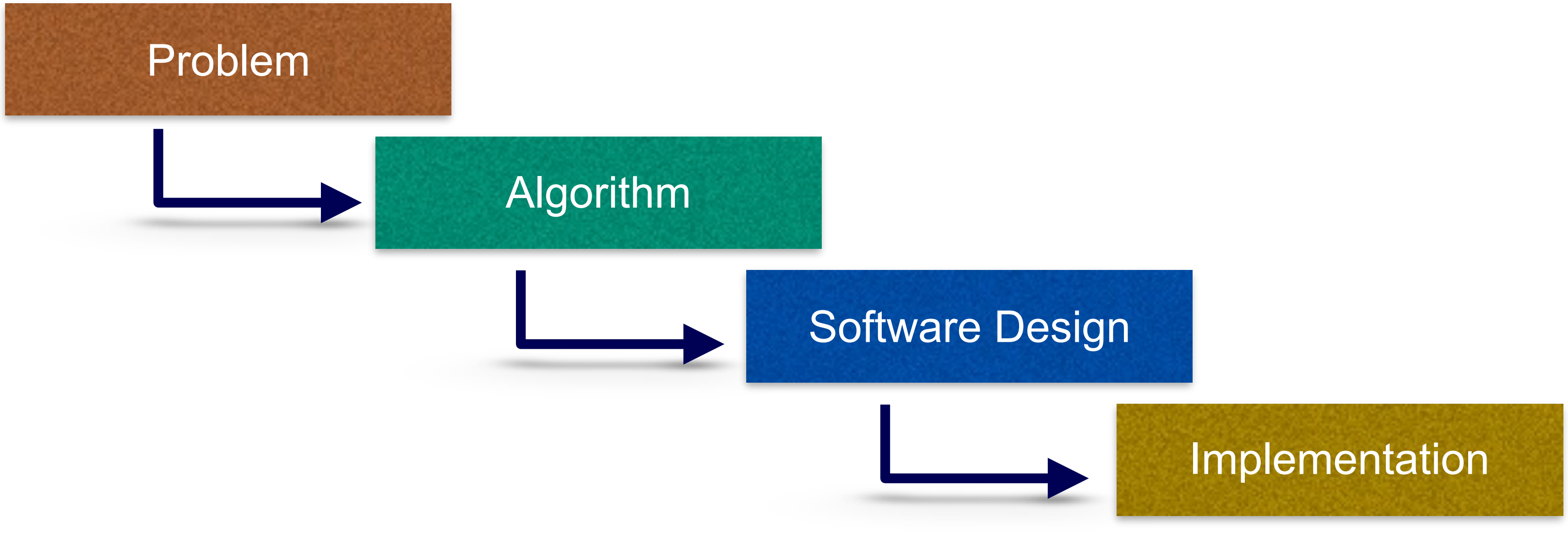
C.A.R. Hoare on Software Design

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

-- Sir Charles Anthony Richard Hoare.

Problem Solving

- Problem Solving is about find software solutions to problems
  - It’s an obvious statement, but what does it actually mean?



Problem Solving

- Problem Solving != Debugging.
- Problem Solving != ad-hoc compile-and-tweak.
- There are usually many possible solutions to a problem.
- The design of algorithms and software are hardest parts of the problem solving process.
  - Actual implementation is often simple
- A designers three most important tools are
  - Knowledge
  - Experience
  - Good judgement

Object Ownership

RMIT UNIVERSITY

Object Ownership

- ***Ownership*** is the concept of who (which module/object/code) has responsibility for managing the life-cycle of memory allocated on the heap
  - That is, who controls a piece of programmer-managed memory*
- Aspects of ownership include:
  - Who controls which other code can modify a piece of memory
  - Who must de-allocate (delete) memory when it is no longer required

**For the purposes of this discussion, memory allocated on the heap will be referred to as "objects". However this may include:*

- *Primitive Types*
- *Arrays*
- *Objects of Classes*

Managing Owned Objects

- The ***owner*** of programmer-allocated objects must consider include:
  - How is an object allocated?
  - What values can be read?
  - What values can be modified?
  - Who can read or modify values?
  - When can values be read or modified?
  - Is it clear that the code manages the life-cycle of the object?
  - When should the object be de-allocated?

Accessing Non-Owned Objects

- A user of a non-owned programmer-allocated object must consider:
  - Is the object allocated and available to use?
  - Am I permitted to access, read and/or modify the object?
  - How should I read access or modify the object?
  - Can the object be modified by someone else?
  - What happens is someone else modified the object?
  - When does the object get de-allocated?
  - Am I aware of when or if an object is de-allocated?
  - Is it possible for me to access an object *after* it has been de-allocated?

