

# Research Computing with C++

Matt Clarkson

Jonathan Cooper  
James Hetherington

Roma Klapaukh

# Contents

|                                      |           |
|--------------------------------------|-----------|
| <b>1 C++ for Research</b>            | <b>29</b> |
| Course Overview . . . . .            | 29        |
| Part 1 . . . . .                     | 29        |
| Part 2 . . . . .                     | 29        |
| Course Aims . . . . .                | 29        |
| Pre-requisites . . . . .             | 30        |
| Course Notes . . . . .               | 30        |
| Course Assessment . . . . .          | 30        |
| Course Community . . . . .           | 30        |
| C++ In Research . . . . .            | 30        |
| Problems In Research . . . . .       | 30        |
| C++ Disadvantages . . . . .          | 31        |
| C++ Advantages . . . . .             | 31        |
| Research Programming . . . . .       | 31        |
| Development Methodology? . . . . .   | 31        |
| Approach . . . . .                   | 32        |
| 1. Types of Code . . . . .           | 32        |
| 2. Maximise Your Value . . . . .     | 32        |
| 3. Ask Advice . . . . .              | 32        |
| Another Example - NiftyCal . . . . . | 33        |
| Debunking The Excuses . . . . .      | 33        |
| What Isn't This Course? . . . . .    | 33        |

|   |    |
|---|----|
| What Is This Course? . . . . .                  | 33 |
| Git . . . . .                                   | 34 |
| Git Introduction . . . . .                      | 34 |
| Git Resources . . . . .                         | 34 |
| Git Walk Through - 1 . . . . .                  | 34 |
| Git Walk Through - 2 . . . . .                  | 34 |
| Git Walk Through - 3 . . . . .                  | 35 |
| Homework - 1 . . . . .                          | 35 |
| CMake . . . . .                                 | 35 |
| Ever worked in Industry? . . . . .              | 35 |
| Ever worked in Research? . . . . .              | 35 |
| Research Software Engineering Dilemma . . . . . | 36 |
| Build Environment . . . . .                     | 36 |
| CMake Introduction . . . . .                    | 36 |
| CMake Usage Linux/Mac . . . . .                 | 36 |
| CMake Usage Windows . . . . .                   | 37 |
| Homework - 2 . . . . .                          | 37 |
| Homework - 3 . . . . .                          | 37 |
| CMake Basics . . . . .                          | 38 |
| Compiling Basics . . . . .                      | 38 |
| How does a Compiler Work? . . . . .             | 38 |
| CMake - Directory Structure . . . . .           | 38 |
| CMake - Define Targets . . . . .                | 38 |
| CMake - Order Dependent . . . . .               | 39 |
| Homework - 4 . . . . .                          | 39 |
| Intermediate CMake . . . . .                    | 39 |
| What's next? . . . . .                          | 39 |
| Homework - 5 . . . . .                          | 40 |
| Homework - 6 . . . . .                          | 40 |
| Looking forward . . . . .                       | 40 |
| Unit Testing . . . . .                          | 40 |

|  |    |
|--|----|
| What is Unit Testing? . . . . .                          | 40 |
| Benefits of Unit Testing? . . . . .                      | 41 |
| Drawbacks for Unit Testing? . . . . .                    | 41 |
| Unit Testing Frameworks . . . . .                        | 41 |
| Unit Testing Example . . . . .                           | 42 |
| How To Start . . . . .                                   | 42 |
| C++ Frameworks . . . . .                                 | 42 |
| Worked Example . . . . .                                 | 42 |
| Code . . . . .   | 42 |
| Principles . . . . .                                     | 43 |
| Catch / GoogleTest . . . . .                             | 43 |
| Tests That Fail . . . . .                                | 44 |
| Fix the Failing Test . . . . .                           | 44 |
| Test Macros . . . . .                                    | 44 |
| Testing for Failure . . . . .                            | 45 |
| Setup/Tear Down . . . . .                                | 45 |
| Setup/Tear Down in Catch . . . . .                       | 46 |
| Unit Testing Tips . . . . .                              | 46 |
| C++ design . . . . .                                     | 46 |
| Test Driven Development (TDD) . . . . .                  | 47 |
| TDD in practice . . . . .                                | 47 |
| Behaviour Driven Development (BDD) . . . . .             | 47 |
| TDD Vs BDD . . . . .                                     | 47 |
| Anti-Pattern 1: Setters/Getters . . . . .                | 48 |
| Anti-Pattern 1: Suggestion. . . . .                      | 48 |
| Anti-Pattern 2: Constructing Dependent Classes . . . . . | 49 |
| Anti-Pattern 2: Suggestion . . . . .                     | 49 |
| Summary BDD Vs TDD . . . . .                             | 49 |
| Any Questions? . . . . .                                 | 49 |
| Homework - Overview . . . . .                            | 49 |
| Homework - 7 . . . . .                                   | 50 |
| Homework - 8 . . . . .                                   | 50 |

|                                       |           |
|---------------------------------------|-----------|
| <b>2 Including Libraries</b>          | <b>51</b> |
| Why use Libraries? . . . . .          | 51        |
| Scientific Coding . . . . .           | 51        |
| What are libraries? . . . . .         | 51        |
| Aim for this chapter . . . . .        | 51        |
| Choosing libraries . . . . .          | 52        |
| Libraries are Awesome . . . . .       | 52        |
| Libraries for Efficiency . . . . .    | 52        |
| Software Licenses . . . . .           | 52        |
| Third Party Licenses . . . . .        | 52        |
| Third Party Licenses . . . . .        | 53        |
| Choosing a License . . . . .          | 53        |
| Choose Stability . . . . .            | 53        |
| Is Library Updated? . . . . .         | 54        |
| Are Developers Reachable? . . . . .   | 54        |
| Is Code Tested? . . . . .             | 54        |
| Is Library High Quality? . . . . .    | 54        |
| Library Features . . . . .            | 54        |
| Summary . . . . .                     | 55        |
| Library Basics . . . . .              | 55        |
| Aim . . . . .                         | 55        |
| Linux/Mac . . . . .                   | 55        |
| Compiler switches . . . . .           | 55        |
| Check Native Build Platform . . . . . | 56        |
| Windows Compiler Switches . . . . .   | 56        |
| Location Issues . . . . .             | 56        |
| Compilation Issues . . . . .          | 56        |
| A Few Good Libraries . . . . .        | 57        |
| Linking libraries . . . . .           | 57        |
| Linking . . . . .                     | 57        |
| Static Linking . . . . .              | 57        |

|   |    |
|---|----|
| Dynamic Linking . . . . .               | 57 |
| Dynamic Loading . . . . .               | 58 |
| Space Comparison . . . . .              | 58 |
| For Scientists . . . . .                | 58 |
| Packaging . . . . .                     | 58 |
| How to Check . . . . .                  | 58 |
| Using libraries . . . . .               | 59 |
| Package Managers . . . . .              | 59 |
| Windows . . . . .                       | 59 |
| Package Managers . . . . .              | 59 |
| Problems . . . . .                      | 59 |
| Build Your Own . . . . .                | 60 |
| External / Individual Build . . . . .   | 60 |
| Meta-Build / Super-Build . . . . .      | 60 |
| Pro's / Con's . . . . .                 | 61 |
| Examples . . . . .                      | 61 |
| Reminder . . . . .                      | 61 |
| Header Only? . . . . .                  | 61 |
| CMake - Example . . . . .               | 61 |
| CMake - Header Only . . . . .           | 62 |
| CMake - Header Only, External . . . . . | 62 |
| CMake - find_package . . . . .          | 62 |
| Types of find_package . . . . .         | 63 |
| find_package - refer to docs . . . . .  | 63 |
| find_package - Result . . . . .         | 63 |
| find_package - Usage . . . . .          | 63 |
| find_package - Fixing . . . . .         | 63 |
| find_package - Tailoring . . . . .      | 64 |
| Provide Build Flags . . . . .           | 64 |
| Check Before Committing . . . . .       | 64 |
| Summary . . . . .                       | 65 |

|                                 |           |
|---------------------------------|-----------|
| CMakeCatch2 . . . . .           | 65        |
| Intro . . . . .                 | 65        |
| Homework - 9a . . . . .         | 65        |
| Homework - 9b . . . . .         | 65        |
| Homework - 9c . . . . .         | 66        |
| CMakeCatchTemplate . . . . .    | 66        |
| Intro . . . . .                 | 66        |
| Features . . . . .              | 66        |
| SuperBuild . . . . .            | 66        |
| Homework - 10 . . . . .         | 67        |
| Summary . . . . .               | 67        |
| No Magic Answer . . . . .       | 67        |
| A Few Good Libraries . . . . .  | 67        |
| Ways to Use . . . . .           | 67        |
| Git Submodule Anyone? . . . . . | 68        |
| <b>3 Better C++</b>             | <b>69</b> |
| Better C++ . . . . .            | 69        |
| The Story So Far . . . . .      | 69        |
| Todays Lesson . . . . .         | 69        |
| <b>4 Patterns</b>               | <b>70</b> |
| Patterns . . . . .              | 70        |
| The Story So Far . . . . .      | 70        |
| Todays Lesson . . . . .         | 70        |
| RAII Pattern . . . . .          | 70        |
| What is it? . . . . .           | 70        |
| Why is it? . . . . .            | 71        |
| Example . . . . .               | 71        |
| Program To Interfaces . . . . . | 71        |
| Why? . . . . .                  | 71        |
| Example . . . . .               | 71        |

|  |    |
|--|----|
| Comments . . . . .                           | 72 |
| Inheritance . . . . .                        | 72 |
| Don't overuse Inheritance . . . . .          | 72 |
| Surely Its Simple? . . . . .                 | 73 |
| Liskov Substitution Principal . . . . .      | 73 |
| What to Look For . . . . .                   | 74 |
| Composition Vs Inheritance . . . . .         | 74 |
| Examples . . . . .                           | 74 |
| But Why? . . . . .                           | 75 |
| Dependency Injection . . . . .               | 75 |
| Construction . . . . .                       | 75 |
| Unwanted Dependencies . . . . .              | 75 |
| Dependency Injection . . . . .               | 76 |
| Constructor Injection Example . . . . .      | 76 |
| Setter Injection Example . . . . .           | 76 |
| Advantages of Dependency Injection . . . . . | 77 |
| Construction Patterns . . . . .              | 77 |
| Constructional Patterns . . . . .            | 77 |
| Managing Complexity . . . . .                | 77 |
| Summary . . . . .                            | 78 |
| Putting It Together - Subsystems . . . . .   | 78 |
| Putting It Together - OOP . . . . .          | 78 |
| Object Oriented Review . . . . .             | 78 |
| C-style Programming . . . . .                | 78 |
| Function-style Example . . . . .             | 78 |
| Disadvantages . . . . .                      | 79 |
| C Struct . . . . .                           | 79 |
| Struct Example . . . . .                     | 79 |
| C++ Class . . . . .                          | 79 |
| Abstraction . . . . .                        | 79 |
| Abstraction - Grady Booch . . . . .          | 80 |



|  |    |
|--|----|
| Class Example . . . . .                    | 80 |
| Encapsulation . . . . .                    | 80 |
| Public/Private/Protected . . . . .         | 80 |
| Class Example . . . . .                    | 81 |
| Inheritance . . . . .                      | 81 |
| Class Example . . . . .                    | 81 |
| Polymorphism . . . . .                     | 82 |
| Class Example . . . . .                    | 82 |
| Essential Reading . . . . .                | 83 |
| General Advice . . . . .                   | 83 |
| Daily Reading . . . . .                    | 83 |
| Additional Reading . . . . .               | 83 |
| C++ tips . . . . .                         | 84 |
| OO tips . . . . .                          | 84 |
| Using Templates . . . . .                  | 84 |
| What Are Templates? . . . . .              | 84 |
| You May Already Use Them! . . . . .        | 84 |
| Why Are Templates Useful? . . . . .        | 85 |
| Book . . . . .                             | 85 |
| Are Templates Difficult? . . . . .         | 85 |
| Why Templates in Research? . . . . .       | 85 |
| Why Teach Templates? . . . . .             | 86 |
| Function Templates . . . . .               | 86 |
| Function Templates Example . . . . .       | 86 |
| Why Use Function Templates? . . . . .      | 86 |
| Language Definition 1 . . . . .            | 87 |
| Language Definition 2 . . . . .            | 87 |
| Default Argument Resolution . . . . .      | 87 |
| Explicit Argument Resolution - 1 . . . . . | 88 |
| Explicit Argument Resolution - 2 . . . . . | 88 |
| Beware of Code Bloat . . . . .             | 89 |

|   |    |
|---|----|
| Two Stage Compilation . . . . .           | 89 |
| Instantiation . . . . .                   | 89 |
| Explicit Instantiation - 1 . . . . .      | 89 |
| Explicit Instantiation - 2 . . . . .      | 90 |
| Explicit Instantiation - 3 . . . . .      | 91 |
| Implicit Instantiation - 1 . . . . .      | 91 |
| Implicit Instantiation - 2 . . . . .      | 91 |
| Class Templates . . . . .                 | 92 |
| Class Templates Example - 1 . . . . .     | 92 |
| Class Templates Example - 2 . . . . .     | 93 |
| Class Templates Example - 3 . . . . .     | 93 |
| Quick Comments . . . . .                  | 93 |
| Template Specialisation . . . . .         | 94 |
| Nested Types . . . . .                    | 94 |
| Smart Pointers . . . . .                  | 94 |
| Use of Raw Pointers . . . . .             | 94 |
| Problems with Raw Pointers . . . . .      | 95 |
| Use Smart Pointers . . . . .              | 95 |
| Further Reading . . . . .                 | 95 |
| Standard Library Smart Pointers . . . . . | 95 |
| Stack Allocated - No Leak. . . . .        | 95 |
| Heap Allocated - Leak. . . . .            | 96 |
| Unique Ptr - Unique Ownership . . . . .   | 96 |
| Unique Ptr - Move? . . . . .              | 97 |
| Unique Ptr - Usage 1 . . . . .            | 97 |
| Unique Ptr - Usage 2 . . . . .            | 98 |
| Shared Ptr - Shared Ownership . . . . .   | 98 |
| Shared Ptr Control Block . . . . .        | 98 |
| Shared Ptr - Usage 1 . . . . .            | 98 |
| Shared Ptr - Usage 2 . . . . .            | 98 |
| Shared Ptr - Usage 3 . . . . .            | 99 |

|  |            |
|--|------------|
| Shared Ptr - Usage 4 . . . . .                       | 99         |
| Weak Ptr - Why? . . . . .                            | 100        |
| Weak Ptr - Example . . . . .                         | 100        |
| Final Advice . . . . .                               | 101        |
| Comment on Boost . . . . .                           | 101        |
| Intrusive Vs Non-Intrusive . . . . .                 | 101        |
| ITK (intrusive) Smart Pointers . . . . .             | 102        |
| Conclusion for Smart Pointers . . . . .              | 102        |
| Error Handling . . . . .                             | 102        |
| Exceptions . . . . .                                 | 102        |
| Exception Handling Example . . . . .                 | 103        |
| What's the Point? . . . . .                          | 103        |
| Error Handling C-Style . . . . .                     | 103        |
| Outcome . . . . .                                    | 104        |
| Error Handling C++ Style . . . . .                   | 104        |
| Outcome . . . . .                                    | 105        |
| Consequences . . . . .                               | 105        |
| Practical Tips For Exception Handling . . . . .      | 106        |
| More Practical Tips For Exception Handling . . . . . | 106        |
| <b>5 HPC Concepts</b>                                | <b>107</b> |
| High Performance Computing Overview . . . . .        | 107        |
| Faster Code . . . . .                                | 107        |
| Doing Instructions Faster . . . . .                  | 107        |
| Doing More Instructions at Once . . . . .            | 107        |
| Concurrency Problems . . . . .                       | 109        |
| Some Better Names . . . . .                          | 109        |
| Flynn's Taxonomy . . . . .                           | 109        |
| Amdahl's Law . . . . .                               | 109        |
| Background Reading . . . . .                         | 109        |
| Essential Reading . . . . .                          | 110        |
| Aim . . . . .  | 110        |

|  |            |
|--|------------|
| <b>6 Shared memory parallelism</b>           | <b>111</b> |
| Shared Memory Parallelism . . . . .          | 111        |
| OpenMP . . . . .                             | 111        |
| About OpenMP . . . . .                       | 111        |
| How it works . . . . .                       | 111        |
| Typical use cases . . . . .                  | 112        |
| OpenMP . . . . .                             | 112        |
| OpenMP basic syntax . . . . .                | 112        |
| OpenMP library . . . . .                     | 112        |
| Compiler support . . . . .                   | 112        |
| CMake Support . . . . .                      | 113        |
| Hello world . . . . .                        | 113        |
| Issues with this example . . . . .           | 113        |
| Slightly improved hello world . . . . .      | 114        |
| Improvements: . . . . .                      | 114        |
| Running OpenMP code for the course . . . . . | 115        |
| References . . . . .                         | 115        |
| Parallelizing loops with OpenMP . . . . .    | 115        |
| Simple example . . . . .                     | 115        |
| Variable scope . . . . .                     | 116        |
| Details of example . . . . .                 | 116        |
| Reduction . . . . .                          | 116        |
| Reduction example . . . . .                  | 116        |
| Races, locks and critical regions . . . . .  | 117        |
| Introduction . . . . .                       | 117        |
| Race conditions . . . . .                    | 117        |
| Barriers and synchronisation . . . . .       | 117        |
| Protecting code and variables . . . . .      | 118        |
| Mutex locks . . . . .                        | 118        |
| OpenMP locks . . . . .                       | 118        |
| Example . . . . .                            | 119        |

|   |     |
|---|-----|
| Multiple locks . . . . .                | 119 |
| Notes . . . . .                         | 120 |
| OpenMP Tasks . . . . .                  | 120 |
| Introduction . . . . .                  | 120 |
| Example . . . . .                       | 120 |
| Code . . . . .                          | 120 |
| Main function . . . . .                 | 121 |
| Advanced usage . . . . .                | 121 |
| Controlling task generation . . . . .   | 122 |
| Scheduling and Load Balancing . . . . . | 122 |
| Number of threads . . . . .             | 122 |
| Load balancing . . . . .                | 122 |
| OpenMP strategies . . . . .             | 123 |
| Which strategy to use . . . . .         | 123 |
| Alternatives to OpenMP . . . . .        | 123 |
| Overview . . . . .                      | 123 |
| C++11 . . . . .                         | 124 |
| Details . . . . .                       | 124 |
| Cilk Plus . . . . .                     | 125 |
| Further Reading . . . . .               | 125 |
| Tutorials . . . . .                     | 125 |
| Wavelet decomposition . . . . .         | 126 |
| Wavelet transforms . . . . .            | 126 |
| Pseudo-code . . . . .                   | 126 |
| Instructions . . . . .                  | 126 |
| Header . . . . .                        | 127 |
| Tests . . . . .                         | 129 |
| Implementation . . . . .                | 132 |

|   |            |
|---|------------|
| <b>7 Distributed memory parallelism</b>               | <b>135</b> |
| Distributed Memory Parallelism . . . . .              | 135        |
| Objectives . . . . .                                  | 135        |
| The basic idea . . . . .                              | 135        |
| MPI in practice . . . . .                             | 136        |
| Specification and implementation . . . . .            | 136        |
| Programming and running . . . . .                     | 136        |
| Hello, world!: hello.cc . . . . .                     | 136        |
| Hello, world!: CMakeLists.txt . . . . .               | 137        |
| Hello, world!: compiling and running . . . . .        | 137        |
| Hello, world! dissected . . . . .                     | 137        |
| MPI with CATCH . . . . .                              | 138        |
| Point to point communication . . . . .                | 139        |
| Many point-2-point communication schemes . . . . .    | 139        |
| Blocking synchronous send . . . . .                   | 139        |
| Blocking send . . . . .                               | 140        |
| Non-blocking send . . . . .                           | 140        |
| Blocking synchronous send . . . . .                   | 141        |
| Blocking receive . . . . .                            | 141        |
| Example: Blocking synchronous example . . . . .       | 142        |
| Example: Do you know your C vs C++ strings? . . . . . | 142        |
| Example: Causing a dead-lock . . . . .                | 143        |
| Blocking send: MPI_Send . . . . .                     | 143        |
| Send vs SSend . . . . .                               | 144        |
| Non-blocking MPI_Isend . . . . .                      | 145        |
| Non-blocking: Isend/Irecv . . . . .                   | 147        |
| Function signatures . . . . .                         | 148        |
| Pass the parcel: SendRecv . . . . .                   | 148        |
| Classroom exercise . . . . .                          | 150        |
| Point to point summary . . . . .                      | 150        |
| Collective Communication . . . . .                    | 151        |

|  |            |
|--|------------|
| Many possible communication schemes . . . . .  | 151        |
| Broadcast: one to many . . . . .               | 151        |
| Scatter: one to many . . . . .                 | 151        |
| Gather: many to one . . . . .                  | 152        |
| All to All: many to many . . . . .             | 152        |
| Reduce operation . . . . .                     | 152        |
| Collective operation API (1) . . . . .         | 153        |
| Example of collective operation (1) . . . . .  | 153        |
| Example of collective operations (2) . . . . . | 154        |
| Causing deadlocks . . . . .                    | 154        |
| Scattering data . . . . .                      | 155        |
| MPI_Reduce . . . . .                           | 155        |
| Exercise: MPI_Reduce . . . . .                 | 156        |
| Splitting the communicators . . . . .          | 156        |
| Splitting communicators: example . . . . .     | 157        |
| Splitting communicators: Exercises . . . . .   | 157        |
| Scatter operation solution . . . . .           | 157        |
| More advanced MPI . . . . .                    | 158        |
| Architecture and usage . . . . .               | 158        |
| Splitting communicators . . . . .              | 158        |
| More MPI data-types . . . . .                  | 159        |
| One sided communication . . . . .              | 159        |
| And also . . . . .                             | 159        |
| <b>8 Python Binding</b>                        | <b>160</b> |
| Python Binding . . . . .                       | 160        |
| Integrate C++ with Python . . . . .            | 160        |
| Overview . . . . .                             | 160        |
| Live Example . . . . .                         | 160        |
| Overview . . . . .                             | 160        |
| Demo / Code Walk Through . . . . .             | 161        |
| Other examples . . . . .                       | 161        |

|   |            |
|---|------------|
| <b>9 Summary 2019</b>                     | <b>162</b> |
| Summary 2019 . . . . .                    | 162        |
| Overview of Course . . . . .              | 162        |
| CMake - cross-platform building . . . . . | 162        |
| Cross-platform is common . . . . .        | 162        |
| CMake summary . . . . .                   | 162        |
| Going Forward . . . . .                   | 163        |
| Resources . . . . .                       | 163        |
| Be a Better Developer . . . . .           | 163        |
| Practices . . . . .                       | 163        |
| Going Forward . . . . .                   | 163        |
| Resources . . . . .                       | 164        |
| Master the Language . . . . .             | 164        |
| Architecture First . . . . .              | 164        |
| Only then Language . . . . .              | 164        |
| Master Parallelism . . . . .              | 164        |
| Basic . . . . .                           | 164        |
| More advanced - GPU . . . . .             | 164        |
| More advanced - Cloud . . . . .           | 165        |
| <b>10 Appendix - Libraries</b>            | <b>166</b> |
| Appendix - C++ Library Examples . . . . . | 166        |
| The Story So Far . . . . .                | 166        |
| Todays Lesson . . . . .                   | 166        |
| Using Eigen . . . . .                     | 166        |
| Introduction . . . . .                    | 166        |
| Tutorials . . . . .                       | 167        |
| Getting started . . . . .                 | 167        |
| C++ Principles . . . . .                  | 167        |
| Matrix Class . . . . .                    | 167        |
| Matrix Class Declaration . . . . .        | 168        |



|  |     |
|--|-----|
| Matrix Class Construction . . . . .    | 168 |
| DenseStorage.h - 1 . . . . .           | 169 |
| DenseStorage.h - 2 . . . . .           | 169 |
| DenseStorage.h - 3 . . . . .           | 169 |
| Eigen Matrix Summary . . . . .         | 170 |
| Eigen Usage - CMake Include . . . . .  | 170 |
| Eigen Usage - CMake Module . . . . .   | 170 |
| Eigen Usage - CMake External . . . . . | 170 |
| Eigen Example - in PCL . . . . .       | 171 |
| PCL - Manual Registration . . . . .    | 171 |
| PCL - SVD class . . . . .              | 171 |
| PCL - Correlation . . . . .            | 172 |
| PCL - Summary . . . . .                | 173 |
| Eigen Summary . . . . .                | 173 |
| Using Boost . . . . .                  | 173 |
| Introduction . . . . .                 | 173 |
| Libraries included . . . . .           | 173 |
| Getting started . . . . .              | 173 |
| Installing pre-compiled . . . . .      | 174 |
| Compiling from source . . . . .        | 174 |
| C++ Principles . . . . .               | 174 |
| C Function Pointers - 1 . . . . .      | 174 |
| C Function Pointers - 2 . . . . .      | 175 |
| C Function Pointers - 3 . . . . .      | 176 |
| C++ Function Objects - 1 . . . . .     | 176 |
| C++ Function Objects - 2 . . . . .     | 177 |
| CMake for Boost . . . . .              | 177 |
| Boost Example . . . . .                | 177 |
| Using Boost odeint . . . . .           | 177 |
| Boost odeint - 1 . . . . .             | 177 |
| Boost odeint - 2 . . . . .             | 178 |

|  |     |
|--|-----|
| Boost odeint - 3 . . . . .                 | 178 |
| Boost odeint - 4 . . . . .                 | 178 |
| Boost odeint - 4 . . . . .                 | 179 |
| Why Boost for Numerics . . . . .           | 180 |
| Using ITK . . . . .                        | 180 |
| Introduction . . . . .                     | 180 |
| C++ Principles . . . . .                   | 181 |
| Architecture Concept . . . . .             | 181 |
| Filter Usage - 1 . . . . .                 | 181 |
| Filter Usage - 2 . . . . .                 | 181 |
| Filter Usage - 3 . . . . .                 | 182 |
| Smart Pointer Intro . . . . .              | 182 |
| Smart Pointer Class . . . . .              | 183 |
| General Smart Pointer Usage . . . . .      | 183 |
| ITK SmartPointer . . . . .                 | 183 |
| Implementing a Filter . . . . .            | 184 |
| Filter Impl - 1 . . . . .                  | 184 |
| Filter Impl - 2 . . . . .                  | 184 |
| Filter Impl - 3 . . . . .                  | 185 |
| Filter Impl - 4 . . . . .                  | 185 |
| Iterators . . . . .                        | 186 |
| Private Constructors? . . . . .            | 186 |
| Static New Method . . . . .                | 187 |
| ObjectFactory::Create . . . . .            | 187 |
| Why Object Factories? . . . . .            | 188 |
| File IO Example . . . . .                  | 188 |
| Object Factory List of Factories . . . . . | 188 |
| PNG IO Factory . . . . .                   | 189 |
| ObjectFactory Summary . . . . .            | 189 |
| ITK Summary . . . . .                      | 189 |
| Summary . . . . .                          | 190 |

|  |            |
|--|------------|
| Learning Objectives . . . . .            | 190        |
| Further Reading . . . . .                | 190        |
| <b>11 Appendix - STL</b>                 | <b>191</b> |
| Appendix - STL . . . . .                 | 191        |
| Standard Template Library . . . . .      | 191        |
| The Standard Template Library . . . . .  | 191        |
| Contents . . . . .                       | 191        |
| Motivation . . . . .                     | 192        |
| Motivation - why STL? . . . . .          | 192        |
| The <b>WRONG</b> way to do it! . . . . . | 192        |
| STL solution . . . . .                   | 193        |
| More conveniences . . . . .              | 194        |
| STL Containers . . . . .                 | 194        |
| Two general types: . . . . .             | 194        |
| Sequences . . . . .                      | 194        |
| Sequences . . . . .                      | 194        |
| More on vector . . . . .                 | 195        |
| Note on C++11 . . . . .                  | 195        |
| Exercise . . . . .                       | 195        |
| Associative containers . . . . .         | 195        |
| Associative containers . . . . .         | 196        |
| More on maps . . . . .                   | 196        |
| More on maps . . . . .                   | 197        |
| Example . . . . .                        | 197        |
| Task 1 . . . . .                         | 198        |
| Task 2 . . . . .                         | 198        |
| Task 2 . . . . .                         | 198        |
| Task 3 . . . . .                         | 199        |
| Task 3 - with function . . . . .         | 199        |
| Task 3 - with composition . . . . .      | 200        |

|  |            |
|--|------------|
| Task 3 - with composition . . . . .            | 200        |
| Accessing containers: iterators . . . . .      | 201        |
| Iterators . . . . .                            | 201        |
| Pairs . . . . .                                | 202        |
| Tuples . . . . .                               | 202        |
| Assignment . . . . .                           | 203        |
| Other STL stuff . . . . .                      | 203        |
| Contents . . . . .                             | 203        |
| Streams . . . . .                              | 203        |
| Function objects . . . . .                     | 204        |
| Function objects . . . . .                     | 204        |
| Function objects . . . . .                     | 205        |
| Algorithms . . . . .                           | 205        |
| <b>12 Appendix - TMP</b>                       | <b>206</b> |
| Appendix - Template Meta-Programming . . . . . | 206        |
| No Longer In Course . . . . .                  | 206        |
| Template Meta-Programming (TMP) . . . . .      | 206        |
| What Is It? . . . . .                          | 206        |
| TMP is Turing Complete . . . . .               | 207        |
| Why Use It? . . . . .                          | 207        |
| Factorial Example . . . . .                    | 207        |
| Factorial Notes: . . . . .                     | 208        |
| Loop Example . . . . .                         | 208        |
| Loop Unrolled . . . . .                        | 209        |
| Policy Checking . . . . .                      | 210        |
| Simple Policy Checking Example . . . . .       | 210        |
| Summary of Policy Checking Example . . . . .   | 212        |
| Traits . . . . .                               | 212        |
| Simple Traits Example . . . . .                | 212        |
| Traits Principles . . . . .                    | 213        |

|   |            |
|---|------------|
| Traits Examples . . . . .                 | 213        |
| Wait, Inheritance Vs Traits? . . . . .    | 213        |
| When to use Traits . . . . .              | 213        |
| TMP Use in Medical Imaging - 1 . . . . .  | 214        |
| TMP Use in Medical Imaging - 2 . . . . .  | 214        |
| TMP Use in Medical Imaging - 3 . . . . .  | 214        |
| TMP Use in Medical Imaging - 4 . . . . .  | 215        |
| TMP Use in Medical Imaging - 5 . . . . .  | 215        |
| Further Reading For Traits . . . . .      | 215        |
| Further Reading In General . . . . .      | 216        |
| Summary . . . . .                         | 216        |
| <b>13 Appendix - MPI</b>                  | <b>217</b> |
| MPI Design Example . . . . .              | 217        |
| Objectives for this chapter . . . . .     | 217        |
| Smooth Life . . . . .                     | 217        |
| Conway's Game of Life . . . . .           | 217        |
| Conway's Game of Life . . . . .           | 217        |
| Smooth Life . . . . .                     | 218        |
| Smooth Life . . . . .                     | 218        |
| Smooth Life on a computer . . . . .       | 219        |
| Smooth Life . . . . .                     | 219        |
| Exercise . . . . .                        | 219        |
| Square domain into a 1-d vector . . . . . | 220        |
| Distances wrap around a torus . . . . .   | 220        |
| Smoothed edge of ring and disk . . . . .  | 220        |
| Automated tests for mathematics . . . . . | 221        |
| Comments . . . . .                        | 221        |
| "Ideal" Domain decomposition . . . . .    | 221        |
| Domain decomposition . . . . .            | 221        |
| Decomposing Smooth Life . . . . .         | 222        |

|  |     |
|--|-----|
| Static and dynamic balance . . . . .             | 222 |
| Communication in Smooth Life . . . . .           | 222 |
| Strong scaling . . . . .                         | 222 |
| Weak scaling . . . . .                           | 222 |
| Exercise: Blocking Collective . . . . .          | 223 |
| Exercise: Halo update . . . . .                  | 223 |
| Header . . . . .                                 | 223 |
| Tests . . . . .                                  | 226 |
| Implementation . . . . .                         | 232 |
| Halo swap . . . . .                              | 238 |
| Where do we put the communicated data? . . . . . | 238 |
| Domains with a halo . . . . .                    | 238 |
| Domains with a halo . . . . .                    | 238 |
| Coding with a halo . . . . .                     | 239 |
| Transferring the halo . . . . .                  | 239 |
| Transferring the halo . . . . .                  | 240 |
| Noncontiguous memory . . . . .                   | 240 |
| Buffering . . . . .                              | 240 |
| Buffering . . . . .                              | 240 |
| Testing communications . . . . .                 | 241 |
| Testing communications . . . . .                 | 241 |
| Testing communications . . . . .                 | 242 |
| Deployment . . . . .                             | 242 |
| Getting our code onto Legion . . . . .           | 242 |
| Scripting deployment . . . . .                   | 243 |
| Writing fabric tasks . . . . .                   | 243 |
| Templating jobscripts . . . . .                  | 243 |
| Mako Template for Smooth Life . . . . .          | 244 |
| Configuration files . . . . .                    | 244 |
| Results . . . . .                                | 244 |
| Derived Datatypes . . . . .                      | 244 |

|  |            |
|--|------------|
| Avoiding buffering . . . . .                           | 244        |
| Wrap Access to the Field . . . . .                     | 244        |
| Copy Directly without Buffers . . . . .                | 245        |
| Defining a Halo Datatype . . . . .                     | 245        |
| Declare Datatype . . . . .                             | 245        |
| Use Datatype . . . . .                                 | 246        |
| Strided datatypes . . . . .                            | 246        |
| Overlapping computation and communication . . . . .    | 246        |
| Wasteful blocking . . . . .                            | 246        |
| Asynchronous communication strategy . . . . .          | 246        |
| Asynchronous communication for 2-d halo: . . . . .     | 247        |
| Asynchronous MPI . . . . .                             | 247        |
| Implementation of Asynchronous Communication . . . . . | 247        |
| <b>14 Appendix - MPI-I/O</b>                           | <b>248</b> |
| Appendix - MPI-I/O . . . . .                           | 248        |
| No Longer In Course . . . . .                          | 248        |
| Parallel Input and Output . . . . .                    | 248        |
| Objectives for the Chapter . . . . .                   | 248        |
| Visualisation . . . . .                                | 248        |
| Visualisation is important . . . . .                   | 248        |
| Visualisation is hard . . . . .                        | 249        |
| Simulate remotely, analyse locally . . . . .           | 249        |
| In-Situ visualisation . . . . .                        | 249        |
| Visualising with NumPy and Matplotlib . . . . .        | 249        |
| Formatted Text IO . . . . .                            | 250        |
| Formatted Text IO . . . . .                            | 250        |
| When and when not to use text IO . . . . .             | 250        |
| Use libraries to generate formatted text . . . . .     | 250        |
| Low-level IO exemplar . . . . .                        | 250        |
| Text Data and NumPy . . . . .                          | 251        |

|  |            |
|--|------------|
| Text File Bloat . . . . .                                  | 251        |
| Binary IO in C++ . . . . .                                 | 252        |
| Binary IO in C++ . . . . .                                 | 252        |
| Binary IO in C++ . . . . .                                 | 252        |
| Binary IO in NumPy . . . . .                               | 252        |
| Endianness and Portability . . . . .                       | 252        |
| Portability . . . . .                                      | 252        |
| Length of datatypes . . . . .                              | 253        |
| Endianness . . . . .                                       | 253        |
| NumPy dtypes . . . . .                                     | 253        |
| XDR . . . . .  | 253        |
| Writing with XDR . . . . .                                 | 254        |
| Writing from a single process . . . . .                    | 254        |
| One process, one file . . . . .                            | 254        |
| One process, one file . . . . .                            | 255        |
| Writing from a single process . . . . .                    | 255        |
| Writing from a single process . . . . .                    | 255        |
| Writing from a single process . . . . .                    | 256        |
| Parallel IO . . . . .                                      | 256        |
| Serialising on a single process . . . . .                  | 256        |
| Parallel file systems . . . . .                            | 256        |
| Introduction to MPI-IO . . . . .                           | 256        |
| Opening parallel file . . . . .                            | 256        |
| Finding your place . . . . .                               | 257        |
| High level research IO libraries . . . . .                 | 257        |
| <b>15 Appendix - Code Quality</b>                          | <b>258</b> |
| Appendix - Code Quality . . . . .                          | 258        |
| No Longer In Course . . . . .                              | 258        |
| Code Quality . . . . .                                     | 258        |
| All tests pass, the code works: are we done yet? . . . . . | 258        |



|  |            |
|--|------------|
| Tools for pretty code . . . . .                                    | 258        |
| The end of brace wars: brainless auto formatting . . . . .         | 258        |
| Linting . . . . .  | 259        |
| Refactoring . . . . .  | 259        |
| Memory leaks . . . . .   | 259        |
| Checking memory allocation intrusively . . . . .                   | 259        |
| Checking memory allocation non-intrusively . . . . .               | 260        |
| Exercise: Traveling salesman solved by Simulated Annealing . . .   | 260        |
| Setting up a docker VM and docker . . . . .                        | 260        |
| Running valgrind on program called <code>awful</code> . . . . .    | 262        |
| Running valgrind on program called <code>less_bad</code> . . . . . | 262        |
| Profiling a program . . . . .                                      | 262        |
| Amdahl's law . . . . .   | 262        |
| Profiling . . . . .  | 263        |
| Exercise: Profiling the correct <code>less_bad</code> . . . . .    | 263        |
| Micro-benchmarking . . . . .                                       | 264        |
| Introduction . . . . .   | 264        |
| Exercise: build and run <code>micro_benchmark</code> . . . . .     | 264        |
| <b>16 Appendix - Cloud</b>   | <b>266</b> |
| Appendix - Cloud computing . . . . .                               | 266        |
| No Longer In Course . . . . .                                      | 266        |
| Big data . . . . .   | 266        |
| Cloud computing . . . . .  | 266        |
| X* as a service . . . . .  | 267        |
| Exercise 1: Working in the cloud . . . . .                         | 267        |
| Spinning up an instance . . . . .                                  | 267        |
| Create an account . . . . .  | 267        |
| Create a key pair . . . . .  | 267        |
| Create a single instance using the web interface . . . . .         | 269        |
| Connect to the instance with SSH . . . . .                         | 269        |

