Program I/O & Version Control

COSC1076
Semester 1 2019

Week 05

RMIT UNIVERSITY

Admin

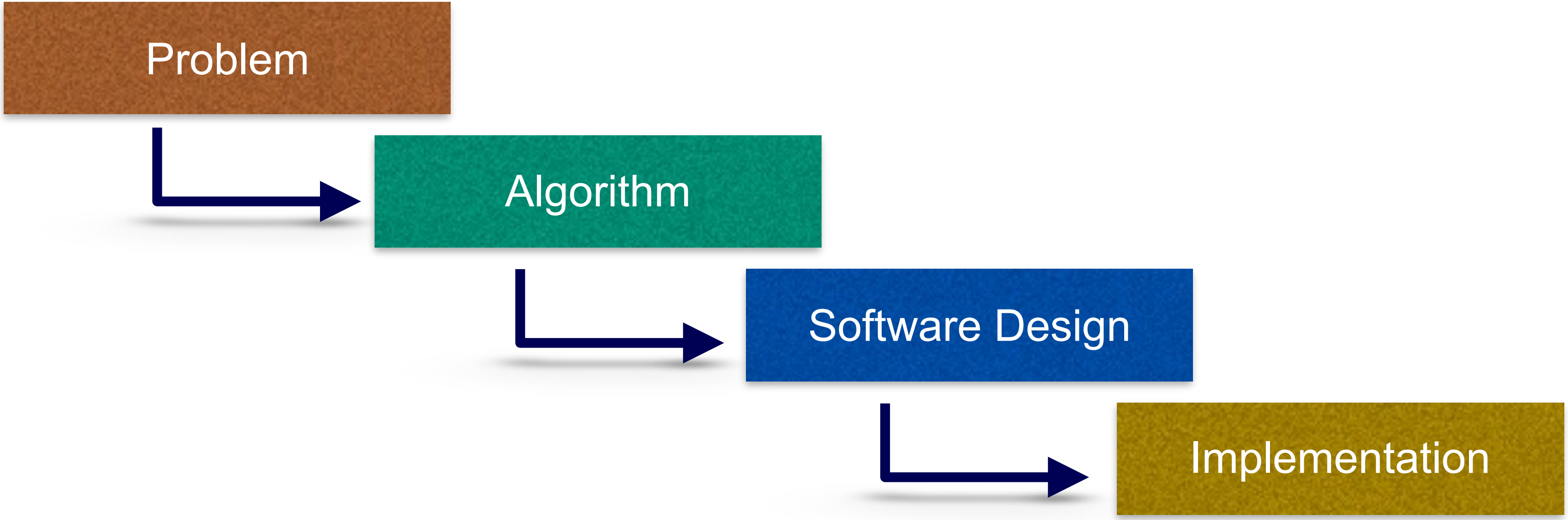
- Assignment 1
  - Due, end of ***this week***
  - Search the forum to see if your questions have already been asked
  - Tim will still read the forum, so ask questions there
- Extra-Help Sessions
  - See Canvas & Register your attendance

Review:
Software Problem Solving

RMIT UNIVERSITY

Problem Solving

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



Review: Program I/O

RMIT UNIVERSITY

What we’ve seen so far

- We have briefly examined how to go simple I/O to the “terminal”
  - Reading in information that a user types from the “terminal”
  - Writing out information to the user on the “terminal”

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 (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
  - (Can also be chained)
- Example

```
int x
std::cin >> x
```

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)

Program I/O

RMIT UNIVERSITY

Input/Output Sources

- There are 2 typical sources of input and output for programs
  - ***Standard I/O*** - aka “the terminal”
  - ***File I/O***
- These are the two we will use in COSC1076
- Other sources of I/O may include:
  - Network I/O (from TCP/UDP connections)
  - External Computer Devices such as:
    - Camera
    - Microphone
    - Speaker

Standard I/O

- Standard I/O is the “standard” communication channel between a program and the operating system
- Generally, the “standard” channel is:
  - Directly connected to the computer’s User I/O
  - That is, the computer keyboard and screen/monitor
  - For programs run through the terminal, the terminal provides the standard keyboard and screen interface
- Standard I/O is divided into 3 streams:
  - stdin (standard input) - typically the keyboard (via the terminal)
  - stdout (standard output) - typically the screen/terminal
  - stderr (standard error) - typically the screen/terminal, explicitly for error reporting

File I/O

- File I/O refers to:
  - Anything stored on the physical hard drive (disk) of the computer
  - Physical external storage devices (including network mounted devices)
  - Printers
- Care must be taken to ensure:
  - The data on the hard drives & external drives is not corrupted
  - Other programs do not modify the files while our program is using them
- In COSC1076, File I/O will deal with “files” stored on the local computer

Abstracting I/O

Does the exact location of the I/O matter to a program?

