Research Computing with C++

Jim Dobson Matt Clarkson Jonathan Cooper
Roma Klapaukh James Hetherington

Contents

Chapter 1

Lecture 1: C++ for Research

Course Overview

Part 1

- Using C++ in research
- Better C++
  - Reliable
  - Reproducible
  - Good science
  - Libraries

Part 2

- HPC concepts
- Shared memory parallelism - OpenMP
- Distributed memory parallelism - MPI

Course Aims

- Teach how to do research with C++
- Optimise your research output
- A taster for various technologies
- Not just C++ syntax, Google/Compiler could tell you that!

Pre-requisites

- Use of command line (Unix) shell
- You are already doing some C++
- You are familiar with your compiler
- (Maybe) You are happy with the concept of classes
- (Maybe) You know C++ up to templates?
- You are familiar with development eg. version control
  - Git: https://git-scm.com/

Course Notes

- Revise some software Engineering: MPHY0021
- Register with Moodle: PHAS0100
  - contact lecturer for key to self-register
  - guest key to look around is "996135"
- Online notes: PHAS0100

Course Assessment

- 2 pieces coursework - 40 hours each
  - See assessment section for details

Course Community

- UCL Research Programming Hub: http://research-programming.ucl.ac.uk
- Slack: https://ucl-programming-hub.slack.com

Lecture 1: C++ In Research

Problems In Research

- Poor quality software
- Excuses
  - I'm not a software engineer
  - I don't have time
  - It's just a prototype
  - I'm unsure of my code (scared to share)

C++ Disadvantages

Some people say:

- Compiled language
  - (compiler versions, libraries, platform specific etc)
- Perceived as difficult, error prone, wordy, unfriendly syntax
- Result: It's more trouble than its worth?

C++ Advantages

- Fast, code compiled to machine code
- Stable, evolving standard, powerful notation, improving
- Lots of libraries, Boost, Qt, VTK, ITK etc.
- Nice integration with CUDA, OpenACC, OpenCL, OpenMP, OpenMPI
- Result: Good reasons to use it, or you may *have* to use it

Research Programming

- Software is always expensive
  - Famous Book: Mythical Man Month
- Research programming is different to product development:
  - What is the end product?

Development Methodology?

- Will software engineering methods help?
  - Waterfall
  - Agile
- At the 'concept discovery' stage, probably too early to talk about product development

Approach

- What am I trying to achieve?
- How do I maximise my research output?
- What is the best pathway to success?
- How do I de-risk (get results, meet deadlines) my research?
- Software is an important part of scientific reproducibility, authorship, credibility.

1. Types of Code

- What are you trying to achieve?
- Divide code:
  - Your algorithm
  - Testing code
  - Data analysis
  - User Interface
  - Glue code
  - Deployment code
  - Scientific output (a paper)

Question: What type of code is C++ good for? Question: Should all code be in C++?

2. Maximise Your Value

- Developer time is expensive
- Your brain is your asset
- Write as little code as possible
- Focus tightly on your hypothesis
- Write the minimum code that produces a paper

Don't fall into the trap "Hey, I'll write a framework for that"

3. Ask Advice

- Before contemplating a new piece of software
  - Ask advice - Slack Channel
  - Review libraries and use them.
  - Check libraries are suitable, and sustainable.
  - Read Libraries section from Software Engineering course
  - Ask about best practices

Debunking The Excuses

- I'm not a software engineer
  - Learn effective, minimal tools
- I don't have time

  - Unit testing to save time
  - Choose your battles/languages wisely
- I'm unsure of my code
  - Share, collaborate

What Isn't This Course?

We are NOT suggesting that:

- C++ is the solution to all problems.
- You should write all parts of your code in C++.

What Is This Course?

We aim to:

- Improve your C++ (and associated technologies).
- Introduction to High Performance Computing (HPC).

So that:

- Apply it to research in a pragmatic fashion.
- You use the right tool for the job.

Git

Git Introduction

- This is a practical course
- We will use git for version control
- Submit git repository for coursework
- Here we provide a very minimal introduction

Git Resources

- Complete beginner - Try Git
- Git book by Scott Chacon
- Git section of MPHYG001
- MPHYG001 repo

Git Walk Through - 1

(demo on command line)

- Creating your own repo:
  - git init
  - git add
  - git commit
  - git log
  - git status
  - git remote add
  - git push

Git Walk Through - 2

(demo on command line)

- Working on existing repo:
  - git clone
  - git add
  - git commit
  - git log
  - git status
  - git push
  - git pull

Git Walk Through - 3

- Cloning or forking:
  - If you have write permission to a repo: clone it, and make modifications
  - If you don't: fork it, to make your own version, then clone that and make modifications.

Homework - 1

- Register Github
- Register for private repos - free for academia.
- Find project of interest - try cloning it, make edits, can you push?
- Find project of interest - try forking it, make edits, can you push?

CMake

Ever worked in Industry?

- (0-3yrs) Junior Developer - given environment, team support
- (4-6yrs) Senior Developer - given environment, leading team
- (7+ years) Architect - chose tools, environment, design code
- Only cross-platform if product/business demands it
- All developers told to use the given platform no choice

Ever worked in Research?

- All prototyping, no scope
- Start from scratch, little support
- No end product, no nice examples
- Cutting edge use of maths/science/technology
- Share with others on other platforms
- Develop on Windows, run on cluster (Linux)

Research Software Engineering Dilemma

- Comparing Research with Industry, in Research you have:
  - Least experienced developers
  - with the least support
  - developing cross-platform
  - No clear specification or scope
- Struggle of C++ is often not the language its the environment

Build Environment

- Windows: Visual Studio solution files
- Linux: Makefiles
- Mac: XCode projects / Makefiles

Question: How was your last project built?

CMake Introduction

- This is a practical course

- We will use CMake as a build tool
- CMake produces
  - Windows: Visual Studio project files
  - Linux: Make files
  - Mac: XCode projects, Make files
- So you write 1 build language (CMake) and run on multi-platform.
- This course will provide most CMake code and boiler plate code for you, so we can focus more on C++. But you are expected to google CMake issues and work with CMake.

CMake Usage Linux/Mac

Demo an "out-of-source" build

```
mkdir cmake_demo
cd cmake_demo
git clone https://github.com/MattClarkson/CMakeHelloWorld
cd CMakeHelloWorld
mkdir build
cd build
cmake ..
make
```

Homework - 2

- Build https://github.com/MattClarkson/CMakeHelloWorld.git
- Ensure you do "out-of-source" builds
- Use CMake to configure separate Debug and Release versions
- Add code to hello.cpp:
  - On Linux/Mac re-compile just using make

Homework - 3

- Build https://github.com/MattClarkson/CMakeHelloWorld.git
- Exit all code editors
- Rename hello.cpp
- Change CMakeLists.txt accordingly
- Notice: The executable name and .cpp file name can be different
- In your build folder, just try rebuilding.
- You should see that CMake is re-triggered, so you get a cmake/compile cycle.

CMake Basics

Compiling Basics

Question: How does a compiler work?

How does a Compiler Work?

Question: How does a compiler work?

- (Don't quote any of this in a compiler theory course!)
- Preprocessing .cpp/.cxx into pure source files
- Source files compiled, one by one into .o/.obj
- executable compiled to .o/.obj
- executable linked against all .o, and all libraries

That's what you are trying to describe in CMake.