|   |     |
|---|-----|
| Terminate the server . . . . .                                  | 270 |
| Exercise 2: Working in the cloud . . . . .                      | 270 |
| Again, from the command line. . . . .                           | 270 |
| Configure the tools . . . . .                                   | 270 |
| Test our connection to Amazon Web Services . . . . .            | 271 |
| Create a key pair . . . . .                                     | 271 |
| Create a security group . . . . .                               | 271 |
| Configure the security group . . . . .                          | 272 |
| Locate an appropriate Machine Image . . . . .                   | 272 |
| Launch an instance . . . . .                                    | 272 |
| View the instance . . . . .                                     | 273 |
| Connect to the instance . . . . .                               | 273 |
| Terminate the instance . . . . .                                | 273 |
| Virtualisation . . . . .  | 274 |
| Reproducible research . . . . .                                 | 274 |
| Virtual machines . . . . .                                      | 274 |
| Virtual environments . . . . .                                  | 274 |
| Exercise 3: Virtualisation . . . . .                            | 274 |
| Reproducible research . . . . .                                 | 274 |
| Create a new security group . . . . .                           | 275 |
| Spin up a cloud instance . . . . .                              | 275 |
| Connect with SSH . . . . .                                      | 275 |
| Install and run Docker on the remote system . . . . .           | 276 |
| Docker environments are created by a set of commands . . . . .  | 276 |
| Pull, build, and run the Docker image . . . . .                 | 277 |
| Connect to the IPython Notebook . . . . .                       | 277 |
| Now use the IPython Notebook to complete the exercise . . . . . | 277 |
| Distributed computing . . . . .                                 | 277 |
| Distributed computing . . . . .                                 | 277 |
| MapReduce . . . . .   | 279 |
| Distributed file systems . . . . .                              | 279 |

|  |            |
|--|------------|
| Hadoop . . . . .                                 | 280        |
| Exercise 4: MapReduce . . . . .                  | 280        |
| The ‘Hello World’ of MapReduce . . . . .         | 280        |
| Choose a book . . . . .                          | 280        |
| Hadoop streaming . . . . .                       | 280        |
| Mapper . . . . .                                 | 281        |
| Reducer . . . . .                                | 281        |
| Demonstrate locally . . . . .                    | 282        |
| Exercise 5: Distributed computing . . . . .      | 282        |
| Again, with Hadoop . . . . .                     | 282        |
| Hadoop in the cloud . . . . .                    | 283        |
| Create an S3 bucket . . . . .                    | 283        |
| Copy data and code to S3 . . . . .               | 283        |
| Launch the compute cluster . . . . .             | 284        |
| Get the cluster ID . . . . .                     | 284        |
| Get the public DNS name of the cluster . . . . . | 284        |
| Connect to the cluster . . . . .                 | 284        |
| Run the analysis . . . . .                       | 285        |
| View the results . . . . .                       | 285        |
| Terminate the cluster . . . . .                  | 286        |
| Delete the bucket . . . . .                      | 286        |
| <b>17 Appendix - Accelerators</b>                | <b>287</b> |
| Appendix - Accelerators . . . . .                | 287        |
| No Longer In Course . . . . .                    | 287        |
| Introduction to Accelerators . . . . .           | 287        |
| Gamers vs Researchers . . . . .                  | 287        |
| What is an accelerator? . . . . .                | 287        |
| Why would I want to use one? . . . . .           | 288        |
| CPU Vectorisation . . . . .                      | 288        |
| Using an accelerator within the CPU . . . . .    | 288        |

|   |            |
|---|------------|
| CPUs as multicore vector processors . . . . .                 | 288        |
| Compiler Autovectorisation . . . . .                          | 289        |
| Autovectorisation results . . . . .                           | 289        |
| Support for Autovectorisation . . . . .                       | 290        |
| Support for Autovectorisation . . . . .                       | 290        |
| Using a GPU as an Accelerator . . . . .                       | 290        |
| GPUs for 3D graphics . . . . .                                | 290        |
| GPUs as multicore vector processors . . . . .                 | 291        |
| GPU Hardware . . . . .  | 291        |
| General Purpose Programming for GPUs . . . . .                | 291        |
| GPU Architecture . . . . .                                    | 292        |
| SIMT . . . . .  | 292        |
| Executing code on the GPU . . . . .                           | 292        |
| GPU-accelerating your code . . . . .                          | 293        |
| Streaming computations . . . . .                              | 293        |
| <i>Single Instruction</i> , Multiple Data . . . . .           | 293        |
| Is this single instruction? . . . . .                         | 294        |
| Single Instruction, <i>Multiple Contiguous Data</i> . . . . . | 294        |
| The joys of indexing . . . . .                                | 294        |
| Structure of Arrays or Arrays of structures . . . . .         | 295        |
| Just USES . . . . .   | 295        |
| Broadcasting (Vectorization) . . . . .                        | 295        |
| Broadcasting (Vectorization) . . . . .                        | 296        |
| A word on transfer rate . . . . .                             | 296        |
| Exercise: Broadcasting . . . . .                              | 297        |
| Exercise: Line of Sight . . . . .                             | 297        |
| <b>18 Installation . . . . .</b>                              | <b>299</b> |
| Installation . . . . .  | 299        |
| Introduction . . . . .  | 299        |
| What you'll need by the end of the course . . . . .           | 299        |

|   |            |
|---|------------|
| Eduroam . . . . .                               | 299        |
| Linux . . . . .                                 | 300        |
| Git . . . . .                                   | 300        |
| CMake . . . . .                                 | 300        |
| Editor and shell . . . . .                      | 300        |
| Windows . . . . .                               | 300        |
| Compiler / Editor . . . . .                     | 300        |
| Git . . . . .                                   | 301        |
| CMake . . . . .                                 | 301        |
| Locating your install . . . . .                 | 301        |
| Telling Shell where to find the tools . . . . . | 301        |
| Finding your terminal . . . . .                 | 301        |
| Checking which tools you have . . . . .         | 302        |
| Tell Git about your editor . . . . .            | 302        |
| Mac . . . . .                                   | 303        |
| XCode and command line tools . . . . .          | 303        |
| A note on Homebrew Versus MacPorts . . . . .    | 303        |
| Installing Homebrew . . . . .                   | 303        |
| Git . . . . .                                   | 303        |
| CMake . . . . .                                 | 303        |
| Editor and shell . . . . .                      | 304        |
| Boost . . . . .                                 | 304        |
| Linux Users . . . . .                           | 304        |
| Mac Users . . . . .                             | 304        |
| Windows Users . . . . .                         | 304        |
| CMake and Boost . . . . .                       | 305        |
| Docker . . . . .                                | 305        |
| Optional - Ask Lecturer . . . . .               | 305        |
| <b>19 Assessment</b>                            | <b>306</b> |
| Assessment Structure . . . . .                  | 306        |

# Chapter 1

## C++ for Research

### Course Overview

#### Part 1

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

#### 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: [MPHY0022](#)
  - key is “performance”
- Online notes: [MPHY0022](#)

## Course Assessment

- 3 hour exam
- 2 pieces coursework - 40 hours each
  - 1 due 3rd week March
  - 1 due last week April
  - (roughly)

## Course Community

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

## 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](#)
  - Famous People: [Edsger W. Dijkstra](#)
- 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: [NiftyReg](#)
  - Testing code
  - Data analysis
  - User Interface
  - Glue code
  - Deployment code
  - Scientific [paper](#) production

Examples: NiftyReg 300 citations in 5 years!

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

## Another Example - NiftyCal

- We should: Practice What We Preach
- Small, algorithms
- Unit tested
- Version controlled
- Small number of libraries
- Increased research output

## 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).
- Do 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

```
cd ~/build
git clone https://github.com/MattClarkson/CMakeHelloWorld
mkdir CMakeHelloWorld-build
cd CMakeHelloWorld-build
ccmake ../CMakeHelloWorld
make
```

## CMake Usage Windows

Demo an “out-of-source” build

- git clone <https://github.com/MattClarkson/CMakeHelloWorld>
- Run cmake-gui.exe
- Select source folder (CMakeHelloWorld downloaded above)
- Specify new build folder (CMakeHelloWorld-build next to, but not inside CMakeHelloWorld)
- Hit *configure*
- When asked, specify compiler
- Set flags and repeatedly *configure*
- When *generate* option is present, hit *generate*
- Compile, normally using Visual Studio

## 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
- On Windows, build separate 32 and 64 bit versions
- Add code to hello.cpp:
  - On Linux/Mac re-compile just using make
  - On Windows, re-compile within Visual Studio
  - (i.e. you don’t need to repeatedly run CMake)

## 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!)
- .cpp/.cxx 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 cpp
#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 cpp
#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_NO_THROW( expression )  
    CHECK_NO_THROW( 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

# Including Libraries

### Why use Libraries?

#### Scientific Coding

- Explain your science
  - Should not just distribute binaries and papers
  - Source code is best description of your science
- Reproducible science
  - Collaborators must know how you build your project and libraries
- Time is short
  - Don't re-invent the wheel
  - Use libraries if they provide bits you need

### What are libraries?

- Libraries provide collections of useful classes and functions, ready to use
- C++ libraries can be somewhat harder to use than pure Python modules
- This lecture will get you started

### Aim for this chapter

- How to choose a library
  - Licensing, longevity, developer community, technical implementation, **feature list** etc.

- Working with libraries
  - Including them
  - C++ concepts
  - Not an exhaustive product specific tutorial

## Choosing libraries

### Libraries are Awesome

A [great set of libraries](#) allows for a very powerful programming style:

- Write minimal code yourself
  - Choose the right libraries
  - Plug them together
  - Create impressive results

### Libraries for Efficiency

Not only is this efficient with your programming time, it's also more efficient with computer time.

The chances are any general algorithm you might want to use has already been programmed better by someone else.

### Software Licenses

- Consider both:
  - License for 3rd party code / dependencies
  - License for your code when you distribute it
- So, even if you aren't distributing code yet, you need to understand the licenses of your dependencies.

### Third Party Licenses

CAVEAT: This is not legal advice. If in doubt, seek your own legal advice.

- When you distribute your code, the licenses of any libraries you use takes effect
- If library has:

- [MIT](#) and [BSD](#) are permissive. So you can do what you want, including sell it on.
- [Apache](#) handles multiple contributors and patent rights, but is basically permissive.

## Third Party Licenses

CAVEAT: This is not legal advice. If in doubt, seek your own legal advice.

- When you distribute your code, the licenses of any libraries you use takes effect
- If library has:
  - [GPL](#) requires you to open-source your code, including changes to the library you imported, and your work is considered a “derivative work”, so must be shared.
  - [LGPL](#) for libraries, but use dynamic not static linkage. If you use static linking its basically as per [GPL](#).
- Still some debate on [GPL/LGPL and derivative works](#) - only true test is in court.

## Choosing a License

CAVEAT: This is not legal advice. If in doubt, seek your own legal advice.

- When you plan to distribute code:
  - Read [this book](#), and/or [GitHub’s advice](#), and [OSI](#) for choosing your own license.
  - Don’t write your own license, unless you use legal advice.
  - Try to pick one of the standard ones for compatibility.
- Note: Once a 3rd party has your code under a license agreement, their restrictions are determined by that version of the code.

## Choose Stability

- So, take care in your choice of 3rd party library
  - Don’t want to redo work later.
  - Don’t want to rely too heavily on non-distributable code.
  - But if you do, understand what that means.

## Is Library Updated?

- Code that is not developed, rots.
  - When was the last commit?
  - How often are there commits?
  - Is the code on an open version control tool like GitHub?

## Are Developers Reachable?

- Can you find the lead contributor on the internet?
  - Do they respond when approached?
- Are there contributors other than the lead contributor?

## Is Code Tested?

- Are there many unit tests, do they run, do they pass?
- Does the library depend on other libraries?
- Are the build tools common?
- Is there a sensible versioning scheme (e.g. [semantic versioning](#)).
- Is there a suitable release schedule?

## Is Library High Quality?

- Shouldn't need to look excessively closely, but consider
  - Documentation
  - Number of ToDo's
  - Dependencies (recursively)
  - Data Structures? How to import/export image
  - Can you write a convenient wrapper?

## Library Features

- Then look at features
  - Manual
  - Easy to use

## Summary

- In comparison with languages such as MATLAB/Python
  - In C++ prefer few well chosen libraries
  - Be aware of their licenses for future distribution
  - Keep a log of any changes, patches etc. that you make
  - Be able to compile all your own code, including libraries
    - \* so need common build environment. (eg. CMake, Make, Visual Studio).

## Library Basics

### Aim

You need your compiler to find:

- Headers: `#include`
- Library:
  - Dynamic: `.dylib`, `.so`, `.lib` / `.dll`
  - Static: `.a`, `.lib`

### Linux/Mac

```
g++ -c -I/users/me/myproject/include main.cpp
g++ -o main main.o -L/users/me/myproject/lib -lmylib
```

- `-I` to specify include folder
- `-L` to specify library folder
- `-l` to specify the actual library

### Compiler switches

So, that means

- Include directory
  - `-I /some/directory`
- Library directory
  - `-L /some/directory`



- Library
  - `-l library`

Similar concept on Windows, Linux and Mac.

## Check Native Build Platform

- Look inside
  - Makefile
  - Visual Studio options
- Basically constructing `-I`, `-L`, `-l` switches to pass to command line compiler.

## Windows Compiler Switches

- Visual Studio (check version)
- Project Properties
  - C/C++ -> Additional Include Directories.
  - Configuration Properties -> Linker -> Additional Library Directories
  - Linker -> Input -> Additional Dependencies.
- Check compile line - its equivalent to Linux/Mac, `-I`, `-L`, `-l`

## Location Issues

When you use a library:

- Where is it?
- Header only?
- System version or your version?
- What about bugs? How do I upgrade? Do I need to build it myself?

## Compilation Issues

When you use a library:

- Which library version?
- Which compiler version?
- Debug or Release?
- [Static or Dynamic?](#)
- 32 bit / 64 bit?
- Platform specific flags?
- Pre-installed, or did you compile it?

## A Few Good Libraries

Due to all those issues shown above, again, the main advice for libraries:

- As few as possible
- As good as possible

## Linking libraries

### Linking

- From first lecture
  - Code is split into functions/classes
  - Related functions get grouped into libraries
  - Libraries get compiled / archived into one file
- End User needs
  - Header file = declarations (and implementation if header only)
  - Object code / library file = implementations

### Static Linking

- Windows (.lib), Mac/Linux (.a)
- Compiled code from static library is copied into the current translation unit.
- Increases disk space compared with dynamic linking.
- Current translation unit then does not depend on that library.

### Dynamic Linking

- Windows (.dll), Mac (.dylib), Linux (.so)
- Compiled code is left in the library.
- At runtime,
  - OS loads the executable
  - OS / Linker finds any unresolved libraries
  - Recursive process
- Saves disk space compared with static linking.
- Faster compilation/linking times?
- Current translation unit has a known dependency remaining.

## Dynamic Loading

- System call to load a library (dlopen/LoadLibrary)
- Dynamically discover function names and variables
- Execute functions
- Normally used for plugins
- Not covered here

## Space Comparison

- If you have many executables linking a common set of libraries
- Static
  - Code gets copied - each executable bigger
  - Doesn't require searching for libraries at run-time
- Dynamic
  - Code gets referenced - smaller executable
  - Requires finding libraries at run-time
- Space - less of a concern these days

## For Scientists

- Ease of use
- Ease of distribution to collaborators
- Prefer static if possible for ease of deployment

## Packaging

- Packaging large apps takes effort
- hire Research Software Engineers

## How to Check

- Windows - Dependency Walker
- Linux - ldd
- Mac - otool -L
- (live demo on Mac)

## Using libraries

### Package Managers

- Package Manager (Linux/Mac)
  - Precompiled
  - Stable choice
  - Inter-dependencies work
- Linux
  - `sudo apt-get install`
  - `sudo yum install`
- Mac
  - `sudo port install`
  - `brew install`

### Windows

- Libraries typically:
  - Randomly installed location
  - In system folders
  - In developer folders
  - In build folder
- Please clean your machine!!
- Try [Chocolatey](#) package manager

### Package Managers

- So, if you can use standard versions of 3rd party libraries
- Package managers are a good way to go
- You just need to specify what versions so your collaborator can check

### Problems

- As soon as you hit a bug in a library
  - How to update?
  - Knock on effects
    - \* Cascading updates
    - \* Inconsistent development environment

## Build Your Own

- 2 basic approaches
  - External / Individual build
    - \* Build dependencies externally
    - \* Point your software at those packages
  - SuperBuild / Meta-Build
    - \* Write code to download all dependencies
    - \* The correct version numbers is stored in code

### External / Individual Build

For example

```
C:\build\ITK-v1
C:\build\ITK-v1-build
C:\build\ITK-v1-install
C:\build\VTK-v2
C:\build\VTK-v2-build
C:\build\VTK-v2-install
C:\build\MyProject
C:\build\MyProject-build
```

We setup `MyProject-build` to know the location of ITK and VTK install folder.

### Meta-Build / Super-Build

For example

```
C:\build\MyProject
C:\build\MyProject-SuperBuild\ITK\src
C:\build\MyProject-SuperBuild\ITK\build
C:\build\MyProject-SuperBuild\ITK\install
C:\build\MyProject-SuperBuild\VTK\src
C:\build\MyProject-SuperBuild\VTK\build
C:\build\MyProject-SuperBuild\VTK\install
C:\build\MyProject-SuperBuild\MyProject-build
```

We setup `MyProject-build` to know the location of ITK and VTK that it itself compiled.

## Pro's / Con's

- External Build
  - Pro's - build each dependency once
  - Con's - collaborators will do this inconsistently
  - Con's - how to manage multiple versions of all dependencies
- Meta Build
  - Pro's - all documented, all self-contained, easier to share
  - Con's - Slow build? Not a problem if you only run `make` in sub-folder `MyProject-build`

## Examples

### Reminder

- Fundamentally, we need:
  - Access to header files - declaration
  - Access to compiled code - definition

### Header Only?

- After all this:
  - Static/Dynamic
  - Package Managers / Build your own
  - External build / Internal Build
  - Release / Debug
  - `-I`, `-L`, `-l`
- Header only libraries are very attractive!

### CMake - Example

- `catch.hpp` Header files in project [CMakeCatch2](#)
- From `CMakeCatch2/CMakeLists.txt`

```
include_directories(${CMAKE_SOURCE_DIR}/Code/)
add_subdirectory(Code)
if(BUILD_TESTING)
    include_directories(${CMAKE_SOURCE_DIR}/Testing/)
    add_subdirectory(Testing)
endif()
```

## CMake - Header Only

- For example:

```
CMakeCatchTemplate/3rdParty/libraryA/version1/Class1.hpp  
CMakeCatchTemplate/3rdParty/libraryA/version1/Class2.hpp
```

- Add path to CMakeLists.txt

```
include_directories(${CMAKE_SOURCE_DIR}/3rdParty/libraryA/version1/)
```

- You'd have audit trail (via git repo) of when updates to library were made.

## CMake - Header Only, External

- If you think library is too large:

```
C:\3rdParty\Eigen  
C:\build\MyProject  
C:\build\MyProject-build
```

- Add to CMakeLists.txt

```
include_directories("C:\3rdParty\Eigen\install\include\eigen3\
```

- Hard-coded, but usable if you write detailed build instructions
- Not very platform independent
- Not very flexible

## CMake - find\_package

- In general, in CMake, each dependency requires a bit of code to look up include directories, and libraries.

```
find_package(OpenCV REQUIRED)  
include_directories(${OpenCV_INCLUDE_DIRS})  
list(APPEND ALL_THIRD_PARTY_LIBRARIES ${OpenCV_LIBS})  
add_definitions(-DBUILD_OpenCV)
```

- So a 3rd party package can provide information on how you should use it

## Types of `find_package`

- `find_package` has several different modes:
  - *config* mode: Use 3rd party projects own config, eg. `VTKConfig.cmake`
  - *module* mode: Use a FindModule, some come with CMake
  - *module* mode: Write your own FindModule
  - *module* mode: Write your own FindModule with generated / substituted variables

## `find_package` - refer to docs

A Basic Example (for a full example - see docs)

- Given:

```
find_package(SomeLibrary [REQUIRED])
```

- CMake will search
  - all directories in `CMAKE_MODULE_PATH`
  - for `SomeLibraryConfig.cmake` - does *config* mode
  - for `FindSomeLibrary.cmake` - does *module* mode
  - case sensitive

## `find_package` - Result

- `find_package(SomeLibrary)` should return `SomeLibrary_FOUND:BOOL=TRUE` if library was found
- Sets any other variables necessary to use the library
- Check `CMakeCache.txt` to see result

## `find_package` - Usage

- So many 3rd party libraries are CMake ready.
- If things go wrong, you can debug it - CMake is all text based.

## `find_package` - Fixing

- You can provide patched versions
- Add your source/build folder to the `CMAKE_MODULE_PATH`



```

set(CMAKE_MODULE_PATH
    ${CMAKE_SOURCE_DIR}/CMake
    ${CMAKE_BINARY_DIR}
    ${CMAKE_MODULE_PATH}
)

```

- So CMake will find your version before the system versions

## find\_package - Tailoring

- You can write your own
  - e.g. FindEigen in [CMakeCatchTemplate](#)
- Use CMake to substitute variables
  - Force include/library dirs
  - Useful for vendors API that isn't CMake compatible
  - Useful for meta-build. Force directories to match the package you just compiled.

## Provide Build Flags

- If a package is found, you can add compiler flag.
 

```
add_definitions(-DBUILD_OpenCV)
```
- So, you can optionally include things:
 

```
#ifdef BUILD_OpenCV
#include <cv.h>
#endif
```
- Best not to do too much of this.
- Useful to provide build options, e.g. for running on clusters

## Check Before Committing

- Before you commit code to git,
- Make sure you are compiling what you think you are!
 

```
#ifdef BUILD_OpenCV
blah blah
#include <cv.h>
#endif
```
- should fail compilation

## Summary

- Basic aim:
  - `include_directories()` generates `-I`
  - `link_directories()` generates `-L`
  - `target_link_libraries(mylibrary PRIVATE ${libs})` generates `-l` for each library
- Might not need `link_directories()` if libraries fully qualified
- Try default CMake `find_package`
- Or write your own and add location to `CMAKE_MODULE_PATH`

## CMakeCatch2

### Intro

- Demo project on [GitHub: CMakeCatch2](#)
- No functional code, other than adding 2 numbers
- Sets up Eigen and Catch, header only, checked into project

### Homework - 9a

- Download `CMakeCatch2`
- Practice compiling `CMakeCatch2` project, testing the flags above.
- Choose a another project relevant to you - eg. `glog/gflags/VTK/OpenCV`
- Compile it in a separate folder, as per their build instructions
- Modify some C++ in `CMakeCatch2`, recompile
- Try to include external library, e.g. `glog/gflags/VTK/OpenCV` and call a simple function
  - See example for Eigen, in [Command Line App](#)

### Homework - 9b

These were covered earlier - here's how you set them.

- `CMAKE_BUILD_TYPE:String=[Debug|Release]`, sets different optimisation levels, and debug symbols
- `BUILD_SHARED_LIBS:BOOL=[OFF|ON]`, switch between static linking and dynamic linking
- Compile flags

- CMAKE\_C\_FLAGS, CMAKE\_CXX\_FLAGS, CMAKE\_EXE\_LINKER\_FLAGS all `String`
- You can add extra compile flags as you wish.

## Homework - 9c

- 32 or 64 bit?
- Generally stick with default for your platform
- If you need to change:
  - Visual Studio - chose generator, check in Visual Studio (x86, Win64)
  - Linux CMAKE\_C\_FLAGS = `-m32` etc. [For example](#)

## CMakeCatchTemplate

### Intro

- Demo project on [GitHub: CMakeCatchTemplate](#)
- No functional code, other than adding 2 numbers
- Basically shows how to use CMake, via various examples.

### Features

- Full feature list in [README.md](#)
- SuperBuild:
  - Downloads Boost, Eigen, glog, gflags, OpenCV, PCL, FLANN, VTK
- Example GUI apps (beyond scope of course)
- Unit testing

### SuperBuild

- See flag: `BUILD_SUPERBUILD:BOOL=[ON|OFF]`
- If OFF
  - Just compiles *this* project in current folder
- If ON
  - Dependencies in current folder
  - Compiles *this* project in sub-folder
- Try it, in separate build folders.

## Homework - 10

Setup folders like:

```
# Source folder  
/User/me/build/CMakeCatchTemplate  
  
# SuperBuild folder  
/User/me/build/CMakeCatchTemplate-SuperBuild
```

- Ensure BUILD\_SUPERBUILD=ON
- Try turning a small library like BUILD\_gflags to ON
- Run the build in the SuperBuild folder
- Look on disk, see how the dependencies are compiled in the CMakeCatchTemplate-SuperBuild folder.
- Then look in sub-folder MYPROJECT-build to see how this project is build, using the dependencies in the folder above.

## Summary

### No Magic Answer

- C++ is more tricky than MATLAB / Python
- While package managers make it easier, you still need to understand what you're building.

### A Few Good Libraries

Main advice for libraries:

- As few as possible
- As high quality as possible

### Ways to Use

- Easiest - header only
  - just include directly in your source code
  - use cmake `include_directories()`
  - compile it in
- use packages from package manager

- use cmake to `find_package`
  - set variables to your installed version
- use build system to build everything
  - you control ALL flags
  - cmake does `find_package` on things you just compiled

## Git Submodule Anyone?

- git submodule
  - If all dependencies are cmake'd
  - Put each project in a git submodule
  - Cmake can configure whole project
  - Not really used on this course

## Chapter 3

# Better C++

### Better C++

#### The Story So Far

- Git
- CMake
- Use libraries with CMake

#### Today's Lesson

- Beginners C++ not sufficient
- Look at
  - C++ recap
  - Templates - generic programming
  - Exceptions - error handling

# Chapter 4

# Patterns

## Patterns

### The Story So Far

- Git
- CMake
- Use libraries
- C++ features
- Seeing those C++ in libraries

### Today's Lesson

- How to assemble classes
- Beginner mistakes
  - 1 class - all functionality
  - Deep inheritance trees
- You need
  - How to assemble / organise classes

## 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](#)

## Program To Interfaces

### Why?

- In research code we often “just start hacking”
- You tend to mix interface and implementation
- Results in client 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
  - e.g. camera calibration depends on OpenCV
  - 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 truly 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”
- Can something truly be substituted?
- If someone else filled a vector of type T, would I care what type I have?
- Look for:
  - Preconditions cannot be strengthened
  - Postconditions cannot be weakened
  - Invariants preserved

## 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 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.
- Department ‘has-a’ Professor. If a department is shutdown (deleted), Professor should not be deleted. Needs assigning to another department.

## But Why?

- Good article: [Choosing Composition or Inheritance](#)
- Inheritance has much tighter definition than you realise
- Composition is more flexible

## 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

## 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

## 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

## 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);
}

```

## Essential Reading

### General Advice

- Do a version control course such as [Software Carpentry](#), or [MPHYG001](#).
- Contribute to online open-source project
- For your repository - pick a coding style, and stick to it

### Daily Reading

- Every C++ developer should keep repeatedly reading at least:
  - [Effective C++](#), Meyers
  - [More Effective C++](#), Meyers
  - [Effective STL](#), Meyers

### Additional Reading

- Recommended
  - [Accelerated C++](#), Koenig, Moo.
  - [Design Patterns \(1994\)](#), Gamma, Helm, Johnson and Vlissides
  - [Modern C++ Design](#), Andrei Alexandrescu

## C++ tips

Numbers in brackets refer to Scott Meyers “Effective C++” book.

- Top C++ tips such as:
  - Declare data members private (22)
  - Use `const` whenever possible (3)
  - Make interfaces easy to use correctly and hard to use incorrectly (18)
  - Avoid returning “handles” to object internals (28)
  - Initialise objects properly. Throw exceptions from constructors. Fail early. (4)

## OO tips

- More general OO tips such as:
  - Never throw exceptions from destructors
  - Prefer non-member non-friend functions to member functions (better encapsulation) (23)
  - Make sure public inheritance really models “is-a” (32)
  - Learn alternatives to polymorphism (Template Method, Strategy) (35)
  - Model “has-a” through composition (38)

## 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

- You should read “[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?

- Generalise 2D, 3D, n-dimensions, (e.g. [ITK](#) )
- Test numerical code with simple types, apply to complex/other types
- Several useful libraries for research

## 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](http://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 - 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 - 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 - 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>(int); // instantiates f<int>(int), template argument deduced
```

## Explicit Instantiation - 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 - 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 - 1

- Instantiated as they are used
- Normally via `#include` header files.
- Given (library) header, that contains implementation:

```

#ifdef 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 - 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

```

## Class Templates

### Class Templates Example - 1

- If you understand template functions, then template classes are easy!
- Referring 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 - 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 - 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
```

## Nested Types

In libraries such as [ITK](#), we see:

```
template< typename T, unsigned int NVectorDimension = 3 >  
class Vector:public FixedArray< T, NVectorDimension >  
{  
    public:  
        // various stuff  
        typedef T ValueType;  
        // various stuff  
        T someMemberVariable;
```

- typedef is just an alias
- using nested typedef, must be qualified by class name
- can also refer to a real variable

## 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=0x209f010
f=0x209f010, 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)

## Comment on Boost

- Boost has become a sandbox for standard C++
- Boost features become part of standard C++, (different name space)
- So if you are forced to use old compiler
  - You could use boost - lecture 5.

## Intrusive Vs Non-Intrusive

- Intrusive - Base class maintains a reference count eg. [ITK](#)
- Non-intrusive
  - `std::unique_ptr`
  - `std::shared_ptr`
  - `std::weak_ptr`
  - works for any class

## ITK (intrusive) Smart Pointers

```
class MyFilter : // Other stuff
{
public:
    typedef MyFilter Self
    typedef SmartPointer<Self> Pointer;
    itkNewMacro(Self);
protected:
    MyFilter();
    virtual ~MyFilter();
};

double someFunction(MyFilter::Pointer p)
{
    // stuff
}

int main()
{
    MyFilter::Pointer p = itk::MyFilter::New();
}
```

## 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

## Error Handling

### Exceptions

- Exceptions are the C++ or Object Oriented way of Error Handling
- Read [this](#) example

## 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?

- Have separated error handling logic from application logic
- 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;
    }
}

```

## Outcome

- Code that throws does not worry about the catcher
- Exceptions are classes, can carry data
- Exceptions can form class hierarchy

## Consequences

- More suited to larger libraries of re-usable functions
- Many different catchers, all implementing different error handling
- Lends itself to layered software (draw diagram)
- 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

## 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\(\)](#)
- Beware side-effects
  - Perform validation before updating any member variables

## Chapter 5

# HPC Concepts

### High Performance Computing Overview

#### Faster Code

How can we make a computer go faster?

- Do instructions faster
- Do more instructions at once

#### Doing Instructions Faster

- How?
  - Reduce wait time. I.E cache misses, data loading, etc.
    - \* Can be complex and hardware dependant
    - \* Often automated
  - Write machine code rather than interpreted code
    - \* JIT does this for you
    - \* Allows for cool tracing optimisations
  - Increase CPU speed
    - \* CPU instructions happen faster
    - \* Data might not **move** faster

#### Doing More Instructions at Once

Image variant, original by Benjamin D. Esham / Wikimedia Commons, CC BY-SA 3.0



Figure 5.1: Dining Philosophers

## Concurrency Problems

- Deadlock
  - All philosophers have one fork each and are waiting for another to appear
- Starvation
  - A philosopher never gets to eat
- Race conditions
  - A bug where the order of operations matters. This is usually either a algorithm design bug, or two philisophers both using the same fork at the same time.

## Some Better Names

- Philosopher: thread, process, task, etc.
- Fork: resource, lock, etc.
- Spaghetti: Critical section.

## Flynn's Taxonomy

- **SISD**: You doing something on your own
- **SIMD**: You and your clones doing the same things on different inputs
- **MISD**: You and your friends doing different things with the same inputs
- **MIMD**: You and your friends doing unrelated things
- **SIMT**: You and your clones doing the same things on different inputs, but you can choose to skip steps

## Amdahl's Law

You cannot make your code go faster than the non-parallel bit.

## Background Reading

- This section is based on background reading outside the classroom
- This will save time in the classroom for practical work
- But you do need to know this content
- Read:
  - [Some background history on Wikipedia](#)
  - [Blaise Barney's overview of parallel computing](#)

## Essential Reading

- For the exam you will need:
  - [Amdahl's law](#)
  - [Flynn's Taxonomy](#)
- For tutorial's and practical work you will need:
  - Use of unix shell
  - Submitting jobs on a cluster
  - See [RITS HPC Training](#)
  - We'll recap this now

## Aim

For the remainder of the course, we need to develop

- Skills to run jobs on a cluster, e.g. Legion.
- A locally installed development environment, so you can develop
- Familiarity with new technologies, OpenMP, MPI, Accelerators, Cloud, so you can make a reasoned choice

## Chapter 6

# Shared memory parallelism

### Shared Memory Parallelism

#### OpenMP

- Shared memory only parallelization
- Useful for parallelization on a single cluster node
- MPI next week for inter-node parallelization
- Can write hybrid code with both OpenMP and MPI

#### About OpenMP

- Extensions of existing programming languages
- Standardized by international [committee](#)
- Support for Fortran, C and C++
- C/C++ uses the same syntax
- Fortran is slightly different

#### How it works

- Thread based parallelization
- A master thread starts executing the code.
- Sections of the code is marked as parallel
  - A set of threads are forked and used together with the master thread
  - When the parallel block ends the threads are killed or put to sleep



## Typical use cases

- A loop with independent iterations
- Hopefully a significant part of the execution time
- More complicated if an iterations have dependencies

## OpenMP

### OpenMP basic syntax

- Annotate code with `#pragma omp ...`
  - This instruct the compiler in how to parallize the code
  - `#pragmas` are a instructions to the compiler
  - Not part of the language
  - i.e. like `#pragma once` alternative to include guards
  - Compiler will usually ignore pragmas that it doesn't understand
  - All OpenMP pragmas start with `#pragma omp`
- OpenMP must typically be activated when compiling code

### OpenMP library

- OpenMP library:
  - It provides utility functions.
  - `omp_get_num_threads()` ...
  - Use with `#include <omp.h>`

### Compiler support

OpenMP is supported by most compilers.

- OpenMP must typically be activated with a command line flags at compile time. Different for different compilers. Examples:
  - Intel, Linux, Mac `-openmp`
  - Intel, Windows `/Qopenmp`
  - GCC/G++, `-fopenmp`

Clang OpenMP is slightly different from other OpenMP code, and as such may not be able to work together (as clang does not support 128-bit floats).

## CMake Support

CMake knows how to deal with OpenMP, mostly:

```
find_package(OpenMP)

add_program(my_threaded_monster main.cc)
if(OPENMP_FOUND)
    target_compile_options(my_threaded_monster PUBLIC "${OpenMP_CXX_FLAGS}")
    target_link_libraries(my_threaded_monster PUBLIC "${OpenMP_CXX_FLAGS}")
endif()
```

## Hello world

```
#include <iostream>
#include <omp.h>

int main(int argc, char ** argv)
{
    #pragma omp parallel
    {
        int threadnum = 0;
        int numthreads = 0;
        threadnum = omp_get_thread_num();
        numthreads = omp_get_num_threads();
        std::cout << "Hello World, I am " << threadnum
                  << " of " << numthreads << std::endl;
    }
}
```

- `#pragma omp parallel` marks a block is to be run in parallel
- In this case all threads do the same
- No real work sharing

## Issues with this example

- `std::cout` is not thread safe. Output from different threads may be mixed
  - Try running the code
  - Mixed output?
- All threads call `omp_get_num_threads()` with the same result
  - Might be wasteful if this was a slow function
  - Everybody stores a copy of `numthreads`
  - Waste of memory

## Slightly improved hello world

```
#include <iostream>
#ifdef _OPENMP
#include <omp.h>
#endif

int main(int argc, char ** argv)
{
    int threadnum = 0;
    int numthreads = 0;
    #pragma omp parallel shared(numthreads), private(threadnum)
    {
        #ifdef _OPENMP
            threadnum = omp_get_thread_num();
            #pragma omp single
            {
                numthreads = omp_get_num_threads();
            }
        #endif
        #pragma omp critical
        {
            std::cout << "Hello World, I am " << threadnum <<
                " of " << numthreads << std::endl;
        }
    }
}
```

## Improvements:

- Use `#pragma omp critical` to only allow one thread to write at a time
  - Comes with a performance penalty since only one thread is running this code at a time
- Use Preprocessor `#ifdef _OPENMP` to only include code if OpenMP is enabled
  - Code works both with and without OpenMP
- Variables defined outside parallel regions
  - Must be careful to tell OpenMP how to handle them
  - `shared`, `private`, `first private`
  - More about this later
- `#pragma omp single`
  - Only one thread calls `get_num_threads()`

## Running OpenMP code for the course

If you have a multicore computer with GCC or other suitable compiler you can run it locally.

## References

- [OpenMP homepage](#)
- [OpenMP cheat sheet](#)
- [OpenMP specifications](#)

## Parallelizing loops with OpenMP

### Simple example

Integrate:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

```
#include <iostream>

int main(int argc, char ** argv)
{
    double pi,sum,x;
    const int N = 10000000;
    const double w = 1.0/N;

    pi = 0.0;
    sum = 0.0;
    #pragma omp parallel private(x), firstprivate(sum), shared(pi)
    {
        #pragma omp for
        for (int i = 0; i < N; ++i)
        {
            x = w*(i-0.5);
            sum = sum + 4.0/(1.0 + x*x);
        }
        #pragma omp critical
        {
            pi = pi + w*sum;
        }
    }
    std::cout << "Result is " << pi << std::endl;
}
```

## Variable scope

- Private: Each thread has its own copy. Any values from before the `parallel` block are ignored, and are not affected after.
- Shared: Only one shared variable. Shared both between threads and with the non parallel code.
- Firstprivate: Private variable but initialized with value from before the `parallel` block.