Abstracting I/O

Does the exact location of the I/O matter to a program?

- No!
  - Reading/Writing from files or standard locations is the same concept
- I/O is abstracted into ***I/O streams***

I/O Streams

- A stream is:
  - A method to communicate with any ***device***
  - A consistent interface for the programmer
  - Independent of the actual device being used
  - A level of abstraction between the programmer and the device
  - Can write to disk file or another type of device (e.g. console)
  - Has two types: (1) text streams; (2) binary streams
- A device may be:
  - Standard I/O
  - Files (local & on external hard drives)
  - Network connections
  - External devices

Binary Streams

- Binary Streams
  - A sequence of bytes (1's and 0's)
  - No character translation occurs
  - There is a 1-to-1 correspondence between bytes of the stream and the actual device
  - May contain a certain number of null bytes at the end
    - For example, for padding so the file fills a sector on a disk

Text Streams

- Text Streams
  - A sequence of characters
  - Can be organised into lines terminated by a newline character
    - Optional for the last line
  - Character translation may occur as required by the host environment
  - For example:
    - newline → carriage return / linefeed [when writing]
    - Carriage return / linefeed ← newline [when reading]
  - Not necessarily a 1-to-1 relationship between characters of the stream and the actual device

Program I/O in C++

Writing (ostream)

- An output stream (`ostream`) allows content to be "written to" the stream
- Output is performed with the << operator

```
std::ostream& outputStream = std::cout;
outputStream << "Some output" << endl;
```

- An ostream is a generic C++ class, of which there can be multiple implementations
  - The output operator is defined as:

```
ostream& operator<<(T& val);
```

  - Where T is some type
  - Note how it returns a reference to an output stream!

Creating an Output Stream

- The output stream for *standard output* is `std::cout`
  - This the form of output we have used most often

Writing to a File

- To write to a file, a file output stream is required
  - `std::ofstream` class
  - Found in `fstream` header file
- Similar to using `std::cout`, except:
  - Before writing, must ***open*** the file (for writing)
    - When opening a file, provide the file name
  - When done, must ***close*** the file
    - This is so your OS knows you are finished using the file
- When opening, there are two modes
  - Normal - creates a new files, erasing any exiting file with the same name
  - Append - add to the end of an existing file

Writing to a File

```
std::string filename("file.txt");          ← Filename - Relative Path
std::ofstream outFile;                     ← File output stream
outFile.open(filename);                    ← Open for normal output
outFile << "Writing to File" << endl;      ← Write
outFile.close();                           ← Close file

outFile.open(filename, std::ofstream::app); ← Open for Appending
outFile << "Appending to File" << endl;
outFile.close();
```

Checking the Status of Output Streams

- It may be necessary to check the "status" of the output stream
  - The `ostream` class has methods to do this
- It may also be necessary to "flush" the stream to ensure the contents is written
  - Typically a stream automatically flushes after every newline

`good()`	Check whether state of stream is good
`fail()`	Check whether either failbit or badbit is set
`flush()`	Flush output stream buffer

Reading (istream)

- An input stream (`istream`) allows content to be "read from" the stream
- Input can be performed with the >> operator

```
std::istream& inputStream = std::cin;
double value;
inputStream >> value;
std::cout << "Read: " << value << std::endl;
```

- An istream is a generic C++ class:
  - The output operator is defined as:

```
istream& operator>>(T& val);
```

  - Where T is some type
  - Note how it returns a reference to an input stream!

Creating an Read Stream

- The output stream for *standard input* is `std::cin`

Reading from a File

- To read from a file, a file input stream is required
  - `std::ifstream` class
  - Found in `fstream` header file
- Functions similar to `std::ofstream`:
  - File must be opened/closed
  - Reading may fail if the opened file does not exist

Checking the Status of Read Streams

- The "status" of an input stream is checked similar to output streams
- A special check is if the end-of-input (^D character) is reached

`good()`	Check whether state of stream is good
`fail()`	Check whether either failbit or badbit is set
`eof()`	Check if EOF is reached

Technical Details

- The >> read operator only reads the ***next valid token*** from the input stream
  - The validity of a token is based on the type of the variable that is being read
  - Token are *always* separated by whitespace
  - Whitespace is not read!
    - This includes spaces and newlines
    - Even if `std::string` is used, whitespace is ignored

Alternative Read methods

- The `istream` class provides alternative methods for reading, depending on precisely how the input needs to be handled