CMake - Directory Structure

- CMake starts with top-level CMakeLists.txt
- CMakeLists.txt is read top-to-bottom
- All CMake code goes in CMakeLists.txt or files included from a CMakeLists.txt
- You can sub-divide your code into separate folders.
- If you `add_subdirectory`, CMake will go into that directory and start to process the CMakeLists.txt therein. Once finished it will exit, go back to directory above and continue where it left off.
- e.g. top level CMakeLists.txt

```
project(MYPROJECT VERSION 0.0.0)
add_subdirectory(Code)
if(BUILD_TESTING)
  set()
  include_directories()
  add_subdirectory(Testing)
endif()
```

CMake - Define Targets

- Describe a target, e.g. Library, Application, Plugin

```
add_executable(hello hello.cpp)
```

- Note: You don't write compile commands
- You tell CMake what things need compiling to build a given target. CMake works out the compile commands!

CMake - Order Dependent

- You can't say "build Y and link to X" if X not defined
- So, imagine in a larger project

```
add_library(libA a.cpp b.cpp c.cpp)
add_library(libZ x.cpp y.cpp z.cpp)
target_link_libraries(libZ libA)
add_executable(myAlgorithm algo.cpp) # contains main()
target_link_libraries(myAlgorithm libA libZ ${THIRD_PARTY_LIBS})
```

- So, logically, its a big, ordered set of build commands.

Homework - 4

- Build https://github.com/MattClarkson/CMakeLibraryAndApp.git
- Look through .cpp/.h code. Ask questions if you don't understand it.
- What is an "include guard"?
- What is a namespace?
- Look at .travis.yml and appveyor.yml - cross platform testing, free for open-source
- Look at myApp.cpp, does it make sense?
- Look at CMakeLists.txt, does it make sense?
- Look for examples on the web, e.g. VTK

Intermediate CMake

What's next?

- Most people learn CMake by pasting snippets from around the web
- As project gets larger, its more complex
- Researchers tend to just "stick with what they have."
- i.e. just keep piling more code into the same file.
- Want to show you a reasonable template project.

Homework - 5

- Build https://github.com/MattClarkson/CMakeCatch2.git
- If open-source, use travis and appveyor from day 1.
- We will go through top-level CMakeLists.txt in class.
- See separate `Code` and `Testing` folders
- Separate `Lib` and `CommandLineApps` and `3rdParty`
- You should focus on
  - Write a good library
  - Unit test it
  - Then it can be called from command line, wrapped in Python, used via GUI.

Homework - 6

- Try renaming stuff to create a library of your choice.
- Create a github account, github repo, Appveyor and Travis account
- Try to get your code running on 3 platforms
- If you can, consider using this repo for your research
- Discuss
  - Debug / Release builds
  - Static versus Dynamic
  - declspec import/export
  - Issues with running command line app? Windows/Linux/Mac

Looking forward

In the remainder of this course we cover

- Some compiler options
- Using libraries
- Including libraries in CMake
- Unit testing
- i.e. How to put together a C++ project
- in addition to actual C++ and HPC

Unit Testing

What is Unit Testing?

At a high level

- Way of testing code.
- Unit
  - Smallest 'atomic' chunk of code
  - i.e. Function, could be a Class
- See also:
  - Integration/System Testing
  - Regression Testing
  - User Acceptance Testing

Benefits of Unit Testing?

- Certainty of correctness
- (Scientific Rigour)
- Influences and improves design
- Confidence to refactor, improve

Drawbacks for Unit Testing?

- Don't know how
  - This course will help
- Takes too much time
  - Really?
  - IT SAVES TIME in the long run

Unit Testing Frameworks

Generally, all very similar

- JUnit (Java), NUnit (.net?), CppUnit, phpUnit,
- Basically
  - Macros (C++), methods (Java) to test conditions
  - Macros (C++), reflection (Java) to run/discover tests
  - Ways of looking at results.
    - Java/Eclipse: Integrated with IDE
    - Log file or standard output

Unit Testing Example

How To Start

We discuss

- Basic Example
- Some tips

Then its down to the developer/artist.

C++ Frameworks

To Consider:

- Catch
- GoogleTest
- QTestLib
- BoostTest
- CppTest
- CppUnit

Worked Example

- Borrowed from
  - Catch Tutorial
  - and Googletest Primer
- We use Catch, so notes are compilable
- But the concepts are the same

Code

To keep it simple for now we do this in one file:

```
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cp
#include "../catch/catch.hpp"

unsigned int Factorial( unsigned int number ) {
    return number <= 1 ? number : Factorial(number-1)*number;
}
```

```
TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

Produces this output when run:

```
===============================================================================
All tests passed (4 assertions in 1 test case)
```

Principles

So, typically we have

- Some `#include` to get test framework
- Our code that we want to test
- Then make some assertions

Catch / GoogleTest

For example, in Catch:

```
// TEST_CASE(<unique test name>, <test case name>)
TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
}
```

In GoogleTest:

```
// TEST(<test case name>, <unique test name>)
TEST(FactorialTest, HandlesPositiveInput) {
  EXPECT_EQ(2, Factorial(2));
  EXPECT_EQ(6, Factorial(3));
}
```

all done via C++ macros.

Tests That Fail

What about Factorial of zero? Adding

```
    REQUIRE( Factorial(0) == 1 );
```

Produces something like:

```
    factorial2.cc:9: FAILED:
    REQUIRE( Factorial(0) == 1 )
    with expansion:
    0 == 1
```

Fix the Failing Test

Leading to:

```
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cp
#include "../catch/catch.hpp"

unsigned int Factorial( unsigned int number ) {
    //return number <= 1 ? number : Factorial(number-1)*number;
    return number > 1 ? Factorial(number-1)*number : 1;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(0) == 1 );
    REQUIRE( Factorial(1) == 1 );
    REQUIRE( Factorial(2) == 2 );
    REQUIRE( Factorial(3) == 6 );
    REQUIRE( Factorial(10) == 3628800 );
}
```

which passes:

```
===============================================================================
All tests passed (5 assertions in 1 test case)
```

Test Macros

Each framework has a variety of macros to test for failure. Catch has:

```
REQUIRE(expression); // stop if fail
CHECK(expression);   // doesn't stop if fails
```

If an exception is thrown, it's caught, reported and counts as a failure.

Examples:

```
CHECK( str == "string value" );
CHECK( thisReturnsTrue() );
REQUIRE( i == 42 );
```

Others:

```
REQUIRE_FALSE( expression )
CHECK_FALSE( expression )
REQUIRE_THROWS( expression ) # Must throw an exception
CHECK_THROWS( expression ) # Must throw an exception, and continue testing
REQUIRE_THROWS_AS( expression, exception type )
CHECK_THROWS_AS( expression, exception type )
REQUIRE_NOTHROW( expression )
CHECK_NOTHROW( expression )
```

Testing for Failure

To re-iterate:

- You should test failure cases
  - Force a failure
  - Check that exception is thrown
  - If exception is thrown, test passes
  - (Some people get confused, expecting test to fail)
- Examples
  - Saving to invalid file name
  - Negative numbers passed into double arguments
  - Invalid Physical quantities (e.g. -300 Kelvin)

Setup/Tear Down

- Some tests require objects to exist in memory
- These should be set up
  - for each test
  - for a group of tests
- Frameworks do differ in this regards

Setup/Tear Down in Catch

Referring to the Catch Tutorial:

