Advanced Research Computing

MPhil in Data Intensive Science - Trial Year

Dr James Fergusson and Dr Amelia Drew

Prerequisites

- **Research Computing Topics**
  - Linux and Bash
  - Python
  - Best Practices in Design and Development
  - Performance (profiling, optimisation)
  - Multi-language Programming
  - Public Release (including Docker)

Course Content

- **Advanced Programming**
  - Interpreters and Compilers
  - Review of C++
  - Overview of Fortran
  - Compilation (Makefiles, build automation tools)
  - Debugging
  - Profiling
  - Running on Supercomputers
  - Advanced Build and CI Management
  - Co-ordinating Large Software Projects

Course Content

- **Parallelisation**
  - Supercomputer Architectures and Overview of Parallelisation
  - Predictive Performance Models - Roofline, Threads and Processes
  - Vectorisation
  - OpenMP and MPI
  - Debugging
  - Parallel I/O
  - GPUs and Heterogeneous Programming (Introduction to DPC++)
- **Visualisation**
  - Advanced Visualisation Approaches
  - Paraview

Course Content

LUMI Supercomputer, Finland
Largest in Europe (TOP500)

Course Content

- If you would like to test out the material on your laptop, please do the following:
  - Download VSCode (https://code.visualstudio.com/)
  - Download and configure Docker
  - Pull Docker file from my GitHub page (https://github.com/amelialdrew/advanced_research_computing_docker)
  - Build Docker image in VSCode and run

Advanced Programming

Interpreters and Compilers

Advanced Programming

Interpreters and Compilers

- *Example:* we would like to write a computer program to add two numbers together

- We first pick a programming language that suits our needs best eg. C++, Python

python Copy code

```python
# Ask the user for the first number
num1 = input("Enter the first number: ")

# Ask the user for the second number
num2 = input("Enter the second number: ")

# Add the two numbers together
result = float(num1) + float(num2)

# Print the result
print("The sum of", num1, "and", num2, "is", result)
```

Advanced Programming

Interpreters and Compilers

- Computers cannot understand these languages directly - they can only execute *machine code*
- Each computer processor (different CPUs, GPUs etc.) has its own machine language
- The processor requires certain instructions which tell it eg. to find an address in memory and perform an operation on it
- Looks something like:

```
01001000 01100101 01101100 01101100 01101111 00100001
```

Advanced Programming

Interpreters and Compilers

- It would be very difficult to write this program in machine (eg. binary) code…

- To be able to program with any efficiency, we need a way to translate between the human-readable code, and the binary machine code

- This is primarily performed in two ways:

  - Use an **interpreter** - these go through code line by line and interpret each command for the processor so that it can be executed

  - Use a **compiler** - these translate code, check for errors and create an ‘executable,’ which can then be run by the processor. This is usually the fastest method

Advanced Programming

Interpreters and Compilers

- So far, you are familiar with Python (taught in the Research Computing module)
  - It is recommended that you use a *virtual environment* for Python projects
  - This is a *folder structure* that allows you to run a Python environment in a way that:
    - Any packages installed in your operating system's (OS's) global Python installation are not mixed up/overwritten
    - Updating your OS does not affect running your scripts
    - You can easily switch between Python versions/dependencies (eg. usually you cannot have two versions of the same library if you have one place to install packages)

Advanced Programming

Interpreters and Compilers

Continued..

- Recommended reading: https://realpython.com/python-virtual-environments-a-primer
- In our examples, we use a Docker image of Ubuntu with Python installed via pip - for more flexibility with versions, we would ideally use a virtual environment *within* the Docker container

```
# Get the base Ubuntu image from Docker Hub
FROM ubuntu:latest

# Update apps on the base image
RUN apt-get -y update && apt-get install -y

# Install Python
RUN apt-get -y install python3-pip

# Install iPython
RUN pip3 install ipython
```

Advanced Programming

Interpreters and Compilers

- Python is a high-level language that can be run directly using an **interpreter** (CPython)
- If using a virtual environment, the interpreter will be stored in `bin/`
- The interpreter performs certain steps:
  - Checks code for syntax and errors
  - Converts Python code into ‘byte code’
  - Initialises a *virtual machine* that converts byte code into binary code and executes
- For a Python script `test.py`, we run using the command `python3 test.py` (where `python3` is the command to call the interpreter)