`int get()`	Extract a single character from the stream
`istream& getline(char* s, streamsize n );`	Read a whole line into a string, including whitespace and newline characters, up to a maximum number of characters

- The string STL library provides an alternative get line that works with strings

`std::getline(istream&  is, string& str);`	Read a whole line into a string, including whitespace and newline characters

String Processing

String Streams (stringstream)

- Input and Output streams can be generated from `std::string`'s
- They use the same operators/methods:
  - << operator
  - >> operator
  - get
  - good/bad/eof
- They do no require opening/closing
- String streams are useful for:
  - Loading data before processing it
  - Modularisation of I/O processing
- Located in `sstream` STL file

C-String

- Recall that a c-string is just an array of characters

`h`	`e`	`l`	`l`	`o`	`!`	`\0`

std::string Class - Indexing

- A `std::string` is a class that “nicely” wraps a c-style string
  - Provides useful methods for interacting with the string
  - Can still interact with the `std::string` in the style of an array
  - We don’t “see” the \0 terminating character
- Each character in the string can be individually accessed

`[<index>]`	Lookup cell (index) using square bracket notation, similar to using an array. No bounds checking!
`at(<index>)`	Method that is the same as using square brackets. With bounds checking

std::string Class - Methods

- Other methods on the class include

`length()`	Returns the number of characters
`reverse()`	Reverse the string
`append(string)`	Append a given string to the end of this string
`substr(int,int)`	Return the sub-string between two locations
`c_str()`	Get a c-style version of the string

std::string Class - Operators

- The class can also be used with typical operators

=	Assignment. Copies one string into another
+	Combine two string into a third string. Copies the contents
+=	Append
== !=	Compare for equality/inequality
< > <= >=	Compare for lexical ordering

- We will see how to do this in week 10.

Lexical Comparison of Strings

- Two string can be compared for ***lexical ordering***.
  - That is not necessarily the order which the two strings appear in a dictionary
  - The comparison uses ASCII (or unicode) values
- Each character of the string is compared one-at-a-time, in order until two characters differ
  - The character with the small ASCII value is ***lexicographically first***
  - Otherwise, the smaller string comes first

string 1:	h	e	l	l	o	!
compare (<)	<	<	<	<		
string 2:	h	e	l	P		
string 3:	h	e	l	l		

Command Line Arguments

RMIT UNIVERSITY

Command Line Arguments

- ***Command line arguments*** are options placed on the terminal after the name of the program
  - Arguments are separated by any amount of whitespace

```
./program_name arg1 arg2 arg3
```

Command
(Program Name)

Argument(s)

- Technically the name of the command/program is also an argument!

Command Line Arguments

- Arguments are passed to a C++ program using a special set of parameters for the main function

```
int main(int argc, char** argv) {
}
```

Number of Arguments

Arguments as 2D Array of strings

- If a program does not require the arguments, the `void` type is used

```
int main(void) {
}
```

Command Line Arguments

```
int main(int argc, char** argv) {
}
```

- The program name is always the first argument
- `argc`
  - Will always be at least 1
- `argv`
  - An array of c-style strings
  - Each string in argv is guaranteed to be null-terminated
  - For our purposes, the easiest thing to do is to convert the c-style string into `std::string`’s

C++ Online Documentation

Online C++ References

- For classes and header files that we are using, links will be given to the relevant documentation. These are listed on Canvas for each week
  - <string> header
    - std::string
  - <iostream> header
    - std::istream
    - std::osftream
  - <fstream> header
- Main websites:
  - Official Language Reference
  - C++.com

Online C++ References

- Be careful
  - These sites are the definitive documentation and can be quite confusing
  - Most classes we have seen are complex generic types
    - The types that we use are generated by typedef's!
  - But don't be alarmed, the information you need is there and simple to extract

Group Development & Version Control (Git)