Transferring Ownership of Objects

- Ownership over an object may be ***transferred***
  - The previous owner is no longer responsible for managing the object
  - The new owner assume management responsibility
- Third-party users of the object (code that is not involved in the transfer) should be aware of the transfer of ownership
- Transferring ownership can be fraught with danger:
  - It must be clear that ownership is transferred
  - Failing to correctly transfer ownership leaks to memory leaks and segmentation faults

Self-Managed Objects

- A ***fully self-managed object*** internally decides:
  - Who may access and modify its contents
  - When to delete itself
- Generally, when using a self managed object, it must be assumed that
  - Other code can access and modify the object
  - “You” have no control over when the object is destroyed
- Generally, a self-managed object will internally track references to itself’s, and automatically detect when there are no references to it.
  - Then it will delete itself
- By default, Java uses self-managed objects
  - We will discuss implementing self-managed objects later in the course

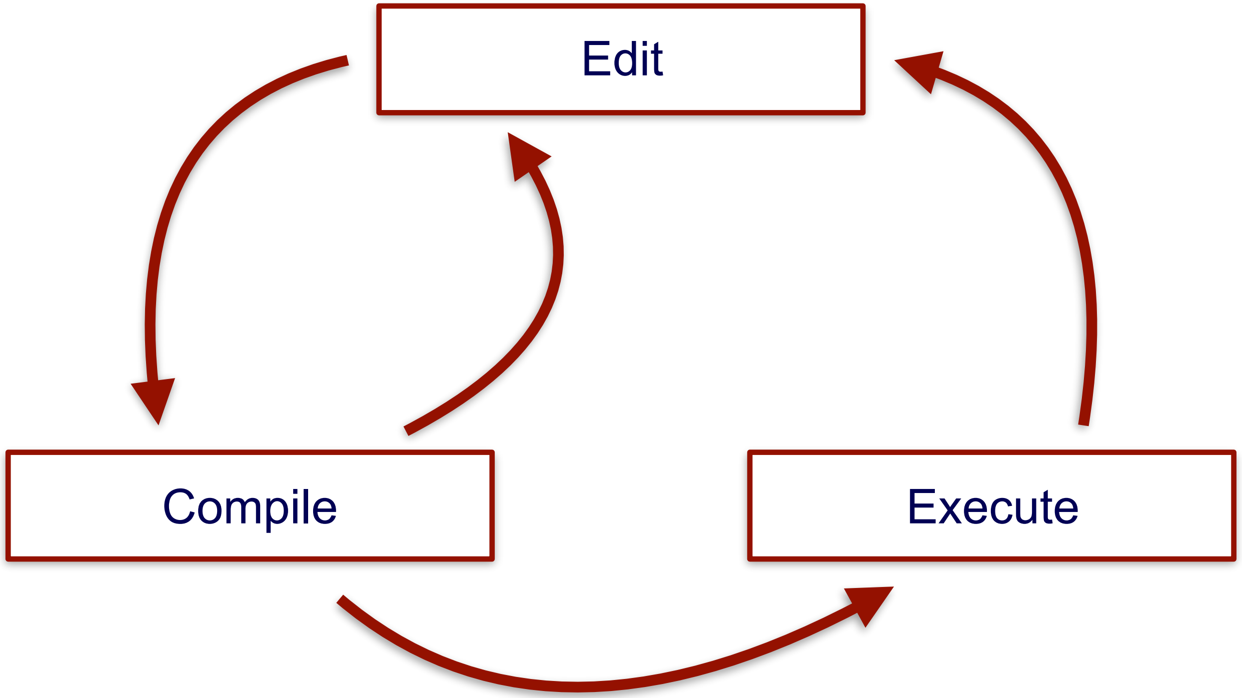
Partial Compilation

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++!*

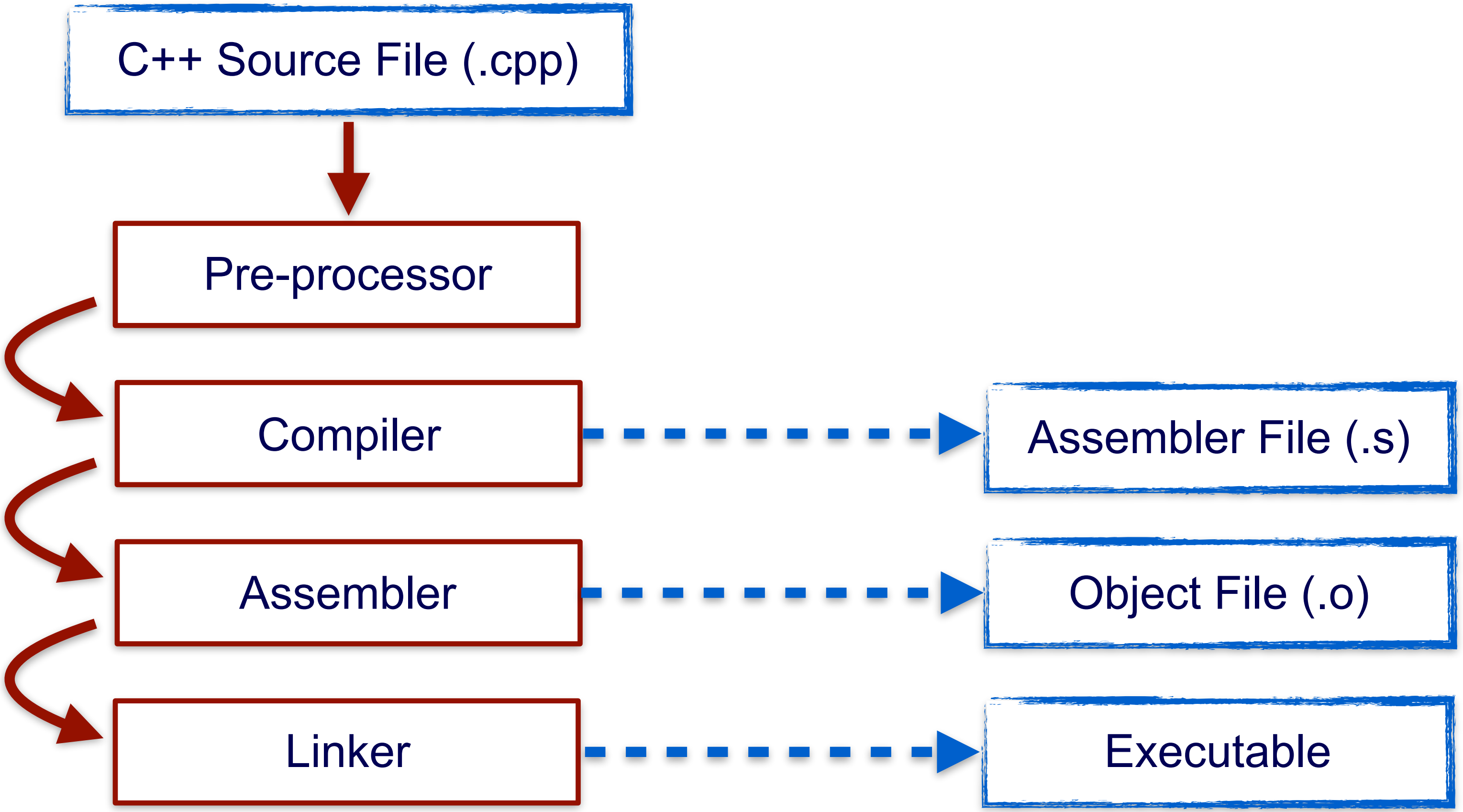
Development Cycle



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



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!*

Automated Build

RMIT UNIVERSITY

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

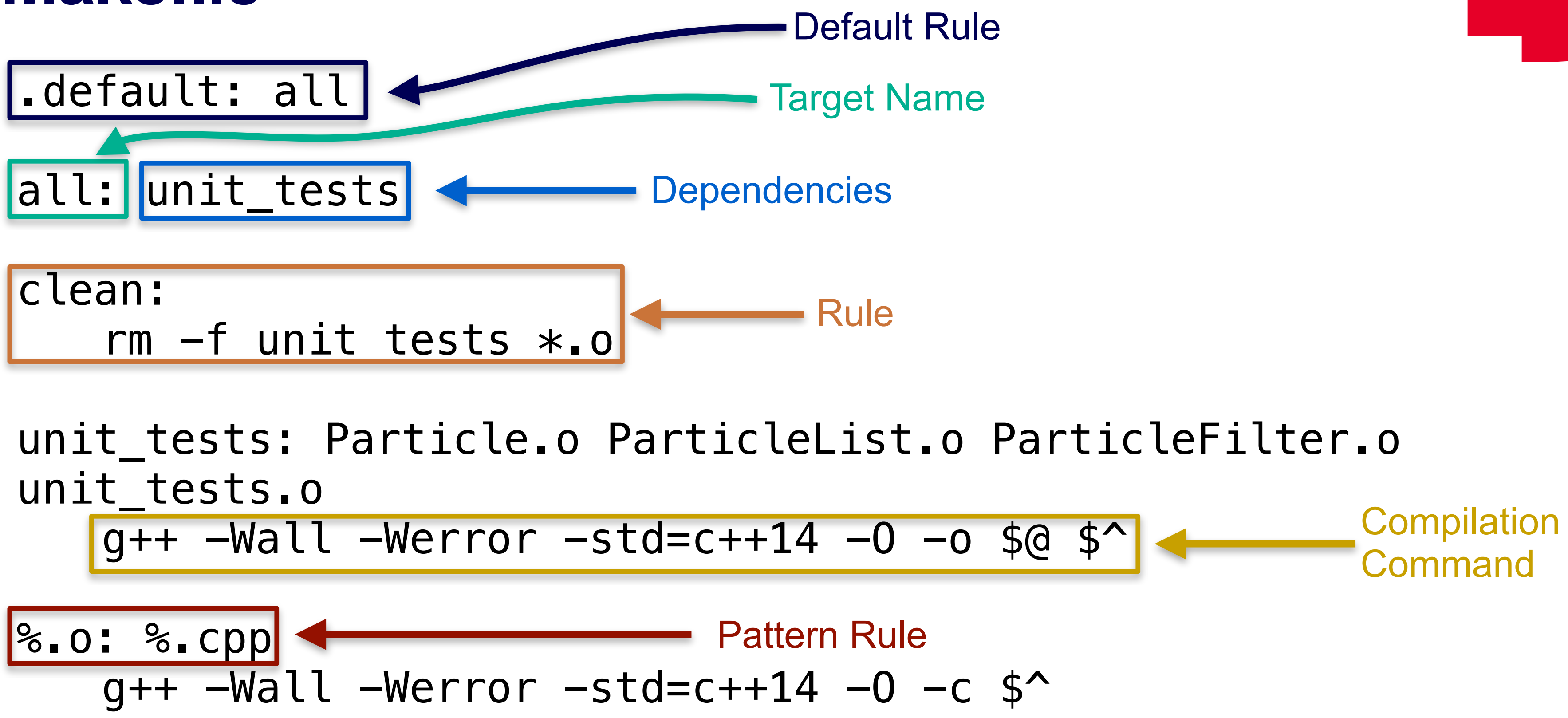
Make

- 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 $^