## Details of example

- Important that `x` and `sum` are private
  - Try making them shared and see what happens
- Note that the default is shared
  - Can be controlled with the default clause
  - `default(none)` is safer
  - “Explicit is better than implicit”
- We use a critical region to add safety without a race condition

## Reduction

- Aggregating a result from multiple threads with a single mathematical operation
- Is a very common pattern
- OpenMP has build in support for doing this
- Simplifies the code and avoids the explicit critical region
- Easier to write and may perform better

## Reduction example

```
#include <iostream>

int main(int argc, char ** argv)
{
    double pi,sum,x;
    const int N = 10000000;
    const double w = 1.0/N;

    pi = 0.0;
    sum = 0.0;
```

```

#pragma omp parallel private(x), reduction(+:sum)
{
    #pragma omp for
    for (int i = 0; i < N; ++i)
    {
        x = w*(i-0.5);
        sum = sum + 4.0/(1.0 + x*x);
    }
}
pi = w*sum;
std::cout << "Result is " << pi << std::endl;
}

```

## Races, locks and critical regions

### Introduction

In the best of worlds our calculations can be done independently. However, even in our simplest examples we saw issues.

- `std::cout` is not thread safe. Garbage mixed output
- Needs to use `critical` to merge output
- Real world examples may be more complicated
- Incorrectly shared variable leads to random and typically wrong results

### Race conditions

When the result of a calculation depends on the timing between threads.

- Example: threads writing to same variable
- Can be hard to detect
- May only happen in rare cases
- May only happen on specific platforms
- Or depend on system load from other applications

### Barriers and synchronisation

Typically it is necessary to synchronize threads. Make sure that all threads are done with a piece of work before moving on. Barriers synchronizes threads.

- Parallel regions such as `omp for` have an implicit barrier at the end

- Threads wait for the last to finish before moving on
- May waste significant amount of time
- We will return to look at load balancing later
- Sometimes there is no need to wait
- Disable implicit barrier with `nowait`
- Sometimes you need a barrier where there is no implicit barrier
  - `#pragma omp barrier` inserts a barrier
  - Overusing this may lead to a performance drop

## Protecting code and variables

- `#pragma omp critical`
  - Only one task can execute at a time
  - Protect non thread-safe code
- `#pragma omp single`
  - Only one thread executes this block
  - The first thread that arrives will execute the code
- `#pragma omp master`
  - Similar to `single` but uses the master thread
- `#pragma omp atomic`
  - Protect a variable by changing it in one step.

## Mutex locks

Sometimes the critical regions are not flexible enough to implement your algorithm.

Examples:

- Need to prevent two different pieces of code from running at the same time.
- Need to lock only a fraction of a large array.

## OpenMP locks

OpenMP locks is a general way to manage resources in threads.

- A thread tries to set the lock.

- If the lock is not held by any other thread it is successful and free to carry on.
- If not it will wait until the lock becomes unset.
- Important to remember to unset the lock when done.
- Might otherwise result in a deadlock. Program hangs.

## Example

Replace the critical region with a lock. In this case there is no real gain from using a lock.

```
#include <iostream>
#include <omp.h>

int main(int argc, char ** argv)
{
    double pi,sum,x;
    const int N = 10000000;
    const double w = 1.0/N;
    omp_lock_t writelock;
    pi = 0.0;
    sum = 0.0;
    #pragma omp parallel private(x), firstprivate(sum), shared(pi)
    {
        #pragma omp for
        for (int i = 0; i < N; ++i)
        {
            x = w*(i-0.5);
            sum = sum + 4.0/(1.0 + x*x);
        }
        omp_set_lock(&writelock);
        pi = pi + w*sum;
        omp_unset_lock(&writelock);
    }
    omp_destroy_lock(&writelock);
    std::cout << "Result is " << pi << std::endl;
}
```

## Multiple locks

Sometimes it is useful to lock multiple resources with different locks.

- Use multiple locks protecting different resources



- Can result in deadlocks if two threads needs both needs the same locks
- One thread holds one lock and the other one holds the other
- Both are waiting for a lock to be free

## Notes

OpenMP implements two types of locks. We have only considered simple locks. Consult the [OpenMP specifications](#) for nested locks.

## OpenMP Tasks

### Introduction

- Not all problems are easily expressed as for loops.
- The task construction creates a number of tasks
- The tasks are added to a queue
- Threads take a task from the queue

### Example

Calculate Fibonacci numbers by recursion.

- Only as an example:
  - Hard to get any performance improvement. Usually slower than serial code
  - Inefficient algorithm in any case. Why?
  - Consider limiting the number of tasks. Why?
- Use taskwait to ensure results are done before adding their results together

### Code

```
#include <iostream>
#ifdef _OPENMP
#include <omp.h>
#endif

int fib(int n)
{
    if (n < 2)
```

```

        return n;
    int x;
    int y;
    const int tune = 40;
    #pragma omp task firstprivate(n) shared(x)
    {
        x = fib(n-1);
    }
    #pragma omp task firstprivate(n) shared(y)
    {
        y = fib(n-2);
    }
    #pragma omp taskwait

    return x + y;
}

```

## Main function

```

int main(int argc, char ** argv)
{
    #ifdef _OPENMP
    omp_set_dynamic(0);
    #endif
    const int num = 20;
    int a;
    #pragma omp parallel shared(a)
    {
        #pragma omp single nowait
        {
            a = fib(num);
        }
    }
    std::cout << "fib " << num << " is " << a << std::endl;
}

```

Note only one thread initially creates tasks. Tasks are still running in parallel.

## Advanced usage

- Task dependency:
  - Depends on child tasks. `#taskwait`

- Real cases may be more complicated
- May need to explicitly set dependency
- `#pragma omp task depends(in/out/inout:variable)`
- See OpenMP docs for details
- `taskyield` Allows a task to be suspended in favour of a different task:
  - Could be useful together with locks

## Controlling task generation

- `if(expr) expr==false` create an undeferred task
  - Suspend the present task and execute the new task immediately on the same thread
- `final(expr) expr==true` This is the final task
  - All child tasks are included in the present task
- `mergeable`
  - Included and undeferred tasks may be merged into the parent task

May useful to avoid creating too many small tasks. I.e. in our Fibonacci example.

## Scheduling and Load Balancing

### Number of threads

The number of threads executing an OpenMP code is determined by the environmental variable `OMP_NUM_THREADS`.

Normally `OMP_NUM_THREADS` is equal to the number of CPU cores

### Load balancing

Consider our earlier example of a for loop. 10,000,000 iterations split on 4 cores

Two obvious strategies:

- Split in 4 chunks:
  - 2,500,000 iterations for each core
  - Minimal overhead for managing threads
  - Probably a good solution if the cost is independent of thread

- But what if the cost depends on the thread.
- One thread might be slower than the rest
- Give each thread one iteration at a time
  - No idling thread
  - But huge overhead

The best solution is probably somewhere in between.

## OpenMP strategies

OpenMP offers a number of different strategies for load balancing set by the following key words. The default is static with one chunk per thread.

- **static**: Iterations are divided into chunks of size `chunk_size` and assigned to threads in round-robin order
- **dynamic**: Each thread executes a chunk of iterations then requests another chunk until none remain
- **guided**: Like dynamic but the chunk size depends on the number of remaining iterations
- **auto**: The decision regarding scheduling is delegated to the compiler and/or runtime system
- **runtime**: The schedule and chunk size are controlled by runtime variables

## Which strategy to use

It is hard to give general advice on the strategy to use. Depends on the problem and platform. Typically needs benchmarking and experimentation.

- If there is little variation in the runtime of iteration static with a large chunk size minimizes overhead
- In other cases it might make sense to reduce the chunk size or use dynamic or guided
- Note that both dynamic and guided comes with additional overhead to schedule the distribution of work

## Alternatives to OpenMP

### Overview

- OpenACC:

- Similar to OpenMP but intended for accelerators.
- MPI:
  - For distributed memory systems. Next lecture
- POSIX and Windows threads. Not portable across operation systems
- C++ 11 threads
- [Intel Threading Building Blocks](#) C++ only template library
  - Somewhat more complicated. Requires good understanding of templated code
- [Intel Cilk Plus](#) C and C++, Intel and GCC >= 4.9 only

## C++11

Simple example:

```
#include <thread>
#include <iostream>

void f()
{
    std::cout << "Hello" << std::endl;
};

void g()
{
    std::cout << "world" << std::endl;
};

int main(int argc, char ** argv)
{
    std::thread t1 {f};
    std::thread t2 {g};

    t1.join();
    t2.join();
}
```

## Details

- Same problem as first OpenMP example. `std::cout` is not thread safe
  - Use mutex and locks

- Queues
- Futures and promise
- packed\_task

Likely more suited for multi threaded desktop applications than scientific software.

## Cilk Plus

```
#include <iostream>
#include <time.h>
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>

int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}

int main(int argc, char ** argv)
{
    const int n = 35;
    if (argc > 1)
    {
        // Set the number of workers to be used
        __cilkrts_set_param("nworkers", argv[1]);
    }

    int a = fib(n);
    std::cout << "fib(" << n << ") is " << a << std::endl;
}
```

## Further Reading

### Tutorials

Any additional tutorials can be added here:

- [Blaise Barney, Lawrence Livermore National Laboratory](#)

## Wavelet decomposition

### Wavelet transforms

- [Wavelet transforms](#) decompose a signal into details of increasingly large scales
- Given a signal  $s[j]$ , with  $j \in [1, N]$ , we want to split
  - an approximation  $A_s[i]$  with  $i \in [1, N/2]$
  - the details  $D_s[i]$  with  $i \in [1, N/2]$
- Repeat the process on the approximation

### Pseudo-code

- loop over levels in  $[0, L^{\max}]$

For simplicity, the signal is periodic:  $s[j] = s[j + N]$ .

- set  $N' = N$  if  $N$  is even,  $N' = N + 1$  otherwise
- with  $j \in [0, N' - 1], i \in [0, n - 1]$
- set  $D^s[j] = \sum_{i=0}^{n-1} s[2 * j + i] h[i]$
- set  $A^s[j] = \sum_{i=0}^{n-1} s[2 * j + i] l[i]$
- set  $s = A^s$

### Instructions

Given a header and the unit-tests:

1. reconstruct the serial version using the jigsaw implementation file
2. Parallelize with openmp

How many parallelization schemes did you come up with?

If you are not having enough fun, figure out the inverse operation. Wavelets transforms are bijective.

## Header

```
#ifndef CPP_COURSE_WAVELETS_H

#include <cassert>
#include <iostream>
#include <vector>

namespace wavelets {

    ///! \brief Underlying type of all real numbers in the wavelets
    ///! \details Defining a "type hierarchy" makes it easy to modify all the basic
    ///! types underlying a coding project
    typedef double Scalar;

    ///! \brief Represents 1-D signal of any size
    typedef std::vector<Scalar> Signal;

    ///! \brief High and low-pass filter data
    ///! \details
    ///! Note
    ///! ----
    ///!
    ///! This structure declares it's attribute public. And later we declare global
    ///! variables as instances of this type. Both of these aspects are frowned
    ///! upon, *except* in this one case. The wavelet filters are constants for a
    ///! given Daubechey wavelet. For constants of nature and constants of math,
    ///! whether scalar or vectorial (and small), it's okay to use global variable
    ///! and public attributes. Otherwise, beware!
    ///!
    ///! Premature Optimization?
    ///! -----
    ///!
    ///! We could have `std::array<Scalar, N> high_pass` and the same for
    ///! `low_pass`. However, `N`, the number of coefficients, depends on the actual
    ///! wavelet. And in `std::array<Scalar, N>`, `N` needs to be known at compile
    ///! time. We could make this change and the compiler might be able to make use
    ///! of this extra information to produce faster code. However, it would mean
    ///! that `DaubecheyData` needs to be declared as `template<int N> struct
    ///! DaubecheyData`, and all the functions taking `DaubecheyData<N>` as input
    ///! would also need to be templates.
    ///!
    ///! Templating is cool, fun, but viral. So we might as well use the simpler
    ///! implementation and wait for benchmarks and profiling results to tell us
    ///! whether we really need the extra complexity.
    struct DaubecheyData {
```



```

std::vector<Scalar> low_pass;
std::vector<Scalar> high_pass;
};

///! \brief Data for the Daubechey wavelets of type 1
///! \details This data does not change from code to code. In fact, I got them
///! from wikipedia. Constants of nature and constants of math are the only
///! variable that should be declared global (and `const`).
extern DaubecheyData const daubechy1;
///! Data for the Daubechey wavelets of type 2
extern DaubecheyData const daubechy2;

///! \brief Applies a filter starting from a given location of the signal
///! \details The signal is defined by the range `start`, and `end`. `location`
///! must be a position inside that range. The filter is applied starting from
///! that location of the signal.
Scalar apply_cyclical_filter(Signal::const_iterator const &start, Signal::const_iterator loc
                             Signal::const_iterator const &end, Signal const &filter);

///! \brief Applies a filter starting from a given location of the signal
///! \details Accumulates the result of `signal[location + i] * filter[i]` for i
///! in `[0, filter.size()[`. If `location + i` goes out of range, then it
///! cycles back to the begining of the signal. In practice, this means the
///! signal is periodic.
///!
///! This is a thin wrapper around the function that does the work. Its purpose
///! is to provide a user-friendly interface for end users and for testing.
Scalar
apply_cyclical_filter(Signal const &signal, Signal::size_type location, Signal const &filter

///! \brief Applies the wavelet transform once to the signal
///! \details The output iterator should point to a valid range of the same size
///! as the input signal *if the size of the signal is even*, and one larger than the signal
///! *if the size of the signal is odd*. The result is undefined when the signal
///! and output arrays overlap.
void single_direct_transform(Signal::const_iterator const &start, Signal::const_iterator con
                             Signal::iterator const &out, DaubecheyData const &wavelet);

///! \brief Applies high and low pass once to the signal range
///! \details This wrapper is also to make testing somewhat simpler.
Signal single_direct_transform(Signal const &signal, DaubecheyData const &wavelet);

///! \brief Applies the wavelet transform `levels` time to the signal
///! \details The number of coefficients is given by the function
///! `number_of_coefficients`.
void direct_transform(Signal::const_iterator start, Signal::const_iterator end,

```

```

        Signal::iterator out, DaubechyData const &wavelet, unsigned int levels)

    ///! \brief Figures out number of coeffs
    unsigned int number_of_coefficients(unsigned int signal_size, unsigned int levels);

    ///! \brief Transforms the input signal
    Signal direct_transform(Signal const &signal, DaubechyData const &wavelet, unsigned int levels)
    {
    }
    #endif

```

## Tests

```

#define CATCH_CONFIG_MAIN // This tells Catch to provide a main() - only do this in one cpp
#include "wavelets.h"
#include <algorithm>
#include <catch/catch.hpp>
#ifdef _OPENMP
#include <omp.h>
#endif

TEST_CASE("Application of Cyclical Filters")
{
    wavelets::Signal const signal{ 1, 2, 3, 5, 5, 6 };
    wavelets::Signal const filter{ 3, -3 };

    SECTION("No cyclical overlap problem")
    {
        CHECK(wavelets::apply_cyclical_filter(signal, 0, filter) == Approx(-3));
        CHECK(wavelets::apply_cyclical_filter(signal, 1, filter) == Approx(-3));
        CHECK(wavelets::apply_cyclical_filter(signal, 2, filter) == Approx(-6));
    }
    SECTION("Filter reaches exactly end of signal")
    {
        CHECK(wavelets::apply_cyclical_filter(signal, 4, filter) == Approx(-3));
    }
    SECTION("Filter reaches past end of signal")
    {
        CHECK(wavelets::apply_cyclical_filter(signal, 5, filter) == Approx(15));
    }

    SECTION("Empty filter")
    {
        CHECK(wavelets::apply_cyclical_filter({ 5, 2 }, 0, {}) == Approx(0));
    }
    SECTION("Empty signal")

```

```

    {
        CHECK(wavelets::apply_cyclical_filter({}, 0, { 1, 2 }) == Approx(0));
    }
    SECTION("Empty signal and filter")
    {
        CHECK(wavelets::apply_cyclical_filter({}, 0, {}) == Approx(0));
    }
    SECTION("Filter much much bigger than signal")
    {
        CHECK(wavelets::apply_cyclical_filter({ 1, 2 }, 0, { 1, 2, 3, 4, 5 }) == Approx(21));
    }
}

TEST_CASE("Single pass wavelet transform")
{
    // Use unnormalized haar wavelets, because simple
    // Normalize by multiplyint the coeffs by std::sqrt(2)
    wavelets::DaubechyData const haar{ { 1, 1 }, { 1, -1 } };
    SECTION("Haar")
    {
        // Odd number of elements to test periodicity
        auto const actual = single_direct_transform({ 1, 2, 3, 5, 5, 6, 8 }, haar);
        //                                     high pass             low pass
        //                                     =                       =
        //                                     details             approximation
        wavelets::Signal const expected{ -1, -2, -1, 7, /* */ 3, 8, 11, 9 };
        REQUIRE(actual.size() == expected.size());
        CHECK(std::equal(actual.begin(), actual.end(), expected.begin()));
    }

    SECTION("Empty signal")
    {
        CHECK(wavelets::single_direct_transform({}, haar).size() == 0);
    }
}

TEST_CASE("Multi-pass wavelet transform")
{
    wavelets::DaubechyData const haar{ { 1, 1 }, { 1, -1 } };
    SECTION("Empty signal")
    {
        CHECK(wavelets::direct_transform({}, haar, 2).size() == 0);
    }
    SECTION("Level 0 is a copy")
    {
        wavelets::Signal const signal{ 1, 2, 3, 4 };
    }
}

```

```

    auto const actual = wavelets::direct_transform(signal, haar, 0);
    REQUIRE(actual.size() == signal.size());
    CHECK(std::equal(actual.begin(), actual.end(), signal.begin()));
}
SECTION("Haar")
{
    wavelets::Signal const signal{ 1, 2, 3, 5, 5, 6, 8 };
    std::vector<wavelets::Signal> const expecteds{
        signal, // level 0
        { -1, -2, -1, 7, 3, 8, 11, 9 }, // level 1
        { -1, -2, -1, 7, -5, 2, 11, 20 }, // level 2
        { -1, -2, -1, 7, -5, 2, -9, 31 }, // level 3
        { -1, -2, -1, 7, -5, 2, -9, 0, 62 }, // level 4
        { -1, -2, -1, 7, -5, 2, -9, 0, 0, 124 }, // level 5
        { -1, -2, -1, 7, -5, 2, -9, 0, 0, 0, 248 }, // level 6
    };
    for (decltype(expecteds.size()) levels(0); levels < expecteds.size(); ++levels) {
        auto const actual = direct_transform(signal, haar, levels);
        REQUIRE(actual.size() == expecteds[levels].size());
        CHECK(std::equal(actual.begin(), actual.end(), expecteds[levels].begin()));
    }
}

#ifdef _OPENMP
// Catch offers to print out the duration of a test, so we can use it as a
// make-do benchmarking framework
// Run the executable with --duration yes
TEST_CASE("OpenMP")
{
    wavelets::Signal input(1000000);
    wavelets::Signal output(1000000);
    std::generate(input.begin(), input.end(), std::rand);
    std::fill(output.begin(), output.end(), 0);
    SECTION("Serial")
    {
        auto const nthreads = omp_get_max_threads();
        omp_set_num_threads(1);
        for (int i(0); i < 1000; ++i)
            wavelets::direct_transform(
                input.begin(), input.end(), output.begin(), wavelets::daubechy2, 8);
        omp_set_num_threads(nthreads);
    }
    SECTION("Parallel")
    {
        for (int i(0); i < 1000; ++i)

```

```

        wavelets::direct_transform(
            input.begin(), input.end(), output.begin(), wavelets::daubechy2, 8);
    }
}
#endif

```

## Implementation

```

#include "wavelets.h"
#include <cassert>
#include <numeric>
#ifdef _OPENMP
#include <omp.h>
#endif

namespace wavelets {

DaubechyData const daubechy1{
    { 7.071067811865475244008443621048490392848359376884740365883398e-01,
      7.071067811865475244008443621048490392848359376884740365883398e-01 },
    { -7.071067811865475244008443621048490392848359376884740365883398e-01,
      7.071067811865475244008443621048490392848359376884740365883398e-01 }
};

DaubechyData const daubechy2{
    { 4.829629131445341433748715998644486838169524195042022752011715e-01,
      8.365163037378079055752937809168732034593703883484392934953414e-01,
      2.241438680420133810259727622404003554678835181842717613871683e-01,
      -1.294095225512603811744494188120241641745344506599652569070016e-01 },
    { 1.294095225512603811744494188120241641745344506599652569070016e-01,
      2.241438680420133810259727622404003554678835181842717613871683e-01,
      -8.365163037378079055752937809168732034593703883484392934953414e-01,
      4.829629131445341433748715998644486838169524195042022752011715e-01 }
};

Scalar apply_cyclical_filter(
    Signal const& signal, Signal::size_type location, Signal const& filter)
{
    return apply_cyclical_filter(
        signal.begin(), signal.begin() + location, signal.end(), filter);
}

Signal single_direct_transform(Signal const& signal, DaubechyData const& wavelet)
{
    Signal coefficients(number_of_coefficients(signal.size(), 1));

```

```

    single_direct_transform(signal.begin(), signal.end(), coefficients.begin(), wavelet);
    return coefficients;
}

Signal direct_transform(
    Signal const& signal, DaubechyData const& wavelet, unsigned int levels)
{
    // shifting the bit left is the same as multiplying by two
    Signal coefficients(number_of_coefficients(signal.size(), levels));
    direct_transform(signal.begin(), signal.end(), coefficients.begin(), wavelet, levels);
    return coefficients;
}

unsigned int number_of_coefficients(unsigned int signal_size, unsigned int levels)
{
    if (levels == 0)
        return signal_size;
    auto const even_size = signal_size + signal_size % 2;
    if (levels == 1)
        return even_size;
    return even_size / 2 + number_of_coefficients(even_size / 2, levels - 1);
}

Scalar apply_cyclical_filter(
    Signal::const_iterator const& start,
    Signal::const_iterator location,
    Signal::const_iterator const& end,
    Signal const& filter)
{
    if (start == end)
        return 0;

    assert(location >= start);
    assert(location < end);

    Signal::value_type result(0);
    auto i_filter = filter.begin();
    while (i_filter != filter.end()) {
        for (; i_filter != filter.end() and location != end; ++i_filter, ++location)
            result += (*i_filter) * (*location);
        location = start;
    }
    return result;
}

void single_direct_transform(

```

```

        Signal::const_iterator const& start, Signal::const_iterator const& end,
        Signal::iterator const &out, DaubechyData const& wavelet)
{
    assert(start <= end);
    int const half = ((end - start) + (end - start) % 2) / 2;
    #pragma omp parallel
    {
        #pragma omp for
        for (int i=0; i < half; ++i)
            *(out + i) = apply_cyclical_filter(start, start + 2 * i, end, wavelet.high_pass);
        #pragma omp for
        for (int i=0; i < half; ++i)
            *(out + i + half) = apply_cyclical_filter(start, start + 2 * i, end, wavelet.low_pass);
    }
}

void direct_transform(
    Signal::const_iterator start, Signal::const_iterator end,
    Signal::iterator out, DaubechyData const& wavelet,
    unsigned int levels)
{
    if (start == end)
        return

        assert(start <= end);
    if (levels == 0) {
        std::copy(start, end, out);
        return;
    }

    single_direct_transform(start, end, out, wavelet); // first iteration

    auto const half = [](Signal::size_type n) { return (n + n % 2) / 2; };
    auto approx_size = half(end - start);
    Signal work_array(approx_size);
    auto const i_work = work_array.begin();
    for (unsigned int i(1); i < levels; ++i, approx_size = half(approx_size)) {
        out += approx_size;
        std::copy(out, out + approx_size, i_work);
        single_direct_transform(i_work, i_work + approx_size, out, wavelet);
    }
}
}

```

## Chapter 7

# Distributed memory parallelism

### Distributed Memory Parallelism

#### Objectives

- Understand concepts of distributed memory parallelism
- Gain experience in common communication patterns
- Learn the fundamentals of programming with MPI

#### The basic idea

- Many processes, each with their own data



- Each process is independent
- Processes can send messages to one another



## MPI in practice

### Specification and implementation

- In practice, we use MPI, the [Message Passing Interface](#)
- MPI is a **specification** for a **library**
- It is implemented by separate vendors/open-source projects
  - [OpenMPI](#)
  - [MPICH](#)
- It is a C library with many many bindings:
  - Fortran (part of official MPI specification)
  - Python: [boost](#), [mpi4py](#)
  - R: [Rmpi](#)
  - C++: [boost](#)

### Programming and running

- An MPI program is executed with `mpiexec -n N [options] nameOfProgram [args]`
- MPI programs call methods from the mpi library

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
              MPI_Comm comm)
```

- Vendors provide wrappers (mpiCC, mpic++) around compilers. Wrappers point to header file locations and link to the right libraries. An MPI program can be (easily) compiled by substituting `(g++|icc) -> mpiCC`

### Hello, world!: hello.cc

```
#include <mpi.h>
#include <iostream>

int main(int argc, char * argv[]) {
    /// Must be first call
    MPI_Init(&argc, &argv);
    /// Now MPI calls possible

    /// Size of communicator and process rank
    int rank, size;
```

```

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

std::cout << "Processor " << rank << " of " << size << " says hello\n";

/// Must be last MPI call
MPI_Finalize();
/// No more MPI calls from here
return 0;
}

```

## Hello, world!: CMakeLists.txt

```

find_package(MPI REQUIRED)

add_executable(hello hello.cc)
target_include_directories(hello SYSTEM PUBLIC ${MPI_INCLUDE_DIRS})
target_link_libraries(hello PUBLIC ${MPI_LIBRARIES})

```

## Hello, world!: compiling and running

On `aristotle.rc.ucl.ac.uk`:

- Load modules:

```

module swap compilers/gnu
module unload mpi/intel/2015/update3/intel
module load mpi/openmpi/3.0.0/gnu-4.9.2
module load cmake

```

- Create files “hello.cc” and “CMakeLists.txt” in some directory
- Create build directory: `mkdir build && cd build`
- Run cmake and make: `cmake .. && make`
- Run the code: `mpiexec -n 4 hello`

## Hello, world! dissected

- MPI calls **must** appear between `MPI_Init` and `MPI_Finalize`

- Groups of processes are handled by a communicator. `MPI_COMM_WORLD` han-



dles the group of all processes.

- All know size of group and rank (order) of process in group
- By **convention**, process of rank 0 is **special** and called **root**

## MPI with CATCH

Running MPI unit-tests requires `MPI_Init` and `MPI_Finalize` before and after the test framework (**not** inside the tests).

```
#include <mpi.h>
// Next line tells CATCH we will use our own main function
#define CATCH_CONFIG_RUNNER
#include "catch.hpp"

TEST_CASE("Just test I exist") {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    CHECK(size > 0); CHECK(rank >= 0);
}

int main(int argc, char * argv[]) {
    MPI_Init(&argc, &argv);
    int result = Catch::Session().run(argc, argv);
    MPI_Finalize();
    return result;
}
```

## Point to point communication






### Many point-2-point communication schemes

Can you think of two behaviours for message passing?







- Process 0 can (i) give message and then either (ii) leave or (iii) wait for acknowledgements
- Process 1 can (i) receive message
- MPI can (i) receive message, (ii) deliver message, (iii) deliver acknowledgements

### Blocking synchronous send




| Stage                         | Figure   |
|-------------------------------|--|
| a. 0, 1, and MPI stand ready: |  |
| b. message dropped off by 0:  |  |
| c. transit:                   |  |
| d. message received by 1      |  |
| e. receipt received by 0      |  |



| Stage | Figure |
|-------|--------|
|-------|--------|

## Blocking send

| Stage                         | Figure   |
|-------------------------------|--|
| a. 0, 1, and MPI stand ready: |    |
| b. message dropped off by 0:  |    |
| c. transit, 0 leaves          |   |
| d. message received by 1      |  |

## Non-blocking send

| Stage                          | Figure   |
|--------------------------------|--|
| a. 0, 1, and MPI stand ready:  |  |
| b. 0 leaves message in safebox |  |
| c. transit                     |  |

| Stage                        | Figure   |
|------------------------------|--|
| d. message received by 1     |  |
| e. receipt placed in safebox |  |

## Blocking synchronous send

```
int MPI_Ssend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm)
```

| Parameter | Meaning   |
|-----------|---|
| buf       | Pointer to buffer. Always void because practical C is not type safe.                                  |
| count     | Size of the buffer. I.e. length of the message to send, in units of the specified datatype (not byte) |
| datatype  | Encodes type of the buffer. MPI_INT for integers, MPI_CHAR for characters. Lots of others.            |
| dest      | Rank of the <b>receiving</b> process  |
| tag       | A tag for message book-keeping  |
| comm      | The communicator – usually just MPI_COMM_WORLD  |
| return    | An error tag. Equals MPI_SUCCESS on success.  |

## Blocking receive

```
int MPI_Recv(void *buf, int bufsize, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

Good for both synchronous and asynchronous communication

| Parameter | Meaning  |
|-----------|--|
| buf       | Pointer to receiving <b>pre-allocated</b> buffer                                     |
| bufsize   | Size of the buffer. I.e. maximum length of the message to receive. See MPI_Get_count |
| datatype  | Inform on the type of the buffer   |

| Parameter | Meaning                                       |
|-----------|---|
| source    | Rank of the <b>sending</b> process            |
| tag       | A tag for message book-keeping                |
| status    | MPI_STATUS_IGNORE for now. See MPI_Get_count. |
| comm      | The communicator                              |
| return    | Error tag                                     |

## Example: Blocking synchronous example

Inside a new section in the test framework:

```
std::string const peace = "I come in peace!";
if (rank == 0) {
    int const error = MPI_Ssend(
        (void*) peace.c_str(), peace.size() + 1, MPI_CHAR, 1, 42, MPI_COMM_WORLD);
    // Here, we guarantee that Rank 1 has received the message.
    REQUIRE(error == MPI_SUCCESS);
}
if (rank == 1) {
    char buffer[256];
    int const error = MPI_Recv(
        buffer, 256, MPI_CHAR, 0, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    REQUIRE(error == MPI_SUCCESS);
    CHECK(std::string(buffer) == peace);
}
```

Common bug: Set both sender and receiver to 0. What happens?

## Example: Do you know your C vs C++ strings?

Why the +1?

```
int const error = MPI_Ssend(
    (void*) peace.c_str(), peace.size() + 1, MPI_CHAR, 1, 42, MPI_COMM_WORLD);

...

```

Because C and C++ `char const*` strings are null-terminated to indicate the string is finished, which adds an extra character. However, `std::string` abstracts it away. And so its length does **not** include the null-termination.

## Example: Causing a dead-lock

Watch out for order of send and receive!

Bad:




```
if (rank == 0) {  
    MPI_Ssend (sendbuf, count, MPI_INT, 1, tag, comm);  
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);  
} else {  
    MPI_Ssend (sendbuf, count, MPI_INT, 0, tag, comm);  
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);  
}
```

...


Good:

```
if (rank == 0) {  
    MPI_Ssend (sendbuf, count, MPI_INT, 1, tag, comm);  
    MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);  
} else {  
    MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);  
    MPI_Ssend (sendbuf, count, MPI_INT, 0, tag, comm);  
}
```

## Blocking send: MPI\_Send

| Stage                         | Figure   |
|-------------------------------|--|
| a. 0, 1, and MPI stand ready: |  |
| b. message dropped off by 0:  |  |
| c. transit, 0 leaves          |  |



| Stage                    | Figure   |
|--------------------------|--|
| d. message received by 1 |  |

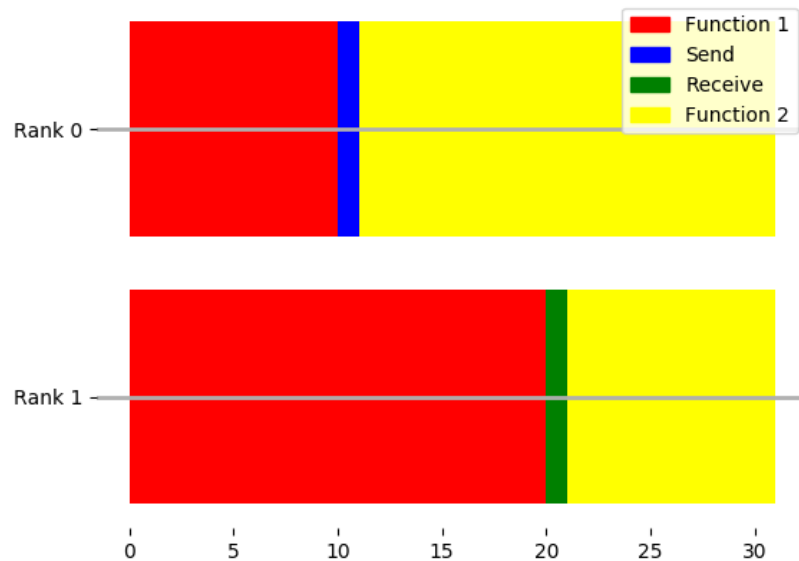
## Send vs SSend

Why would we use Send instead of SSend?

```
std::string peace = "I come in peace!";
if (rank == 0) {
    int const error = MPI_Send(
        (void*) peace.c_str(), peace.size() + 1, MPI_CHAR, 1, 42, MPI_COMM_WORLD);
    // We do not guarantee that Rank 1 has received the message yet
    // But nor do we necessarily know it hasn't.
    // But we are definitely allowed to change the string, as MPI promises
    // it has been buffered
    peace = "Shoot to kill!"; // Safe to reuse the send buffer.
    REQUIRE(error == MPI_SUCCESS);
}
if (rank == 1) {
    char buffer[256];
    int const error = MPI_Recv(
        buffer, 256, MPI_CHAR, 0, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    REQUIRE(error == MPI_SUCCESS);
    CHECK(std::string(buffer) == peace);
}
```

Both guarantee the buffer is safe to reuse. Send makes no guarantee as to whether it returns early or not. But **SSend** forces a **synchronisation point**: the codes reach the matching places, with all processes waiting until all reach that point.

It may come out slightly faster to use Send, since having a synchronisation point when you don't need one can slow things down.



Suppose (A) runs slightly faster than (B) does; at the end, they've both been running fully efficiently.

---

With a synchronisation point in between, you'll have wasted time:

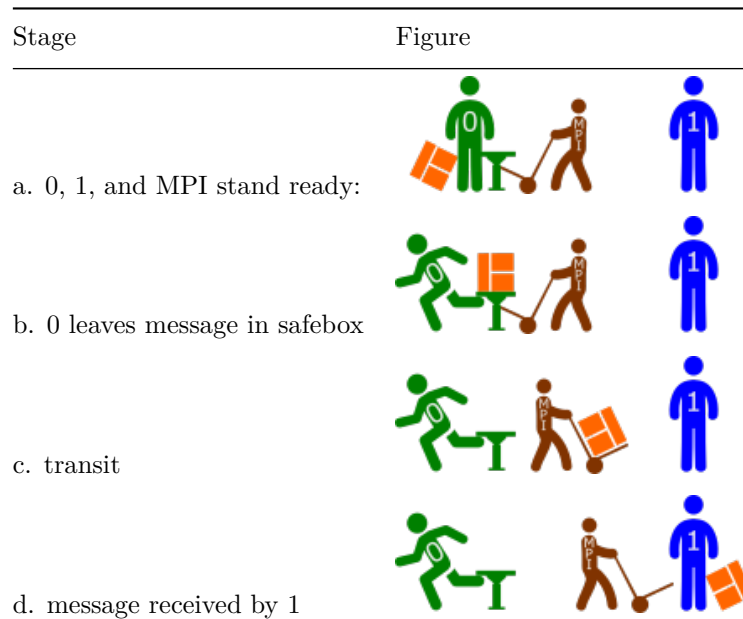
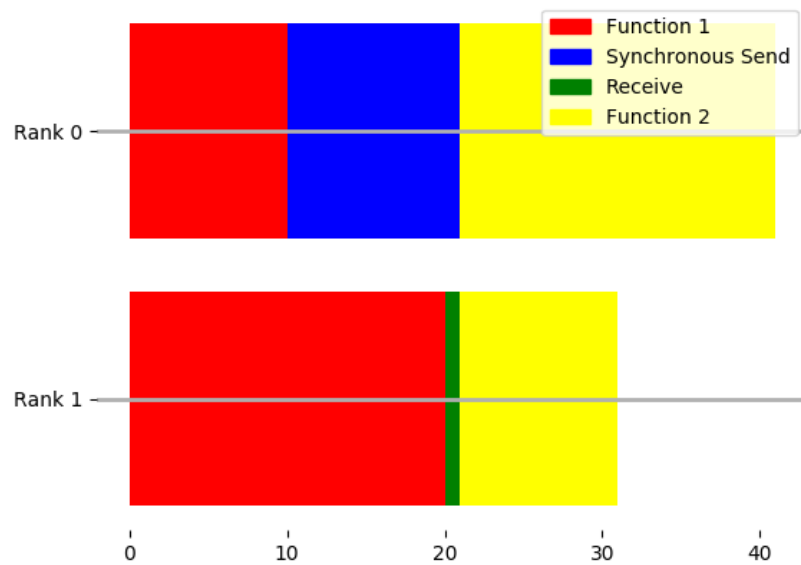
---


This is only important when there is noise or variability in the execution time on different processes, but this is often the case.

So unnecessary synchronisation points are bad. The MPI Implementation may choose to buffer, or synchronise in Send; you're letting MPI guess.

However, if you want to fine tune this to get the best performance, you should use Isend.

## Non-blocking MPI\_Isend



| Stage                        | Figure   |
|------------------------------|--|
| e. receipt placed in safebox |  |

## Non-blocking: Isend/Irecv

With Isend, we indicate when we want the message to set off.

We receive a handle to the message, of type `MPI_Request*` which we can use to require it has been received, or check.

This produces more complicated code, but you can write code which **overlaps calculation with communication**: the message is travelling, while you get on with something else. We'll see a practical example of using this next lecture.