```
TEST_CASE( "vectors can be sized and resized", "[vector]" ) {

    std::vector<int> v( 5 );

    REQUIRE( v.size() == 5 );
    REQUIRE( v.capacity() >= 5 );

    SECTION( "resizing bigger changes size and capacity" ) {
        v.resize( 10 );

        REQUIRE( v.size() == 10 );
        REQUIRE( v.capacity() >= 10 );
    }
    SECTION( "resizing smaller changes size but not capacity" ) {
        v.resize( 0 );

        REQUIRE( v.size() == 0 );
        REQUIRE( v.capacity() >= 5 );
    }
    SECTION( "reserving bigger changes capacity but not size" ) {
        v.reserve( 10 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 10 );
    }
    SECTION( "reserving smaller does not change size or capacity" ) {
        v.reserve( 0 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );
    }
}
```

So, Setup/Tear down is done before/after each section.

Unit Testing Tips

C++ design

- Stuff from above applies to Classes / Functions

- Think about arguments:
  - Code should be hard to use incorrectly.
  - Use `const`, `unsigned` etc.
  - Testing forces you to sort these out.

Test Driven Development (TDD)

- Methodology
  1. Write a test
  2. Run test, should fail
  3. Implement/Debug functionality
  4. Run test
     1. if succeed goto 5
     2. else goto 3
  5. Refactor to tidy up

TDD in practice

- Aim to get good coverage
- Some people quote 70% or more
- What are the downsides?
- Don't write 'brittle' tests

Behaviour Driven Development (BDD)

- Behaviour Driven Development (BDD)
  - Refers to a whole area of software engineering
  - With associated tools and practices
  - Think about end-user perspective
  - Think about the desired behaviour not the implementation
  - See Jani Hartikainen article.

TDD Vs BDD

- TDD
  - Test/Design based on methods available
  - Often ties in implementation details
- BDD

  - Test/Design based on behaviour
  - Code to interfaces (later in course)

- Subtly different
- Aim for BDD

Anti-Pattern 1: Setters/Getters

Testing every Setter/Getter.

Consider:

```
class Atom {

  public:
    void SetAtomicNumber(const int& number) { m_AtomicNumber = number; }
    int GetAtomicNumber() const { return m_AtomicNumber; }
    void SetName(const std::string& name) { m_Name = name; }
    std::string GetName() const { return m_Name; }
  private:
    int m_AtomicNumber;
    std::string m_Name;
};
```

and tests like:

```
TEST_CASE( "Testing Setters/Getters", "[Atom]" ) {

    Atom a;

    a.SetAtomicNumber(1);
    REQUIRE( a.GetAtomicNumber() == 1);
    a.SetName("Hydrogen");
    REQUIRE( a.GetName() == "Hydrogen");
```

- It feels tedious
- But you want good coverage
- This often puts people off testing
- It also produces "brittle", where 1 change breaks many things

Anti-Pattern 1: Suggestion.

- Focus on behaviour.

  - What would end-user expect to be doing?
  - How would end-user be using this class?
  - Write tests that follow the use-case
  - Gives a more logical grouping
  - One test can cover > 1 function
  - i.e. move away from slavishly testing each function
- Minimise interface.
  - Provide the bare number of methods
  - Don't provide setters if you don't want them
  - Don't provide getters unless the user needs something
  - Less to test. Use documentation to describe why.

Anti-Pattern 2: Constructing Dependent Classes

- Sometimes, by necessity we test groups of classes
- Or one class genuinely Has-A contained class
- But the contained class is expensive, or could be changed in future

Anti-Pattern 2: Suggestion

- Read up on Dependency Injection
- Enables you to create and inject dummy test classes
- So, testing again used to break down design, and increase flexibility

Summary BDD Vs TDD

Aim to write:

- Most concise description of requirements as unit tests
- Smallest amount of code to pass tests
- ... i.e. based on behaviour

Any Questions?

Homework - Overview

- Example git repo, CMake, Catch template project:
  - CMakeCatch2 - Simple

  - CMakeCatchTemplate - Complex

- You should
  - Clone, Build.
  - Add unit test in Testing
  - Run via ctest
  - Find log file

Homework - 7

- Imagine a simple function, e.g. to add two numbers.
- Play with unit tests until you understand the difference between:

```
int AddTwoNumbers(int a, int b);
int AddTwoNumbers(const int& a, const int&b);
void AddTwoNumbers(int* a, int*b, int* output);
void AddTwoNumbers(const int* const a, const int* const b);
```

Now imagine, instead of integers, the variables all contained a large Image. Which type of function declaration would you use?

Homework - 8

- Write a Fraction class
- Write a print function to print nicely formatted fractions
- Does the print function live inside or outside of the class?
- Write a method `simplify()` which will simplify the fraction.
- Unit test until you have at least got the hang of unit testing
- Review your function arguments and return types

Chapter 2

Lecture 2: Modern C++ (1)

Lecture 2: Overview

The Story So Far

- Git
- CMake
- Catch2

Todays Lesson

- Recap of C++ features
- OO concepts:
  - Encapsulation and data abstraction
  - Inheritance
  - Polymorphism

Recap of C++ features

C++ is evolving

- C++98
- Introduced templates

- STL containers and algorithms
- Strings and IO/streams

- C++11
- Many new features introduced, feels like a different programming language
- Move semantics
- `auto`
- Lambda functions
- `constexpr`
- Smart pointers
- `std::array`
- Support for multithreading
- Regular expressions
- C++14
- `auto` works in more places
- Generalised lambda functions
- `std::make_unique`
- Reader/writer locks
- C++17
- fold expressions
- `std::any` and `std::variant`
- The Filesystem library
- and more...

C++ constantly evolving, you don't just learn once and then stop you need keep up with language developments. For this course we will be using up until C++14.

Some excercises using

- Now some excercises manipulating `vectors`, using range based for loops and some some algorithms from `<algorithm>`.

Homework - 9

- Use https://github.com/MattClarkson/CMakeLibraryAndApp.git and create a new library to perform various operations on a std::vector
- Using a range base for loop Write a function that prints all the elements on the vector to screen and call this from `myApp`
- Write a function using a range based for loop that counts the number of elements equal to a value 5

- Now repeat but instead use the `std::count` algorithm from the `<algorithm>` library

Homework - 10

- Again using https://github.com/MattClarkson/CMakeLibraryAndApp.git
- Write a function `add_elements(vector<int> v, int val, int ntimes)` that takes a `vector<int>` and appends `ntimes` new elements with value `val`

- Print contents of the input vector to screen before and after calling the function as well from within the `add_elements` function, what is the problem?
- Try passing by reference `add_elements(vector<int> &v, ...` instead, does it work now?
- What are the advantages/disadvantages to passing by reference?
- Try passing as a `const` reference `add_elements(const vector<int> &v, ...`, what happens now and why would you use this?

Object Oriented Review

C-style Programming

- Procedural programming
- Pass data to functions

Function-style Example

```
double compute_similarity(double *imag1, double* image2, double *params)
{
  // stuff
}

double calculate_derivative(double* params)
{
  // stuff
}

double convert_to_decimal(double numerator, double denominator)
{
```

```
  // stuff
}
```

Disadvantages

- Can get out of hand as program size increases
- Can't easily describe relationships between bits of data
- Relies on method documentation, function and variable names
- Can't easily control/(enforce control of) access to data

C Struct

- So, in C, the struct was invented
- Basically a class without methods
- This at least provides a logical grouping

Struct Example