```

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

COSC1076 automated builds

- For this course we will use only use `make`
  - (But starting after assignment 1)
  - 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!

Debugging
(Not the best term)

Debugging

- Programs are either correct or wrong
  - So Debugging isn’t the best term…
- The best debugging technique is to not write errors in the first place!
- Ideally all code would be correct the first time it is written
  - In practice, this is not the case
  - Debugging is about finding *every error* in a program

Static Techniques

- Static techniques involve inspecting code *without* executing the code
- Code Style
  - It is easier to spot errors in code that is well formatted
  - This is why we are pedantic about code style
- Static Code Analysis
  - Read source code and to follow it the way the computer does (ie. mechanically, step-by-step) to see what it really does.
  - Take the time to understand the error before you try to fix it. Remember the language rules and the software requirements before fixing bugs
  - Explain your code to someone else!
- Proving Code Correctness
  - Using static mathematical analysis, prove that a code block is correct

Dynamic Techniques

- Dynamic techniques find errors by executing the code
- Test-then-Develop paradigm to software development
  1. Decide upon the purpose of a code-block
  2. *First* develop a series of tests for that code-block
  3. Write the code
  4. Evaluate the code against the pre-developed tests
- Writing tests *before* writing the code is critical
  - This maintains the independence of the tests
  - You do not want the tests to be influenced by the development process
  - It is even better to use a third-party developer to write your tests!

Dynamic Techniques

- Black-box testing
  - Testing based on Input-Output
  - A test:
    - Provides input to a code-block
    - Provides the expected output for the code-block
  - The tests pass if the output precisely matches the expected output
- Unit Tests
  - Strictly unit tests target a specific function, module or code-block
  - Same as Test-then-Develop paradigm with Black-box testing

Dynamic Techniques (GDB / LLDB)

- A program may be inspected “live" as it is executed
  - Effectively the execution of the program is paused
  - The program state may be inspected
  - The evaluation of the code may be incrementally “stepped” through.
- The C/C++ live-debugging tool is GDB
  - Program is lldb on MacOS

Dynamic Techniques (GDB / LLDB)

- gdb is a "symbolic" debugger which means that you can examine your program during execution using the symbols of your source code.
  - These symbols do not get saved by compilation by default. To save the symbols, compile with the -g option

```
g++ … -g -o …
```

- gdb can be run with the command:

```
gdb myprog
```

- gdb can also be used to inspect a program state immediately after a segmentation fault has occurred
- Once gdb is started, it has a number of commands to control the dynamic execution of the code
  - Type 'h' (help) to get a summary of the commands

Further Discussion: Assignment 1

RMIT UNIVERSITY

Topics To Discuss

- Header files
- Unit Tests
- Object/memory Ownership

RMIT
UNIVERSITY