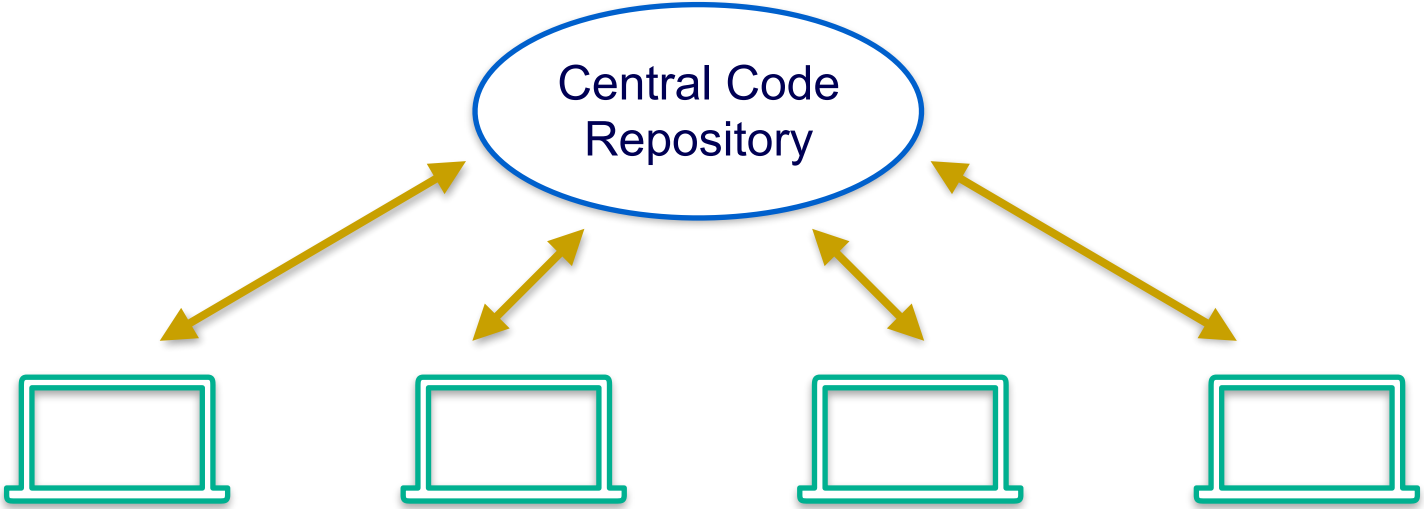
RMIT UNIVERSITY

Version Control with Git

- Some of you will be familiar with this already
- Many software manage system (SMS's) exist to help developers manage and collaborate on creating software.
- Git is a popular tool (for now)
  - Git is highly configurable for many different purposes
  - Git has many different functionalities
  - We will use Git in a very simple model
    - You can go as far "down the rabbit hole" as you want

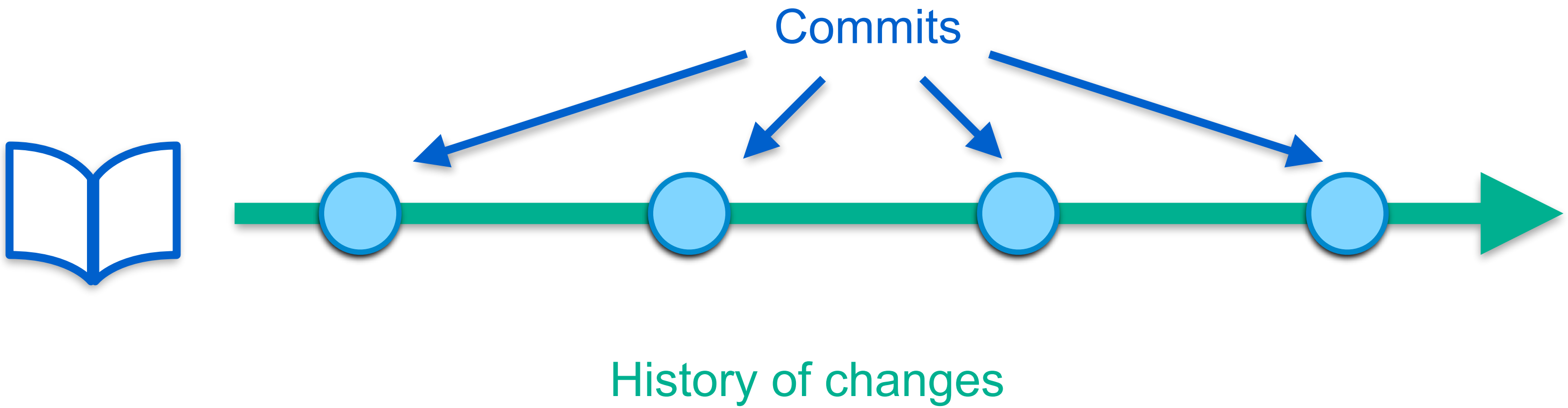
Centralised Git Model

- A simple model uses a ***central repository*** for a code base
  - Users ***push*** and ***pull*** changes to the code base to/from the repository
  - Code changes are grouped into a block termed a ***commit***