```
struct Fraction {
  int numerator;
  int denominator;
};

double convertToDecimal(const Fraction& f)
{
  return f.numerator/static_cast<double>(f.denominator);
}
```

C++ Class

- C++ provides the class to enhance the language with user defined types
- Once defined, use types as if native to the language

Abstraction

- C++ class mechanism enables you to define a type
  - independent of its data
  - independent of its implementation
  - class defines concept or blueprint
  - instantiation creates object

Abstraction - Grady Booch

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

Class Example

```
class Atom {
public:
  Atom(){};
  ~Atom(){};
  int GetAtomicNumber();
  double GetAtomicWeight();
};

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

Encapsulation

- Encapsulation is:
  - Bundling together methods and data
  - Restricting access, defining public interface
- Describes how you correctly use something

Public/Private/Protected

- For class methods/variables:
  - `private`: only available in this class
  - `protected`: available in this class and derived classes
  - `public`: available to anyone with access to the object
- (public, protected, private inheritance comes later)

Class Example

```
// Defining a User Defined Type.
class Fraction {
```

```
public: // access control
  // How to create
  Fraction();
  Fraction(const int &num, const int &denom);

  // How to destroy
  ~Fraction();

  // How to access
  int numerator() const;
  int denominator() const;

  // What you can do
  const Fraction operator+(const Fraction &another);

private: // access control
  // The data
  int m_Numerator;
  int m_Denominator;
};
```

Inheritance

- Used for:
  - Defining new types based on a common type
- Careful:
  - Beware - "Reduce code duplication, less maintenance"
  - Types in a hierarchy MUST be related
  - Don't over-use inheritance
  - We will cover other ways of object re-use

Class Example

```
class Shape {
public:
  Shape();
  void setVisible(const bool &isVisible) { m_IsVisible = isVisible; }
  virtual void rotate(const double &degrees) = 0;
  virtual void scale(const double &factor) = 0;
  // + other methods
private:
```

```
  bool m_IsVisible;
  unsigned char m_Colour[3]; // RGB
  double m_CentreOfMass[2];
};

class Rectangle : public Shape {
public:
  Rectangle();
  virtual void rotate(const double &degrees);
  virtual void scale(const double &factor);
  // + other methods
private:
  double m_Corner1[2];
  double m_Corner2[2];
};

class Circle : public Shape {
public:
  Circle();
  virtual void rotate(const double &degrees);
  virtual void scale(const double &factor);
  // + other methods
private:
  float radius;
};
```

Polymorphism

- Several types:
  - (normally) "subtype": via inheritance
  - "parametric": via templates
  - "ad hoc": via function overloading
- Common interface to entities of different types
- Same method, different behaviour

Class Example

```
#include "shape.h"
int main(int argc, char** argv)
{
  Circle c1;
  Rectangle r1;
  Shape *s1 = &c1;
```

```
  Shape *s2 = &r1;

  // Calls method in Shape (as not virtual)
  bool isVisible = true;
  s1->setVisible(isVisible);
  s2->setVisible(isVisible);

  // Calls method in derived (as declared virtual)
  s1->rotate(10);
  s2->rotate(10);
}
```

Homework - 11

- Use https://github.com/MattClarkson/CMakeCatch2.git and create a simple shape class for a square with methods to calculate the area
- Create an object of the class from within an app and print the result of the area calculation to screen
- Try to write some unit tests to check the class behaves as expected

Homework - 12

- Again using `CMakeCatch2` as a basis use inheritance and polymorphism to create a shape base class and a set of derived classes for a square, rectangle
- Draw a class inheritance diagram before starting
- Confirm that the get area functions behave differently for the different dervied classes

Chapter 3

Lecture 3: Modern C++ (2)

Lecture 3: Overview

The Story So Far

- Git, CMake, Catch2
- Recap of Modern C++ features
- OO concepts

Todays Lesson

- C++ Standard Library
- Smart pointers and move semantics
- Lambda expressions
- Exceptions

C++ Standard Library

What is it

- A collection of:
- Containers (e.g. `std::vector`, `std::array`, `std::map`, ...)
- Methods to manipulate these containers (e.g. `std::sort`, `std::find_if`)

- Strings (`std::string`) and streams (e.g. `std::cout`, `std::endl`, ...)
- Function objects (e.g. arithmetic operations, comparisons and logical operations, ...)
- Part of the ISO Standard itself so it has to be implemented to be called C++
- Can include in your code by simply adding the relevant `#include <headername>` and then using the `std::` namespace
- A list of these headers can be found at https://en.cppreference.com/w/cpp/header

Homework - 13

- Write short code snippets that use:
- Containers `<array>`, `<vector>` and `<map>`

- The `<random>` number generator library
- The `<iostream>` to read in a string from the terminal and output it to screen

Smart Pointers

Use of Raw Pointers

- Given a pointer passed to a function

```
void DoSomethingClever(int *a)
{
  // write some code
}
```

- How do we use the pointer?
- What problems are there?

Problems with Raw Pointers

- From "Effective Modern C++", Meyers, p117.
  - If you are done, do you destroy it?

  - How to destroy it? Call `delete` or some method first: `a->Shutdown();`
  - Single object or array?
  - `delete` or `delete[]`?
  - How to ensure the whole system only deletes it once?
  - Is it dangling, if I don't delete it?

Use Smart Pointers

- `new/delete` on raw pointers not good enough
- So, use Smart Pointers
  - automatically delete pointed to object
  - explicit control over sharing
  - i.e. smarter
- Smart Pointers model the "ownership"

Further Reading

- Notes here are based on these:
  - David Kieras online paper
  - "Effective Modern C++", Meyers, ch4

Standard Library Smart Pointers

- Here we teach Standard Library
  - std::unique_ptr - models *has-a* but also unique ownership
  - std::shared_ptr - models *has-a* but shared ownership
  - std::weak_ptr - temporary reference, breaks circular references

Stack Allocated - No Leak.

- To recap:

```
#include "Fraction.h"
int main() {
  Fraction f(1,4);
}
```

- Gives:

```
I'm being deleted
```

- So stack allocated objects are deleted, when stack unwinds.

Heap Allocated - Leak.

- To recap:

```
#include "Fraction.h"
int main() {
  Fraction *f = new Fraction(1,4);
}
```

- Gives:

- So heap allocated objects are not deleted.
- Its the pointer (stack allocated) that's deleted.

Unique Ptr - Unique Ownership

- So:

```
#include "Fraction.h"
#include <memory>
int main() {
  std::unique_ptr<Fraction> f(new Fraction(1,4));
}
```

- Gives:

```
I'm being deleted
```

- And object is deleted.
- Is that it?

Unique Ptr - Move?

- Does move work?

```
#include "Fraction.h"
#include <memory>
#include <iostream>

int main() {
  std::unique_ptr<Fraction> f(new Fraction(1,4));
  // std::unique_ptr<Fraction> f2(f); // compile error

  std::cerr << "f=" << f.get() << std::endl;

  std::unique_ptr<Fraction> f2;
  // f2 = f; // compile error
  // f2.reset(f.get()); // bad idea

  f2.reset(f.release());
  std::cout << "f=" << f.get() << ", f2=" << f2.get() << std::endl;

  f = std::move(f2);
  std::cout << "f=" << f.get() << ", f2=" << f2.get() << std::endl;
}
```

- Gives:

```
f=0, f2=0x1712010
f=0x1712010, f2=0
I'm being deleted
```

- We see that API makes difficult to use incorrectly.