---

```

std::string peace = "I come in peace!";
if (rank == 0) {
    MPI_Request request;
    int error = MPI_Isend(
        (void*) peace.c_str(), peace.size() + 1, MPI_CHAR, 1, 42,
        MPI_COMM_WORLD, &request);
    // We do not guarantee that Rank 1 has received the message yet
    // We can carry on, and ANY WORK WE DO NOW WILL OVERLAP WITH THE
    // COMMUNICATION
    // BUT, we can't safely change the string.
    REQUIRE(error == MPI_SUCCESS);
    // Do some expensive work here
    for (int i=0; i<1000; i++) {}; // BUSYNESS FOR EXAMPLE
    MPI_Status status;
    error = MPI_Wait(&request, &status);
    REQUIRE(error == MPI_SUCCESS);
    // Here, we run code that requires the message to have been
    // successfully sent.
}
if (rank == 1) {
    char buffer[256];
    int const error = MPI_Recv(
        buffer, 256, MPI_CHAR, 0, 42, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```

```

        REQUIRE(error == MPI_SUCCESS);
        CHECK(std::string(buffer) == peace);
    }

```

## Function signatures

```

int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm, MPI_Request *request)

```

```

int MPI_Irecv(void *buf, int bufsize, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Request *request)

```

...

```

int MPI_Recv(void *buf, int bufsize, MPI_Datatype datatype, int source, int tag,
            MPI_Comm comm, MPI_Status *status)

```

## Pass the parcel: SendRecv

Consider a group of N processes in a ring: each has a value, and wants to “pass the parcel” to the left. How would you achieve this with SSend and Receive?

```

int message = rank*rank;
int received = -7;

// Define the ring
int left = rank - 1;
int right = rank + 1;
if (rank == 0) {
    left = size - 1;
}
if (rank == size - 1){
    right = 0;
}

```

---

With synchronous calls each process can only either be sending or receiving. So the even processes need to send, while the odd ones receive, then vice-versa. This is clearly inefficient.

```

if (rank % 2 == 0) {
    int error = MPI_Ssend(
        &message, 1, MPI_INT, left, rank, MPI_COMM_WORLD);
    REQUIRE(error == MPI_SUCCESS);

    error = MPI_Recv(
        &received, 1, MPI_INT, right, right, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    REQUIRE(error == MPI_SUCCESS);
}
if (rank % 2 == 1) {

    int error = MPI_Recv(
        &received, 1, MPI_INT, right, right, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    REQUIRE(error == MPI_SUCCESS);

    error = MPI_Ssend(
        &message, 1, MPI_INT, left, rank, MPI_COMM_WORLD);
    REQUIRE(error == MPI_SUCCESS);
}
REQUIRE( received == right*right );

```

---

With ISend/IRecv, this can be achieved in one go: each process posts its send, then posts its receive, then waits for completion.

```

MPI_Request send_req;
// Everyone sets up their messages to send
int error = MPI_Isend(
    &message, 1, MPI_INT, left, rank, MPI_COMM_WORLD, &send_req);
REQUIRE(error == MPI_SUCCESS);

// Recv acts as our sync-barrier
error = MPI_Recv(
    &received, 1, MPI_INT, right, right, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
REQUIRE(error == MPI_SUCCESS);

// But let's check our send completed:
int completed;
error = MPI_Test(&send_req, &completed, MPI_STATUS_IGNORE);
REQUIRE(error == MPI_SUCCESS);
REQUIRE(completed == true);

REQUIRE( received == right*right );

```

---

However, this is such a common pattern, that there is a separate MPI call to make this easier:

```
int MPI_Sendrecv(void *sendbuf, int scount, MPI_Datatype stype, int dest, int stag,
                 void *recvbuf, int rcount, MPI_Datatype rtype, int source, int rtag,
                 MPI_Comm comm, MPI_Status *status)
```

Each argument is duplicated for the send and receive payloads.

## Classroom exercise

Implement ring-send using Sendrecv.

## Point to point summary

Sending messages:

| name      | Blocking | forces synchronisation point | Buffer-safe |
|-----------|----------|------------------------------|-------------|
| MPI_Ssend | yes      | yes                          | yes         |
| MPI_Send  | maybe    | no                           | yes         |
| MPI_Isend | no       | no                           | no          |

Receiving messages:

| name      | blocking |
|-----------|----------|
| MPI_Recv  | yes      |
| MPI_Irecv | no       |

(There are a few others, rarely used - see [MPI manuals](#)!)

## Collective Communication







### Many possible communication schemes

Think of two possible forms of **collective** communications:







- give a beginning state
- give an end state



### Broadcast: one to many




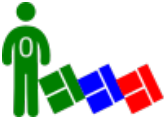


| State                      | Figure  |  |   |
|----------------------------|---|--|---|
|                            |  |   |  |
| data in 0, no data in 1, 2 |   |  |   |
|                            |  |  |  |
| data from 0 sent to 0, 1   |   |  |   |

### Scatter: one to many




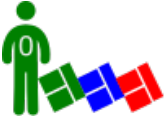


| State                  | Figure  |   |   |
|------------------------|---|---|---|
|                        |  |  |  |
| data in 0              |   |   |   |
|                        |  |  |  |
| data from 0 in 0, 1, 2 |   |   |   |









### Gather: many to one

| State           | Figure  |  |   |
|-----------------|---|--|---|
| data in 0, 1, 2 |  |  |  |
|                 |  |   |  |

### All to All: many to many

| State           | Figure  |  |   |
|-----------------|---|--|---|
| data in 0, 1, 2 |   |   |   |
|                 |  |  |  |

### Reduce operation

| State           | Figure  |   |   |
|-----------------|---|---|---|
| data in 0, 1, 2 |  |  |  |
|                 |  |  |  |

Wherefrom the baby bunny?

...

Sum, difference, or any other **binary** operator:

$$\left( \left( \text{green square} + \text{blue square} \right) + \text{red square} \right) = \text{baby bunny} \quad \left( \left( \text{green square} - \text{blue square} \right) - \text{red square} \right) = \text{baby bunny}$$

There are [predefined operators](#) or you can define your own.

The predefined operators are all both associative and commutative. User-defined ones can be specified as non-commutative (e.g. difference).

## Collective operation API (1)

Group synchronisation:

```
int MPI_Barrier(MPI_Comm comm);
```

---

Broadcasting:

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root,
             MPI_Comm comm)
```

---

| Parameter | Content                             |
|-----------|-------------------------------------|
| buf       | Pointer to sending/receiving buffer |
| count     | Size of the buffer/message          |
| datatype  | Informs on the type of the buffer   |
| root      | Sending process                     |
| comm      | The communicator!                   |
| return    | Error tag                           |

---

## Example of collective operation (1)

Insert into a new `CATCH` section the following commands

```

std::string const peace = "I come in peace!";
std::string message = "";
int error;
if (rank == 0) {
    message = peace;
    error = MPI_Bcast(
        (void*) peace.c_str(), peace.size() + 1, MPI_CHAR, 0, MPI_COMM_WORLD);
} else {
    char buffer[256];
    error = MPI_Bcast(buffer, 256, MPI_CHAR, 0, MPI_COMM_WORLD);
    message = std::string(buffer);
}

```

## Example of collective operations (2)

And then insert the following right after it

```

for (int i(0); i < size; ++i) {
    if (rank == i) {
        INFO("Current rank is " << rank);
        REQUIRE(error == MPI_SUCCESS);
        CHECK(message == peace);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}

```

## Causing deadlocks

Explain why the following two codes fail.

1. Replace the loop in the last fragment with:

```

for (int i(1); i < size; ++i) ...

```

---

2. Refactor and put everything inside the loop

```

std::string const peace = "I come in peace!";
std::string message;

```

```

for (int i(0); i < size; ++i) {
    if (i == 0 && rank == 0) { /* broadcast */ }
    else if (rank == i) { /* broadcast */ }
    if (rank == i) { /* testing bit */ }
    MPI_Barrier(MPI_COMM_WORLD);
}

```

NOTE: a loop with a condition for `i == 0` is a common anti-pattern (i.e. bad)

## Scattering data

```

int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype, int root,
               MPI_Comm comm)

```

| Parameter | Content  |
|-----------|--|
| sendbuf   | Pointer to sending buffer (only significant at root) |
| sendcount | Size of a <b>single</b> message                      |
| datatype  | Type of the buffer                                   |
| recvbuf   | Pointer to receiving buffers (also at root)          |
| recvcount | Size of the receiving buffer                         |
| recvtype  | Informs on the type of the receiving buffer          |

Exercise: Have the root scatter “This....” “message.” “is.split.” to 3 processors (including itself).

## MPI\_Reduce

```

int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm)

```

| Parameter | Content                     |
|-----------|-----------------------------|
| sendbuf   | Pointer to sending buffer   |
| recvbuf   | Pointer to receiving buffer |
| count     | Size of the buffer/message  |

| Parameter | Content                           |
|-----------|-----------------------------------|
| datatype  | Informs on the type of the buffer |
| op        | The binary operation to perform   |
| root      | Receiving process                 |
| comm      | The communicator!                 |
| return    | Error tag                         |

## Exercise: MPI\_Reduce

This example uses the Gregory-Leibniz Series to calculate  $\pi$ .

```
const int TERMS_PER_PROCESS = 1000;
double my_denominator = rank * TERMS_PER_PROCESS * 2 + 1;
double sign = 1;
double my_result = 0.0;
for (int i(0); i<TERMS_PER_PROCESS; ++i)
{
    my_result += sign / my_denominator;
    my_denominator += 2;
    sign = -sign;
}
```

Can you write the parallel code that combines these results and checks the accuracy of the calculation?

See [the MPI\\_Reduce docs](#) to help you.

Extension: use [MPI\\_Allreduce](#) instead so all processes get the final answer.

## Splitting the communicators

Groups of processes can be split according to **color**:

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

| Parameter | Content  |
|-----------|--|
| comm      | Communicator that contains all the processes to be split |

| Parameter | Content  |
|-----------|--|
| color     | All processes with same color end up in same group |
| int key   | Controls rank in final group                       |
| newcomm   | Output communicator                                |

## Splitting communicators: example

The following splits processes into two groups with ratio 1:2.

```
bool const is_apple = rank % 3 == 0;
SECTION("split 1:2 and keep same process order") {
    MPI_Comm apple_orange;
    MPI_Comm_split(MPI_COMM_WORLD, is_apple ? 0 : 1, rank, &apple_orange);

    int nrank, nsize;
    MPI_Comm_rank(apple_orange, &nrank);
    MPI_Comm_size(apple_orange, &nsize);

    int const div = (size - 1) / 3, napples = 1 + div;
    if (is_apple) {
        CHECK(nsize == napples);
        CHECK(nrank == rank / 3);
    } else {
        CHECK(nsize == size - napples);
        CHECK(nrank == rank - 1 - (rank / 3));
    }
}
```

## Splitting communicators: Exercises

1. Use “-rank” as the key: what happens?
2. Split into three groups with ratios 1:1:2
3. Use one of the collective operations on a single group

## Scatter operation solution

```
std::string const message = "This message is going to come out in separate channels";
if (message.size() < (unsigned)size) return;
```



```
int N = message.size() / size;

char buffer[256];
int const error = MPI_Scatter(
    (void*) message.c_str(), N, MPI_CHAR, // Only significant at root
    buffer, 256, MPI_CHAR, // Receiver side
    0, MPI_COMM_WORLD
);
REQUIRE(error == MPI_SUCCESS);
CHECK(message.substr(rank*N, N) == std::string(buffer, N));
```

## More advanced MPI

### Architecture and usage

- Depending on library, MPI processes can be **placed** on a specific node...
- ... and even chained to specific cores
- Fewer processes than cores means we can do MPI + openMP:
- some data is distributed (MPI)
- some data is shared (openMP)
- MPI-3 allows for creating/destroying processes dynamically

### Splitting communicators

`MPI_Group_*` specify operations to create sets of processes. In practice, they define operations on sets:

- union

- intersection
- difference

And allow the creation of a communicator for the resulting group.

`MPI_Comm_group` does the converse, giving you a group from a communicator.

## More MPI data-types

It is possible to declare complex data types

- strided vectors, e.g. only one of every element (`MPI_Type_vector`)
- sub-matrices (strided in  $n$  axes,  $n \geq 2$ ) (`MPI_Type_create_struct`)
- irregular strides (`MPI_Type_indexed`)

## One sided communication

Prior to MPI-3, both sending and receiving processes must be aware of the communication.

One-sided communication allow processes to define a buffer that other processes can access without their explicit knowledge.

## And also

- Cartesian grid topology where process  $(1, 1)$  is neighbour of  $(0, 1)$ ,  $(1, 0)$ ,  $(2, 1)$ ,  $(1, 2)$ . With simplified operations to send data EAST, WEST, UP, DOWN...
- More complex graph topologies
- Non-blocking collective operations



## Chapter 8

# Python Binding

### Python Binding

#### Integrate C++ with Python

Useful if you want to:

- Integrate algorithms in a C++ library, with a wider Python eco-system?
- Optimise a small piece of code, but otherwise use Python

It does suit research rather well.

#### Overview

The basic idea:

- Python interpreter can access functions/classes in a compiled C/C++ shared library
- Write your C/C++ function/class
- Wrap it using a suitable library/macro
- Compile
- import module into Python

### Live Example

#### Overview

- [CMakeCatchTemplate](#) also has Python examples

- Use either [BoostPython](#) or [PyBind11](#)
- [PyBind11](#) is header only, in theory, should be easier
- But Boost seems to have better examples on the internet

## Demo / Code Walk Through

We shall step through a simple function:

- Adds 2 integers
- [Boost.Python example](#)
- [PyBind11 example](#)

## Other examples

- OpenCV uses numpy ndarray (Python) and converts to `cv::Mat` (C++)
- See [Gregory Kramida's pyboostconverter](#), integrated [here](#), using Boost.Python.
- Also see [scikit-surgeryopencv](#), using Boost.Python, OpenCV and [Gregory Kramida's pyboostconverter](#).
- Also see [scikit-surgerypcl](#), using Boost.Python, Boost.Numpy and [PCL](#).

## Chapter 9

# Summary 2019

### Summary 2019

#### Overview of Course

- Building - cmake for multi-platform
- Testing - ctest / catch2 for correctness
- Better C++, smart pointers, exceptions, debugging
- Shared memory parallelism
- Distributed memory parallelism

### CMake - cross-platform building

#### Cross-platform is common

- Researchers deal with small prototypes
- Develop on Windows/Mac, deploy to Linux cluster
- i.e. cross-platform is common
- Researchers have little programming support

#### CMake summary

- CMake generates Makefiles / Visual Studio solution files.
- **make** or Visual Studio program runs your compiler
- CMake is NOT compiling your code

So: Learn 1 build system (CMake), and general build code for each platform (Windows, Linux, Mac)

## Going Forward

- There is a CMake book, but its outdated
- Learn by example
- On this course, I gave as much CMake as I could
- But look at commit log for [CMakeTemplate](#), as I'm still hacking CMake stuff myself.

## Resources

- [CMake homepage](#)
- [CMake hello world](#)
- [CMake library and app](#)
- [CMake with Catch2](#)
- [CMakeTemplate](#) with [SuperBuild](#) / [MetaBuild](#) and the associated [CMakeTemplateRenamer](#) to generate new projects.

## Be a Better Developer

### Practices

- Unit testing: [Catch2](#)
- Continuous Integration: [Travis](#) and [Appveyor](#), all my examples use it.
- Style: [KWStyle](#)
- Linting: [cppcheck](#)
- Memory checking: [valgrind](#) (Linux only)

See also clang tools such as `clang-tidy`, `clang-format`.

## Going Forward

- Once you start C++, read 1 chapter Scott Meyers per day.
- Then dip into it as needed.
- Start each project with good layout and good template, e.g. [CMakeTemplate](#)
- For scientific computing, consider small C++ libraries, over large projects.

## Resources

- All [Scott Meyers](#)
- Also [Andrei Alexandrescu](#)

## Master the Language

### Architecture First

- Understand why code is organised as it is.
- What makes a design extensible?
- What makes a design easy to test?

### Only then Language

- Pick up language features as you go
- Understand the range of compilers you can support
  - [gcc features](#)
  - [MSVC](#)
  - [clang](#)

## Master Parallelism

### Basic

- Only optimise the most necessary bits
- Use basic timing, profiling or [Amdahls law](#)
- [Shared Memory Parallelism - OpenMP](#)
- [Distributed Memory Parallelism - OpenMPI](#)

### More advanced - GPU

- Most people do shared memory on single card
- First try a domain specific library that uses GPU e.g [OpenCV](#)
- Then go more specific if need be: [CUDA](#), [OpenCL](#), [\[ArrayFire\]\[ArrayFire\]](#)
- Can also be distributed across servers, [OpenMPI](#)
- But its hard to deploy, or have access to cards
- You'll need further training courses

## More advanced - Cloud

- Cloud is more about availability of nodes/resources
- i.e. same programming, just can scale differently
- If code uses 1 node, batch processing
- If code uses N nodes, very parallel, but difficult to get hold of.

## Chapter 10

# Appendix - Libraries

## Appendix - C++ Library Examples

### The Story So Far

- Git
- CMake
- Use libraries
- More C++

### Today's Lesson

- C++ libraries
  - Language features
  - Using libraries

## Using Eigen

### Introduction

- [Eigen](#) is:
  - C++ template library,
  - Linear algebra, matrices, vectors, numerical solvers, etc.

| Module             | Header file                                     | Contents   |
|--------------------|---|--|
| <b>Core</b>        | <code>#include &lt;Eigen/Core&gt;</code>        | <b>Matrix</b> and <b>Array</b> classes, basic linear algebra (including triangular and selfadjoint products), array manipulation     |
| <b>Geometry</b>    | <code>#include &lt;Eigen/Geometry&gt;</code>    | <b>Transform</b> , <b>Translation</b> , <b>Scaling</b> , <b>Rotation2D</b> and 3D rotations ( <b>Quaternion</b> , <b>AngleAxis</b> ) |
| <b>LU</b>          | <code>#include &lt;Eigen/LU&gt;</code>          | Inverse, determinant, LU decompositions with solver ( <b>FullPivLU</b> , <b>PartialPivLU</b> )                                       |
| <b>Cholesky</b>    | <code>#include &lt;Eigen/Cholesky&gt;</code>    | LLT and LDLT Cholesky factorization with solver  |
| <b>Householder</b> | <code>#include &lt;Eigen/Householder&gt;</code> | Householder transformations; this module is used by several linear algebra modules   |
| <b>SVD</b>         | <code>#include &lt;Eigen/SVD&gt;</code>         | SVD decomposition with least-squares solver ( <b>JacobiSVD</b> )   |
| <b>QR</b>          | <code>#include &lt;Eigen/QR&gt;</code>          | QR decomposition with solver ( <b>HouseholderQR</b> , <b>ColPivHouseholderQR</b> , <b>FullPivHouseholderQR</b> )                     |
| <b>Eigenvalues</b> | <code>#include &lt;Eigen/Eigenvalues&gt;</code> | Eigenvalue, eigenvector decompositions ( <b>EigenSolver</b> , <b>SelfAdjointEigenSolver</b> , <b>ComplexEigenSolver</b> )            |
| <b>Sparse</b>      | <code>#include &lt;Eigen/Sparse&gt;</code>      | Sparse matrix storage and related basic linear algebra ( <b>SparseMatrix</b> , <b>DynamicSparseMatrix</b> , <b>SparseVector</b> )    |
|                    | <code>#include &lt;Eigen/Dense&gt;</code>       | Includes Core, Geometry, LU, Cholesky, SVD, QR, and Eigenvalues header files   |
|                    | <code>#include &lt;Eigen/Eigen&gt;</code>       | Includes Dense and Sparse header files (the whole Eigen library)   |

## Tutorials

Obviously, you can read:

- the existing [manual pages](#)
- tutorials ([short](#), [long](#)).
- the [Quick Reference](#)

## Getting started

- Header only, just need `#include`
- Uses CMake, but that's just for
  - documentation
  - run unit tests
  - do installation.

## C++ Principles

(i.e. why introduce Eigen on this course)

- Eigen uses
  - Templates
  - Loop unrolling, traits, template meta programming

## Matrix Class

- This:



```

#include <iostream>
#include <Eigen/Dense>
using Eigen::MatrixXd;
int main()
{
    MatrixXd m(2,2);
    m(0,0) = 3;
    m(1,0) = 2.5;
    m(0,1) = -1;
    m(1,1) = m(1,0) + m(0,1);
    std::cout << m << std::endl;
}

```

- Produces:

```

3   -1
2.5 1.5

```

## Matrix Class Declaration

Matrix Class

```

template<typename _Scalar, int _Rows, int _Cols, int _Options, int _MaxRows, int _MaxCols>
class Matrix
: public PlainObjectBase<Matrix<_Scalar, _Rows, _Cols, _Options, _MaxRows, _MaxCols> >
{

```

So, its templates, [so review last weeks lecture](#).

## Matrix Class Construction

But in documentation:

All combinations are allowed: you can have a matrix with a fixed number of rows and a dynamic number of columns, e are all valid:

```

Matrix<double, 6, Dynamic>           // Dynamic number of columns (heap allocation)
Matrix<double, Dynamic, 2>           // Dynamic number of rows (heap allocation)
Matrix<double, Dynamic, Dynamic, RowMajor> // Fully dynamic, row major (heap allocation)
Matrix<double, 13, 3>                 // Fully fixed (usually allocated on stack)

```

Figure 10.1: Matrix construction

It took a while but I searched and found:

```

src/Core/util/Constants.h:const int Dynamic = -1;

```

and both fixed and dynamic Matrices come from same template class???

How do they do that?

## DenseStorage.h - 1

In src/Core/DenseStorage.h:

```
template <typename T, int Size, int MatrixOrArrayOptions,
         int Alignment = (MatrixOrArrayOptions&DontAlign) ? 0
                        : (((Size*sizeof(T))%16)==0) ? 16
                        : 0 >
struct plain_array
{
    T array[Size];
```

So, a plain\_array structure containing a stack allocated array.

## DenseStorage.h - 2

In src/Core/DenseStorage.h:

```
// purely fixed-size matrix
template<typename T, int Size, int _Rows, int _Cols, int _Options> class DenseStorage
{
    internal::plain_array<T,Size,_Options> m_data;
```

There is a default template class for DenseStorage, and specialisation for fixed arrays.

## DenseStorage.h - 3

In src/Core/DenseStorage.h:

```
// purely dynamic matrix.
template<typename T, int _Options> class DenseStorage<T, Dynamic, Dynamic, Dynamic, _Options>
{
    T *m_data;
    DenseIndex m_rows;
    DenseIndex m_cols;
```

There is a default template class for DenseStorage, and specialisation for Dynamic arrays.

## Eigen Matrix Summary

- Templated type supports dynamic and fixed arrays seamlessly on stack or heap
- typedef's to make life easier: `Matrix3d` = 3 by 3 of double
- Uses TMP to generate code at compile time
- Benefit from optimisations such as loop unrolling when using fixed size constant arrays

## Eigen Usage - CMake Include

- Need to set include path
- You could download and 'install' eigen into your project, and commit it. e.g.

```
include_directories(${CMAKE_SOURCE_DIR}/session03/cpp/Eigen/eigen-3.2.3/include/eigen3)
```

## Eigen Usage - CMake Module

- CMake (3.1) does not have a `Find Module` for eigen, but eigen provides one.
- So, in your source tree

```
mkdir CMake
cp <path_to_eigen>/cmake/FindEigen3.cmake ./CMake
```

- Then in your `CMakeLists.txt`

```
set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/CMake;${CMAKE_MODULE_PATH}")
find_package(Eigen3)
include_directories(${EIGEN3_INCLUDE_DIR})
```

## Eigen Usage - CMake External

- [NiftySeg](#) uses

```
option(USE_SYSTEM_EIGEN "Use an already installed version of the Eigen library" OFF)
if(USE_SYSTEM_EIGEN)
    find_package(EIGEN REQUIRED)
else()
    set(${PROJECT_NAME}_VERSION_EIGEN "ffa86ffb5570" CACHE STRING "Version of EIGEN" FORCE)
```

```

set(${PROJECT_NAME}_MD5_SUM_EIGEN 9559c34af203dde5f3f1d976d859c5b3 CACHE STRING "MD5 check
set(${PROJECT_NAME}_LOCATION_EIGEN
    "http://cmic.cs.ucl.ac.uk/platform/dependencies/eigen-eigen-${${PROJECT_NAME}_VERSION_E
    CACHE STRING "Location of Eigen" FORCE)
ExternalProject_Add(Eigen
    URL ${${PROJECT_NAME}_LOCATION_EIGEN}
    URL_MD5 ${${PROJECT_NAME}_MD5_SUM_EIGEN}
    PREFIX ${PROJECT_BINARY_DIR}/Eigen
    DOWNLOAD_DIR ${PROJECT_BINARY_DIR}/Eigen/download
    SOURCE_DIR ${PROJECT_BINARY_DIR}/Eigen/source
    STAMP_DIR ${PROJECT_BINARY_DIR}/Eigen/stamps
    TMP_DIR ${PROJECT_BINARY_DIR}/Eigen/tmp
    BINARY_DIR ${PROJECT_BINARY_DIR}/Eigen/build
    CMAKE_ARGS
        ${CMAKE_PROPAGATED_VARIABLES}
        -DCMAKE_INSTALL_PREFIX:PATH=${PROJECT_BINARY_DIR}/Eigen/install
        -DBUILD_TESTING=1
    )
set(Eigen_INCLUDE_DIR ${PROJECT_BINARY_DIR}/Eigen/install/include/eigen3)
endif()
include_directories(${Eigen_INCLUDE_DIR})

```

## Eigen Example - in PCL

[Point Cloud Library](#) uses [Eigen](#). Lets look at point based registration of two, same length, point sets.

## PCL - Manual Registration

- Class to hold point lists
- Callbacks (not shown here) to add points to list

```

class ManualRegistration : public QMainWindow
{
protected:
    pcl::PointCloud<pcl::PointXYZ>    src_pc_;
    pcl::PointCloud<pcl::PointXYZ>    dst_pc_;
    Eigen::Matrix4f                   transform_;

```

## PCL - SVD class

- In apps/src/manual\_registration/manual\_registration.cpp
- Create a class to estimate SVD of two point sets

- See paper [Arun et. al. 1987](#)
- Its an example of the orthogonal procrustes problem

```
pcl::registration::TransformationEstimationSVD<pcl::PointXYZ, pcl::PointXYZ> tfe;
tfe.estimateRigidTransformation(src_pc_, dst_pc_, transform_);
```

## PCL - Correlation

After subtracting each point from the mean point we have in `registration/include/pcl/registration/impl/`

```
template <typename PointSource, typename PointTarget, typename Scalar> void
pcl::registration::TransformationEstimationSVD<PointSource, PointTarget, Scalar>::getTransf
    const Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic> &cloud_src_demean,
    const Eigen::Matrix<Scalar, 4, 1> &centroid_src,
    const Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic> &cloud_tgt_demean,
    const Eigen::Matrix<Scalar, 4, 1> &centroid_tgt,
    Matrix4 &transformation_matrix) const
{
    transformation_matrix.setIdentity ();

    // Assemble the correlation matrix H = source * target'
    Eigen::Matrix<Scalar, 3, 3> H = (cloud_src_demean * cloud_tgt_demean.transpose ()).topLeft

    // Compute the Singular Value Decomposition
    Eigen::JacobiSVD<Eigen::Matrix<Scalar, 3, 3> > svd (H, Eigen::ComputeFullU | Eigen::Comput
    Eigen::Matrix<Scalar, 3, 3> u = svd.matrixU ();
    Eigen::Matrix<Scalar, 3, 3> v = svd.matrixV ();

    // Compute R = V * U'
    if (u.determinant () * v.determinant () < 0)
    {
        for (int x = 0; x < 3; ++x)
            v (x, 2) *= -1;
    }

    Eigen::Matrix<Scalar, 3, 3> R = v * u.transpose ();

    // Return the correct transformation
    transformation_matrix.topLeftCorner (3, 3) = R;
    const Eigen::Matrix<Scalar, 3, 1> Rc (R * centroid_src.head (3));
    transformation_matrix.block (0, 3, 3, 1) = centroid_tgt.head (3) - Rc;
}
```

## PCL - Summary

- Use of Eigen implements [Arun et. al. 1987](#) in one main function.
- Relatively close matching of algorithm to code.
- Template notation a bit awkward, but now, not so insurmountable.
- So, using library, we gain power and benefit of very experienced library programmers.

## Eigen Summary

- Header only
- Use CMake to set the include path
- Templated, so its compiled in, no link or run-time dependencies
- Simple to use linear algebra library
- Advise not to mix with GUI code
- Consider static linking as using templates anyway - ease of distribution

## Using Boost

### Introduction

- [Boost](#) is “...one of the most highly regarded and expertly designed C++ library projects in the world.”
- A [large \(121+\)](#) collection of C++ libraries
- Aim to establish standards, contribute to C++11, C++17 etc.
- Hard to use C++ without bumping into Boost at some point
- It's heavily templated
- Many libraries header only, some require compiling.

### Libraries included

- Log, FileSystem, Asio, Serialization, Pool (memory) ...
- Regexp, String Algo, DateTime, ...
- Math, Odeint, Graph, Polygon, Rational, ...
- Each has good documentation, and tutorial, and unit tests, and is widely compiled.

### Getting started

- Default build system: `bjam`

- Also CMake version of boost project, possible deprecated.
- Once installed, many header only libraries, so similar to Eigen.

## Installing pre-compiled

- Linux:
  - `sudo apt-get install boost`
  - `sudo apt-get install libboost1.53-dev`
- Mac
  - Homebrew (brew) or Macports (port)
- Windows
  - Precompiled binaries? Probably you need to build from source.

## Compiling from source

- [Follow build instructions here](#)
- Or use bigger project with it as part of build system
  - NifTK, MITK, Slicer, GImias (medical imaging)

## C++ Principles

(i.e. why introduce Boost on this course)

- Boost uses
  - Templates
  - Widespread use of:
    - \* Generic Programming
    - \* Template Meta-Programming
  - Functors

## C Function Pointers - 1

- Useful if using [Numerical Recipes in C](#)
- See [Wikipedia](#) article and tutorials online

This:

```

#include <stdio.h> /* for printf */
double cm_to_inches(double cm) {
    return cm / 2.54;
}
int main(void) {
    double (*func1)(double) = cm_to_inches;
    printf("Converting %f cm to %f inches by calling function.\n", 5.0, cm_to_inches(5.0));
    printf("Converting %f cm to %f inches by deref pointer.\n", 15.0, func1(15.0));
    return 0;
}

```

Produces:

```

Converting 5.000000 cm to 1.968504 inches by calling function.
Converting 15.000000 cm to 5.905512 inches by deref pointer.

```

## C Function Pointers - 2

- Function pointers can be passed to functions

This:

```

#include <stdio.h>
#include <math.h>
double integrate(double (*funcp)(double), double lo, double hi) {
    double sum = 0.0;
    for (int i = 0; i <= 100; i++)
    {
        sum += (*funcp)(i / 100.0 * (hi - lo) + lo);
    }
    return sum / 100.0;
}
int main(void) {
    double (*fp)(double) = sin;
    printf("sum(sin): %f\n", integrate(fp, 0.0, 1.0));
    return 0;
}

```

Produces:

```

sum(sin): 0.463901

```



## C Function Pointers - 3

- Function pointers
  - often used for callbacks, cost functions in optimisation etc.
  - called by name, or dereference pointer
  - are generally stateless

## C++ Function Objects - 1

- We can define an object to represent a function
  - Called [Function Object](#) or Functor

This:

```
#include <vector>
#include <algorithm>
#include <iostream>
struct IntComparator
{
    bool operator()(const int &a, const int &b) const
    {
        return a < b;
    }
};
/*
template <class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
*/
int main()
{
    std::vector<int> items;
    items.push_back(1);
    items.push_back(3);
    items.push_back(2);
    std::sort(items.begin(), items.end(), IntComparator());
    std::cout << items[0] << "," << items[1] << "," << items[2] << std::endl;
    return 0;
}
```

Produces:

1,2,3

## C++ Function Objects - 2

- But function objects can
  - Have state
  - Have member variables
  - Be complex objects, created by any means
  - e.g. Cost function, similarity between two images

## CMake for Boost

- If installed correctly, should be something like:

```
set(Boost_ADDITIONAL_VERSIONS 1.53.0 1.54.0 1.53 1.54)
find_package(Boost 1.53.0)
if(Boost_FOUND)
    include_directories(${Boost_INCLUDE_DIRS})
endif()
```

## Boost Example

- With 121+ libraries, can't give tutorial on each one!
- Pick one small numerical example
- Illustrate the use of functors in ODE integration

## Using Boost odeint

- Its a numerical example, as we are doing scientific computing!
- As with many libraries, just include right header

```
#include <boost/numeric/odeint.hpp> // Include ODE solver library
                                   // just to check our build system found it
```

- See [this tutorial](#)

## Boost odeint - 1

Given these global definitions:

```
const double gam = 0.15;
typedef std::vector< double > state_type;

// This is functor class
```

## Boost odeint - 2

First define a functor for the function to integrate:

```
class harm_osc {
    double m_gam; // class can have member variables, state etc.
public:
    harm_osc( double gam ) : m_gam(gam) { }

    // odeint integrators normally call f(x, dxdt, t)
    void operator()( const state_type &x , state_type &dxdt , const double /* t */ )
    {
        dxdt[0] = x[1];
        dxdt[1] = -x[0] - m_gam*x[1];
    }
};

// This is observer to record output, and is also a functor class
```

## Boost odeint - 3

Define an observer to collect graph-points:

```
struct push_back_state_and_time
{
    std::vector< state_type >& m_states;
    std::vector< double >& m_times;

    push_back_state_and_time( std::vector< state_type > &states , std::vector< double > &times
    : m_states( states ) , m_times( times ) { }

    void operator()( const state_type &x , double t )
    {
        m_states.push_back( x );
        m_times.push_back( t );
    }
};
```

## Boost odeint - 4

The run it:

```

int main(void) {

    state_type x(2);
    x[0] = 1.0; // start at x=1.0, p=0.0
    x[1] = 0.0;

    std::vector<state_type> x_vec; // vector of vectors
    std::vector<double> times; // stores each time point

    harm_osc harmonic_oscillator(0.15);
    size_t steps = boost::numeric::odeint::integrate(
        harmonic_oscillator ,
        x , 0.0 , 10.0 , 0.1 ,
        push_back_state_and_time( x_vec , times ) );

    for( size_t i=0; i<=steps; i++ )
    {
        std::cout << times[i] << '\t' << x_vec[i][0] << '\t' << x_vec[i][1] << '\n';
    }

}

```

## Boost odeint - 4

Produces:

```

0    1    0
0.1 0.995029    -0.0990884
0.342911    0.942763    -0.32773
0.59639 0.832373    -0.537274
0.865439    0.662854    -0.714058
1.15445 0.436595    -0.839871
1.44346 0.184494    -0.892171
1.70128 -0.0446646    -0.875731
1.9591  -0.262234    -0.80313
2.21692 -0.454545    -0.681207
2.48815 -0.616993    -0.510476
2.77613 -0.733844    -0.296963
3.06411 -0.7866    -0.0685387
3.35209 -0.773719    0.155753
3.64008 -0.699015    0.358007
3.92806 -0.571123    0.522852
4.21604 -0.402601    0.638582
4.50402 -0.20875    0.697943

```

```

4.792    -0.0062679  0.698535
5.07998  0.188159   0.642795
5.36797  0.359199   0.537596
5.65595  0.494053   0.39351
5.94393  0.583381   0.223796
6.23191  0.621907   0.0432166
6.51989  0.608673   -0.133212
6.80787  0.546935   -0.291443
7.09586  0.44372   -0.419492
7.39491  0.303457   -0.510857
7.69397  0.143005   -0.553869
7.99302  -0.0228226 -0.546913
8.29208  -0.179394  -0.492786
8.59113  -0.313525  -0.398255
8.89019  -0.414555  -0.273299
9.18925  -0.475166  -0.130101
9.4883   -0.49187   0.0181101
9.78736  -0.465143  0.158247
10  -0.421907   0.246407

```

## Why Boost for Numerics

- Broader question is
  - Why someone else's library? Boost or some other.
- Advanced use of Template Meta Programming, Traits
  - Performance optimisations
  - Alternative implementations
    - \* CUDA via Thrust
    - \* MPI
    - \* etc
- You just focus on your bit

## Using ITK

### Introduction

- [Insight Segmentation and Registration Toolkit](#)
- Insight Journal for library additions
- Large community in medical image processing
- Deliberately no visualisation, see [VTK](#).

## C++ Principles

- Heavy use of Generic Programming
- Use of Template Meta-Programming
- Often perceived by “scientific programmers” (Matlab) as difficult
- Aim: demonstrate here, that we can now use it!
- Of particular interest
  - typedefs - make life easier
  - SmartPointers - reduce leaking memory
  - Iterators - fast image access
  - Object Factories - extensibility

## Architecture Concept

- Use of pipeline of filters
- Simple to plug image processing filters together
- Sometimes difficult to manage memory for huge images

## Filter Usage - 1

We work through a simple filter program. First, typedefs are aliases.