Advanced Programming

Interpreters and Compilers

Practical Task 1:

- Run the test code from previous slide (in Docker container)

Practical Task 2:

- Start `iPython` on your laptop with the docker image - this invokes the interpreter so you can interact with it
- Try out some simple command eg. `1+2`, return
- The interactive interpreter executes the statement

Advanced Programming

Interpreters and Compilers

- C++ is another high-level language that must be **compiled** in order to run
- Like an interpreter, a **compiler** takes human-written code and converts it to be machine-readable
- The main difference to interpreters is that compilation is performed *in advance* of running the code
- We must install a **C++ compiler** on our machine (sort of the equivalent of installing Python)
- This can be done eg. on Mac, via Homebrew

Advanced Programming

Interpreters and Compilers

- Like an interpreter, the compiler performs certain steps:
  - Preprocesses source files, eg.
    - Inserts content from *header files* added via `#include`. These are replaced with entire content of included file.
    - Any `false` conditional compilation sections removed (`#ifdef, #endif`)
    - This creates a *translation unit*

Advanced Programming

Interpreters and Compilers

Continued..

- Compiles *object file* from the preprocessed translation unit
    - This is a machine code file
    - References to functions, symbols used are not yet defined - they have no memory address
- Object files are then *linked* into an executable
    - We must link any object file that specifies eg. a function, to the file in which the function is defined

- Unless you use specific compilation flags, all of these steps will be done automatically
- However, it is useful to know the steps to be able to diagnose compilation errors

Advanced Programming

Interpreters and Compilers

Example:

- We write some source code for a ‘Hello World’ program, `hello_world.cpp`

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

- There is no interpreter to run this directly, so we must compile first

Advanced Programming

Interpreters and Compilers

- We install a C++ compiler called `g++` and run the command `g++ hello_world.cpp -o hello_world`
- This will give us an ‘executable’ file called `hello_world` - without the `-o` option, the executable is given some default name `a.out`
- We can then run this executable using the command `./hello_world`

Advanced Programming

Interpreters and Compilers

Practical Task:

- Compile and run the `hello_world.cpp` in the Docker container
- Create and examine the *translation unit,* using `g++ -E hello_world.cpp -o whatever_preproc_name`
  - You can count the number of lines using `wc -l whatever_preproc_name`
- Create and examine the *object file,* using `g++ -c hello_world.cpp` (will create `hello_world.o`)
  - You can use the command `nm hello_world.o` to see which symbols are specified (they are not yet linked)

Advanced Programming

Interpreters and Compilers

- To recap - what has actually happened here?

    1. Compiler replaces `#include <iostream>` with entire content of included file
    2. Source file compiled into a translation unit, then an ‘object’ file
    3. Object file has been linked with the file that contains the `std::cout` and `std::endl` functions
    4. The code has been run

- We now have an idea of how interpreters and compilers work with different programming languages

Advanced Programming

Review of C++

Advanced Programming

Review of C++

Some recommended books:

- *Accelerated C++*, A. Koenig and B. E. Moo (slightly outdated but a good approach)

- Books by Bjarne Stroustrup, e.g. *Principles and Practice Using C++* and *The C++ Programming Language 4th Edition* (includes C++11 concepts)

- *Moving Planets Around - An Introduction to N-Body Simulations Applied to Exoplanetary Systems*, J. Roa et al. (how to write an N-body code)

Websites:

- http://en.cppreference.com

- ChatGPT or https://stackoverflow.com/ (general questions)

Advanced Programming

Review of C++

Aim:

- To refresh your knowledge of, or to learn, C++
- Primarily cover basic concepts, with some more advanced
- Led by examples and by what you are likely to need to do as a researcher/developer:
  - Analysing data
  - Using other people's code

Advanced Programming

Review of C++

- C++ is a language developed in 1979 by Bjarne Stroustrup
- The first C++ ‘standard’ was developed in 1998
- It is an *object-oriented* language (we will unpack this later)
- Changes/additions to the standard are made as new C++ versions appear (C++98, C++03, C++11, C++14, C++17, C++20)
- The current most up-to-date version is C++20, but in practice, actively developed codes are probably using up to C++14/17
- New versions usually provide new advanced features

Advanced Programming

Review of C++

Advantages:

- C++ can be optimised well by modern compilers - it is **fast**
- Object-oriented approach means code is (in theory) easy to read
- The basic features are relatively intuitive

Disadvantages:

- In large codes, object-orientation can mean you must go down ‘rabbit holes’ to work out how the code works
- Can be complex if you want to implement advanced features

Advanced Programming

Review of C++

- We have written some code to output the text, ‘`Hello, world!`’, to the terminal (using ChatGPT)
- What does this program demonstrate?