Unique Ptr - Usage 1

- Forces you to think about ownership
  - No copy constructor
  - No assignment
- Consequently
  - Can't pass pointer by value
  - Use move semantics for placing in containers

Unique Ptr - Usage 2

- Put raw pointer STRAIGHT into unique_ptr
- see `std::make_unique` in C++14.

```
#include "Fraction.h"
#include <memory>
int main() {
  std::unique_ptr<Fraction> f(new Fraction(1,4));
}
```

Shared Ptr - Shared Ownership

- Many pointers pointing to same object
- Object only deleted if no pointers refer to it
- Achieved via reference counting

Shared Ptr Control Block

- Won't go to too many details:
- From "Effective Modern C++", Meyers, p140

Shared Ptr - Usage 1

- Place raw pointer straight into shared_ptr
- Pass to functions, reference or by value
- Copy/Move constructors and assignment all implemented

Shared Ptr - Usage 2

```
#include "Fraction.h"
#include <memory>
#include <iostream>
void divideBy2(const std::shared_ptr<Fraction>& f)
{
  f->denominator *= 2;
}
void multiplyBy2(const std::shared_ptr<Fraction> f)
{
  f->numerator *= 2;
```

```
}
int main() {
  std::shared_ptr<Fraction> f1(new Fraction(1,4));
  std::shared_ptr<Fraction> f2 = f1;
  divideBy2(f1);
  multiplyBy2(f2);
  std::cout << "Value=" << f1->numerator << "/" << f1->denominator << std::endl;
  std::cout << "f1=" << f1.get() << ", f2=" << f2.get() << std::endl;
}
```

Shared Ptr - Usage 3

- Watch out for exceptions.
- "Effective Modern C++", Meyers, p140

```
#include "Fraction.h"
#include <memory>
#include <stdexcept>
#include <vector>
int checkSomething(const std::shared_ptr<Fraction>& f, const int& i)
{
  // whatever.
}
int computeSomethingFirst()
{
  // what if this throws?
}
int main()
{
  std::vector<std::shared_ptr<Fraction> >  spaceForLotsOfFractions;
  int result = checkSomething(std::shared_ptr<Fraction>(new Fraction(1,4)),
                              computeSomethingFirst()
                             );
}
```

Shared Ptr - Usage 4

- Prefer `std::make_shared`
- Exception safe

```
#include "Fraction.h"
#include <memory>
```

```
#include <stdexcept>
#include <vector>
int checkSomething(const std::shared_ptr<Fraction>& f, const int& i)
{
  // whatever.
}
int computeSomethingFirst()
{
  // what if this throws?
}
int main()
{
  std::vector<std::shared_ptr<Fraction> >  spaceForLotsOfFractions;
  int result = checkSomething(std::make_shared<Fraction>(1,4),
                              computeSomethingFirst()
                             );
}
```

Weak Ptr - Why?

- Like a shared pointer, but doesn't actually own anything
- Use for example:
  - Caches
  - Break circular pointers
- Limited API
- Not terribly common as most code ends up as hierarchies

Weak Ptr - Example

- See David Kieras online paper

```
#include "Fraction.h"
#include <memory>
#include <iostream>
int main() {
  std::shared_ptr<Fraction> s1(new Fraction(1,4));
  std::weak_ptr<Fraction> w1;      // can point to nothing
  std::weak_ptr<Fraction> w2 = s1; // assignment from shared
  std::weak_ptr<Fraction> w3(s1);  // construction from shared

  // Can't be de-referenced!!!
  // std::cerr << "Value=" << w1->numerator << "/" << w1->denominator << std::endl;
```

```
  // Needs converting to shared, and checking
  std::shared_ptr<Fraction> s2 = w1.lock();
  if (s2)
  {
    std::cout << "Object w1 exists=" << s2->numerator << "/" << s2->denominator << std::endl
  }

  // Or, create shared, check for exception
  std::shared_ptr<Fraction> s3(w2);
  std::cout << "Object must exists=" << s3->numerator << "/" << s3->denominator << std::endl
}
```

Final Advice

- Benefits of immediate, fine-grained, garbage collection
- Just ask Scott Meyers!
  - Use `unique_ptr` for unique ownership
  - Easy to convert `unique_ptr` to `shared_ptr`
  - But not the reverse
  - Use `shared_ptr` for shared resource management
  - Avoid raw `new` - use `make_shared`, `make_unique`
  - Use `weak_ptr` for pointers that can dangle (cache etc)

Conclusion for Smart Pointers

- Default to standard library, check compiler
- Lots of other Smart Pointers
  - Boost (use STL).
  - ITK
  - VTK
  - Qt Smart Pointers
- Don't be tempted to write your own
- Always read the manual
- Always consistently use it

Homework - 14

- Create a memory leak using bare pointers and `new` to heap allocate and then repeat using using smart pointers

- Think of a use case where a resource should be uniquely owned and implement as a short application
- Think of a use case where the resource is shared between multiple and implement
- *Note: in lecture 6 you will learn how to use tools such as* ***`valgrind`*** *to consistently check for memory leaks*

Lambda expressions

Game changer for C++

- Lambda expressions allow you to create an unnamed function object within code
- Expression can be defined as they are passed to a function as an argument
- Useful for varions STL `_if` and comparison functions: `std::find_if`, `std::remove_if`, `std::sort`, `std::lower_bound` etc
- Can be used for a one off call for a context specific function
- From "Effective Modern C++", Meyers, p215
- A game changer for C++ despite bringing no new expressive power to the language
- Everything you can do with a lambda could be done by hand with a bit more typing
- But the impact on day to day C++ software development is enormous

- Allow expressions to be defined as they are being passed as an argument to a function

```
std::find_if(container.begin(), container.end(),
             [](int val) { return 0 < val && val < 10; });
```

Basic syntax

- `[ captures ] { body };`
- `captures` is comma separated list of variable from enclosing scope that the lambda can use
- `body` is where the function is defined

- `[x,y] { return x + y; }`
- Captures can be by value `[x]` or by reference `[&x]`
- `[=]` and '[&] are default capture modes for all variables in enclosing scope -> discouraged as can lead to dangling references and are not thread safe ("Effective Modern C++", Meyers, p216)
- `[ captures ] ( params ) { body };`
- `params` list of params as with named functions except cannot have default value
- `[] (int x, int y) { return x*x + y*y; }(1,2)` would return `5`
- `[ captures ] ( params ) -> ret { body };`
- `ret` is the return type, if not specified inferred from the return statement in the function
- `[] () -> float { return "a"; }` would give `error: cannot convert const char* to float in return`
- It is possible to copy and reuse lambdas

```
auto c1 = [](int y) { return y*y; };
auto c2 = c1; // c2 is a copy of c1
cout << "c1(2) = " << c1(2) << endl;
cout << "c2(4) = " << c2(4) << endl;
```