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

    const unsigned int Dimension = 2;
    typedef int PixelType;
    typedef itk::Image<PixelType, Dimension> ImageType;
    typedef itk::AddImageFilter<ImageType, ImageType> AddFilterType;
    typedef itk::ImageFileReader<ImageType> ImageReaderType;
    typedef itk::ImageFileWriter<ImageType> ImageWriterType;
```

## Filter Usage - 2

Objects are constructed:

```
ImageReaderType::Pointer reader1 = ImageReaderType::New();
ImageReaderType::Pointer reader2 = ImageReaderType::New();
AddFilterType::Pointer addFilter = AddFilterType::New();
```

```
ImageWriterType::Pointer writer = ImageWriterType::New();

// eg. if not using typedefs
//itk::ImageFileWriter< itk::Image<int, 2> >::Pointer writer
// = itk::ImageFileWriter< itk::Image<int, 2> >::New();
```

## Filter Usage - 3

Pipeline is executed:

```
reader1->SetFileName("inputFileName1.nii");
reader2->SetFileName("inputFileName2.nii");
addFilter->SetInput(0, reader1->GetOutput());
addFilter->SetInput(1, reader2->GetOutput());
writer->SetInput(addFilter->GetOutput());
writer->SetFileName("outputFileName1.nii");
//writer->Update(); // commented out, as filenames are fake.
// and build system for lecture notes
// tries to run the program.

return 0;
}
```

More information on ITK Pipeline can be found in the [ITK Software Guide](#).

## Smart Pointer Intro

Lets look at some interesting features.

- Smart Pointer
  - Class, like a pointer, but ‘Smarter’ (clever)
  - Typically, once allocated will automatically destroy the pointed to object
  - Implementations vary, STL, ITK, VTK, Qt, so read the docs
- So, in each class e.g. itkAddImageFilter

```
typedef AddImageFilter      Self
typedef SmartPointer<Self> Pointer
```

and so, its used like

```
ClassName::Pointer variableName = ClassName::New();
```

## Smart Pointer Class

In the SmartPointer itself

```
/** Constructor to pointer p */
SmartPointer (ObjectType *p):
    m_Pointer(p)
{ this->Register(); }

/** Destructor */
~SmartPointer ()
{
    this->UnRegister();
    m_Pointer = ITK_SP_NULLPTR;
}
```

and

```
private:
/** The pointer to the object referred to by this smart pointer. */
ObjectType *m_Pointer;

void Register()
{
    if ( m_Pointer ) { m_Pointer->Register(); }
}
```

## General Smart Pointer Usage

- Avoid use of explicit pairs of `new/delete`
- Immediately assign object to SmartPointer
- Consistently (i.e. always) use SmartPointer
  - Pass (reference to) SmartPointer to function.
  - Can (but should you?) return SmartPointer from function.
  - Don't use raw pointer, and don't store raw pointers to objects.
  - You can't test raw pointer to check if object still exists.
- Object is deleted when last SmartPointer reference goes out of scope

## ITK SmartPointer

- ITK keeps reference count in `itk::LightObject` base class
- So, it can only be used by sub-classes of `itk::LightObject`



- Reference is held in the object
- Same method used in MITK, as MITK uses ITK concepts
- [VTK](#) has a SmartPointer that requires calling Delete explicitly (!!)
- STL has much clearer definition of different types of smart pointer
- Read [THIS](#) tutorial

## Implementing a Filter

- ITK provides many image processing filters.
- But you can write your own easily
  - Single Threaded - override GenerateData()
  - Multi-Threaded - override ThreadedGenerateData()
- Now we see an example - thresholding, as we want to study the C++ not the image processing.

### Filter Impl - 1

Basic filter:

```
namespace itk
{
template< class TInputImage, class TOutputImage = TInputImage>
class MyThresholdFilter:public ImageToImageFilter< TInputImage, TOutputImage >
{
public:
```

### Filter Impl - 2

Boilerplate nested typedefs :

```
/** Standard class typedefs. */
typedef MyThresholdFilter                               Self;
typedef ImageToImageFilter< TInputImage, TOutputImage > Superclass;
typedef SmartPointer< Self >                             Pointer;
typedef typename TInputImage::PixelType                  InputPixelType;
typedef typename TOutputImage::PixelType                 OutputPixelType;

/** Method for creation through the object factory. */
itkNewMacro(Self);
```

```

/** Run-time type information (and related methods). */
itkTypeMacro(ImageFilter, ImageToImageFilter);

```

## Filter Impl - 3

Look at ITK Macros :

```

itkSetMacro(Low, InputPixelType);
itkGetMacro(Low, InputPixelType);
itkSetMacro(High, InputPixelType);
itkGetMacro(High, InputPixelType);

```

```

protected:
    MyThresholdFilter(){}
    ~MyThresholdFilter(){}

```

## Filter Impl - 4

The main method :

```

/** Does the real work. */
virtual void GenerateData()
{
    TInputImage *inputImage = static_cast< TInputImage * >(&this->ProcessObject::GetInputImage());
    TOutputImage *outputImage = static_cast< TOutputImage * >(&this->ProcessObject::GetOutputImage());

    ImageRegionConstIterator<TInputImage> inputIterator = ImageRegionConstIterator<TInputImage>(inputImage);
    ImageRegionIterator<TOutputImage> outputIterator = ImageRegionIterator<TOutputImage>(outputImage);

    for (inputIterator.GoToBegin(),
         outputIterator.GoToBegin();
         !inputIterator.IsAtEnd() && !outputIterator.IsAtEnd();
         ++inputIterator,
         ++outputIterator)
    {
        if (*inputIterator >= m_Low && *inputIterator <= m_High)
        {
            *outputIterator = 1;
        }
        else

```

```

        {
            *outputIterator = 0;
        }
    }
}

private:
    MyThresholdFilter(const Self &); //purposely not implemented
    void operator=(const Self &); //purposely not implemented
    InputPixelType m_Low;
    InputPixelType m_High;
};
} // end namespace

int main(int argc, char** argv)
{
    // Not providing a real example,
    // as I dont know how to read/write images within the
    // dxy framework.
    return 0;
}

```

## Iterators

- ITK provides many iterators
- Generic Programming means:
  - Suitable for n-dimensions
  - Suitable for all types of data
- Also, different image access concepts
  - Region of Interest
  - Random subsampling
  - No change in code
- So iterators enable you to traverse image and encapsulate the traversal mechanism in an iterator
- Similar concept to STL `.begin()`, `.end()`
- See [ITK Software Guide](#)

## Private Constructors?

If you look at an ITK filter, you may notice for example

```
protected:
    AddImageFilter() {}
    virtual ~AddImageFilter() {}

private:
    AddImageFilter(const Self &);
    void operator=(const Self &);
```

- Copy constructor and copy assignment are private and not implemented
- Constructor and Destructor private. So how do you use?

## Static New Method

You will then see

```
/** Method for creation through the object factory. */
itkNewMacro(Self);
```

which if you hunt for long enough, you find this snippet

```
#define itkSimpleNewMacro(x)
static Pointer New(void)
{
    Pointer smartPtr = ::itk::ObjectFactory< x >::Create();
    if ( smartPtr.GetPointer() == ITK_NULLPTR )
    {
        smartPtr = new x;
    }
    return smartPtr;
}
```

So, either this ObjectFactory creates it, or a standard new call.

## ObjectFactory::Create

In `itk::ObjectFactory` we ask factory to `CreateInstance` using a `char*`

```
static typename T::Pointer Create()
{
    LightObject::Pointer ret = CreateInstance( typeid( T ).name() );
    return dynamic_cast< T * >( ret.GetPointer() );
}
```

`CreateInstance` works with either a base class name, or a class name to return either a specific class, or a family of classes derived from a common base class.

## Why Object Factories?

- Rather than create objects directly
- Ask a class (ObjectFactory) to do it
- This class contain complex logic, not just a new operator
- So, we can
  - dynamically load libraries from ITK\_AUTOLOAD\_PATH at runtime
  - Have a list/map of current classes, and provide overrides
  - i.e swap in a GPU version instead of CPU
- More dynamic variant of FactoryMethod, AbstractFactory (See [GoF](#))

## File IO Example

In itkImageFileReader.hxx

```
std::list< LightObject::Pointer > allobjects =  
    ObjectFactoryBase::CreateAllInstance("itkImageIOBase");
```

- We ask the factory for every class that is a sub-class of itkImageIOBase.
- Then we can ask each ImageIOBase sub-class if it can read a specific format.
- First one to reply true reads the image.
- In general case, ask ObjectFactoryBase for any class.

## Object Factory List of Factories

In class ObjectFactoryBase

```
class ObjectFactoryBase:public Object  
{  
public:  
    static std::list< ObjectFactoryBase * > GetRegisteredFactories();
```

This class maintains a static vector of ObjectFactoryBase. These are added programmatically, via static initialisation or dynamically via the ITK\_AUTO\_LOAD\_PATH.

## PNG IO Factory

- Given `itkPNGImageIO.h/cxx` can read PNG images
- We see in `itkPNGImageIOFactory.cxx`

```
PNGImageIOFactory::PNGImageIOFactory()
{
    this->RegisterOverride( "itkImageIOBase",
                           "itkPNGImageIO",
                           "PNG Image IO",
                           1,
                           CreateObjectFunction< PNGImageIO >::New() );
}
```

So, PNG factory says it implements a type of `itkImageIOBase`, will return an `itkPNGImageIO`, and instantiates a function object that calls the right constructor.

## ObjectFactory Summary

- ObjectFactory defines a static vector of ObjectFactory
- ObjectFactory objects loaded:
  - Directly named in code at compile time
  - Via static initialisers when a dynamic library is loaded
  - Or from `ITK_AUTOLOAD_PATH`
- ObjectFactory returns one/all classes that implement a given class
- Static New method now asks factory for a class.
- So, you can override any ITK class.
- Why is above example not an infinite loop?

## ITK Summary

- Pipeline architecture for most filters
- Also includes a registration framework (see [ITK Software Guide](#))
- Smart Pointers - reference counting, automatic deletion
- Static New method with ObjectFactory to enable overriding any class at runtime
- Dynamic loading via `ITK_AUTOLOAD_PATH`
- Pipeline architecture - easy to prototype, once you know C++
- Write your own filter, unit test, generalise to n-dimension, of n-vectors.
- Easy to extend to multi-threading

## Summary

### Learning Objectives

- Use well written libraries
- Understand enough C++ for some common C++ libraries

### Further Reading

Please refer to the following:

- [Short Eigen Tutorial](#)
- [Longer Eigen Tutorial](#)
- [Boost Tutorials for each library](#)
- [ITK Software Guide](#)
- [Simple ITK](#): Template free, wrapper, python, [paper](#)

## Chapter 11

# Appendix - STL

## Appendix - STL

### Standard Template Library

This lecture got compressed into lecture 3.

- Notes provided [here](#) in their entirety, for historical reasons.

## The Standard Template Library

### Contents

[STL reference](#)

- Containers
  - Sequences
  - Associative
- Iterators
- I/O streams
- Functors/functions
- Algorithms



## Motivation

### Motivation - why STL?

Because you have better things to do than rediscovering the wheel.

Task: Read in two files with an unknown number of integers, and sort them altogether.

### The WRONG way to do it!

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return ( *(int*)a - *(int*)b );
}

int main()
{
    FILE* if1 = fopen("90stl/cpp/randomNumbers1.txt","r");
    FILE* if2 = fopen("90stl/cpp/randomNumbers2.txt","r");

    // First determine size of array needed
    int size1=0, size2=0;
    while (!feof(if1)) {
        fscanf(if1, "%d", &size1);
        if (!feof(if1)) size1++;
    }
    rewind(if1);
    while (!feof(if2)) {
        fscanf(if2, "%d", &size2);
        if (!feof(if2)) size2++;
    }
    rewind(if2);

    // Read in the data
    int theArray[size1+size2];
    for (int i=0;i<size1;++i) {
        fscanf(if1, "%d", &theArray[i]);
    }
    for (int i=size1;i<size1+size2;++i) {
        fscanf(if2, "%d", &theArray[i]);
    }
}
```

```

fclose(if1);
fclose(if2);

// Sort and output
qsort(theArray,size1+size2,sizeof(int),compare);
for (int i=0;i<size1+size2;++i) {
    printf("%d ", theArray[i]);
}
}

```

## STL solution

```

#include <vector>
#include <algorithm>
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream if1("90stl/cpp/randomNumbers1.txt",std::ifstream::in);
    std::ifstream if2("90stl/cpp/randomNumbers2.txt",std::ifstream::in);

    // Read in the data.
    int number;
    std::vector<int> theArray;
    while (!if1.eof()) {
        if1 >> number;
        if (!if1.eof()) theArray.push_back(number);
    }
    while (!if2.eof()) {
        if2 >> number;
        if (!if2.eof()) theArray.push_back(number);
    }
    if1.close();
    if2.close();

    // Sort and output
    std::sort(theArray.begin(),theArray.end());
    for (int i=0;i<theArray.size();++i) {
        std::cout << theArray[i] << " ";
    }
}

```

## More conveniences

- Similar API between different STL containers. eg. `myContainer->begin()`, `end()`, `at()`, `erase()`, `clear()`, `size()`....
- Algorithms and data structures optimised for speed and memory.

## STL Containers

### Two general types:

- *Sequences*: Elements are ordered in a linear sequence. Individual elements are accessed by their position in this sequence/index. eg. `myVector[3]`, `myList.front()`, etc.
- *Associative*: Elements are referenced by their key and not by their position in the container. eg. `myMap['key1']`

## Sequences

Properties:

- size: fixed/dynamic
- access: random/sequential
- underlying memory structure: contiguous/not
  - random access in non-contiguous memory is tricky
  - affects how efficiently inserting/removing of elements can be done
  - affects if pointer arithmetic can be done
- optimised insert/remove operations

## Sequences

| Name          | size           | access                   | memory            | efficient insert/remove |
|---------------|----------------|--------------------------|-------------------|-------------------------|
| array         | fixed          | random                   | contiguous        | -                       |
| <b>vector</b> | <b>dynamic</b> | <b>random</b>            | <b>contiguous</b> | <b>at end only</b>      |
| deque         | dynamic        | random                   | non-contiguous    | both ends               |
| list          | dynamic        | sequential               | non-contiguous    | anywhere                |
| forward_list  | dynamic        | sequential, only forward | non-contiguous    | anywhere                |

## More on vector

It's dynamic, so you can add/erase elements:

```
std::vector<int> myVec;
for (int i=0;i<10;++i) {
    myVec.push_back(i);
}
myVec.insert(myVec.begin(),-1);
myVec.erase(1);
```

myVec = -1 1 2 3 4 5 6 7 8 9

... or manipulate its size:

```
std::vector<int> myVec(10);
std::cout << "Vector size before = " << myVec.size();
myVec.resize(5);
std::cout << " after = " << myVec.size() << "\n";
```

Vector size before = 10 after = 5

## Note on C++11

- For all containers, `emplace(const_iterator position, Args&&... args)` is preferable to `insert(const_iterator position, const value_type& val)`, as it doesn't create any copies of the object you add to the container.
- Cases you might prefer something other than `emplace`:
  - backward compatibility
  - `insert` has more constructors
  - at least `emplace_back` might not work as expected in some implementations

## Exercise

Think of cases where you'd use a specific container

## Associative containers

Properties:

- key: is it separate from value?
  - maps: key-value pair
  - sets: value is the key (and thus it's const!)
- ordering. Affects performance:
  - unordered containers fastest to fill and access by key
  - ordered containers fastest to iterate through, and they're already ordered :o)
- unique values?

## Associative containers

|                   | ordered  | unordered          |
|-------------------|----------|--------------------|
| <b>unique</b>     | map      | unordered_map      |
| <b>non-unique</b> | multimap | unordered_multimap |

|                   | ordered  | unordered          |
|-------------------|----------|--------------------|
| <b>unique</b>     | set      | unordered_set      |
| <b>non-unique</b> | multiset | unordered_multiset |

## More on maps

Fill them with

```
std::map<std::string,int> myMap;
myMap["broccoli"] = 2;
myMap["garlic"] = 1;
myMap["broccoli"] = 1; // returns reference to element => element is updated
myMap.insert( std::pair<std::string,int>("bread",4) );
myMap.insert( std::pair<std::string,int>("broccoli",3) ); // returns iterator
// to existing element => element is not updated

typedef std::multimap<std::string,int> MMapType;
MMapType myMMap; // there's no [] for multimap
myMMap.insert( std::pair<std::string,int>("broccoli",2) );
myMMap.insert( std::pair<std::string,int>("bread",4) );
myMMap.insert( std::pair<std::string,int>("broccoli",3) );
```

## More on maps

Access elements with

```
std::cout << "myMap[broccoli] = " << myMap["broccoli"] << "\n";
std::cout << "myMap.find(bread) = " << myMap.find("bread")->second << "\n";
for (std::map<std::string,int>::iterator it=myMap.begin(); it!=myMap.end(); ++it)
    std::cout << it->first << " " << it->second << "\n";

std::cout << "myMMap.count(broccoli) = " << myMMap.count("broccoli") << "\n";
std::pair < MMapType::iterator,MMapType::iterator> range = myMMap.equal_range("broccoli");
for (MMapType::iterator it=range.first; it!=range.second; ++it)
    std::cout << it->first << " " << it->second << "\n";
for (MMapType::iterator it=myMMap.begin(); it!=myMMap.end(); ++it)
    std::cout << it->first << " " << it->second << "\n";

myMap[broccoli] = 1
myMap.find(bread) = 4
bread 4
broccoli 1
garlic 1
myMMap.count(broccoli) = 2
broccoli 2
broccoli 3
bread 4
broccoli 2
broccoli 3
```

## Example

Read in file with unknown number of particle-momentum pairs.

```
proton 4.5
electron 17.8
pion 4.6
muon 12.0
pion 3.2
neutrino 8.2
pion 23.7
proton 9.4
neutrino 6.7
```

- Then print out

1. list of all particle-momentum pairs in alphabetical order
  - how about in ascending momentum order?
2. list of types of particles in the file in alphabetical order
  - how about in mass order?
3. list of particle-max momentum pairs

## Task 1

```
#include <map>
#include <string>
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream ifs("90stl/cpp/particleList.txt",std::ifstream::in);
    // Read in the data
    std::multimap<std::string,double> theParticles;
    std::string name;
    double momentum;
    while (!ifs.eof()) {
        ifs >> name >> momentum;
        if (!ifs.eof())
            theParticles.insert( std::pair<std::string,double>(name,momentum) );
    }
    ifs.close();
    // Output - it's already sorted!
    std::multimap<std::string,double>::iterator iter = theParticles.begin();
    for ( ; iter!=theParticles.end(); ++iter) {
        std::cout << iter->first << " " << iter->second << std::endl;
    }
}
```

## Task 2

- Hint1: use a set
- Hint2: write a custom comparator

## Task 2

- Hint1: use a set
- Hint2: write a custom comparator

```

class compMass {
public:
    compMass() :
        m_particlesOrdered({"neutrino", "electron", "muon", "pion", "kaon", "proton"})
    {};
    bool operator() (const std::string& s1, const std::string& s2) const
    {
        int index1=-1, index2=-1;
        for (int i=0; i<m_particlesOrdered.size(); ++i) {
            if (m_particlesOrdered[i]==s1) index1 = i;
            if (m_particlesOrdered[i]==s2) index2 = i;
        }
        return index1<index2; // unknown particles appear first (index=-1)
    }
private:
    std::vector<std::string> m_particlesOrdered;
};

```

then use it in the set constructor:

```
std::set<std::string, compMass> theParticles;
```

### Task 3

Hint: extend the map class. - How?

- inherit from `std::map` - **NO!**
  - STL containers are *not* designed to be polymorphic, as they don't have virtual destructors (*Meyers 1:7*)
- composition - write a class that contains either an `std::map` object or smart pointer to it.
- free functions with STL containers/iterators as arguments

### Task 3 - with function

```

void keepMax(std::map<std::string, double>& theMap, const std::string key, const double value)
{
    if (theMap.find(key)==theMap.end()) //element doesn't already exist in map
        theMap[key] = value;
    else
        // any logic can go in here, eg a counter, an average, etc...
}

```



```

        theMap[key] = std::max(theMap[key],value);
    }

```

Then use it in the loop when filling the container:

```

if (!feof(ifs)) keepMax( theParticles,name,momentum );

```

### Task 3 - with composition

```

class maxMap
{
public:
    typedef std::map<std::string,double>::iterator iterator;

    void insert(const std::pair<std::string,double> theElement)
    {
        std::string key = theElement.first;
        double value = theElement.second;
        if (m_map.find(key)==m_map.end()) //element doesn't already exist in map
            m_map[key] = value;
        else
            // any logic can go in here, eg a counter, an average, etc...
            m_map[key] = std::max(m_map[key],value);
    }
    iterator begin() { return m_map.begin(); }
    iterator end() { return m_map.end(); }
private:
    std::map<std::string,double> m_map;
};

```

### Task 3 - with composition

Then use it in main:

```

int main()
{
    std::ifstream ifs("90stl/cpp/particleList.txt",std::ifstream::in);
    // Read in the data
    maxMap theParticles;
    std::string name;
    double momentum;
    while (!ifs.eof()) {

```

```

    ifs >> name >> momentum;
    if (!ifs.eof())
        theParticles.insert( std::pair<std::string,double>(name,momentum) );
}
ifs.close();
// Output - it's already sorted!
maxMap::iterator iter = theParticles.begin();
for ( ; iter!=theParticles.end(); ++iter) {
    std::cout << iter->first << " " << iter->second << std::endl;
}
}

```

## Accessing containers: iterators

For random-access containers you can do

```

std::vector<int> myVector = {1,2,3,4};
for (int i=0; i<myVector.size(); ++i) {
    std::cout << myVector[i] << std::endl;
}

```

But for sequential-access ones, you can only do

```

#include <iterator>

std::list<int> mylist = {1,2,3,4};
std::list<int>::iterator it=mylist.begin()
for ( ; it!=mylist.end(); ++it) {
    std::cout << *it << std::endl;
}

```

## Iterators

An iterator is any object that, pointing to some element in a container, has the ability to iterate through the elements of that range using a set of operators.

- Minimum operators needed: increment (++) and dereference (\*).
- A pointer is the simplest iterator.
- Brings some container-independence.
  - especially when using typedef

<http://www.cplusplus.com/reference/iterator/>

## Pairs

```
#include <utility>           // std::pair

int main () {
    std::pair<std::string,double> product1;           // default constructor
    std::pair<std::string,double> product2("tomatoes",2.30); // value init

    auto foo = std::make_pair (10,20);               // created a std::pair<int,int>
    product1 = std::make_pair(std::string("lightbulbs"),0.99); // using make_pair

    product2.first = "shoes";                         // the type of first is string
    product2.second = 39.90;                          // the type of second is double

    std::cout << "foo: " << foo.first << ", " << foo.second << '\n';

    return 0;
}
```

## Tuples

```
#include <tuple>             // std::tuple, std::make_tuple, std::get

int main()
{
    std::tuple<int,char> one;           // default
    std::tuple<int,char> two(10,'a');  // initialization

    auto first = std::make_tuple (10,'a');           // tuple < int, char >
    const int a = 0; int b[3];                     // decayed types:
    auto second = std::make_tuple (a,b);             // tuple < int, int* >

    std::cout << "two contains: " << std::get<0>(two);
    std::cout << " and " << std::get<1>(two);
    std::cout << std::endl;

    return 0;
}
```

## Assignment

- Download a human genome file from [ftp://ftp.ensembl.org/pub/release-87/fasta/homo\\_sapiens/dna/](ftp://ftp.ensembl.org/pub/release-87/fasta/homo_sapiens/dna/) . This is a sequence of characters from the dictionary {A,C,G,T,N}.
- List all the possible 3-letter combinations (k-mers for k=3) that appear in this file together with the number of appearances of each in the file, in order of number of appearances.
  - Output should be something like

```
NNN 3345028
AGT 2348
CTT 1578
...
```

- Hint: use only associative containers. You'll need to extend their functionality to some kind of counter.

## Other STL stuff

### Contents

- I/O with streams
- Function objects
- Algorithms

### Streams

- `iostream` for `std::cout`, `std::cerr` and `std::cin`
- `fstream` for file I/O
- `sstream` for string manipulation

```
std::ifstream myfile(filename, std::ifstream::in);
if (!myfile.good()) {
    stringstream mess;
    mess << "Cannot open file " << filename << " . It probably doesn't exist." << endl;
    throw runtime_error(mess.str());
}

myfile.seekg(0, std::ios::end);
m_size = myfile.tellg();
```

```

m_buffer = new char[m_size];
myfile.seekg(0, std::ios::beg);
myfile.read(m_buffer,m_size);
myfile.close();

```

## Function objects

- Remember `std::set<std::string,compMass> theParticles; ?`
- There are several ways to define function-type stuff in c++, and `std::function` wraps them all
  - functions and function pointers
  - functors (i.e. an object with an `operator()` member function)
  - lambdas (i.e. nameless inline functions)
- Such functions can be used as comparators in STL containers or algorithms, among other things.

## Function objects

```

#include <functional>    // std::function, std::negate

// a function:
int half(int x) {return x/2;}

// a function object class:
struct third_t {
    int operator()(int x) {return x/3;}
};

// a class with data members:
struct MyValue {
    int value;
    int fifth() {return value/5;}
};

int main () {
    std::function<int(int)> fn1 = half;           // function
    std::function<int(int)> fn2 = &half;          // function pointer
    std::function<int(int)> fn3 = third_t();      // function object
    std::function<int(int)> fn4 = [](int x){return x/4;}; // lambda expression
    std::function<int(int)> fn5 = std::negate<int>(); // standard function object

    std::cout << "fn1(60): " << fn1(60) << '\n';
    std::cout << "fn2(60): " << fn2(60) << '\n';

```

```

std::cout << "fn3(60): " << fn3(60) << '\n';
std::cout << "fn4(60): " << fn4(60) << '\n';
std::cout << "fn5(60): " << fn5(60) << '\n';

// stuff with members:
std::function<int(MyValue&)> value = &MyValue::value; // pointer to data member
std::function<int(MyValue&)> fifth = &MyValue::fifth; // pointer to member function

MyValue sixty {60};
std::cout << "value(sixty): " << value(sixty) << '\n';
std::cout << "fifth(sixty): " << fifth(sixty) << '\n';
}

```

## Function objects

Output:

```

fn1(60): 30
fn2(60): 30
fn3(60): 20
fn4(60): 15
fn5(60): -60
value(sixty): 60
fifth(sixty): 12

```

## Algorithms

<http://www.cplusplus.com/reference/algorithm/>

- A collection of functions especially designed to be used on ranges of elements.
- Don't start coding any task, before checking if it's already there!

```
std::sort(RandomAccessIter first, RandomAccessIter last, Compare comp)
```

```
std::transform(InIter first, InIter last, OutIter res, UnaryOp op)
```

```
std::transform(InIter1 first1, InIter1 last1, InIter2 first2, OutIter res, BinaryOp op)
```

```
std::for_each (InIter first, InIter last, Function fn)
```

```
std::max_element(ForwardIter first, ForwardIter last, Compare comp)
```

```
std::min_element(ForwardIter first, ForwardIter last, Compare comp)
```

```
std::minmax_element(ForwardIter first, ForwardIter last, Compare comp)
```

## Chapter 12

# Appendix - TMP

## Appendix - Template Meta-Programming

### No Longer In Course

- Notes provided [here](#) for historical reasons.
- Also, read “[Modern C++ Design](#)”

## Template Meta-Programming (TMP)

### What Is It?

- See [Wikipedia](#), [Wikibooks](#), [Keith Schwarz](#)
- C++ Template
  - Type or function, parameterised over, set of types, constants or functions
  - Instantiated at compile time
- Meta Programme
  - Program that produces or manipulates constructs of target language
  - Typically, it generates code
- Template Meta-Programme
  - C++ programme, uses Templates, generate C++ code at compile time

## TMP is Turing Complete

- Given: A [Turing Machine](#)
  - Tape, head, states, program, etc.
- A language is “Turing Complete” if it can simulate a Turing Machine
  - e.g. Conditional branching, infinite looping
- Turing’s work underpins much of “what can be computed” on a modern computer
  - C, C++ no templates, C++ with templates, C++ TMP
  - All Turing Complete
- Interesting that compiler can generate such theoretically powerful code.
- But when, where, why, how to use TMP?
- (side-note: Its not just a C++ pre-processor macro)

## Why Use It?

- Use sparingly as code difficult to follow
- Use for
  - Optimisations
  - Represent Behaviour as a Type
  - Traits classes
- But when you see it, you need to understand it!

## Factorial Example

See [Wikipedia Factorial Example](#)

- This:

```
#include <iostream>
using namespace std;

template <int n>
struct factorial {
    enum { value = n * factorial<n - 1>::value };
};
```



```

template <>
struct factorial<0> {
    enum { value = 1 };
};

int main () {
    std::cout << factorial<0>::value << std::endl;
    std::cout << factorial<8>::value << std::endl;
}

```

- Produces:

```

1
40320

```

## Factorial Notes:

- Compiler must know values at compile time
  - i.e. constant literal or constant expression
  - See also [constexpr](#)
- Generates/Instantiates all functions recursively
- Factorial 16 = 2004189184
- Factorial 17 overflows
- This simple example to illustrate “computation”
- But when is TMP actually useful?
- Notice that parameter was an integer value ... not just “int” type

## Loop Example

- This:

```

#include <iostream>
#include <vector>
using namespace std;

template<typename T>
T Sum(const std::vector<T>& data)
{
    T total = 0;
    for (size_t i = 0; i < data.size(); i++)
    {
        total += data[i];
    }
}

```

```

    }
    return total;
}

int main () {
    size_t numberOfInts = 3;
    size_t numberOfLoops = 1000000000;
    vector<int> a(numberOfInts);
    int total = 0;

    std::cout << "Started" << std::endl;
    for (size_t j = 0; j < numberOfLoops; j++)
    {
        for (size_t i = 0; i < numberOfInts; i++)
        {
            total = Sum(a);
        }
    }
    std::cout << "Finished:" << total << std::endl;
}

```

- Time: numberOfInts=3 took 40 seconds

## Loop Unrolled

- This:

```

#include <iostream>
#include <vector>
using namespace std;

template <typename T, int length>
class FixedVector {
    T data[length];
public:
    FixedVector()
    {
        // Initialise
        for (size_t i = 0; i < length; i++)
        {
            data[i] = 0;
        }
    }
    T Sum()

```

```

    {
        T sum = 0;
        for (size_t i = 0; i < length; i++)
        {
            sum += data[i];
        }
        return sum;
    }
};

int main () {
    const size_t numberOfInts = 3;
    const size_t numberOfLoops = 1000000000;
    FixedVector<int, numberOfInts> a;
    int total = 0;

    std::cout << "Started" << std::endl;
    for (size_t j = 0; j < numberOfLoops; j++)
    {
        for (size_t i = 0; i < numberOfInts; i++)
        {
            total = a.Sum();
        }
    }
    std::cout << "Finished:" << total << std::endl;
}

```

- Time: numberOfInts=3 took 32 seconds when switch to fixed vector, and 23 when a raw array.

## Policy Checking

- Templates parameterised by type not by behaviour
- But you can make a class to represent the behaviour
- See [Keith Schwarz](#) for longer example.

## Simple Policy Checking Example

- This:

```

#include <iostream>
#include <vector>
#include <stdexcept>

```

```

class NoRangeCheckingPolicy {
public:
    static void CheckRange(size_t pos, size_t n) { return; } // no checking
};

class ThrowErrorRangeCheckingPolicy {
public:
    static void CheckRange(size_t pos, size_t n)
    {
        if (pos >= n) { throw std::runtime_error("Out of range!"); }
    }
};

template < typename T
          , typename RangeCheckingPolicy = NoRangeCheckingPolicy
          >
class Vector
: public RangeCheckingPolicy
{
private:
    std::vector<T> data;
public:
    // other methods etc.
    const T& operator[] (size_t pos) const
    {
        RangeCheckingPolicy::CheckRange(pos, data.size());
        return data[pos];
    }
};

int main () {
    Vector<int, ThrowErrorRangeCheckingPolicy> a;
    // a.push_back(1); or similar
    // a.push_back(2); or similar
    try {
        std::cout << a[3] << std::endl;
    } catch (const std::runtime_error& e)
    {
        std::cerr << e.what();
    }
    return 0;
}

```

- Produces:

## Summary of Policy Checking Example

- Define interface for behaviour
- Parameterize over all behaviours
- Use multiple-inheritance to import policies
- e.g. logging / asserts

## Traits

- From C++ standard 17.1.18
  - “a class that encapsulates a set of types and functions necessary for template classes and template functions to manipulate objects of types for which they are instantiated.”
- Basically: Traits represent details about a type
- You may be using them already!
- Start with a simple example

## Simple Traits Example

- This:

```
#include <iostream>

template <typename T>
struct is_void {
    static const bool value = false;
};

template <>
struct is_void<void> {
    static const bool value = true;
};

int main () {
    std::cout << "is_void(void)=" << is_void<void>::value << std::endl;
    std::cout << "is_void(int)=" << is_void<int>::value << std::endl;
    return 0;
}
```

- Produces:

```
is_void(void)=1
is_void(int)=0
```

## Traits Principles

- Small, simple, normally public, eg. struct
- else/if
  - Else template
  - partial specialisations
  - full specialisations
- Probably using them already
  - `std::numeric_limits<double>::max()`
  - ITK has similar `itk::NumericTrait<PixelType>`
- Applies to primitives as well as types

## Traits Examples

- [Simple Tutorial from Aaron Ballman](#)
- [Boost meta-programming support](#)
- [Boost type\\_traits tutorial](#)
- [C++11 has many traits](#)

## Wait, Inheritance Vs Traits?

- We said inheritance is often overused in OO
- We say that too frequent if/switch statements based on type are bad in OO
- C++11 providing many [is\\_X type traits](#) returning bool, leading to if/else
- So, when to use it?

## When to use Traits

- Some advice
  - Sparingly
  - To add information to templated types
  - Get algorithm to work for 1 data type
  - If you extend to multiple data types and consider templates
    - \* When you need type specific behaviour

- traits probably better than template specialisation
- traits better than inheritance based template hierarchies
- Remember
  - Scientist = few use-cases
  - Library designer = coding for the unknown, and potentially limitless use-cases
    - \* More likely of interest to library designers

## TMP Use in Medical Imaging - 1

Declare an [ITK](#) image

```
template< typename TPixel, unsigned int VImageDimension = 2 >
class Image:public ImageBase< VImageDimension >
{
public:
// etc
```

- TPixel, int, float etc.
- VImageDimension = number of dimensions

## TMP Use in Medical Imaging - 2

But what type is origin/spacing/dimensions?

```
template< unsigned int VImageDimension = 2 >
class ImageBase:public DataObject
{
    typedef SpacePrecisionType          SpacingValueType;
    typedef Vector< SpacingValueType, VImageDimension > SpacingType;
```

## TMP Use in Medical Imaging - 3

So now look at Vector

```
template< typename T, unsigned int NVectorDimension = 3 >
class Vector:public FixedArray< T, NVectorDimension >
{
public:
```

## TMP Use in Medical Imaging - 4

Now we can see how fixed length arrays are used

```
template< typename T, unsigned int TVectorDimension >
const typename Vector< T, TVectorDimension >::Self &
Vector< T, TVectorDimension >
::operator+=(const Self & vec)
{
    for ( unsigned int i = 0; i < TVectorDimension; i++ )
    {
        ( *this )[i] += vec[i];
    }
    return *this;
}
```

which may be unrolled by compiler.

## TMP Use in Medical Imaging - 5

- [ITK](#)
  - uses [itk::NumericTraits<>](#) adding mathematical operators like multiplicative identity, additive identity
  - uses traits to describe features of meshes, like `numeric_limits`, but more generalised
- [MITK](#) (requires coffee and a quiet room)
  - uses [mitkPixelTypeList.h](#) for multi-plexing across templated image to non-templated image type
  - uses [mitkGetClassHierarchy.h](#) to extract a list of class names in the inheritance hierarchy
- [TMP in B-spline based registration](#):

## Further Reading For Traits

- [Keith Schwarz](#)
- [Nathan Meyers](#)
- [Todd Veldhuizen](#), traits scientific computing
- [Thaddeaeus Frogley](#), ACCU, traits tutorial
- [Aaron Ballman](#)
- [Andrei Alexandrescu](#)
- [Andrei Alexandrescu](#) traits with state



- [Boost meta-programming support](#)
- [Boost type\\_traits tutorial](#)
- [C++11 has many traits](#)

## Further Reading In General

- [Andrei Alexandrescu's Book](#)
- [Herb Sutter's Guru of The Week](#), especially [71](#) and [this](#) article
- And of course, keep reading [Meyers](#)

## Summary

- Learnt
  - Notation for template function/class/meta-programming
  - Uses and limitations of template function/class
  - Template Meta-Programming
    - \* Optimisation, loop unrolling
    - \* Policy classes
    - \* Traits

## Chapter 13

# Appendix - MPI

### MPI Design Example

#### Objectives for this chapter

In this chapter, we give an extended introduction to a particular parallel programming example, introducing some common programming patterns which occur in working with MPI.

### Smooth Life

#### Conway's Game of Life

The Game of Life is a cellular automaton devised by John Conway in 1970.

A cell, on a square grid, is either alive or dead. On the next iteration:

- An alive cell:
  - remains alive if it has 2 or 3 alive neighbours out of 8
  - dies (of isolation or overcrowding) otherwise
- A dead cell:
  - becomes alive if it has exactly 3 alive neighbours
  - stays dead otherwise

#### Conway's Game of Life

This simple set of rules produces beautiful, complex behaviours:



Figure 13.1: “Gospers glider gun” by Kieff. Licensed under CC BY-SA 3.0 via Wikimedia Commons.

## Smooth Life

Smooth Life, proposed by Stephan Rafler, extends this to continuous variables:

- The neighbourhood is an integral over a ring centered on a point,  $R(r)$ .
- The local state is an integral over the disk inside the ring,  $D(r)$ .

(The ring has outer radius  $3 \times$  inner radius, so that the area ratio of 1:8 matches the grid version.)

## Smooth Life

- Each point has some degree of ‘aliveness’ in  $[0, 1]$
- At the next timestep
  - a ‘fully alive’ cell ( $D(r) = 1$ ) remains alive if the neighbourhood is within the ‘death’ limits, i.e.  $d^{(1)} \leq R(r) \leq d^{(2)}$
  - a ‘fully dead’ cell ( $D(r) = 0$ ) becomes alive if the neighbourhood is within the ‘birth’ limits, i.e.  $b^{(1)} \leq R(r) \leq b^{(2)}$
- We actually define the new aliveness  $S(D(r), R(r))$  as a smoothly varying function blending between these limiting cases, using a ‘sigmoid’ shape

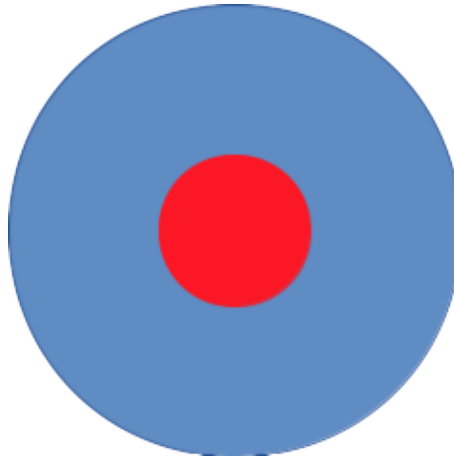


Figure 13.2: A cell and its neighbourhood

## Smooth Life on a computer

We discretise Smooth Life using a grid, so that the integrals become sums. The aliveness variable becomes a floating point number.

To avoid the hard-edges of a “ring” and “disk” defined on a grid, we weight the sum by the fraction of a cell that would fall inside the ring or disk:

If the distance  $d$  from the edge of the ring is within 0.5 units, we weight the integral by  $2d - 1$ , so that it smoothly varies from 1 just inside to 0 just outside.