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- C++ is an *object-oriented programming language* - it is organised around ‘objects’ (in contrast to eg. functional programming)

- Each object will have a defined *type* - these can be already built in (`int`, `double` etc.) or user-defined (`class`)

- The type defines permitted operations and a ‘semantic meaning’ to the object

- You will often see ‘an object is an *instance* of a *class’* - but also true for built-in types

- Objects enable code *abstraction* - unnecessary implementation code can be hidden

- Other codes that mainly use object orientation are Python, Java

Advanced Programming

Review of C++

- In `hello_world.cpp` example, `cout` is an *object* of the *class* `ostream` - it is an object of *type* `ostream`

- Type `ostream` is defined by the external library, `<iostream>`
  - https://cplusplus.com/reference/iostream/

- The `0` returned by the `main()` function is also an object of type `int`

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- The most used built-in C++ data types are:
  - `int` - integer (from -2147483648 to 2147483647, 2-4 bytes memory)
  - `float` - floating point (decimals and exponentials, 4 bytes memory)
  - `double` - floating point with double precision (8 bytes)
  - `bool` - boolean (`true` or `false`, used for conditional statements, 1 byte)
  - `void` - valueless - only used for functions that do not return any data
  - `char` - character (1 byte)
  - [`wchar_t` - wide character (character that takes up more than one byte)]

Advanced Programming

Review of C++

In example program:

- `int` (integer) is a built-in type
- `'Hello, world!'` is made up of several `char`

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- Every C++ program must have a `main()` function that returns `0` when successful
- This is what is called (by the C runtime library initialisation code) when you run the code
- We note the `#include <iostream>` - this is an external library

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- In the example, `std` refers to the ‘standard *namespace*’
- Namespaces are collections of related names
  - `std` is used by standard libraries to contain all the names that it defines, eg. `std::cout`
  - If we decided to use a different namespace, we could have another function eg. `otherlib::cout`

Advanced Programming

Review of C++

- Now that we know how to output text to the terminal, we might like to write a program for a scientific purpose
- Eg. We would like to analyse some data in a text file, `data.csv`

Practical Task: Use ChatGPT to create a C++ program that calculates the average value of the second column of a csv file (comma separated values)

Advanced Programming

Review of C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <string>

double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}

int main() {
    std::ifstream file("data.csv"); // Replace "data.csv" with the path to

    if (!file) {
        std::cerr << "Failed to open the file." << std::endl;
        return 1;
    }
```

```
    std::vector<double> columnData;
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream ss(line);
        std::string cell;

        // Split the line into cells using comma as the delimiter
        std::getline(ss, cell, ','); // Skip the first column
        std::getline(ss, cell, ','); // Read the second column

        // Convert the cell value to double and store it in the vector
        double value;
        std::istringstream(cell) >> value;
        columnData.push_back(value);
    }

    file.close();

    if (columnData.empty()) {
        std::cerr << "No data found in the second column." << std::endl;
        return 1;
    }

    double average = calculateAverage(columnData);
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