- Gives

```
c1(2) = 4
c2(4) = 16
```

Example use

```
std::vector<int> v { 1,2,3,4,5,6,7,8,9,10 };
int neven = std::count_if(v.begin(), v.end(), [](int i){ return i % 2 == 0; });
```

Homework 15

- Create your own lambda expressions for each of the three basic syntax examples given above
- Try to change a param from within, can you see a different behaviour if passed by reference or by value?
- Use `std::count_if` with an appropriate lambda expression to count the number of values in a `vector<int>` that are divisable by 2 or 3

Homework 16

- Create a simple `Student` class that has public member variables storing `string firstname`, `string secondname` and `int age`
- Create a vector and fill it with various instances of the `Student` class
- Create a sorting class `StudentSort` and that has a method `vector<Student> SortByAge(vector<Student> vs);` that returns a `vector<Student>` that has been sorted by age
- Use a `std::sort` and a lambda expression for this
- Add a `bool` switch to `SortByAge(vector<Student> vs, bool reverse = false)` that reverses the sort

Error Handling

Exceptions

- Exceptions are the C++ or Object Oriented way of Error Handling

Exception Handling Example

```
#include <stdexcept>
#include <iostream>
bool someFunction() { return false; }

int main()
{
  try
  {
    bool isOK = false;
    isOK = someFunction();
    if (!isOK)
    {
      throw std::runtime_error("Something is wrong");
    }
  }
  catch (std::exception& e)
  {
    std::cerr << "Caught Exception:" << e.what() << std::endl;
  }
}
```

What's the Point?

- A good summary here:
  - Have separated error handling logic from application logic
  - Forces calling code to recognize an error condition and handle it
  - Stack-unwinding destroys all objects in scope according to well-defined rules
  - A clean separation between code that detects error and code that handles error
- First, lets look at C-style return codes

Error Handling C-Style

```
int foo(int a, int b)
{
  // stuff

  if(some error condition)
  {
    return 1;
  } else if (another error condition) {
    return 2;
  } else {
    return 0;
  }
}

void caller(int a, int b)
{
 int result = foo(a, b);
 if (result == 1) // do something
 else if (result == 2) // do something difference
 else
 {
   // All ok, continue as you wish
 }
}
```

Outcome

- Can be perfectly usable
- Depends on depth of function call stack
- Depends on complexity of program

- If deep/large, then can become unweildy

Error Handling C++ Style

```
#include <stdexcept>
#include <iostream>

int ReadNumberFromFile(const std::string& fileName)
{
  if (fileName.length() == 0)
  {
    throw std::runtime_error("Empty fileName provided");
  }

  // Check for file existence etc. throw io errors.

  // do stuff
  return 2; // returning dummy number to force error
}


void ValidateNumber(int number)
{
  if (number < 3)
  {
    throw std::logic_error("Number is < 3");
  }
  if (number > 10)
  {
    throw std::logic_error("Number is > 10");
  }
}


int main(int argc, char** argv)
{
  try
  {
    if (argc < 2)
    {
      std::cerr << "Usage: " << argv[0] << " fileName" << std::endl;
      return EXIT_FAILURE;
    }

    int myNumber = ReadNumberFromFile(argv[1]);
```

```
    ValidateNumber(myNumber);

    // Compute stuff.

    return EXIT_SUCCESS;

  }
  catch (std::exception& e)
  {
    std::cerr << "Caught Exception:" << e.what() << std::endl;
  }
}
```

Comments

- Code that throws does not worry about the catcher
- Exceptions are classes, can carry data
- More suited to larger libraries of re-usable functions
- Many different catchers, all implementing different error handling
- Generally scales better, more flexible

Practical Tips For Exception Handling

- Decide on error handling strategy at start
- Use it consistently
- Create your own base class exception
- Derive all your exceptions from that base class
- Stick to a few obvious classes, not one class for every single error

Homework 17

- Taking the `Fraction` class from homework 8:
  - Try to call `simplify` for a fraction with a denominator of 0 and see what exception is thrown
    - Try to catch this exception from the calling code
  - Create your own exception class that is thrown instead. It should inherit from `std::exception` (see cppreference/error/exception)
    - Catch this from the calling code
  - Create the fraction and call `simplify` from within a function that is called from the main calling code

    - Check you can catch the exception either in the calling code or from within the function

More Practical Tips For Exception Handling

- Look at C++ standard classes and tutorial
- An exception macro may be useful, e.g. mitk::Exception and mithThrow()

Chapter 4

Lecture 4: Modern C++ (3)

Patterns and Templates

The Story So Far

- Git, CMake, Catch2
- Recap of Modern C++ features
- OO concepts
- C++ Standard Library
- Smart pointers and move semantics
- Lambda expressions
- Error handling

Todays Lesson

- How to assemble/organise classes
  - Beginner mistakes
    - 1 class - all functionality
    - Deep inheritance trees
- Patterns
  - RAII
  - Common OO design patterns
- Templates

- Function templates
- Class templates
- Template specialisation

Program To Interfaces

Why?

- In research code we often "just start hacking"
- You tend to mix interface and implementation
- Results in client/user of a class having implicit dependency on the implementation
- So, define a pure virtual class, not for inheritance, but for clean API

Example

```
#include <memory>
#include <vector>
#include <string>

class DataPlayerI {
public:
  virtual void StartPlaying() = 0;
  virtual void StopPlaying() = 0;
};

class FileDataPlayer : public DataPlayerI {
public:
  FileDataPlayer(const std::string& fileName){}; // opens file    (RAII)
  ~FileDataPlayer(){};                           // releases file (RAII)
public:
  virtual void StartPlaying() {};
  virtual void StopPlaying() {};
};

class Experiment {
public:
  Experiment(DataPlayerI *d) { m_Player.reset(d); } // takes ownership
  void Run() {};
  std::vector<std::string> GetResults() const {};
private:
  std::unique_ptr<DataPlayerI> m_Player;
};
```

```
int main(int argc, char** argv)
{
  FileDataPlayer fdp(argv[1]); // Or some class WebDataPlayer derived from DataPlayerI
  Experiment e(&fdp);
  e.Run();

  // etc.
}
```

Comments

- Useful between sub-components of a system
  - GUI front end, Web back end
  - Logic and Database
- Is useful in general to force loose connections between areas of code
  - e.g. different libraries that have different dependencies
  - define an interface that just exports standard types
  - stops the spread of dependencies
  - Lookup Pimpl idiom

Inheritance

Don't overuse Inheritance

- Inheritance is not just for saving duplication
- It MUST represent derived/related types
- Derived class must truely represent 'is-a' relationship
- eg 'Square' is-a 'Shape'
- Deep inheritance hierarchies are almost always wrong
- If something 'doesn't quite fit' check your inheritance

Surely Its Simple?

- Common example: Square/Rectangle problem, here

```
#include <iostream>

class Rectangle {
public:
```

```
  Rectangle() : m_Width(0), m_Height(0) {};
  virtual ~Rectangle(){};
  int GetArea() const { return m_Width*m_Height; }
  virtual void SetWidth(int w) { m_Width=w; }
  virtual void SetHeight(int h) { m_Height=h; }
protected:
  int m_Width;
  int m_Height;
};

class Square : public Rectangle {
public:
  Square(){};
  ~Square(){};
  virtual void SetWidth(int w) { m_Width=w; m_Height=w; }
  virtual void SetHeight(int h) { m_Width=h; m_Height=h; }
};

int main()
{
  Rectangle *r = new Square();
  r->SetWidth(5);
  r->SetHeight(6);
  std::cout << "Area = " << r->GetArea() << std::endl;
}
```

Liskov Substitution Principal

- Wikipedia
- "if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program"
- i.e. is S is derived class from base class T then objects of type T should be able to be replaced with objects of type S
- Can something truely be substituted?
- If someone else filled a vector of type T, would I care what type I have?