## Smooth Life

Smooth Life shows even more interesting behaviour:

[SmoothLifeVideo](#)

- Gliders moving any direction
- “Tension tubes”

## Exercise

We will create the following two functions:

- a function to compute the two integrals  $D(r)$  and  $R(r)$  as  $D(x_0, y_0) = \sum_{i,j \in \text{grid}} F(i, j) \text{Disk}(r = \|(i - x_0, j - y_0)\|)$  with  $F(i, j)$  the current value at point  $(i, j)$

- the main update function:  
loop over all points  $(x, y)$  in field:
  - compute integrals centered at  $(x, y)$
  - update the field using the transition function

All other functions, including the transition function, Disk, etc are given.

## Square domain into a 1-d vector

We wrap the 2-d grid into a 1-d vector in **row-major** format:

$$F(i, j) \Leftrightarrow F(I)$$

with  $i = I / N_x$  and  $j = I \% N_x$ ,

with  $N_x$  the number of points in direction  $x$ .

## Distances wrap around a torus

We use periodic boundary conditions: the field is a torus.

```
int Smooth::TorusDistance(int x1, int x2, int size) const {
    auto const remainder = std::abs(x1 - x2) % size;
    return std::min(remainder, std::abs(remainder - size));
}
```

## Smoothed edge of ring and disk

Disk( $r$ ):

```
double Smooth::Disk(distance radius) const {
    if(radius > inner + smoothing / 2)
        return 0.0;
    if(radius < inner - smoothing / 2)
        return 1.0;
    return (inner + smoothing / 2 - radius) / smoothing;
}
```

## Automated tests for mathematics

```
///! Smooth step function: 0 at -infty, 1 at +infty
static double Sigmoid(double variable, double center, double width);

SECTION("Sigmoid function is correct") {
    double e = std::exp(1.0);
    REQUIRE(Smooth::Sigmoid(1.0, 1.0, 4.0) == 0.5);
    REQUIRE(std::abs(Smooth::Sigmoid(1.0, 0.0, 4.0) - e / (1 + e)) < 0.0001);
    REQUIRE(Smooth::Sigmoid(10000, 1.0, 4.0) == 1.0);
    REQUIRE(std::abs(Smooth::Sigmoid(0.0, 1.0, 0.1)) < 0.001);
}
```

## Comments

We can see that this is pretty slow:

If the overall grid is  $M$  by  $N$ , and the range of interaction ( $3*$  the inner radius), is  $r$ , then each time step takes  $MNr^2$  calculations: if we take all of these proportional as we “fine grain” our discretisation (a square domain, and a constant interaction distance in absolute units), the problem grows like  $N^4$ !

So let’s parallelize!

## “Ideal” Domain decomposition

### Domain decomposition

One of the most important problems in designing a parallel code is “Domain Decomposition”:

- How will I divide up the calculation between processes?

Design objectives in decomposition are:

- Minimise communication
- Optimise load balance (share out work evenly)

## Decomposing Smooth Life

We'll go for a 1-d spatial decomposition for smooth life, dividing the domain into "Stripes" along the x-axis.

If we have an  $N$  by  $M$  domain, and  $p$ , processes, each process will be responsible for  $NM/p$  cells, and  $NMr^2/p$  calculations.

## Static and dynamic balance

This will achieve perfect **static** load balance: the average work done by each process is the same. If we were not solving in a rectangular grid, this would have been harder.

However, since the calculation can (perhaps) be done quicker when the domain is empty, and our field will vary as time passes, we will not achieve perfect **dynamic** load balance.

## Communication in Smooth Life

Given that each cell needs to know the state of cells within a range  $r = 3r_{\text{in}}d$ , where  $r_{\text{in}}d$  is the inner radius of the neighbourhood ring, and  $r$  the outer radius, we need to get this information to the neighbouring sites.

This is **great**: we only need to transfer information to neighbours, not all the other processes. Such "local" communication results in fast code.

The amount of communication to take place each time step is proportional to  $rMp$ , but assuming an appropriate network topology exists, each pair of neighbours can look after their communication at the same time, so communication will take time proportional to  $rMp/p=rM$ .

## Strong scaling

We therefore expect the time taken for a simulation to vary like:  $Mr(k + Nr/p)$ . (Where  $k$  is a parameter describing the relative time to communicate one cell's state compared to the time for calculating one cell.)

Thus, we see that for a FIXED problem size, the benefit of parallelism will disappear and communication will dominate: this is Amdahl's law again.

## Weak scaling

However, if we consider larger and larger problems, growing  $N$  as  $p$  grows, then we can stop communication overtaking us. This is a common outcome: **local**

problems provide perfect **weak** scaling (until network congestion or IO problems dominate).

Any NON-LOCAL communication, where the total amount of time for communication to take place grows as the number of processes does (such as a gather, which takes time  $p$ , or a reduction, like  $\ln(p)$ , or an all-to-all, like  $p^2$ ) means that perfect weak scaling can't be achieved.

### Exercise: Blocking Collective

We will parallelize the update and integral functions by having each process work on a contiguous strip of the whole field.

For simplicity, each process owns a full replica of the field in memory. This is inefficient since each process owns memory describing a part of the field it will never use. Improving this is fairly easy, but requires some more bookkeeping. Do try it at home!

Exercise: parallelize using a blocking collective.

### Exercise: Halo update

Parallelize using a non-blocking collective and layer computation and calculation:

1. send data (part of the field) other processes need
2. update part of owned field that does not need data from other processes
3. receive data from other processes
4. update part of owned field that needs data from other processes

This is called a halo update and quite common to domain decomposition problems. We can layer communication and computation even more by splitting over data on the left and on the right boundaries.

### Header

```
#include <string>
#include <vector>
#include <mpi.h>

typedef double density;
typedef double distance;
typedef double filling;

class Smooth {
```



```

public:
    Smooth(int sizex = 100, int sizey = 100, distance inner = 21.0, filling birth_1 = 0.278,
           filling birth_2 = 0.365, filling death_1 = 0.267, filling death_2 = 0.445,
           filling smoothing_disk = 0.147, filling smoothing_ring = 0.028);
    int Size() const;
    int Sizex() const;
    int Sizey() const;
    int Range() const;
    int Frame() const;
    const std::vector<density> &Field() const;
    void Field(std::vector<density> const &input);

    /// \brief Piecewise linear function defining the disk
    /// \details
    /// - 1 inside the disk
    /// - 0 outside the disk
    /// -  $0 < x < 1$  in the smoothing region
    double Disk(distance radius) const;
    /// \brief Piecewise linear function defining the ring
    /// \details
    /// - 1 inside the ring
    /// - 0 outside the ring
    /// -  $0 < x < 1$  in the smoothing region
    double Ring(distance radius) const;
    /// "Sigmoid_Signature"
    /// Smooth step function: 0 at -infty, 1 at +infty
    static double Sigmoid(double variable, double center, double width);
    /// end
    ///  $e^{-4x / width}$ : 0 at -infty, 1 at +infty
    static double Sigmoid(double x, double width);
    density Transition(filling disk, filling ring) const;
    /// Find the distance between 2 indices on a 1d torus
    int TorusDistance(int x1, int x2, int size) const;
    /// Find the Euclidean distance between points on a torus
    double Radius(int x1, int y1, int x2, int y2) const;
    /// Value of the integral over a single ring
    double NormalisationRing() const;
    /// Value of the integral over a single disk
    double NormalisationDisk() const;
    /// Sets the playing field to random values
    void SeedRandom();
    /// Sets the playing field to constant values
    void SeedConstant(density constant = 0);
    /// Adds a disk to the playing field
    void AddDisk(int x0 = 0, int y0 = 0);
    /// Adds a ring to the playing field

```

```

void AddRing(int x0 = 0, int y0 = 0);
/// Sets a single pixel in the field
void AddPixel(int x0, int y0, density value);
/// Moves to next step
void Update();
/// Prints current field to standard output
void Write(std::ostream &out);

/// Returns {disk, ring} integrals at point (x, y)
std::pair<density, density> Integrals(int x, int y) const;

/// Linear index from cartesian index
int Index(int i, int j) const;
/// Cartesian index from linear index
std::pair<int, int> Index(int i) const;

private:
    int sizex, sizey;
    std::vector<density> field, work_field;
    filling birth_1, death_1;
    filling birth_2, death_2;
    filling smoothing_disk, smoothing_ring;
    distance inner, outer, smoothing;
    int frame;
    double normalisation_disk, normalisation_ring;

#ifdef HAS_MPI
public:
    MPI_Comm const &Communicator() const { return communicator; }
    void Communicator(MPI_Comm const &comm) { communicator = comm; }

    /// Update which layers computation and communication
    void LayeredUpdate();

    /// Figure start owned sites for given rank
    static int OwnedStart(int nsites, int ncomms, int rank);

    /// \brief Syncs fields between processes
    /// \details Assumes that each rank owns the sites given by OwnedRange.
    static void WholeFieldBlockingSync(std::vector<density> &field, MPI_Comm const &comm);

    /// \brief Syncs fields between processes without blocking
    /// \details Assumes that each rank owns the sites given by OwnedRange.
    static MPI_Request WholeFieldNonBlockingSync(std::vector<density> &field, MPI_Comm const &comm);
private:
    MPI_Comm communicator;

```

```
#endif
};
```

## Tests

```
#ifndef HAS_MPI
#define CATCH_CONFIG_MAIN
#else
#define CATCH_CONFIG_RUNNER
#include <catch.hpp>
#endif

#include <cmath>
#include <random>
#include "catch.hpp"
#include "smooth.h"

TEST_CASE("Compute Integrals") {
    Smooth smooth(300, 300);
    smooth.SeedConstant(0);

    // check for different positions in the torus
    for(auto const x : {150, 298, 0}) {
        for(auto const y : {150, 298, 0}) {
            SECTION("At position (" + std::to_string(x) + ", " + std::to_string(y) + ")") {
                SECTION("Ring only") {
                    smooth.AddRing(150, 150);

                    auto const result = smooth.Integrals(150, 150);
                    // 0.1 accuracy because of smoothing
                    CHECK(std::get<0>(result) == Approx(0).epsilon(0.1));
                    CHECK(std::get<1>(result) == Approx(1).epsilon(0.1));
                }

                SECTION("Disk only") {
                    smooth.AddDisk(150, 150);
                    auto const result = smooth.Integrals(150, 150);
                    CHECK(std::get<0>(result) == Approx(1).epsilon(0.1));
                    CHECK(std::get<1>(result) == Approx(0).epsilon(0.1));
                }

                SECTION("Disk and ring") {
                    smooth.AddRing(150, 150);
                    smooth.AddDisk(150, 150);
                    auto const result = smooth.Integrals(150, 150);
                }
            }
        }
    }
}
```

```

        CHECK(std::get<0>(result) == Approx(1).epsilon(0.1));
        CHECK(std::get<1>(result) == Approx(1).epsilon(0.1));
    }
}
}

TEST_CASE("Update") {
    // just test playing with a single pixel lit up sufficiently that the
    // transition is non-zero in the ring.
    auto const radius = 5;
    Smooth smooth(100, 100, radius);
    smooth.AddPixel(50, 50, 0.3 * smooth.NormalisationRing());
    CHECK(std::get<0>(smooth.Integrals(50, 50))
        == Approx(0.3 * smooth.NormalisationRing() / smooth.NormalisationDisk()));

    // check the integrals are numbers for which Transition gives non-zero result
    // in the ring
    CHECK(std::get<1>(smooth.Integrals(50, 50)) == Approx(0));
    CHECK(std::get<0>(smooth.Integrals(40, 40)) == Approx(0));
    CHECK(std::get<1>(smooth.Integrals(40, 40)) == Approx(0.3));
    CHECK(std::get<0>(smooth.Integrals(42, 39)) == Approx(0));
    CHECK(std::get<1>(smooth.Integrals(42, 39)) == Approx(0.3));

    // Now call update
    smooth.Update();
    auto const field = smooth.Field();
    // And check death in the disk
    CHECK(field[smooth.Index(50, 50)] == Approx(0));
    CHECK(field[smooth.Index(51, 52)] == Approx(0));
    // And check life in the ring
    CHECK(field[smooth.Index(45, 45)] == Approx(smooth.Transition(0, 0.3)));
    CHECK(field[smooth.Index(42, 39)] == Approx(smooth.Transition(0, 0.3)));
    // And check death outside
    CHECK(field[smooth.Index(15, 15)] == Approx(0));
}

#ifdef HAS_MPI
TEST_CASE("Arithmetics for plitting a field on different nodes") {
    CHECK(Smooth::OwnedStart(5, 2, 0) == 0);
    CHECK(Smooth::OwnedStart(5, 2, 1) == 3);

    for(int i(0); i < 5; ++i)
        CHECK(Smooth::OwnedStart(5, 5, i) == i);

    // with too many procs, some procs have empty ranges

```

```

    for(int i(5); i < 10; ++i)
        CHECK(Smooth::OwnedStart(5, 10, i) == 5);
}

TEST_CASE("Sync whole field") {
    int rank, ncomms;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ncomms);

    // Create known field: -1 outside owned range, equal to rank inside
    // Different on each process!
    // Also, we make sure the size does not split evenly with the number of procs,
    // because that is a harder test.
    std::vector<density> field(5 * ncomms + ncomms / 3, -1);
    std::fill(field.begin() + Smooth::OwnedStart(field.size(), ncomms, rank),
              field.begin() + Smooth::OwnedStart(field.size(), ncomms, rank + 1), rank);

    SECTION("Blocking synchronisation") {
        Smooth::WholeFieldBlockingSync(field, MPI_COMM_WORLD);

        for(int r(0); r < ncomms; ++r)
            CHECK(std::all_of(field.begin() + Smooth::OwnedStart(field.size(), ncomms, r),
                              field.begin() + Smooth::OwnedStart(field.size(), ncomms, r + 1),
                              [r](density d) { return std::abs(d - r) < 1e-8; }));
    }

    SECTION("Non blocking synchronisation") {
        auto request = Smooth::WholeFieldNonBlockingSync(field, MPI_COMM_WORLD);
        MPI_Wait(&request, MPI_STATUS_IGNORE);

        for(int r(0); r < ncomms; ++r)
            CHECK(std::all_of(field.begin() + Smooth::OwnedStart(field.size(), ncomms, r),
                              field.begin() + Smooth::OwnedStart(field.size(), ncomms, r + 1),
                              [r](density d) { return std::abs(d - r) < 1e-8; }));
    }
}

TEST_CASE("Serial vs parallel") {
    Smooth serial(100, 100, 5);
    Smooth parallel(100, 100, 5);
    parallel.Communicator(MPI_COMM_WORLD);

    // generate one field for all Smooth instances
    std::vector<density> field(100 * 100);
    std::random_device rd; // Will be used to obtain a seed for the random number engine
    std::mt19937 gen(rd());

```

```

std::uniform_real_distribution<> randdist(0, 1);
std::generate(field.begin(), field.end(), [&randdist, &gen]() { return randdist(gen); });
MPI_Bcast(field.data(), field.size(), MPI_DOUBLE, 0, MPI_COMM_WORLD);

// set the fields for both Smooth instances
serial.Field(field);
parallel.Field(field);

int rank, ncomms;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &ncomms);

auto const start = Smooth::OwnedStart(field.size(), ncomms, rank);
auto const end = Smooth::OwnedStart(field.size(), ncomms, rank);
// if this is false, then the test itself is wrong
CHECK(std::equal(serial.Field().begin() + start, serial.Field().begin() + end,
                 parallel.Field().begin() + start));

SECTION("Blocking synchronization") {
    // check fields are the same in parallel and in serial for a few iterations
    for(int i(0); i < 3; ++i) {
        serial.Update();
        parallel.Update();
        CHECK(std::equal(serial.Field().begin() + start, serial.Field().begin() + end,
                        parallel.Field().begin() + start));
    }
}

SECTION("Layered communication-computation") {
    for(int i(0); i < 3; ++i) {
        serial.Update();
        parallel.LayeredUpdate();
        CHECK(std::equal(serial.Field().begin() + start, serial.Field().begin() + end,
                        parallel.Field().begin() + start));
    }
}

TEST_CASE("Smooth model can be instantiated and configured", "[Smooth]") {

    SECTION("Smooth can be constructed") {
        Smooth smooth;
        REQUIRE(smooth.Size() == 10000);
        REQUIRE(smooth.Field().size() == smooth.Size());
    }
}

```

```

TEST_CASE("Smooth mathematical functions are correct", "[Smooth]") {
    Smooth smooth;
    SECTION("Disk support function is correct") {
        REQUIRE(smooth.Disk(500) == Approx(0));
        REQUIRE(smooth.Disk(21.6) == 0.0);
        REQUIRE(smooth.Disk(21.4) > 0.0);
        REQUIRE(smooth.Disk(21.4) < 1.0);
        REQUIRE(smooth.Disk(20.6) > 0.0);
        REQUIRE(smooth.Disk(20.6) < 1.0);
        REQUIRE(smooth.Disk(20.4) == Approx(1.0));
        REQUIRE(smooth.Disk(19.0) == Approx(1.0));
        REQUIRE(smooth.Disk(21.0) == Approx(0.5));
    }
    SECTION("Ring support function is correct") {
        REQUIRE(smooth.Ring(22) == 1.0);
        REQUIRE(smooth.Ring(21.6) == 1.0);
        REQUIRE(smooth.Ring(21.4) > 0.0);
        REQUIRE(smooth.Ring(21.4) < 1.0);
        REQUIRE(smooth.Ring(20.6) > 0.0);
        REQUIRE(smooth.Ring(20.6) < 1.0);
        REQUIRE(smooth.Ring(20.4) == 0.0);
        REQUIRE(smooth.Ring(21.0) == 0.5);
        REQUIRE(smooth.Ring(64.0) == 0.0);
        REQUIRE(smooth.Ring(63.6) == 0.0);
        REQUIRE(smooth.Ring(63.4) > 0.0);
        REQUIRE(smooth.Ring(63.4) < 1.0);
        REQUIRE(smooth.Ring(62.6) > 0.0);
        REQUIRE(smooth.Ring(62.6) < 1.0);
        REQUIRE(smooth.Ring(62.4) == 1.0);
        REQUIRE(smooth.Ring(63.0) == 0.5);
    }
}

/// Sigmoid_Test
SECTION("Sigmoid function is correct") {
    double e = std::exp(1.0);
    REQUIRE(Smooth::Sigmoid(1.0, 1.0, 4.0) == 0.5);
    REQUIRE(std::abs(Smooth::Sigmoid(1.0, 0.0, 4.0) - e / (1 + e)) < 0.0001);
    REQUIRE(Smooth::Sigmoid(10000, 1.0, 4.0) == 1.0);
    REQUIRE(std::abs(Smooth::Sigmoid(0.0, 1.0, 0.1)) < 0.001);
}

/// end
SECTION("Transition function is correct") {
    REQUIRE(std::abs(smooth.Transition(1.0, 0.3) - 1.0) < 0.1);
    REQUIRE(smooth.Transition(1.0, 1.0) == Approx(0));
    REQUIRE(std::abs(smooth.Transition(0.0, 0.3) - 1.0) < 0.1);
}

```

```

    REQUIRE(std::abs(smooth.Transition(0.0, 0.0)) < 0.1);
}
SECTION("Wraparound Distance is correct") {
    REQUIRE(smooth.TorusDistance(95, 5, 100) == 10);
    REQUIRE(smooth.TorusDistance(5, 96, 100) == 9);
    REQUIRE(smooth.TorusDistance(5, 10, 100) == 5);
    REQUIRE(smooth.Radius(10, 10, 13, 14) == 5.0);
}
}

TEST_CASE("NormalisationsAreCorrect") {
    Smooth smooth(100, 100, 10);
    SECTION("Disk Normalisation is correct") {
        // Should be roughly pi*radius*radius,
        REQUIRE(std::abs(smooth.NormalisationDisk() - 314.15) < 1.0);
    }
    SECTION("Ring Normalisation is correct") {
        // Should be roughly pi*outer*outer-pi*inner*inner, pi*100*(9-1), 2513.27
        REQUIRE(std::abs(smooth.NormalisationRing() - 2513.27) < 2.0);
    }
}

TEST_CASE("FillingsAreUnityWhenSeeded") {
    Smooth smooth;
    smooth.SeedConstant(0);
    SECTION("Disk Filling Unity With Disk Seed") {
        smooth.AddDisk();
        REQUIRE(std::get<0>(smooth.Integrals(0, 0)) == Approx(1).epsilon(0.1));
    }

    SECTION("Disk Filling Zero With Ring Seed") {
        smooth.AddRing();
        REQUIRE(std::get<0>(smooth.Integrals(0, 0)) == Approx(0).epsilon(0.1));
    }
    SECTION("Ring Filling Unity With Ring Seed") {
        smooth.AddRing();
        REQUIRE(std::get<1>(smooth.Integrals(0, 0)) == Approx(1).epsilon(0.1));
    }
}

TEST_CASE("FillingFieldHasRangeofValues") {
    Smooth smooth(300, 300);
    smooth.SeedConstant(0);
    smooth.AddRing();
    double min = 1.0;
    double max = 0.0;

```



```

    for(int x = 0; x < 300; x++) {
        double filling = std::get<1>(smooth.Integrals(x, 0));
        min = std::min(min, filling);
        max = std::max(max, filling);
    }
    REQUIRE(min < 0.2);
    REQUIRE(max > 0.4);
}

int main(int argc, const char **argv) {
    // There must be exactly once instance
    Catch::Session session;

    MPI_Init(&argc, const_cast<char ***>(&argv));
    auto const result = session.run();

    MPI_Finalize();

    return result;
}
#endif

```

## Implementation

```

#include <cassert>
#include <cmath>
#include <cstdlib>
#include <iostream>

#include "smooth.h"

Smooth::Smooth(int sizex, int sizey, distance inner, filling birth_1, filling birth_2,
               filling death_1, filling death_2, filling smoothing_disk, filling smoothing_ring,
               : sizex(sizex), sizey(sizey), field(sizex * sizey), work_field(sizex * sizey), inner(inner),
               birth_1(birth_1), birth_2(birth_2), death_1(death_1), death_2(death_2),
               smoothing_disk(smoothing_disk), smoothing_ring(smoothing_ring), outer(inner * 3),
               smoothing(1.0)
#ifdef HAS_MPI
    ,
    communicator(MPI_COMM_SELF)
#endif
{
    normalisation_disk = NormalisationDisk();
    normalisation_ring = NormalisationRing();
}

```

```

const std::vector<density> &Smooth::Field() const { return field; };
void Smooth::Field(std::vector<density> const &input) {
    assert(field.size() == input.size());
    field = input;
}

int Smooth::Range() const { return outer + smoothing / 2; }

int Smooth::SizeX() const { return sizex; }
int Smooth::SizeY() const { return sizey; }
int Smooth::Size() const { return sizex * sizey; }

/// "Disk_Smoothing"
double Smooth::Disk(distance radius) const {
    if(radius > inner + smoothing / 2)
        return 0.0;
    if(radius < inner - smoothing / 2)
        return 1.0;
    return (inner + smoothing / 2 - radius) / smoothing;
}
/// end

double Smooth::Ring(distance radius) const {
    if(radius < inner - smoothing / 2)
        return 0.0;
    if(radius < inner + smoothing / 2)
        return (radius + smoothing / 2 - inner) / smoothing;
    if(radius < outer - smoothing / 2)
        return 1.0;
    if(radius < outer + smoothing / 2)
        return (outer + smoothing / 2 - radius) / smoothing;
    return 0.0;
}

double Smooth::Sigmoid(double variable, double center, double width) {
    return Sigmoid(variable - center, width);
}

double Smooth::Sigmoid(double x, double width) { return 1.0 / (1.0 + std::exp(-4.0 * x / width)); }

density Smooth::Transition(filling disk, filling ring) const {
    auto const sdisk = Sigmoid(disk - 0.5, smoothing_disk);
    auto const t1 = birth_1 * (1.0 - sdisk) + death_1 * sdisk;
    auto const t2 = birth_2 * (1.0 - sdisk) + death_2 * sdisk;
    return Sigmoid(ring - t1, smoothing_ring) * Sigmoid(t2 - ring, smoothing_ring);
}

```

```

int Smooth::Index(int i, int j) const { return i * Sizex() + j; }
std::pair<int, int> Smooth::Index(int i) const { return {i / Sizex(), i % Sizex()}; }

/// "Torus_Difference"
int Smooth::TorusDistance(int x1, int x2, int size) const {
    auto const remainder = std::abs(x1 - x2) % size;
    return std::min(remainder, std::abs(remainder - size));
}
/// end

double Smooth::Radius(int x1, int y1, int x2, int y2) const {
    int xdiff = TorusDistance(x1, x2, sizex);
    int ydiff = TorusDistance(y1, y2, sizey);
    return std::sqrt(xdiff * xdiff + ydiff * ydiff);
}

double Smooth::NormalisationDisk() const {
    double total = 0.0;
    for(int x = 0; x < sizex; x++)
        for(int y = 0; y < sizey; y++)
            total += Disk(Radius(0, 0, x, y));
    return total;
}

double Smooth::NormalisationRing() const {
    double total = 0.0;
    for(int x = 0; x < sizex; x++)
        for(int y = 0; y < sizey; y++)
            total += Ring(Radius(0, 0, x, y));
    return total;
}

void Smooth::Update() {
#ifdef HAS_MPI
    int rank, ncomms;
    MPI_Comm_rank(Communicator(), &rank);
    MPI_Comm_size(Communicator(), &ncomms);

    WholeFieldBlockingSync(field, communicator);
    auto const start = OwnedStart(Size(), ncomms, rank);
    auto const end = OwnedStart(Size(), ncomms, rank + 1);
#else
    auto const start = 0;
    auto const end = field.size();
#endif
}

```

```

    for(int i(start); i < end; ++i) {
        auto const xy = Index(i);
        auto const integrals = Integrals(xy.first, xy.second);
        work_field[i] = Transition(integrals.first, integrals.second);
    }

    std::swap(field, work_field);
    frame++;
}

#ifdef HAS_MPI
void Smooth::LayeredUpdate() {
    int rank, ncomms;
    MPI_Comm_rank(Communicator(), &rank);
    MPI_Comm_size(Communicator(), &ncomms);

    // Start synchronising the field
    auto request = WholeFieldNonBlockingSync(field, communicator);
    auto const start = OwnedStart(Size(), ncomms, rank);
    auto const end = OwnedStart(Size(), ncomms, rank + 1);
    // Determine which locations could be affected by other processes
    auto const interaction = Size() * static_cast<int>(std::floor(outer + smoothing / 2 + 1));

    // Define a lambda function to update part of our field
    auto const set_work_field_at_index = [this](int i) {
        auto const xy = Index(i);
        auto const integrals = Integrals(xy.first, xy.second);
        work_field[i] = Transition(integrals.first, integrals.second);
    };

    // Update those locations that cannot be affected by other processes
    for(int i(start + interaction); i < end - interaction; ++i)
        set_work_field_at_index(i);

    // Wait for field sync to finish
    MPI_Wait(&request, MPI_STATUS_IGNORE);

    // Update the remaining locations
    for(int i(start); i < std::min(end, start + interaction); ++i)
        set_work_field_at_index(i);
    for(int i(std::min(end, end - interaction)); i < end; ++i)
        set_work_field_at_index(i);

    std::swap(field, work_field);
    frame++;
}

```

```

}
#endif

std::pair<density, density> Smooth::Integrals(int x, int y) const {
    density ring_total(0), disk_total(0);
    for(std::vector<density>::size_type i(0); i < field.size(); ++i) {
        auto const cartesian = Index(i);
        int deltax = TorusDistance(x, cartesian.first, sizex);
        if(deltax > outer + smoothing / 2)
            continue;

        int deltay = TorusDistance(y, cartesian.second, sizey);
        if(deltay > outer + smoothing / 2)
            continue;

        double radius = std::sqrt(deltax * deltax + deltay * deltay);
        double fieldv = field[i];
        ring_total += fieldv * Ring(radius);
        disk_total += fieldv * Disk(radius);
    }
    return {disk_total / NormalisationDisk(), ring_total / NormalisationRing()};
}

void Smooth::SeedRandom() {
    for(int x = 0; x < sizex; x++)
        for(int y = 0; y < sizey; y++)
            field[Index(x, y)] += (static_cast<double>(rand()) / static_cast<double>(RAND_MAX));
}

void Smooth::SeedConstant(density constant) { std::fill(field.begin(), field.end(), constant); }
void Smooth::AddDisk(int x0, int y0) {
    for(int x = 0; x < sizex; x++)
        for(int y = 0; y < sizey; y++)
            field[Index(x, y)] += Disk(Radius(x0, y0, x, y));
}

void Smooth::AddRing(int x0, int y0) {
    for(int x = 0; x < sizex; x++)
        for(int y = 0; y < sizey; y++)
            field[Index(x, y)] += Ring(Radius(x0, y0, x, y));
}

void Smooth::AddPixel(int x0, int y0, density value) { field[Index(x0, y0)] = value; }

void Smooth::Write(std::ostream &out) {
    for(int x = 0; x < sizex; x++) {

```

```

        for(int y = 0; y < sizey; y++)
            out << field[Index(x, y)] << " , ";
        out << std::endl;
    }
    out << std::endl;
}

int Smooth::Frame() const { return frame; }

#ifdef HAS_MPI
int Smooth::OwnedStart(int nsites, int ncomms, int rank) {
    assert(nsites >= 0);
    assert(ncomms > 0);
    assert(rank >= 0 and rank <= ncomms);
    return rank * (nsites / ncomms) + std::min(nsites % ncomms, rank);
}

void Smooth::WholeFieldBlockingSync(std::vector<density> &field, MPI_Comm const &comm) {
    int rank, ncomms;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &ncomms);

    if(ncomms == 1)
        return;

    std::vector<int> displacements{0}, sizes;

    for(int i(0); i < ncomms; ++i) {
        displacements.push_back(Smooth::OwnedStart(field.size(), ncomms, i + 1));
        sizes.push_back(displacements.back() - displacements[i]);
    }

    MPI_Allgather(MPI_IN_PLACE, sizes[rank], MPI_DOUBLE, field.data(), sizes.data(),
                  displacements.data(), MPI_DOUBLE, comm);
}

MPI_Request Smooth::WholeFieldNonBlockingSync(std::vector<density> &field, MPI_Comm const &comm,
int rank, ncomms;
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &ncomms);

std::vector<int> displacements{0}, sizes;

for(int i(0); i < ncomms; ++i) {
    displacements.push_back(Smooth::OwnedStart(field.size(), ncomms, i + 1));
    sizes.push_back(displacements.back() - displacements[i]);
}

```

```

}

MPI_Request request;
MPI_Iallgatherv(MPI_IN_PLACE, sizes[rank], MPI_DOUBLE, field.data(), sizes.data(),
                displacements.data(), MPI_DOUBLE, comm, &request);
return request;
}
#endif

```

## Halo swap

### Where do we put the communicated data?

The data we need to receive from our neighbour needs to be put in a place where it can be conveniently used to calculate the new state of cells within distance  $r$  of the boundary.

The standard design pattern for this is to use a **Halo Swap**: we extend the memory buffer used for our state, adding new space either side of our main domain to hold a **halo**.

### Domains with a halo

Thus, each process now holds  $N + 2r$  cells:

- $0 \leq x < r$ , the left halo, holding data calculated by the left neighbour
- $r \leq x < 2r$ , data which we calculate, which will form our **left neighbour's right halo**
- $2r \leq x < N$ , data which we calculate, unneeded by neighbours
- $N \leq x < N + r$ , data which we calculate, which will form our **right neighbour's left halo**
- $N + r \leq x < N + 2r$ , the right halo.

### Domains with a halo

```

range(outer + smoothing / 2), local_x_size(sizex / mpi_size),
local_x_size_with_halo(local_x_size + 2 * range), total_x_size(sizex),
x_coordinate_offset(rank * local_x_size - range), local_x_min_calculate(range),
local_x_max_needed_left(2 * range), local_x_max_calculate(range + local_x_size),
local_x_min_needed_right(local_x_size),

```

## Coding with a halo

We will thus **update** the field only from  $r$  to  $N + r$ , but we will **access** the field from 0 to  $N + 2r$ :

```
int from_x=0; int to_x=local_x_size_with_halo

void Smooth::QuickUpdateStripe(int from_x, int to_x) {
    for(int x = from_x; x < to_x; x++) {
        for(int y = 0; y < sizey; y++) {
            double ring_total = 0.0;
            double disk_total = 0.0;
            for(int x1 = 0; x1 < local_x_size_with_halo; x1++) {
                int deltax
                    = TorusDifference(x + x_coordinate_offset, x1 + x_coordinate_offset, total_x_size);
                if(deltax > outer + smoothing / 2)
                    continue;

                for(int y1 = 0; y1 < sizey; y1++) {
                    int deltax = TorusDifference(y, y1, sizey);
                    if(deltax > outer + smoothing / 2)
                        continue;

                    double radius = std::sqrt(deltax * deltax + deltax * deltax);
                    double fieldv = Field(x1, y1);
                    ring_total += fieldv * Ring(radius);
                    disk_total += fieldv * Disk(radius);
                }
            }

            SetNewField(x, y,
                transition(disk_total / normalisation_disk, ring_total / normalisation_ring));
        }
    }
}
```

## Transferring the halo

We need to pass data left at the same time as we receive data from the right.

If we use separate **Send** and **Recv** calls, we'll have a deadlock: process 2 will be sending to process 1, while process 3 is sending to process 2. No process will be executing a **Recv**.



Fortunately, MPI provides **Sendrecv**: expressing that we want to do a blocking **Send** to one process while we simultaneously do a blocking **Recv** from another. Exactly what we need for pass-the-parcel.

## Transferring the halo

```
void Smooth::CommunicateMPI() {
    BufferLeftHaloForSend();
    MPI_Sendrecv(send_transport_buffer, range * sizey, MPI_DOUBLE, left, rank,
                 receive_transport_buffer, range * sizey, MPI_DOUBLE, right, right, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    UnpackRightHaloFromReceive();
    BufferRightHaloForSend();
    MPI_Sendrecv(send_transport_buffer, range * sizey, MPI_DOUBLE, right, mpi_size + rank,
                 receive_transport_buffer, range * sizey, MPI_DOUBLE, left, mpi_size + left,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    UnpackLeftHaloFromReceive();
}
```

## Noncontiguous memory

Our serial solution uses a C++ `vector<vector<double> >` to store our field.

That means each column of data starts at a different location in memory; our data is not contiguous in memory, the outer vector holds a series of pointers to the start of each inner vector.

This means we can't just transmit  $Mr$  doubles in one go by sending from the address of `&field1[0][0]`.

## Buffering

We'll get around this by copying all the data into a send buffer, and unpacking from a receive buffer. This adds to our communication overhead, but doesn't change its scaling behaviour, both overheads are  $O(Mr)$ .

We'll look in a later section how this can be avoided.

## Buffering

```
void Smooth::BufferLeftHaloForSend() {
    for(int x = local_x_min_calculate; x < local_x_min_calculate + range; x++) {
```

```

        for(int y = 0; y < sizey; y++) {
            send_transport_buffer[y * range + x - local_x_min_calculate] = Field(x, y);
        }
    }
}

void Smooth::UnpackRightHaloFromReceive() {
    for(int x = local_x_max_calculate; x < local_x_size_with_halo; x++) {
        for(int y = 0; y < sizey; y++) {
            SetField(x, y, receive_transport_buffer[y * range + x - local_x_max_calculate]);
        }
    }
}

```

## Testing communications

We implement a copy of our buffers which doesn't use MPI:

```

void Smooth::CommunicateLocal(Smooth &left, Smooth &right) {
    BufferLeftHaloForSend();
    std::memcpy(left.receive_transport_buffer, send_transport_buffer,
                sizeof(density) * range * sizey);
    left.UnpackRightHaloFromReceive();
    BufferRightHaloForSend();
    std::memcpy(right.receive_transport_buffer, send_transport_buffer,
                sizeof(density) * range * sizey);
    right.UnpackLeftHaloFromReceive();
}

```

which allows us to check our halos are all set up correctly.

## Testing communications

We test that our copy works as expected:

```

TEST_CASE("CommunicationBufferingFunctionsCorrectly") {
    Smooth smooth(200, 100, 5, 0, 2);
    Smooth smooth2(200, 100, 5, 1, 2);
    REQUIRE(smooth.LocalXSize() == 100);
    REQUIRE(smooth.LocalXSizeWithHalo() == 130);
}

```

```

    REQUIRE(smooth.Radius(0, 0, 0, 0) == 0);
    REQUIRE(smooth2.Radius(0, 0, 0, 0) == 0);
    smooth.SeedDisk(); // Half the Seeded Disk falls in smooth2's domain, so total filling wi
                      // half a disk.
    REQUIRE(smooth.Field(15, 0) == 1.0);
    REQUIRE(std::abs(smooth.FillingDisk(15, 0) - 0.5) < 0.1);
    REQUIRE(smooth2.FillingDisk(85, 0) == 0.0);
    smooth.CommunicateLocal(smooth2, smooth2); // Transport the data
    REQUIRE(std::abs(smooth.FillingDisk(15, 0) - 0.5) < 0.1);
    REQUIRE(smooth2.Field(115, 0) == 1.0);
    REQUIRE(std::abs(smooth2.FillingDisk(115, 0) - 0.5) < 0.1);
}

```

## Testing communications

We test that the MPI comms results are the same as serial

```

    smooth.QuickUpdate();
    paraSmooth.CommunicateMPI();
    paraSmooth.QuickUpdate();
    paraSmooth.CommunicateMPI();
    for(unsigned int x = 0; x < 50; x++) {
        for(unsigned int y = 0; y < 100; y++) {
            REQUIRE(std::abs(smooth.Field(x + 15 + rank * 50, y) - paraSmooth.Field(x + 15, y))
                    < 0.00001);
        }
    }
}

```

## Deployment

### Getting our code onto Legion

We now have to

- Clone our code base onto our cluster
- Load appropriate modules to get the compilers we need
- Build our code
- Construct an appropriate submission script
- Submit the submission script
- Wait for the job to queue and run

- Copy the data back from the cluster
- Analyse it to display our results.

This is a **pain in the neck**

## Scripting deployment

There are various tools that can be used to automate this process.

You should use one.

Since I like Python, I use [Fabric](#) to do this.