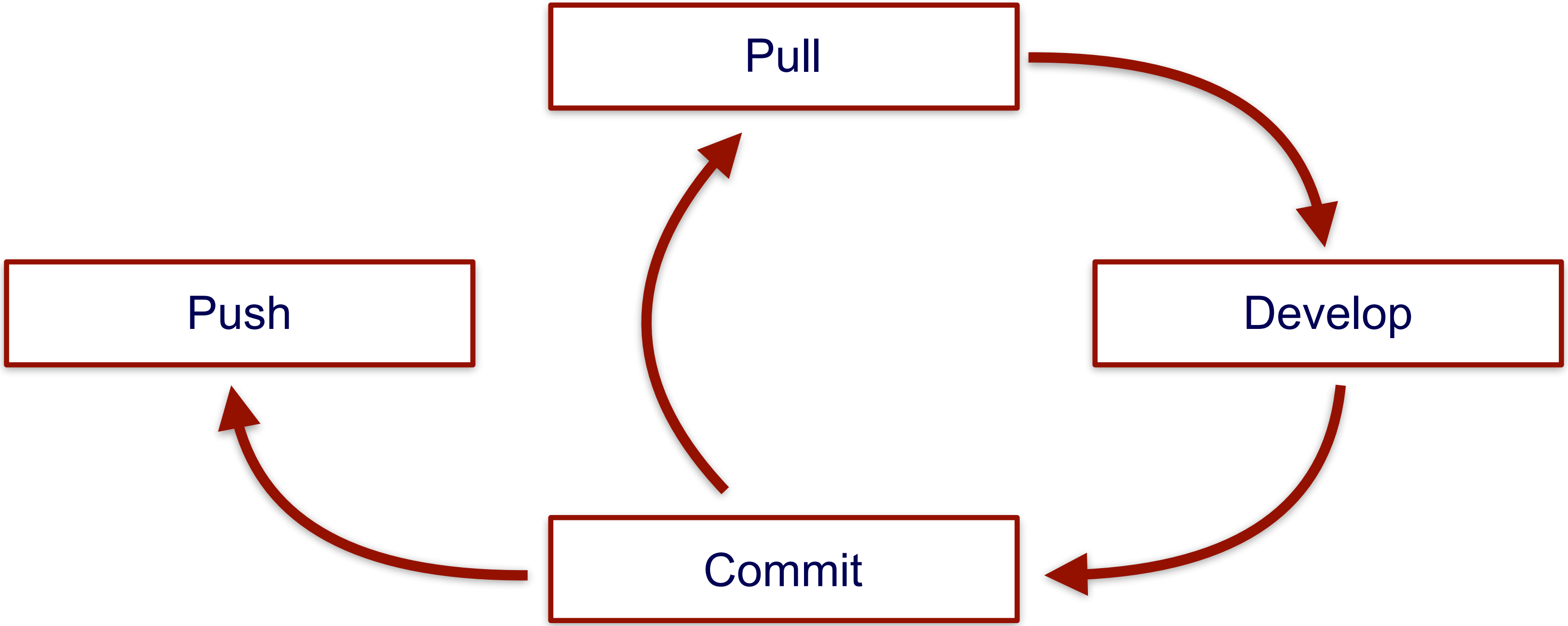
Commits

- A ***commit*** is a "***diff***" (difference) in how the files of the code base have changed from their previous state, to their new state
- Commits are chained together to form a ***history*** of code changes
- The head of the history is the most current (most up-to-date) version of the codebase, applying all of the commit from the start of the history



Standard Development Cycle

- The Standard Develop Cycle with Version Control is



Advanced Git: Branches

- Code changes from different developers can be divided into different ***branches***
- Every Git repository always has at least one branch
  - The main development branch is called ***master***
- For now, we will stick to working with one branch

Git Resources

- RMIT has a partnership with Atlassian to use BitBucket.
  - You should be able to use your RMIT account details to log-into BitBucket

- Other Resources:
  - Git Cheat Sheet (on Canvas, thanks to Atlasssian)
  - Install Git (for Mac, Linux, & Windows)

Debugging
(Not the best term)

RMIT UNIVERSITY

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

Dynamic Techniques (GDB / LLDB)

- After you start up gdb you can issue commands to gdb to control, monitor and modify your program's execution. There are a large number of commands available. Here are some as example (the abbreviation for each command is shown in square brackets):

  `[l]ist` – display source code

  `[br]eak <function> | linenum`

  - sets a breakpoint at 'function' or at linenum of your source code
  - when the program reaches this breakpoint, execution is suspended and you can examine your program using other gdb commands

- For example:

  `br myFunc` – sets breakpoint at function 'myFunc'

  `br 120` – sets breakpoint at line no. 120

Further Discussion: Assignment 1

RMIT UNIVERSITY

RMIT UNIVERSITY