What to Look For

- If you have:
  - Methods you don't want to implement in derived class
  - Methods that don't make sense in derived class
  - Methods that are unneeded in derived class

  - If you have a list of something, and someone else swapping in a derived type would cause problems
- Then you have probably got your inheritance wrong

Composition Vs Inheritance

- Lots of Info online eg. wikipedia
- In basic OO Principals
  - 'Has-a' means 'pointer or reference to'
  - eg.`Car` has-a `Engine`
- But there is also:
- Composition: Strong 'has-a'. Component parts are owned by thing pointing to them.
- Aggregation: Weak 'has-a'. Component part has its own lifecycle.
- Association: General term, referring to either composition or aggregation, just a 'pointer-to'

Examples

- House 'has-a' set of Rooms. Destroying House, you destroy all Room objects. No point having a House on its own.
- Room 'has-a' Computer. If room is being refurbished, Computer should not be thrown away. Can go to another room

But Why?

- Good article: Choosing Composition or Inheritance
- Composition is more flexible
- Inheritance has much tighter definition than you realise

Dependency Injection

Construction

- What could be wrong with this:

```
#include <memory>

class Bar {
```

```
};

class Foo {
public:
  Foo()
  {
    m_Bar = new Bar();
  }

private:
  Bar* m_Bar;
};

int main()
{
  Foo a;
}
```

Unwanted Dependencies

- If constructor instantiates class directly:
  - Hard-coded class name
  - Duplication of initialisation code

Dependency Injection

- Read Martin Fowler's Inversion of Control Containers and the Dependency Injection Pattern
- Type 2 - Constructor Injection
- Type 3 - Setter Injection

Constructor Injection Example

```
#include <memory>

class Bar {
};

class Foo {
public:
  Foo(Bar* b)
```

```
  : m_Bar(b)
  {
  }

private:
  Bar* m_Bar;
};

int main()
{
  Bar b;
  Foo a(&b);
}
```

Setter Injection Example

```
#include <memory>

class Bar {
};

class Foo {
public:
  Foo()
  {
  }
  void SetBar(Bar *b) { m_Bar = b; }

private:
  Bar* m_Bar;
};

int main()
{
  Bar b;
  Foo a();
  a.SetBar(&b);
}
```

Question: Which is better?

Advantages of Dependency Injection

- Using Dependency Injection
  - Removes hard coding of `new ClassName`
  - Creation is done outside class, so class only uses public API
  - Leads towards fewer assumptions in the code

Homework 18

- Taking the `Student` class from homework 16:
- Create a new `Laptop` class that has a string `os` data member for the operating system name and an integer `year` data member for the year produced.
- `Laptop` should have both a default constructor that sets `year` to 0 and name to "Not set" as well as an overloaded constructor that initialises both `year` and `os`
- Modify `Student` to have a `Laptop` data member
- Try out the two types of dependency injection above: constructor, setter
- Confirm that the `Student` class is now invarient to changes in how you instantiate `Laptop`

RAII Pattern

What is it?

- Resource Allocation Is Initialisation (RAII)
- Obtain all resources in constructor
- Release them all in destructor

Why is it?

- Guaranteed fully initialised object once constructor is complete
- Objects on stack are guaranteed to be destroyed when an exception is thrown and stack is unwound
  - Including smart pointers to objects

Example

- You may already be using it: STL example
- Another example

Homework 19

- Create a simple class `Foo` that contains a data member that is a raw pointer `bptr` to another class `Bar` that contains an integer as a data member
- Add a `std::cout` to the constructor and destructor of both classes so that you know when they have been called
- Implement the RAII pattern to create and destroy the `Bar` object that `bptr` points to
- Create an instance of `Foo foo` in your application and confirm that if an exception is thrown before Foo goes out of scope that the destructor for both `Foo` and `Bar` are called and the `Bar` object is released

Construction Patterns

Constructional Patterns

- Other methods include
  - See Gang of Four book
  - Strategy Pattern
  - Factory Pattern
  - Abstract Factory Pattern
  - Builder Pattern

- Also look up Service Locator Pattern

Managing Complexity

- Rather than monolithic code (bad)
- We end up with many smaller classes (good)
- So, its more flexible (good)
- It does look more complex at first (don't panic)
- You get used to thinking in small objects

Using Templates

What Are Templates?

- C++ templates allow functions/classes to operate on generic types.
- See: Generic Programming.

- Write code, where 'type' is provided later
- Types instantiated at compile time, as they are needed
- (Remember, C++ is strongly typed)

You May Already Use Them!

You probably use them already. Example type (class):

```
std::vector<int> myVectorInts;
```

Example algorithm: C++ sort

```
std::sort(myVectorInts.begin(), myVectorInts.end());
```

Aim: Write functions, classes, in terms of future/other/generic types, type provided as parameter.

Why Are Templates Useful?

- Generic programming:
  - not pre-processor macros
  - so maintain type safety
  - separate algorithm from implementation
  - extensible, optimisable via Template Meta-Programming (TMP)

Book

- For more infomation see "Modern C++ Design"
- 2001, but still excellent text on templates, meta-programming, policy based design etc.
- This section of course, gives basic introduction for research programmers

Are Templates Difficult?

- Some say: notation is ugly
  - Does take getting used to
  - Use `typedef` to simplify
- Some say: verbose, confusing error messages
  - Nothing intrinsically difficult

  - Take small steps, compile regularly
  - Learn to think like a compiler
- Code infiltration
  - Use sparingly
  - Hide usage behind clean interface

Why Templates in Research?

- Minimise code duplication: think about how useful `std::vector` is
- For example methods that generalise from 2D, 3D, n-dimensions, (e.g. ITK)
- Test numerical code with simple types, apply to complex/other types

Why Teach Templates?

- Standard Template Library uses them
- More common in research code, than business code
- In research, more likely to 'code for the unknown'
- Boost, Qt, EIGEN uses them

Function Templates

Function Templates Example

- Credit to www.cplusplus.com

```
// function template
#include <iostream>
using namespace std;

template <class T> // class|typename
T sum (T a, T b)
{
  T result;
  result = a + b;
  return result;
}

int main () {
  int i=5, j=6;
```

```
  double f=2.0, g=0.5;
  cout << sum<int>(i,j) << '\n';
  cout << sum<double>(f,g) << '\n';
  return 0;
}
```

- And produces this output when run

```
11
2.5
```

Why Use Function Templates?

- Instead of function overloading
  - Reduce your code duplication
  - Reduce your maintenance
  - Reduce your effort
  - Also see this Additional tutorial.

Language Definition 1

- From the language reference

```
template < parameter-list > function-declaration
```

- so

```
template < class T >  // note 'class'
void MyFunction(T a, T b)
{
  // do something
}
```

- or

```
template < typename T1, typename T2 >  // note 'typename'
T1 MyFunctionTwoArgs(T1 a, T2 b)
{
  // do something
}
```

Language Definition 2

- Also
  - Can use `class` or `typename`.
  - I prefer `typename`.
  - Template parameter can apply to references, pointers, return types, arrays etc.