You create `fabfile.py` in your top level folder, and at the shell you can write:

```
fab legion.cold
fab legion.sub
fab legion.stat
fab legion.fetch
```

to build and run code on Legion, without ever using ssh. You can be in any folder below the `fabfile.py` level, such as inside a build folder; the command line tool recurses upward to look for this file, (Just like `git` looks for the `.git` folder.)

## Writing fabric tasks

If you know Python, writing fabric tasks is easy:

`fabfile.py`:

```
@task
def build():
    with cd('/home/ucgajhe/smooth/build'):
        with prefix('module load cmake'):
            run('make')

fab build
```

## Templating jobscripts

Editing a jobscript every time you want to change the number of cores you want to run on is tedious. I use a templating tool [Mako](#) to generate the jobscript:

```
## -pe openmpi ${processes}
```

the templating tool fills in anything in `${}` from a variable in the fabric code.

## Mako Template for Smooth Life

### Configuration files

Avoid using lots of command line arguments to configure your program. It's easier to just have one argument, a configuration file path.

- The configuration file can be kept with results for your records.
- Configuration files can be shipped back and forth to the cluster with fabric.

Even if your code is using simple C++ I/O rather than a nice formatting library, it's best to design your config file in a format which other frameworks can easily read. My favourite is [Yaml](#).

```
width: 200
height: 100
range: 5
```

### Results

[It works](#)

## Derived Datatypes

### Avoiding buffering

We've still got our ugly re-buffering of the halo data into contiguous memory, arising from our use of `vector<vector< double > >`

We can get around that by changing to a flat 1-d array of memory, and storing the  $(x, y)$  element at `Field[ $Mx + y$ ]`. This works fine, especially if we refactor all access to get and update from the field into accessors, so we only need make the change in one place.

### Wrap Access to the Field

```
density Smooth::Field(int x, int y) const { return (*field)[sizey * x + y]; }
```

```

void Smooth::SetNewField(int x, int y, density value) { (*fieldNew)[sizey * x + y] = value;

void Smooth::SetField(int x, int y, density value) { (*field)[sizey * x + y] = value; }

void Smooth::SeedField(int x, int y, density value) {
    SetField(x, y, value + Field(x, y));
    if(Field(x, y) > 1.0) {
        SetField(x, y, 1.0);
    }
}

```

## Copy Directly without Buffers

```

void Smooth::CommunicateMPIUnbuffered() {
    MPI_Sendrecv((*field) + sizey * range, range * sizey, MPI_DOUBLE, left, rank,
                (*field) + sizey * local_x_max_calculate, range * sizey, MPI_DOUBLE, right,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv((*field) + sizey * local_x_min_needed_right, range * sizey, MPI_DOUBLE, right,
                mpi_size + rank, (*field), range * sizey, MPI_DOUBLE, left, mpi_size + left,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

```

## Defining a Halo Datatype

It's usually the case in programming that thinking of a whole body of data as a single entity produces cleaner, faster code than programming at the level of the individual datum.

We want to be able to think of the **Halo** as a single object to be transferred, rather than as a series of doubles.

We can do this using an MPI Derived Datatype:

## Declare Datatype

```

void Smooth::DefineHaloDatatype() {
    MPI_Type_contiguous(sizey * range, MPI_DOUBLE, &halo_type);
    MPI_Type_commit(&halo_type);
}

```

## Use Datatype

```
void Smooth::CommunicateMPIDerivedDatatype() {
    MPI_Sendrecv(*field + sizey * local_x_min_calculate, 1, halo_type, left, rank,
                 *field + sizey * local_x_max_calculate, 1, halo_type, right, right, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    MPI_Sendrecv(*field + sizey * local_x_min_needed_right, 1, halo_type, right, mpi_size + rank,
                 *field, 1, halo_type, left, mpi_size + left, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

## Strided datatypes

Supposing we wanted to use a 2-d decomposition. Our y-direction halo's data would not be contiguous in memory.

Let's imagine we used  $Ny + x$  to index into the field instead of  $Mx + y$ : we can define a derived datatype which specifies data as a series of stretches, with gaps.

`MPI_Type_Vector( $M, r, N$ )` would define the relevant type for this:  $M$  chunks, each  $r$  doubles long, each  $N$  doubles apart in memory.

## Overlapping computation and communication

### Wasteful blocking

We are calculating our whole field, then sharing all the halo.

This is wasteful: we don't need our neighbour's data to calculate the data in the **middle** of our field.

We can start transmitting our halo and receiving our neighbour's, while calculating our middle section. Thus, our communication will not take up extra time, as it is overlapped with calculation.

To do this, we need to use asynchronous communication.

### Asynchronous communication strategy

- Start sending/receiving data
- Do the part of calculation independent of needed data
- Wait until the communication is complete (hopefully instant, if it's already finished)
- Do the part of the calculation that needs communicated data

## Asynchronous communication for 2-d halo:

- Start clockwise send/receive
- Calculate middle
- Finish clockwise send
- Start anticlockwise send
- Calculate right
- Finish anticlockwise send
- Calculate left

## Asynchronous MPI

```
void Smooth::InitiateLeftComms() {
    MPI_Isend(*field + sizey * local_x_min_calculate, 1, halo_type, left, rank, MPI_COMM_WORLD,
              &request_left);
    MPI_Irecv(*field + sizey * local_x_max_calculate, 1, halo_type, right, right, MPI_COMM_WORLD,
              &request_left);
}
```

```
void Smooth::ResolveLeftComms() { MPI_Wait(&request_left, MPI_STATUS_IGNORE); }
```

## Implementation of Asynchronous Communication

```
void Smooth::UpdateAndCommunicateAsynchronously() {

    InitiateLeftComms();
    // Calculate Middle Stripe
    QuickUpdateStripe(local_x_max_needed_left, local_x_min_needed_right);
    ResolveLeftComms(); // So we now have the right halo
    InitiateRightComms();
    QuickUpdateStripe(local_x_min_needed_right, local_x_max_calculate);
    ResolveRightComms();
    QuickUpdateStripe(local_x_min_calculate, local_x_max_needed_left);
    SwapFields();
}

density *Smooth::StartOfWritingBlock() { return &(*field)[local_x_min_calculate * sizey]; }
```



## Chapter 14

# Appendix - MPI-I/O

## Appendix - MPI-I/O

### No Longer In Course

- Notes provided [here](#) for historical reasons.

## Parallel Input and Output

### Objectives for the Chapter

In this chapter, we discuss how to get information *out* of our parallel programs, and how to configure them.

Because we need to reconcile the outputs from different processes, and distribute configuration amongst them, this can be quite complex.

We will look at a few different choices of how to do this.

## Visualisation

### Visualisation is important

We've put a lot of emphasis onto using **Unit Tests** to verify your code.

However, it's just as important to *visualise* your results.

“The graph looks OK” is not a good enough standard of truth.

But “the graph looks wrong” is a strong indication of problems!

## Visualisation is hard

Three-D slices through your data, contour surfaces, animations...

These are all really important for understanding scientific output.

But doing the visualisation in C++ can be very very difficult.

## Simulate remotely, analyse locally

Visualise locally if you can.

Use a tool like fabric to organise your data and make it easy to ship it back from the cluster.

Use dynamic languages like Python or Matlab to visualise your data, or tools like [Paraview](#) or [VisIt](#).

## In-Situ visualisation

When working at the highest scales of parallel, saving out raw simulation data becomes impossible: compute power scales faster than storage.

Under these circumstances, it is necessary to do **data reduction** on the cluster.

In-situ visualisation, where animations or graphs are constructed as part of the analysis, is one approach.

## Visualising with NumPy and Matplotlib

The Python toolchain for visualising quantitative data is powerful, fast and easy.

Here's how we turn a NumPy Matrix view of our data, with axes (frames, width, height), into an animation and save it to disk as an mp4:

```
import matplotlib.animation
import matplotlib.pyplot
def plot(frames, outpath):
    figure = matplotlib.pyplot.figure()
    def _animate(frame_id):
        print "Processing frame", frame_id
        matplotlib.pyplot.imshow(frames[frame_id], vmin=0, vmax=1)

    anim = matplotlib.animation.FuncAnimation(figure, _animate, len(frames), interval=100)
    anim.save(outpath)
```

Try doing that with C++ libraries!

## Formatted Text IO

### Formatted Text IO

- Is easy
- Is portable
- Is human-readable
- Is slow
- Wastes space

### When and when not to use text IO

Use text IO:

- Metadata
- Configuration
- Logging

Don't use it:

- For output of large numerical datasets
- For initial conditions
- For checkpoint/restart

### Use libraries to generate formatted text

For a templating library like Mako in C++ try [CTemplate](#)

This is a great way to create XML and YAML files.

Raw CSV file generation with built-in C++ `<iostream>` is not very robust.

Libraries will automatically quote strings which contain commas.

Boost's [Spirit](#) library is good for this too.

### Low-level IO exemplar

Nevertheless, in simple cases where text is of a known format, C++'s built in formatted IO is quick and easy:

```

void TextWriter::Header(int frames) {
    *outfile << smooth.LocalXSize() << " , " << smooth.Sizey() << " , " << rank << " , " << size
        << std::endl;
}

void TextWriter::Write() {
    for(int x = smooth.Range(); x < smooth.LocalXSize() + smooth.Range(); x++) {
        for(int y = 0; y < smooth.Sizey(); y++) {
            *outfile << smooth.Field(x, y) << " , ";
        }
        *outfile << std::endl;
    }
    *outfile << std::endl;
}

```

## Text Data and NumPy

Parsing text data in NumPy is also very easy:

```

buffer=numpy.genfromtxt(data,delimiter=",")[:, :-1]

frames_data=buffer.reshape([frame_count, width/size, height])

```

this is how the animations you saw last lecture were created.

## Text File Bloat

However, representing  $1.3455188104 \cdot 10^{-10}$  in text:

```
1.3455188104e-10,
```

uses many bytes per number, (one per character in ASCII, more in unicode) whereas recording in a binary representation, even at double precision, typically uses 8.

It's also harder to do parallel IO in such files, as the distance of a certain quantity from the start of the file can't be predicted.

## Binary IO in C++

### Binary IO in C++

To do basic binary file IO in C++, we set `std::ios::binary` as we open the file.

However, we can't use the nice `<<` operator, as these still generate formatted chars. We have to use `ostream::write()`, which is a low-level, C-style function.

### Binary IO in C++

```
void BinaryWriter::Header(int frames) {
    outfile->write(reinterpret_cast<char *>(&sizeX), sizeof(int));
    outfile->write(reinterpret_cast<char *>(&sizeY), sizeof(int));
    outfile->write(reinterpret_cast<char *>(&rank), sizeof(int));
    outfile->write(reinterpret_cast<char *>(&size), sizeof(int));
    outfile->write(reinterpret_cast<char *>(&frames), sizeof(int));
}

void BinaryWriter::Write() {
    outfile->write(reinterpret_cast<char *>(smooth.StartOfWritingBlock()),
        local_element_count * sizeof(double));
}
```

### Binary IO in NumPy

```
buffer = numpy.fromfile(data, bulk_type, frame_count*height*width/size)
```

... more on that 'bulk\_type' parameter later.

## Endianness and Portability

### Portability

There's a big problem with binary files: the bytes that get written for a `double` on one platform are different from those on others.

This is particularly problematic in HPC: the architecture of your laptop is unlikely to be the same as on ARCHER, so if when ship your output files back, your nice visualisation code will not work.

## Length of datatypes

The C++ standard **does not** specify how many bytes are used to represent a **double**: just that it is more than a **float**! (And similarly for other datatypes.)

This means that your data will be represented differently on different platforms.

Therefore, don't forget to use `sizeof(type)` whenever working with byte-level routines like MPI, if you want your code to work on both your laptop, and your local supercomputer.

## Endianness

Another problem is Endianness: a **double** is actually **almost** always 8 bytes, with a 1 bit sign, 11 bit exponent, and 52 bit mantissa. (The IEEE standard).

But there's still ambiguity in how these bytes are ordered.

The 4-byte standard signed integer "5" can be represented as:

00000101 00000000 00000000 00000000 (little endian)

or as

00000000 00000000 00000000 00000101 (big endian)

## NumPy dtypes

As long as you're aware of these problems, you can usually make your visualiser compatible with the endianness and byte counts of the data you're getting off your system.

If you're visualising with python you can set your datatype to be e.g. `<f8` for a little endian 8-byte floating point, or `>i4` for a big-endian 4-byte signed integer.

## XDR

XDR, or 'extensible data representation' is a portability standard defined for binary IO. If you write through XDR, data will be converted to the XDR standard representation.

XDR data is big endian, has everything in multiples of 4 bytes, and supports a rich library of appropriate types.

## Writing with XDR

```
#include <cstdio>
#include <rpc/types.h>
#include <rpc/xdr.h>

void XDRWriter::Write() {

    char *start_to_write = reinterpret_cast<char *>(smooth.StartOfWritingBlock());
    xdr_vector(&xdrfile, start_to_write, local_element_count, sizeof(double),
              reinterpret_cast<xdrproc_t>(xdr_double));
}

xdrstdio_create(&xdrfile, myFile, XDR_ENCODE);
```

## Writing from a single process

### One process, one file

So far, we've been writing one process from each file, calling them `frames.dat.0`, `frames.dat.1` etc.

```
fname << "." << rank << std::flush;
```

We've then been reconciling them together in our visualiser code:

```
def append_process_suffix(prefix, process):
    return prefix + '.' + str(process)

def process_many_files(folder, prefix, size, header_type, bulk_type):

    process_frames=[]
    for process in range(size):
        path=os.path.join(folder,append_process_suffix(prefix,process))

    return numpy.concatenate(process_frames, 1)
```

## One process, one file

This is necessary, because we **cannot** simply have multiple processes writing to the same file, without considerable care, as they will:

- Block while waiting for access to the file
- Overwrite each others' content

However, the one-process-one-file approach **does not scale** to large numbers of files: the file system can be overwhelmed once we're running at the thousands-of-processors level.

## Writing from a single process

An alternative approach is to share all the data to a master process. (Which can be done in  $O(\ln p)$  time), and then write from that file.

This avoids the complexity of reconciling datafiles locally, and avoids overwhelming the cluster filesystem with many small files.

## Writing from a single process

```
void SingleWriter::Write() {  
  
    double *receive_buffer;  
  
    if(rank == 0) {  
        receive_buffer = new double[total_element_count];  
    }  
  
    #pragma GCC diagnostic ignored "-Wmaybe-uninitialized"  
    MPI_Gather(smooth.StartOfWritingBlock(), local_element_count, MPI_DOUBLE, receive_buffer,  
              local_element_count, MPI_DOUBLE, 0, MPI_COMM_WORLD);  
    #pragma GCC diagnostic pop  
  
    if(rank == 0) {  
        xdr_vector(&xdrfile, reinterpret_cast<char *>(receive_buffer), total_element_count,  
                  sizeof(double), reinterpret_cast<xdrproc_t>(xdr_double));  
    }  
}
```



## Writing from a single process

It's important to create the file and write the header only on the master process:

```
SingleWriter::SingleWriter(Smooth &smooth, int rank, int size) : SmoothWriter(smooth, rank,
    if(rank != 0) {
        return;
    }
```

## Parallel IO

### Serialising on a single process

We know that \*any task which is not  $O(1/p)$  will eventually dominate the cost as number of processes increases, preventing scaling.

This is Amdahl's law again.

The master-process-writes approach introduces this problem; IO quickly becomes the dominant part of the task, preventing weak scaling as problem sizes and processor counts increase.

### Parallel file systems

Supercomputers provide **parallel file systems**. These store each file in multiple “stripes”: one can obtain as much parallelism in IO as there are stripes in files.

To make use of this, it is necessary to use MPI's parallel IO library, MPI-IO.

## Introduction to MPI-IO

MPI-IO works by accessing files as data buffers like core MPI\_Send and so on:

```
MPI_File_write(outfile, smooth.StartOfWritingBlock(), local_element_count, MPI_DOUBLE,
    MPI_STATUS_IGNORE);
```

### Opening parallel file

All processes work together to open and create the file:

```
MPI_File_open(MPI_COMM_WORLD, const_cast<char *>(fname.str().c_str()),
              MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &outfile);
```

## Finding your place

The hard part is synchronising things so that each process writes it's section of the file:

```
int offset = 4 * sizeof(int) + // The header
            rank * local_element_count * sizeof(double) + // Offset within the j
            smooth.Frame() * total_element_count * sizeof(double); // Frame offset in the j

MPI_File_seek(outfile, offset, MPI_SEEK_SET);
```

## High level research IO libraries

Standard libraries for scientific data formats such as [HDF5](#) support parallel IO.

You should use these if you can, as you'll get the benefits of both endianness-portability and parallel IO, together with built-in metadata support and compatibility with other tools.

Introduction to HDF5 or NetCDF is beyond the scope of this course, but familiarising yourself with these is strongly encouraged!

## Chapter 15

# Appendix - Code Quality

## Appendix - Code Quality

### No Longer In Course

- Notes provided [here](#) for historical reasons.

## Code Quality

### All tests pass, the code works: are we done yet?

Lots of changes can be made to a code, from cosmetic to crucial:

- formatting, linting, and refactoring
- checking for memory leaks
- profiling and performance
- benchmarking

## Tools for pretty code

### The end of brace wars: brainless auto formatting

Code is read more often than written:

```
cpp int GCD(int a,int b) {int r;while(b){r=a%b; a=b;b=r;}return a;}
```

[clang-format](#) is one possible code formatter. Add it (or any equivalent) to your editor for automatic formatting.

## Linting

Software to check code for correctness:

- the compilers themselves: “-Wall”
- [clang-tidy](#)
- [cppcheck](#)

example:

```
cpp if(FFTW_plan_flag != FFTW_ESTIMATE | FFTW_PRESERVE_INPUT) {  
... }
```

## Refactoring

Once tests exist, it is easy and safe to modify the code:

Refactoring means rewriting:

- to simplify existing code
- to simplify future development
- for legibility
- to decrease tech-debt
- to consolidate similar code (avoid copy-pasta)

## Memory leaks

### Checking memory allocation intrusively

There are a variety of compiler flags to check standard memory errors:

- g++: `-fsanitize=address`
- clang: [address sanitizer](#)

Good when debugging/testing, but may impact performance. May not detect all memory errors (e.g. read before initialization).

## Checking memory allocation non-intrusively

[Valgrind](#) is an instrumentation framework for Linux and (older) Mac.

It detects memory errors and leaks by intercepting every memory access, allocation, and deallocation.

Unfortunately, Valgrind currently does not work with Mac OS/X > 10.11.

So let's use docker (on Linux) and docker-machine (Windows, Mac OS/X)!

## Exercise: Traveling salesman solved by Simulated Annealing

Traveling Salesman Problem:

A salesman living in an n-dimensional world must visit N cities. What is the shortest path?

Simulated Annealing:

- Start from a candidate A
- Create a neighbor B of A
- if `path(A) > path(B)`, then swap A and B
- else if `exp(beta * (path(B) - path(A))) > random()`, then swap A and B
- loop until satisfied

## Setting up a docker VM and docker

First download/update to the latest course:

```
git clone https://github.com/UCL-RITS/research-computing-with-cpp
```

Creating a virtual machine is optional on Linux, and necessary on Windows and Mac OS/X

```
> docker-machine create cpp_course \
    --driver virtualbox \
    --virtualbox-memory 4000 \
    --virtualbox-cpu-count 2
> eval $(docker-machine env cpp_course)
```

The last lines lets docker know on which VM it should create containers.

Then ssh into the machine and look for your home directory:

```

> docker-machine ssh cpp_course
> pwd
# Mac users
> ls /Users/
# Linux users
> ls /home
# Windows users
# uhm, no idea :(, look around and let me know!

```

Then, create a Dockerfile specifying the container we want:

```

> mkdir docker_dir
> cat > docker_dir/Dockerfile <<EOF
FROM ubuntu:latest
RUN apt-get update && apt-get install -y cmake g++ valgrind
EOF

```

Build an image of the container

```

> docker build -t course_container /path/to/docker_dir

```

Now build the code in 11Performance/cpp using an instance (a.k.a container) of the image.

First, check you can see the directory with the source:

```

> docker run --rm \
    -v /path/to/source/on/vm:/path/to/source/on/container \
    -w /path/to/source/on/container \
    course_container \
    ls

```

This should print the content of the directory on your machine, if:

- the virtual box VM was set-up to mount your home directory (automatic)
- the container was set up to mount the VM directory (-v path/VM:/path/container)

Finally, replace the ls command to:

1. create a build directory in the source code directory
2. run cmake from the build directory
3. run make in the build directory

## Running valgrind on program called awful

Assuming everything went well, there should be a compiled program called awful.

It can only run inside the container!

It has memory leaks and bugs. Investigate and correct using valgrind:

```
> docker run --rm \
  -v /path/to/source/on/vm:/path/to/source/on/container \
  -w /path/to/source/on/container \
  course_container \
  valgrind -v --leak-check=full --show-leak-kinds=all \
  --track-origin=yes ./awful
```

## Running valgrind on program called less\_bad

Even programs written without explicit memory allocations can have memory bugs.

The next version uses Eigen to solve the same problem.

1. add `libeigen3-dev` to the Dockerfile
2. rebuild the image
3. re-build the code
4. run valgrind on `less_bad`
5. investigate and correct the code

## Profiling a program

### Amdahl's law

Maximum theoretical speedup for parallelizing a given task.

- Given a program that takes a time  $T$  to execute
- When parallelizing a task  $A$  taking time  $P$  over  $n$  threads
- Then the maximum speedup is:

$$S = \frac{T}{T - P + \frac{P}{n}}$$

Some simple cases:

- $P \mapsto 0, n \mapsto \infty$ , then  $S \rightarrow 1$
- $P = 0.5, n \mapsto \infty$ , then  $S \rightarrow 2$

In practice, it means programmers/researchers should measure performance before jumping to “optimize” code (a.k.a. apply the scientific method?).

## Profiling

Refers to measuring how much time the program spends in each function:

- Valgrind, via [callgrind](#) provides an accurate, non-intrusive solution
- [gprof](#) require “instrumenting” the program, e.g. recompiling with the gprof library. It polls the program every so often, a.k.a. “sampling”
- [gperftools](#), intrusive, thread-capable, can select parts of code to profile, info can be visualized by kcachegrind and/or [pprof](#)
- [XCode instruments](#) provides “sampling” without requiring instrumentation

Other considerations to take into account: threads, GPU-specifics, MPI...

## Exercise: Profiling the correct `less_bad`

1. install `kcachegrind` or `qcachegrind` (latter on Mac + Homebrew, or Windows)
2. recompile in release mode + debug info `cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo`
3. Run the following command:

```
> docker run --rm \
  -v /path/to/source/on/vm:/path/to/source/on/container \
  -w /path/to/source/on/container \
  course_container \
  valgrind -v --tool=callgrind ./awful
```

1. Then run `qcachegrind` on the artifact file:

```
> /usr/local/Cellar/qcachegrind/16.12.0/bin/qcachegrind callgrind.out.1
```

Notice the difference between `include` and `self`, e.g. the time spent in the function as whole, vs the time spent strictly in the function excluding sub-function calls.

Questions:



1. How much time is spent on `neighbor`
2. How much time is spent on `TravelDistance::operator()`
3. What happens when you increase the number of cities to 100, or 10000
4. What should we optimize or parallelize?

## Micro-benchmarking

### Introduction

meaningful bit of code, and systematically run performance tests for each Scientific method applied to performance: measure the time taken by each commit.

Profiling can tell us what part to include in a benchmark.

Possible micro-benchmarking frameworks:

- timers in unit test framework (not really accurate)
- [google/benchmark](#)
- [hayai](#), a “google-test”-like framework
- [others](#)

Questions micro-benchmarking can answer:

1. How long does it take on average
2. Standard-deviation from average
3. Worst case

Questions it doesn't always answer:

1. Performance of large subsets or whole application
2. Parallelization: communication vs computation

### Exercise: build and run `micro_benchmark`

`micro_benchmark` reproduces the evaluation function from the travelling salesman problem.

It reproduces how a micro-benchmark framework works:

1. run code N times for warm-up
2. run code N' times for actual measurement

Make sure the code is built in `Release` mode.

Questions:

1. Why is the float implementation faster/slower? Does the speed-up change with the number of dimensions or the number of cities? What happens for `Nrow = 3`?
2. Write a function that computes the distance manually (without Eigen syntactic sugar). Is it faster for `Nrows=2`? What about `Nrows=8`?

Remark:

None of these have tests (this is an exercise in bad code, after all). Do you trust we are solving the travelling salesman problem? I *know* that `awful` does not, beyond the memory bugs... Because I added a bug. But maybe there are further bugs still.

How much easier would it be to test the `manual` code above if we had tests for the evaluation function?

## Chapter 16

# Appendix - Cloud

### Appendix - Cloud computing

#### No Longer In Course

- Notes provided [here](#) for historical reasons.

#### Big data

Some data requires special treatment for collection, storage, and analysis.

The ‘three vs’:

- volume
- velocity
- variety

#### Cloud computing

Cloud computing is an approach built around the concept of shared resources.

- dynamically add and remove computing resources as you need them.
- bring up large clusters of computing power, then shut it own just as quickly

## X\* as a service

Emergence of service oriented approaches:

- Platform as a service (PaaS)
- Software as a service (SaaS)
- Infrastructure as a service (SaaS)

## Exercise 1: Working in the cloud

### Spinning up an instance

This example briefly run through the process of:

- creating an account with a provider of cloud services
- spinning up a single cloud instance using a web interface

### Create an account

A growing number of companies are offering cloud computing services, for example:

- Amazon Web Services: <http://aws.amazon.com>
- Cloudera: <http://www.rackspace.co.uk/cloud>
- Google Cloud Platform: <https://cloud.google.com>
- Microsoft Azure: <http://azure.microsoft.com>

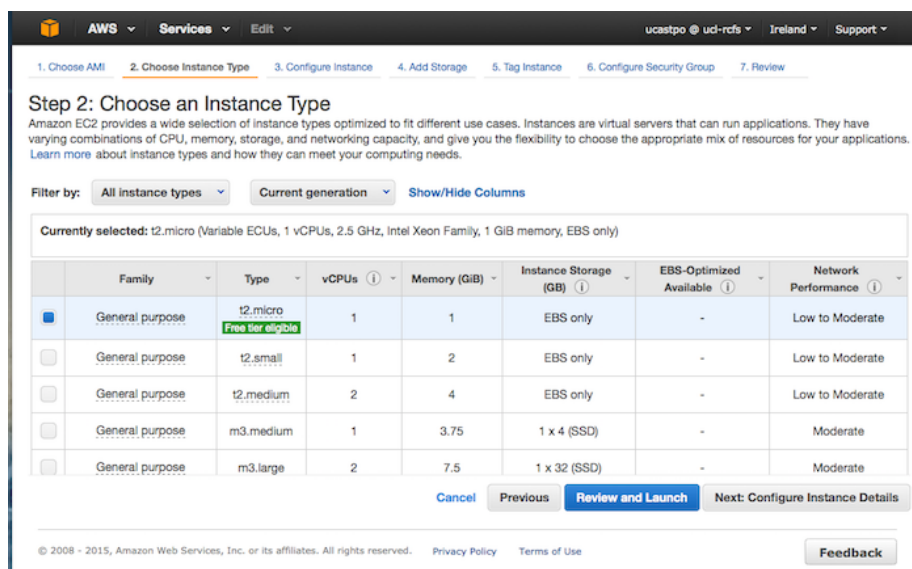
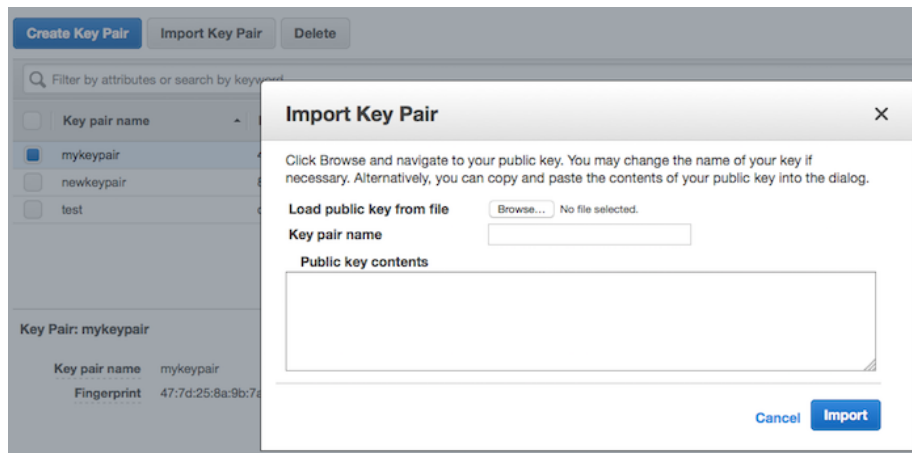
### Create a key pair

In this exercise we will be working with Amazon Web Services.

Amazon uses public key cryptography to authenticate users, so we'll need to create a private/public key pair:

```
# Create a key pair  
$ ssh-keygen -t rsa -f ~/.ssh/ec2 -b 4096
```

We then need to register our public key with Amazon Web Services.



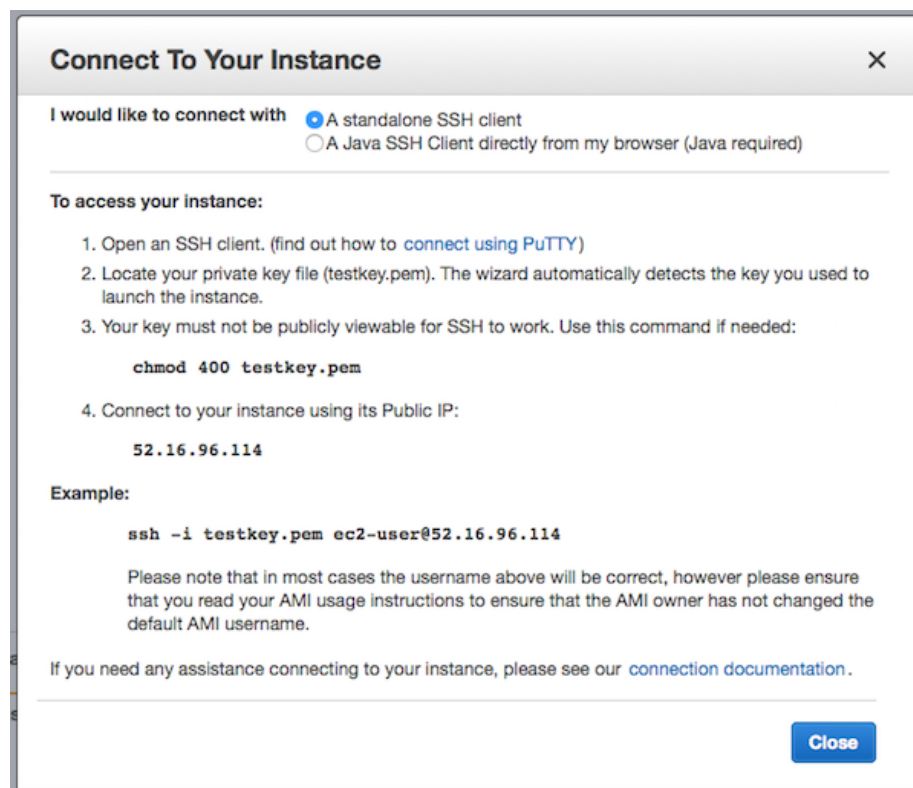
## Create a single instance using the web interface

Navigate to the EC2 Dashboard and create a 'micro' instance (1 CPU, 2.5GHZ, 1GB RAM):

## Connect to the instance with SSH

For Linux instances, the username is 'ec2-user':

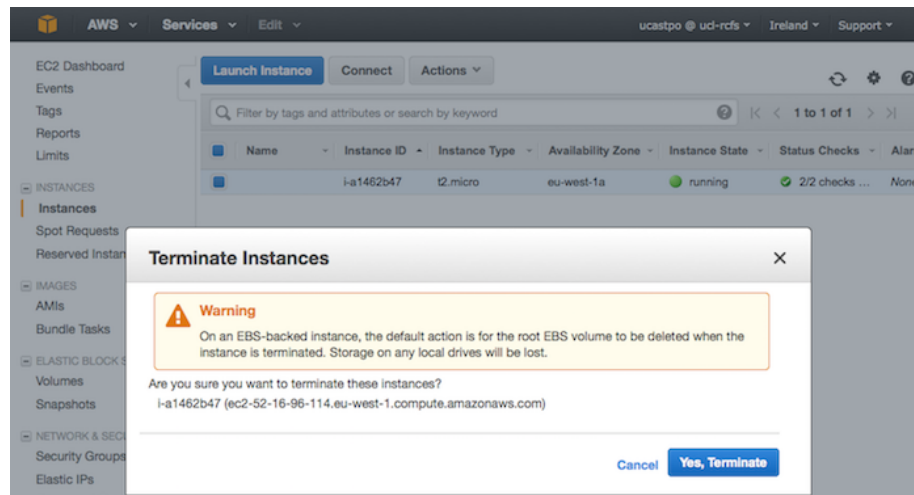
```
# <key_file>: ~/.ssh/ec2
# <public_ID>: 52.16.106.209
$ ssh ec2-user@<public_ID> -i <key_file>
```



```
# Connected
  __|  __|  )
  _| \(\  /  Amazon Linux AMI
  __|\\__|__|
[ec2-user@ip-xxx ~]$
```

## Terminate the server

When finished with the instance, remember to terminate it!



## Exercise 2: Working in the cloud

Again, from the command line...

Install Amazon Web Services Command Line Interface:

<http://docs.aws.amazon.com/cli/latest/userguide/installing.html>

```
# install the tools with pip
sudo pip install awscli
aws ec2 help
```

### Configure the tools

To use the command line tools, you'll need to configure your AWS Access Keys, region, and output format:

<http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html>

```
$ aws configure
```

```
# This configuration is stored locally in a home folder named .aws
# On Unix systems: ~/.aws/config;
```

```
# On Windows: %UserProfile%\aws\config
AWS Access Key ID [*****VDLA]: EXAMPLE
AWS Secret Access Key [*****pa8o]: EXAMPLE
Default region name [eu-west-1]: eu-west-1
Default output format [json]: json
```

## Test our connection to Amazon Web Services

If our connection has been set up correctly, ‘describe-regions’ will return a list of Amazon Web Service regions:

```
# Successful connection will return list of AWS regions
# HTTPSConnectionPool error? Try changing region to eu-west-1
$ aws ec2 describe-regions
```

## Create a key pair

To connect to the instance, we will need a key pair. If you haven’t already done so, create one now:

```
# Create an SSH key pair
$ ssh-keygen -t rsa -f ~/.ssh/ec2 -b 4096
```

Transfer the public key to AWS:

```
# <key_name> is a unique name for the pair (e.g. my-key)
# <key_blob> is the public key: "$(cat ~/.ssh/ec2.pub)"
$ aws ec2 import-key-pair --key-name <key_name> \
  --public-key-material <key_blob>
```

## Create a security group

We’ll also need to create a security group...