Default Argument Resolution

- Given:

```
double GetAverage<typename T>(const std::vector<T>& someNumbers);
```

- then:

```
std::vector<double> myNumbers;
double result = GetAverage(myNumbers);
```

- will call:

```
double GetAverage<double>(const std::vector<double>& someNumbers);
```

- So, if function parameters can inform the compiler uniquely as to which function to instantiate, its automatically compiled.

Explicit Argument Resolution - part 1

- However, given:

```
double GetAverage<typename T>(const T& a, const T& b);
```

- and:

```
int a, b;
int result = GetAverage(a, b);
```

- But you don't want the int version called (due to integer division perhaps), you can:

```
double result = GetAverage<double>(a, b);
```

Explicit Argument Resolution - part 2

- equivalent to

```
GetAverage<double>(static_cast<double>(a), static_cast<double>(b));
```

- i.e. name the template function parameter explicitly.
- Cases for Explicit Template Argument Specification
  - Force compilation of a specific version (eg. int as above)
  - Also if method parameters do not allow compiler to deduce anything eg. `PrintSize()` method.

Beware of Code Bloat

- Given:

```
double GetMax<typename T1, typename T2>(const &T1, const &T2);
```

- and:

```
double r1 = GetMax(1,2);
double r2 = GetMax(1,2.0);
double r3 = GetMax(1.0,2.0);
```

- The compiler will generate 3 different max functions.
- Be Careful
  - Executables/libraries get larger
  - Compilation time will increase
  - Error messages get more verbose

Two Stage Compilation

- Basic syntax checking (eg. brackets, semi-colon, etc), when `#include`'d
- But only compiled when instantiated (eg. check existence of + operator).

Instantiation

- Object Code is only really generated if code is used
- Template functions can be
  - .h file only
  - .h file that includes separate .cxx/.txx/.hxx file (e.g. ITK)
  - .h file and separate .cxx/.txx file (sometimes by convention a .hpp file)
- In general
  - Most libraries/people prefer header only implementations

Explicit Instantiation - part 1

- Language Reference here
- Microsoft Example
- Given (library) header:

```
#ifndef explicitInstantiation_h
#define explicitInstantiation_h
template <typename T> void f(T s);
#endif
```

- Given (library) implementation:

```
#include <iostream>
#include <typeinfo>
#include "explicitInstantiation.h"
template<typename T>
void f(T s)
{
    std::cout << typeid(T).name() << " " << s << '\n';
}
template void f<double>(double); // instantiates f<double>(double)
template void f<>(char); // instantiates f<char>(char), template argument deduced
template void f(int); // instantiates f<int>(int), template argument deduced
```

Explicit Instantiation - part 2

- Given client code:

```
#include <iostream>
#include "explicitInstantiation.h"

int main(int argc, char** argv)
{
  std::cout << "Matt, double 1.0=" << std::endl;
  f(1.0);
  std::cout << "Matt, char a=" << std::endl;
  f('a');
  std::cout << "Matt, int 2=" << std::endl;
  f(2);
//  std::cout << "Matt, float 3.0=" << std::endl;
//  f<float>(static_cast<float>(3.0));  // compile error
}
```

- We get:

```
Matt, double 1.0=
d 1
Matt, char a=
c a
Matt, int 2=
i 2
```

Explicit Instantiation - part 3

- Explicit Instantiation:
  - Forces instantiation of the function
  - Must appear after the definition
  - Must appear only once for given argument list
  - Stops implicit instantiation
- So, mainly used by compiled library providers
- Clients then `#include` header and link to library

```
Linking CXX executable explicitInstantiationMain.x
Undefined symbols for architecture x86_64:
  "void f<float>(float)", referenced from:
```

Implicit Instantiation - part 1

- Instantiated as they are used

- Normally via `#include` header files.
- Given (library) header, that containts implementation:

```
#ifndef explicitInstantiation_h
#define explicitInstantiation_h
#include <iostream>
#include <typeinfo>
template <typename T> void f(T s) { std::cout << typeid(T).name() << " " << s << '\n'; }
#endif
```

Implicit Instantiation - part 2

- Given client code:

```
#include <iostream>
#include "implicitInstantiation.h"

int main(int argc, char** argv)
{
  std::cout << "Matt, double 1.0=" << std::endl;
  f(1.0);
  std::cout << "Matt, char a=" << std::endl;
  f('a');
  std::cout << "Matt, int 2=" << std::endl;
  f(2);
  std::cout << "Matt, float 3.0=" << std::endl;
  f<float>(static_cast<float>(3.0));  // no compile error
}
```

- We get:

```
Matt, double 1.0=
d 1
Matt, char a=
c a
Matt, int 2=
i 2
Matt, float 3.0=
f 3
```

Homework 19

- Write a template function `AGreaterThanB` that compares two input agruments of type `T` and returns a `bool` if `A` is greater than `B`. The function

should be able to handle either `int`, `float` or `string` entries (for the latter you will need to decide how to rank by size)
- Try out the different types of explicit and implicit instantiation
- Advanced/optional: write a template function that performs a binary search on the contents of a vector containing either `int`, `float` or `string` entries (you could adapt this equivalent to) code, N.B. the `vector` will need to be sorted by size

Class Templates

Class Templates Example - part 1

- If you understand template functions, then template classes are easy!
- Refering to this tutorial, an example:

Header:

```
template <typename T> class MyPair {
  T m_Values[2];

public:
  MyPair(const T &first, const T &second);
  T getMax() const;
};
#include "pairClassExample.cc"
```

Class Templates Example - part 2

Implementation:

```
template <typename T>
MyPair<T>::MyPair(const T& first, const T& second)
{
  m_Values[0] = first;
  m_Values[1] = second;
}

template <typename T>
T
MyPair<T>::getMax() const
{
  if (m_Values[0] > m_Values[1])
```

```
    return m_Values[0];
  else
    return m_Values[1];
}
```

Class Templates Example - part 3

Usage:

```
#include "pairClassExample.h"
#include <iostream>

int main(int argc, char** argv)
{
  MyPair<int> a(1,2);
  std::cout << "Max is:" << a.getMax() << std::endl;
}
```

Quick Comments

- Implementation, 3 uses of parameter T
- Same Implicit/Explicit instantiation rules
- Note implicit requirements, eg. operator >
  - Remember the 2 stage compilation
  - Remember code not instantiated until its used
  - Take Unit Testing Seriously!

Template Specialisation

- If template defined for type T
- Full specialisation - special case for a specific type eg. char
- Partial specialisation - special case for a type that still templates, e.g. T*

```
template <typename T> class MyVector {
template <> class MyVector<char> {  // full specialisation
template <typename T> MyVector<T*> { // partial specialisation
```

Homework 20

- Implement the above class `MyPair` template
- Try out with both Implicit and Explicit instantiation

- Add a `Swap()` method that switches the contents of `m_Values[0]` and `m_Values[1]`

Summary

Putting It Together - Subsystems

- Program to interfaces
- Inject dependencies
- Always make fully initialised object
- Always use smart pointers
- Use exceptions for errors
- RAII very useful
- Results in a more flexible, extensible, robust design

Putting It Together - OOP

- Prefer composition over inheritance
- Use constructional patterns to assemble code