```
# creates security group named my-security-group
$ aws ec2 create-security-group \
  --group-name "My security group" \
  --description "SSH access from my local IP address"
```



## Configure the security group

... and allow inbound connections from our local IP address:

```
# create a rule to allow inbound connections on TCP port 22
# find your IP: curl http://checkip.amazonaws.com/
$ aws ec2 authorize-security-group-ingress \
  --group-name "My security group" \
  --cidr <local_IP_address>/32 \
  --port 22 \
  --protocol tcp
```

Note: the /32 at the end of the IP address is the bit number of the [CIDR netmask](#). The /32 mask is equivalent to 255.255.255.255, so defines a single host. Lower values broaden the range of allowed addresses. An IP of 0.0.0.0/0 would allow all inbound connections.

## Locate an appropriate Machine Image

An Amazon Machine Image contains the software configuration (operating system, software...) needed to launch an instance. AMIs are provided by:

- Amazon Web Services
- the user community
- AWS Marketplace

We will search for an Amazon Machine Image ID (AMI-ID) using the command line tools:

```
# Use filter to locate a specific machine (ami-9d23aeaa)
# Filter is a key,value pair
$ aws ec2 describe-images --owners amazon \
  --filters "Name=name,Values=amzn-ami-hvm-2016.09.1.20170119-x86_64-gp2"
```

## Launch an instance

Launch an instance using the Amazon Machine Image ID:

```
# <AMI-ID>: ami-70edb016
# <key_name>: defined when transferring the key
# <group_name>: "My security group"
$ aws ec2 run-instances --image-id <AMI-ID> \
  --key-name <key_name> \
  --instance-type t2.micro \
  --security-groups <group_name>
```

## View the instance

We can check the instance and find the public IP:

```
# View information about the EC2 instances
# e.g. state, root volume, IP address, public DNS name
$ aws ec2 describe-instances
```

## Connect to the instance

Use the public IP to connect:

```
# for Linux instances, the username is ec2-user
# <public_IP>: 52.16.106.209
# <key_file>: ~/.ssh/ec2
$ ssh -i <key_file> ec2-user@<public_IP>
```

You should now be connected!:

```
# Connected
      __|  __|_  )
      _| \(\    /  Amazon Linux AMI
      ---|\\---|---|
[ec2-user@ip-172-31-5-39 ~]$
```

## Terminate the instance

Don't forget to terminate the instance when you have finished:

```
# terminate the instance
# <InstanceId>: i-87086760
$ aws ec2 terminate-instances --instance-ids <InstanceId>

TERMINATINGINSTANCES    i-87086760
CURRENTSTATE             32  shutting-down
PREVIOUSSTATE            16  running
```

## Virtualisation

### Reproducible research

The ability to reproduce the analyses of research studies is increasingly recognised as important.

Several approaches have developed that help researchers to package up code so that their code and dependencies can be distributed and run by others.

### Virtual machines

A popular method for creating a shareable environment is with the use of virtual machines.

The isolated system created by virtual machines can be beneficial, but criticisms include:

- size: virtual machines can be bulky
- performance: virtual machines may use significant system resources

Tools such as Vagrant have helped to simplify the process of creating and using virtual machines:

<https://www.vagrantup.com>

### Virtual environments

Virtual environments offer an alternative to virtual machines. Rather than constructing an entirely new system, virtual environments in general seek to provide reproducible ‘containers’ which are layered on top of an existing environment.

A popular tool for creating virtual environments is Docker:

<https://www.docker.com>

## Exercise 3: Virtualisation

### Reproducible research

In this example, we spin up a single EC2 instance and reproduce the analysis from a study in a virtual environment.

The simple analysis uses `countwords`, a shell script, to count the occurrences of words in a book (in our case `dorian.txt`, The Picture of Dorian Gray).

We will be using Docker to manage our virtual environment: <https://docs.docker.com/userguide/dockerimages/>

## Create a new security group

For this exercise we will be setting up a web connection to the EC2 instance, allowing us to connect to IPython Notebook in a browser. To enable this connection, we will create a new security group:

```
$ aws ec2 create-security-group --group-name "ipython_notebook" \
    --description "web access to ipython notebook"
```

We then need to configure this group to allow inbound connections:

```
# SSH
$ aws ec2 authorize-security-group-ingress \
    --group-name "ipython_notebook" \
    --cidr 0.0.0.0/0 \
    --port 22 \
    --protocol tcp
# port 8888
aws ec2 authorize-security-group-ingress \
    --group-name "ipython_notebook" \
    --cidr 0.0.0.0/0 \
    --port 8888 \
    --protocol tcp
```

## Spin up a cloud instance

As before, we will now spin up a new cloud instance:

```
# <AMI-ID>: e.g. ami-9d23aeea
# <key_name>: e.g. "my_key"
# <group_name>: e.g. "ipython_notebook"
$ aws ec2 run-instances --image-id <AMI-ID> \
    --key-name <key_name> \
    --instance-type t2.micro \
    --security-groups <group_name>
```

## Connect with SSH

When the instance is running, a public IP will be available:

```
$ aws ec2 describe-instances
```

```
"NetworkInterfaces": [
```

```
{
...
  "Association": {
    "PublicIp": "12.34.56.78",
    "PublicDnsName": "ec2-12-34-56-78.eu-west-1.compute.amazonaws.com"
  }
}]
```

Use the public IP to connect over SSH:

```
# <key_file>: e.g. ~/.ssh/ec2
$ ssh ec2-user@<public_IP> -i <key_file>
```

## Install and run Docker on the remote system

Docker needs to be installed on the Cloud instance:

```
[ec2-user@ip-xxx ~]$ sudo yum install -y docker
[ec2-user@ip-xxx ~]$ sudo service docker start
```

## Docker environments are created by a set of commands

Docker environments are created by running a set of commands held within a Dockerfile. A snippet of the Dockerfile used to create our environment is shown below:

```
# Set the source image
FROM ipython/scipystack:master

# Specify commands to run inside the image
RUN apt-get update
RUN apt-get install -y wget

# Create directory for container
RUN mkdir /analysis

# Get the code
ADD https://raw.githubusercontent.com/tompollard/dorian/master/mapper.py \
    ~/analysis/mapper.py

# Run notebook
RUN echo "ipython notebook --ip=0.0.0.0 --port=8888 --no-browser" > /usr/bin/notebook.sh; \
chmod 755 /usr/bin/notebook.sh
```

```
CMD /usr/bin/notebook.sh
```

```
# Open port to the outside world  
EXPOSE 8888  
...
```

## Pull, build, and run the Docker image

Build and run the docker environment:

```
# NB: sudo is not needed when running inside boot2docker  
[ec2-user@ip-xxx ~]$ sudo docker pull tompollard/dorian:master  
[ec2-user@ip-xxx ~]$ sudo docker run -t --rm=true -p 8888:8888 -i \  
    tompollard/dorian:master
```

IPython Notebook is now running in the virtual environment and available to the outside environment on port 8888:

```
[NotebookApp] Using existing profile dir: '/root/.ipython/profile_default'  
[NotebookApp] Serving notebooks from local directory: /analysis  
[NotebookApp] The IPython Notebook is running at: http://0.0.0.0:8888/  
[NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip con  
...
```

## Connect to the IPython Notebook

We can now connect to the IPython Notebook in a browser at the public IP of the Amazon Instance on port 8888 (i.e. `http://{publicIP}:8888`):

## Now use the IPython Notebook to complete the exercise

Using the IPython Notebook, we can reproduce the analysis on both existing and new datasets.

## Distributed computing

### Distributed computing

Dividing a problem into many tasks means analysis can be shared across multiple computers, in a parallel fashion.

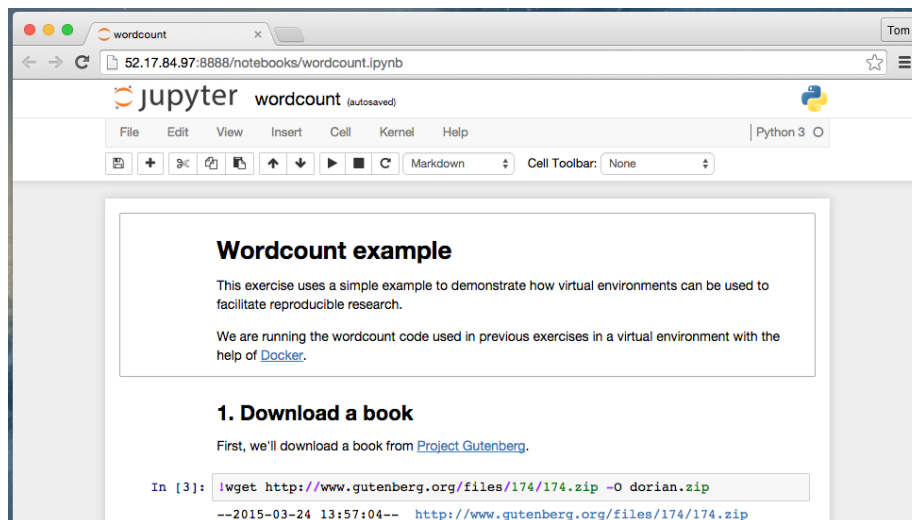


Figure 16.1: IPython Notebook

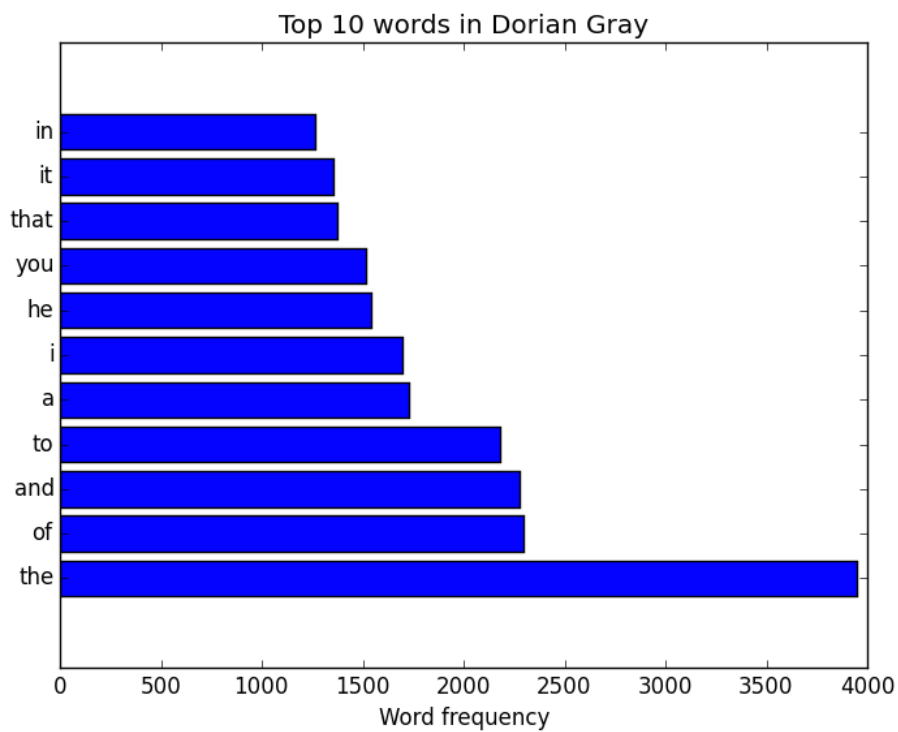


Figure 16.2: Common words in Dorian Gray

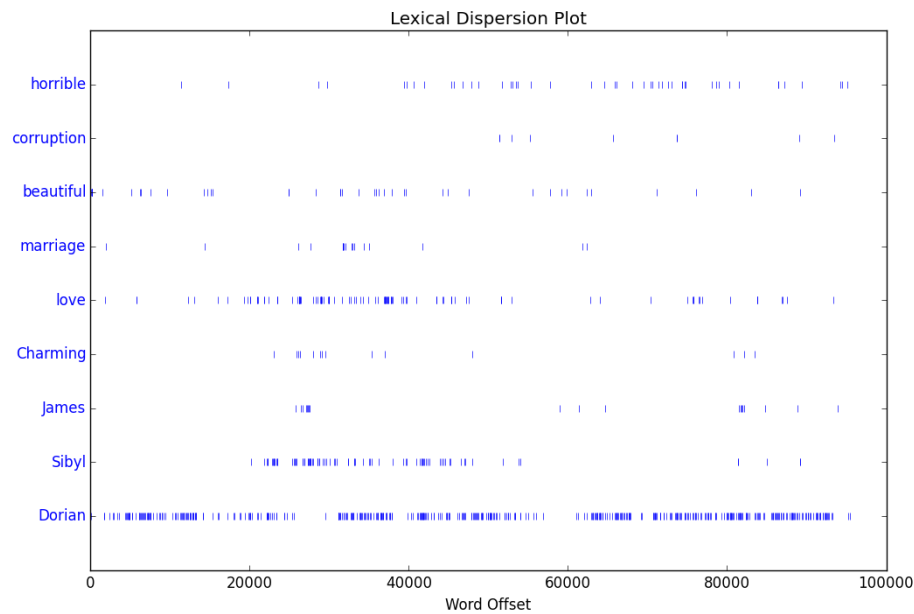


Figure 16.3: Word dispersion

Cloud computing systems have made distributed computing increasingly accessible to individual users.

Introduced by Google in 2004, MapReduce is a popular model that supports distributed computing on large data sets across clusters of computers.

## MapReduce

MapReduce systems are built around the concepts of:

- a mapper, in which the master node takes an input, separates it into sub-problems, and distributes those to worker nodes
- a 'shuffle' step to distribute data from the mapper
- a reducer which collates the answers to the sub-problems and combines them to solve the initial question.

## Distributed file systems

Distributed file systems

- developed to provide reliable data storage across distributed systems



- replicates data across multiple hosts to achieve reliability

## Hadoop

Apache Hadoop:

- framework for distributed processing of large data sets
- scales from single servers to many machines
- includes Hadoop MapReduce and the Hadoop Distributed File System

## Exercise 4: MapReduce

### The ‘Hello World’ of MapReduce

In this example, we will demonstrate how the mapper and reducer can be applied on our local machines to count the number of times each word appears in a book.

#### Choose a book

Find a good book on Project Gutenberg and download it: <http://www.gutenberg.org/browse/scores/top>

```
# The Picture of Dorian Gray
$ wget http://www.gutenberg.org/cache/epub/174/pg174.txt -O dorian.txt
```

```
$ head dorian.txt
```

```
Title: The Picture of Dorian Gray
```

```
The artist is the creator of beautiful things. To reveal art and
conceal the artist is art's aim. The critic is he who can translate
into another manner or a new material his impression of beautiful
things.
```

#### Hadoop streaming

Hadoop streaming enables MapReduce jobs to be run using any executable as the mapper and reducer: <http://hadoop.apache.org/docs/r1.2.1/streaming.html>

The mapper and reducer are written to take line-by-line inputs from stdin and emit the output to stdout.

## Mapper

Our mapper:

- takes lines from our book
- extracts words from the line with a regular expression
- outputs a tab-separated string,value pair for each word (e.g. theword 1)

```
#!/usr/bin/env python
import sys
import re

def mapper(stream):
    pattern = re.compile('[a-zA-Z][a-zA-Z0-9]*')
    for line in stream:
        for word in pattern.findall(line):
            print word.lower() + '\t' + '1'

mapper(sys.stdin)
```

## Reducer

Our reducer:

- takes the word, value pairs generated by the mapper
- sums the count for each word
- outputs a word, count pair

```
#!/usr/bin/env python
import sys

def reducer(stream):
    mydict = {}
    line = stream.readline()
    while line:

        # Get the key/value pair
        line = line.strip()
        word, count = line.split('\t')
        count = int(count)

        # Add to the dict
        if word in mydict:
```

```

        mydict[word] += 1
    else:
        mydict[word] = 1

    # Get the next line
    line = stream.readline()

    # Print the aggregated key/value pairs
    # for word in sorted(mydict.keys()): # order by word
    for word in sorted(mydict, key=mydict.get, reverse=True): # or by count
        print word, '\t', mydict[word]

reducer(sys.stdin)

```

## Demonstrate locally

Download the mapper and reducer:

```
$ git clone https://github.com/tompollard/dorian
```

Create a pipeline to process the book:

```

# sort represents the Hadoop shuffle
$ cat dorian.txt | ./mapper.py | ./reducer.py | sort

the      3948
of       2298
and      2279
to       2181
a        1730
i        1694
he       1544
...

```

## Exercise 5: Distributed computing

### Again, with Hadoop

In the previous exercise, we demonstrated MapReduce on our local computer.

In this exercise we will spin up multiple cloud instances, making use of Hadoop to carry out a distributed MapReduce operation.

We could set up multiple instances in the cloud (for example with EC2), then configure Hadoop to run across the instances. This is not trivial and takes time.

## Hadoop in the cloud

Fortunately, several cloud providers offer configurable Hadoop services. One such service is Amazon Elastic MapReduce (EMR).

Elastic MapReduce provides a framework for:

- uploading data and code to the Simple Storage Service (S3)
- analysing with a multi-instance cluster on the Elastic Compute Cloud (EC2)

## Create an S3 bucket

Create an S3 bucket to hold the input data and our map/reduce functions:

1. Open the Amazon Web Services Console: <http://aws.amazon.com>
2. Select “Create Bucket” and enter a globally unique name
3. Ensure the S3 Bucket shares the same region as other instances in your cluster

Or, through the command line interface:

```
$ aws s3 mb s3://ucl-jh-books-example
```

## Copy data and code to S3

Sample map and reduce functions are available on GitHub:

```
# clone the code from a remote repository
$ git clone https://github.com/tompollard/dorian
```

Copy the data and code to S3:

```
# Copy input code and data to S3
# No support for unix-style wildcards
$ aws s3 cp dorian.txt s3://my-bucket-ucl123/input/
$ aws s3 cp mapper.py s3://my-bucket-ucl123/code/
$ aws s3 cp reducer.py s3://my-bucket-ucl123/code/
```

## Launch the compute cluster

Create a compute cluster with one master instance and two core instances:

```
# Start a EMR cluster
# <ami-version>: version of the machine image to use
# <instance-type>: number and type of Amazon EC2 instances
# <key_name>: "mykeypair"
$ aws emr create-cluster --ami-version 3.11.0 \
    --ec2-attributes KeyName=student01 \
    --instance-groups InstanceGroupType=MASTER,InstanceCount=1,InstanceType=m3.xlarge \
    InstanceGroupType=CORE,InstanceCount=2,InstanceType=m3.xlarge \
    --use-default-roles

"ClusterId": "j-3HGKJHENDODX8"
```

## Get the cluster ID

Get the cluster-id:

```
# Get the cluster-id
$ aws emr list-clusters
...
  "Id": "j-3HGKJHENDODX8",
  "Name": "Development Cluster"
```

## Get the public DNS name of the cluster

When the cluster is up and running, get the public DNS name:

```
# Get the DNS
$ aws emr describe-cluster --cluster-id j-3HGKJHENDODX8 \
    | grep MasterPublicDnsName
...
  "MasterPublicDnsName": "ec2-52-16-235-144.eu-west-1.compute.amazonaws.com"
```

## Connect to the cluster

SSH into the cluster using the username 'hadoop':

```
# SSH into the master node
# <key_file>: ~/.ssh/ec2
# <MasterPublicDnsName>: ec2-52-16-235-144.eu-west-1.compute.amazonaws.com
$ ssh hadoop@<MasterPublicDnsName> -i <key_file>
```

## Run the analysis

View the results

Download the output:

285

## Terminate the cluster

Once our analysis is complete, terminate the cluster:

```
# get cluster id: aws emr list-clusters  
# <cluster_ID>: j-3HGKJHENDODX8  
$ aws emr terminate-clusters --cluster-id <cluster_ID>
```

## Delete the bucket

```
$ aws s3 rb s3://my-bucket-ucl123 --force
```

## Chapter 17

# Appendix - Accelerators

## Appendix - Accelerators

### No Longer In Course

- Notes provided [here](#) for historical reasons.

## Introduction to Accelerators

### Gamers vs Researchers

- Global entertainment industry (2016): \$1800bn
- Global video-game industry (PC and console) (2016): \$100bn
- Global video-game industry – software expenditure (2016): \$50bn
- US government research efforts (pre-Trump), ~\$100bn
- Global HPC industry (2011): \$10bn

### What is an accelerator?

- An accelerator is a piece of computing hardware that performs some function faster than is possible in software running on a general purpose CPU
  - an example is the Floating Point Unit inside a CPU that performs calculations on real numbers faster than the Arithmetic and Logic Unit



- better performance is achieved through concurrency - the ability to perform several operations at once, in parallel
- When an accelerator is separate from the CPU it is referred to as a “hardware accelerator”

## Why would I want to use one?

- Available on many laptop, desktops, phones, and supercomputers
- Can speed-up calculations 2-20x by off-loading from CPU to GPU
- Especially good at streaming
  - 3D graphics
  - MPEG decoding
  - cryptography

## CPU Vectorisation

### Using an accelerator within the CPU

- Consider the following code which performs  $y = a \cdot x + y$  on vectors  $x$  and  $y$ :

```
int saxpy(int n, float a, const float * restrict x, int incx,
          float * restrict y, int incy) {
    if (n < 0)
        return 1;

    for (int i = 0; i < n; i++)
        y[i * incy] += a * x[i * incx];

    return 0;
}
```

- Assuming single precision floating point multiply-add is implemented as one CPU instruction the `for` loop executes  $n$  instructions (plus integer arithmetic for the loop counter, etc.).

### CPUs as multicore vector processors

- In the mid-1990s, Intel were investigating ways of increasing the multimedia performance of their CPUs without being able to increase the clock speed

- Their solution was to implement a set of registers capable of executing the same operation on multiple elements of an array in a single instruction
  - known as SIMD (Single Instruction, Multiple Data)
  - branded as MMX (64-bit, integer only), SSE (128-bit, integer + floating point) and AVX (256-bit)

## Compiler Autovectorisation

- Modern compilers are able to automatically recognise when SIMD instructions can be applied to certain loops
  - elements are known to be contiguous
  - arrays are not aliased

```
int saxpy_fast(int n, float a, const float * restrict x, int incx,
               float * restrict y, int incy) {
    if (n < 0)
        return 1;

    if (incx == 1 && incy == 1) {
        for (int i = 0; i < n; i++)
            y[i] += a * x[i];
    }
    else {
        for (int i = 0; i < n; i++)
            y[i * incy] += a * x[i * incx];
    }

    return 0;
}
```

## Autovectorisation results

- Running this and timing the invocations produces the following output:

```
n = 10000, incx = 1, incy = 1
saxpy: 0.007805ms
saxpy_fast: 0.002801ms
```

- Since Intel's SIMD registers operate on 4 single precision floats at once the first loop executes approximately  $n/4$  instructions
  - resulting in a (near) 4x speedup (Amdahl's Law)

- Can be combined with OpenMP directives (lecture 5) to use SIMD units on multiple cores of the CPU

## Support for Autovectorisation

- GCC
  - use `-ftree-loop-vectorize` to activate
  - use `-fopt-info-vec` to check whether loops are vectorised
  - automatically performed with `-O3`
- ICC
  - use `-vec` (Linux/OSX) or `/Qvec` (Windows) to activate
  - use `-vec-report=n`/`/Qvec-report:n` to check whether loops are vectorised (`n > 0`)
  - automatically performed with `-O2`/`/O2` and higher

## Support for Autovectorisation

- Clang
  - use `-fvectorize` to activate
  - automatically performed at `-O2` and higher
  - no way to see compiler auto-detection
- MSVC
  - activated by default
  - use `/Qvec-report:n` to check whether loops are vectorised (`n > 0`)

## Using a GPU as an Accelerator

### GPUs for 3D graphics

- 3D graphics rendering involves lots of operations on 3 and 4 dimensional vectors
  - positions of vertices (x, y, z)
  - colour and transparency of pixels (RGBA)
- Realtime graphics rendering needs to be fast so accuracy is often sacrificed
  - GPUs are good at 32-bit integer and single precision floating point arithmetic
  - not as good at 64-bit integer and double precision floating point

- 3D graphics operations are independent of one another
  - and can be performed in parallel

## GPUs as multicore vector processors

- Multicore vector processors.
- Much faster memory access to their own memory
- Slow transfer between host and device (~5GB/s)
- Constrained and complex access to different types of device memory
- SIMD: Single Instruction, Multiple Data

| Property         | CPU      | GPU       |
|------------------|----------|-----------|
| No of threads    | <10      | 1000s     |
| SIMD width       | 256 bits | 1024 bits |
| Memory bandwidth | <25 GB/s | <500 GB/s |

## GPU Hardware

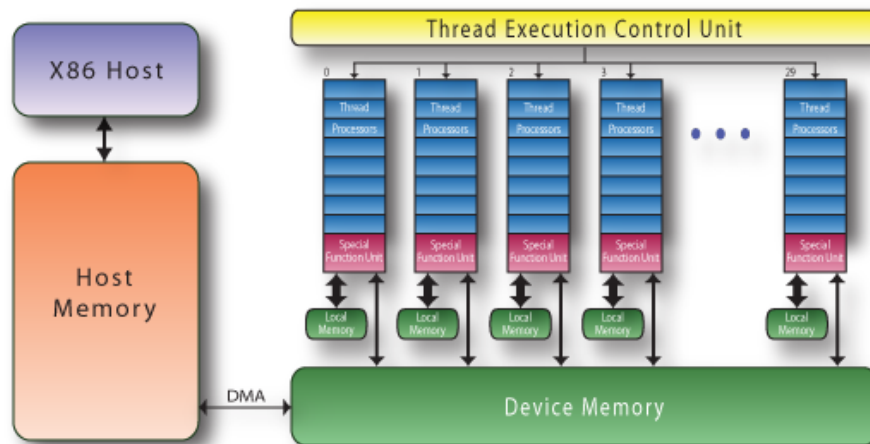
- Each core (multiprocessor - often abbreviated as “SM” or “MP”) has:
  - 128 Single Precision Floating Point Units
  - 32 Double Precision FPU’s (that also perform single precision transcendental functions)
  - 16384 32-bit registers (very fast access)
  - 64KB of “shared” cache memory (fast access)
  - 10KB of “constant” cache memory (writable by the CPU, fast, read-only access by the GPU)
- GPUs also have “global” memory which is the figure quoted by graphics card manufacturers (2GB, etc.)
  - accessed by the CPU across the PCI-Express bus
  - high latency, slow access by the GPU (but still up to 20x faster than CPU RAM access)

## General Purpose Programming for GPUs

- Threads are organised in groups of 32 called “warps”
- Threads also organised “Thread blocks” of several warps
- “Thread blocks” are organised in 1-d, 2-d, or 4-d grids

- “Thread blocks” are assigned to Streaming Multiprocessors
- A GPU cards is contains 1-50 Streaming Multiprocessor

## GPU Architecture



See [more](#)

## SIMT

- Threads within a block have access to the same “shared” memory
- Threads within a block can synchronise
- No communication or synchronisation primitives across blocks/SMs
  - can use atomic operations on variables in global memory (slow)
- This type of programming is a hybrid between threaded programming and SIMD and hence is called SIMT by Nvidia

## Executing code on the GPU

- Functions executed on the GPU and called from the CPU are called “kernels”
  - kernel execution is asynchronous to the CPU
  - CPU must call a function which blocks until the GPU has finished executing the current kernel before attempting to download results

- there is an implicit synchronisation barrier on the GPU at the start of each kernel
- Common programming pattern:
  - upload data from CPU RAM to GPU global memory (slow)
  - execute kernel(s) on GPU (fast)
  - synchronise
  - download results (slow)

## GPU-accelerating your code

- Four options to GPU accelerate code:
  - replace existing libraries with GPU-accelerated ones ([ArrayFire](#), [cuFFT](#), [cuBLAS](#), [Magma](#), ...)
  - use compiler directives to automatically generate GPU-accelerated portions of code (OpenMP 4.5, [OpenACC](#))
  - use CUDA::Thrust C++ template library to build your own kernels
  - write your own kernels in CUDA-C

## Streaming computations

### *Single Instruction, Multiple Data*

All threads in a warp perform the exact same hardware instruction, but on different data.

On thread 0:

```
a[0] = alpha * x[0] + 1;
__syncthreads();
y[0] -= a[31];
```

On thread 31:

```
a[31] = alpha * x[31] + 1;
__syncthreads();
y[31] -= a[0];
```

Is this single instruction?

```
a[thread_id] = ((thread_id < 16) ? 1: -1) + alpha * x[thread_id];
if(thread_id < 16)
    a[thread_id] = alpha * x[thread_id] + 1;
else
    a[thread_id] = alpha * x[thread_id] - 1;

__syncthreads();
y[thread_id] -= a[31 - thread_id];
```

### Single Instruction, *Multiple Contiguous Data*

Each thread in a warp should access a consecutive address in memory. Lets create a kernel for copying a vector using a single block consisting of a single warp (of 32 threads). Assume that the size of the vector is a multiple of 32. `threadIdx.x` is the thread id in Cuda, and automatically passed to each kernel.

The following is a jumble of too many expressions: put it back in order.

```
__global__ void copy(float *odata, const float *idata, int n)
{
    <!-- for (int j = 0; 32 * j < n; ++j) -->
    <!-- for (int j = 0; 32 * j < n; j += 32) -->
    <!-- for (int j = 0; j < n; ++j) -->
    for (int j = 0; j < n; j += 32)

        odata[j + threadIdx.x] = idata[j + threadIdx.x];
    <!-- odata[j + threadIdx.x * 32] = idata[j + threadIdx.x]; -->
    <!-- odata[j + threadIdx.x] = idata[j + threadIdx.x * 32]; -->
    <!-- odata[j + threadIdx.x * 32] = idata[j + threadIdx.x * 32]; -->
}
```

### The joys of indexing

Now imagine rewriting the same code with `n` by `m` by `p` blocks of threads, where each block is `u` by `v` by `w` threads. You have access to the size of the block `blockDim.x`, the index and size of the grid (of blocks) `blockIdx.(x, y, z)` and `gridDim.(x, y, z)`.

To limit memory transfers, each warp should read and write to contiguous arrays in memory.

See also: [shared memory](#), [bank conflicts](#)

## Structure of Arrays or Arrays of structures

```
struct Atom { int x, y, z; };  
std::vector<Atom> molecule;
```

or

```
struct Molecule {  
    std::vector<int> x;  
    std::vector<int> y;  
    std::vector<int> z;  
};
```

## Just USES

Using Somebody Else's Software is a good way to avoid becoming a GPU expert. But which? Check for usability and sustainability:

- Does it do what you need?
- What license is it under?
- Is it simply available on Github/BitBucker/Gitlab?
- Are there automatic tests?
- Is it an active development repo? (number of commits, date of last commit)
- How many people/labs/companies are committing to it? (check pull-requests)
- Is there a community of users? (check issues, wiki)
- Is there documentation?

## Broadcasting (Vectorization)

GPU require running the *same operations* over mutiple data (SIMD). Broadcasting transforms nested loops into a set of matrix or array operations amenable to SIMD, *and* likely to be already GPU-ized by external libraries.

Also useful for MATLAB, Python/numpy, etc...

Example:

$$G = \max(\|A - R_i\|)$$

for all  $i$ , where  $A$  and  $R_i$  are vectors

Naive solution with explicit loops and STL:



```
double naive(std::vector<std::array<double, 3>> const &Rs,
             std::array<double, 3> const &A) {
    double result(0);
    for(auto const &R : Rs) {
        auto norm = (A[0] - R[0]) * (A[0] - R[0]) + (A[1] - R[1]) * (A[1] - R[1])
                    + (A[2] - R[2]) * (A[2] - R[2]);
        if(result < norm)
            result = norm;
    }
    return std::sqrt(result);
}
```

## Broadcasting (Vectorization)

1. transform A from a vector to a matrix A3xn
2. compute pow2 = (R - A3xn)^2 elementwise
3. sum pow2 over columns
4. reduce final vector using max

```
double broadcasting(std::vector<std::array<double, 3>> const &Rs,
                   std::array<double, 3> const &A) {
    af::array A_3x1(A.size(), A.data());
    af::array gpuRs(A.size(), Rs.size(), Rs[0].data());

    auto const A_3xn = af::tile(A_3x1, 1, Rs.size());
    auto const norms = af::sum(af::pow(gpuRs - A_3xn, 2), 0);
    auto const result = af::max(norms);
    std::shared_ptr<double> const host_array(result.host<double>(),
                                             [](double *ptr) { delete[] ptr; });
    return std::sqrt(*host_array);
}
```

## A word on transfer rate

- Transfer to GPU takes time  $T_0$

```
af::array gpuRs(A.size(), Rs.size(), Rs[0].data());
af::array gpuA(A.size(), A.data());
```

- Compute takes  $\frac{C}{n}$ , n the number of GPU threads

```
auto const result
    = af::max(af::sum(af::pow(gpuRs - af::tile(gpuA, 1, Rs.size()), 2), 0));
```

- Transfer to CPU takes time  $T_1$

```
af::array gpuRs(A.size(), Rs.size(), Rs[0].data());
af::array gpuA(A.size(), A.data());
```

- Then, possibly  $T_0 + T_1 + C/n > C$

## Exercise: Broadcasting

$G = \sum_{i, j, k} \cos(K_k \cdot (R_i - R_j))$

Given

```
std::vector<std::array<double, 3>> const Rs = {...};
std::vector<std::array<double, 3>> const Ks = {...};
```

Write code to compute  $G$  (pseudo, or real code):

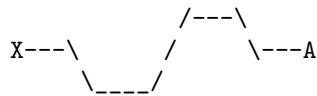
1. Create an  $R_s$  array of  $(3, 1, n)$  using `af::moddims (n = Rs.size())`
2. Create an  $R_s$  array of  $(3, n, 1)$  using `af::moddims`
3. Tile appropriately
4. Compute the dot product using `af::moddims, af::matmul, af::transpose`
5. sum over the cosine of the result

## Exercise: Line of Sight

Given an  $n$  by  $m$  matrix of altitudes, with  $n$  orientations and  $m$  distances:

```
std::vector<double> altitudes(nOrientations * mDistances) = { ... };
```

Compute the whether any point  $i, j$  is in sight. A point is in sight if, for that orientation, all previous angles between X-horizon and X-point are smaller



Angles are computed using the formula `atan(z_i / (stepsize * i))`.

`af::scanf` might come in handy: - given `[a0, a1, a2, ..., aN]` - it computes `[0, a0, a0 + a1, a0 + a1 + a2, ..., a0 + ... + aN]` - leading zero is removed in exclusive scans

This function looks intrinsically difficult to parallelize, but there are well known solutions. It is useful in many fields.

# Chapter 18

## Installation

### Installation

#### Introduction

This document contains instructions for installation of the packages we'll be using during the course. You will be following the training on your own machines, so please complete these instructions. You don't need everything before lecture one: you can refer to these instructions as the course proceeds.

#### What you'll need by the end of the course

- [CMake](#) build tool.
- [Catch](#) unit testing framework.
- [Git](#) and the [GitHub](#) website.
- Some C++ libraries: [The Boost C++ library package](#), and [Eigen linear algebra library](#).
- A code editor of your choice.

#### Eduroam

We will be using UCL's [eduroam](#) service to connect to the internet for this work. So you should ensure you have eduroam installed and working.

## Linux

### Git

If git is not already available on your machine you can try to install it via your distribution package manager (e.g. `apt-get` or `yum`).

On ubuntu or other Debian systems:

```
sudo apt-get install git
```

On RedHat based systems:

```
sudo yum install git
```

### CMake

Again, install the appropriate package with `apt-get` or `yum` (`cmake`). Minimum version 3.5.

### Editor and shell

Many different text editors suitable for programming are available. If you don't already have a favourite, you could look at [Kate](#).

Regardless of which editor you have chosen you should configure git to use it. Executing something like this in a terminal should work:

```
git config --global core.editor NameofYourEditorHere
```

The default shell is usually bash but if not you can get to bash by opening a terminal and typing `bash`.

## Windows

### Compiler / Editor

For code editing we recommend [Visual Studio](#). Please download and install Visual Studio 2017 or 2015. You are advised not to have multiple versions of Visual Studio installed simultaneously.

In addition, it is convenient to have a simple editor that integrates with Git, for the purpose of writing Git commit messages. For this we recommend [Notepad++](#) on windows.

Please download and install Notepad++ and integrate with Git, as shown below.

## Git

Install [Git for Windows](#).

During the installation, you can select “Use Notepad++ as Git’s default editor” if you installed Notepad++ above. Make sure you tick “Use Git from the Windows Command Prompt” so other Unix tools can find git. The defaults should be suitable for other options.

Then install the [GitHub for Windows client](#).

## CMake

Install [cmake](#). Minimum version 3.5.

And choose to add it to the path for all users if so prompted. (You may need to log out and log back in again before this takes effect!)

## Locating your install

Now, we need to find out where Git and Notepad++ have been installed, this will be either in `C:\Program Files (x86)` or in `C:\Program Files`. The former is the norm on more modern versions of windows. If you have the older version, replace `Program\ Files\ \ (x86\)` with `Program\ Files` in the instructions below.

## Telling Shell where to find the tools

We need to tell the new shell installed by MinGW where Notepad++ is.

To do this, use NotePad++ to edit the file at `C:\MinGW\mysys\1.0\etc\profile` and toward the end, above the line `alias clear=cls` add the following:

```
# Path settings from SoftwareCarpentry
export PATH=$PATH:/c/Program\ Files\ \ (x86\)/Notepad++
# End of Software carpentry settings
```

## Finding your terminal

Check this works by opening MinGW shell, with the start menu (Start->All programs->MinGW->MinGW Shell). This should open a *terminal* window, where commands can be typed in directly.

On windows 8 and 10, there may be no app for MinGW. In that case, open the run app and type in

```
C:\MinGW\msys\1.0\msys.bat
```

You can also create a shortcut to this file on your Desktop for quicker access.

## Checking which tools you have

Once you have a terminal open, type

```
which notepad++
```

which should produce readout similar to

```
/c/Program Files (x86)/Notepad++/notepad++.exe
```

Also try:

```
which git
```

which should produce

```
/c/Program Files/Git/cmd/git.exe
```

The `which` command is used to figure out where a given program is located on disk.

## Tell Git about your editor

If you didn't do this as part of the Git install, you now need to update the default editor used by Git.

```
git config --global core.editor "'C:/Program Files (x86)/Notepad++  
/notepad++.exe' -multiInst -nosession -noPlugin"
```

Note that it is not obvious how to copy and paste text in a Windows terminal including Git Bash. Copy and paste can be found by right clicking on the top bar of the window and selecting the commands from the drop down menu (in a sub menu).

You should now have a working version of git and notepad++, accessible from your shell.

## Mac

### XCode and command line tools

Install [XCode](#) using the Mac app store.

Then, go to Xcode...Preferences...Downloads... and install the command line tools option

### A note on Homebrew Versus MacPorts

If you already have either Homebrew or MacPorts installed, do NOT install both. This usually creates a confusing mess for students. If you have either installed already for other courses, then as long as the minimum compiler versions, cmake versions and git versions seem to work, there is no reason to swap. If you have neither installed, we prefer Homebrew, and the notes below are for Homebrew.

### Installing Homebrew

Install Homebrew via typing this at a terminal:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/mxcl/homebrew/go)"
```

### Git

Once Homebrew is installed:

```
brew install git
```

Then install the [GitHub for Mac client](#). (If you have problems with older versions of OSX, it's safe to skip this.)

### CMake

Just do

```
brew install cmake
```

Minimum version 3.5.



## Editor and shell

The default text editor on OS X *textedit* should be sufficient for our use. Alternatively choose from a [list](#) of other good editors.

To setup git to use *textedit* executing the following in a terminal should do.

```
git config --global core.editor  
    /Applications/TextEdit.app/Contents/MacOS/TextEdit
```

The default terminal on OSX should also be sufficient. If you want a more advanced terminal [iTerm2](#) is an alternative.

## Boost

### Linux Users

Install the appropriate package with apt-get or yum, for example:

```
sudo apt-get install boost
```

On Ubuntu 13.04 boost is spilt into multiple packages. The package needed here is “libboost1.53-dev” and to install it you should do:

```
sudo apt-get install libboost1.53-dev
```

Check that your package manager is delivering at least version 1.53, if you have an earlier version, you will need to download from source, following the windows instructions below.

### Mac Users

```
brew install boost
```

### Windows Users

You will need to download and install boost manually.

Create an appropriate folder, somewhere near your working code for the course.

For example, if you are working in `/home/myusername/devel/cppcourse/rsd-cppcourse/reactor` you could do:

```
mkdir -p ~/devel/libraries/boost
cd ~/devel/libraries/boost
```

Now, download and unzip boost, which will take quite a while:

```
curl -L http://sourceforge.net/projects/boost/files/...
...boost/1.54.0/boost_1_54_0.zip/download > boost.zip
unzip boost.zip
```

(All one line, not literally two lines with dots!)

## CMake and Boost

If we download from source, before we next build, we will need to tell our shell where the boost library can be found

```
cd build
export CMAKE_INCLUDE_PATH=/home/myusername/devel/libraries/boost_1_54_0
cmake ..
make
ctest
```

... or your equivalent.

## Docker

### Optional - Ask Lecturer

Mac OS/X:

```
> brew install Caskroom/cask/virtualbox
> brew install docker-machine
> brew install docker
```

or

```
> brew install Caskroom/cask/docker-toolbox
```

Linux: - docker <https://www.docker.com/community-edition> - docker machine  
(optional): <https://docs.docker.com/machine/install-machine/>

Windows or Mac OS/X: - <https://www.docker.com/products/docker-toolbox>

## Chapter 19

# Assessment

### Assessment Structure

- 3 hour exam
  - Past papers from 2015 and later are available, but syllabus does evolve
  - Past marks schemes and standard solutions are not available
- 2 pieces coursework - 40 hours each
  - 1 due 3rd week March
  - 1 due just after Easter