Advanced Programming

Review of C++

- We have defined a new function, `calculateAverage()`
  - What type of output will this function give?
  - This function is called in `main()` and requires a `vector` input

```cpp
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

- This introduces the concept of a *derived data type* - a type created by combining built-in data types

  - `function` - a code segment for a specific purpose, saves us from having to duplicate code

eg. `sometypea calculateAverage(sometypeb parameters) {`

`//Content of file`

`}` (types can be the same, but don't have to be)

called by

`calculateAverage(myparams)`

Advanced Programming

Review of C++

- `array`
  - Set of elements of the same type kept in memory in a continuous way
  - Allows us to store data with a single variable name, in sequence
  - Need to know the number of elements before you define

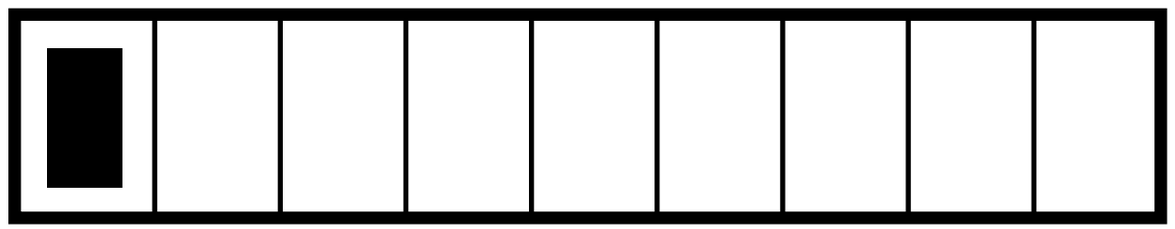
eg. `int time_in_seconds[5] = {0,1,2,3,4};`

Advanced Programming

Review of C++

- `pointer *`
  - a variable that holds the address in memory that another variable is stored
- `reference &` (address-of)

eg.
```
int  black_box;
int* pointer_to_box = &black_box;
```
(the value is `a`)



Advanced Programming

Review of C++

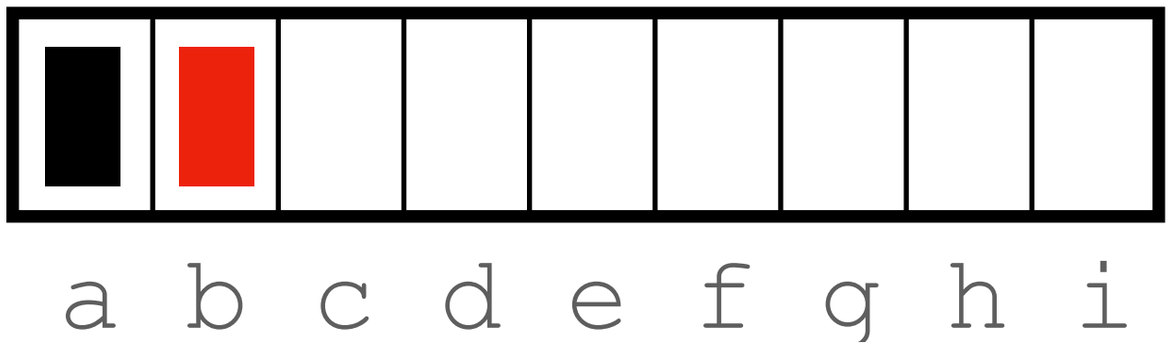
Note, `*` can also be used to *dereference* pointers

eg. `int red_box = *pointer_to_box;`

- Here, `red_box` is an *integer* set equal to ‘the variable pointed to by `pointer_to_box`’

- The new variable will be stored in a new memory address, `b`

- When passed as a function parameter, this is known as ‘passing by value’

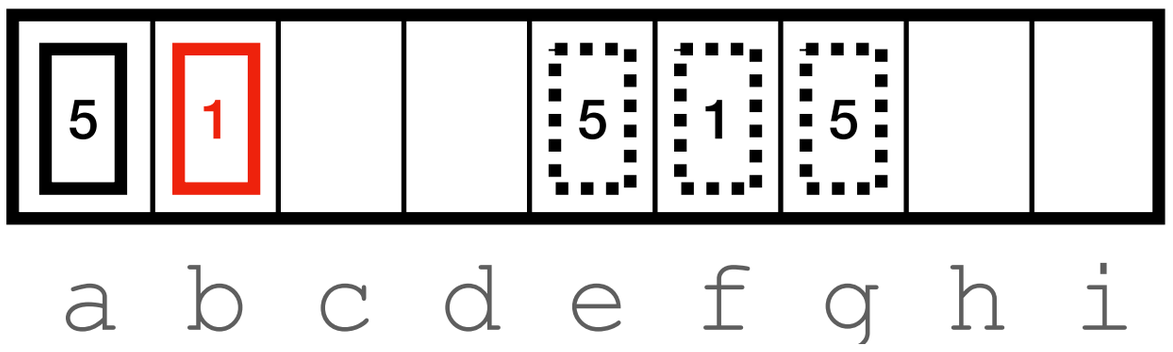


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int x, int y) {
    int temp = x; //eg. stored in location e
    x = y;        //eg. stored in location f
    y = temp;     //eg. stored in location g
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```



Advanced Programming

Review of C++

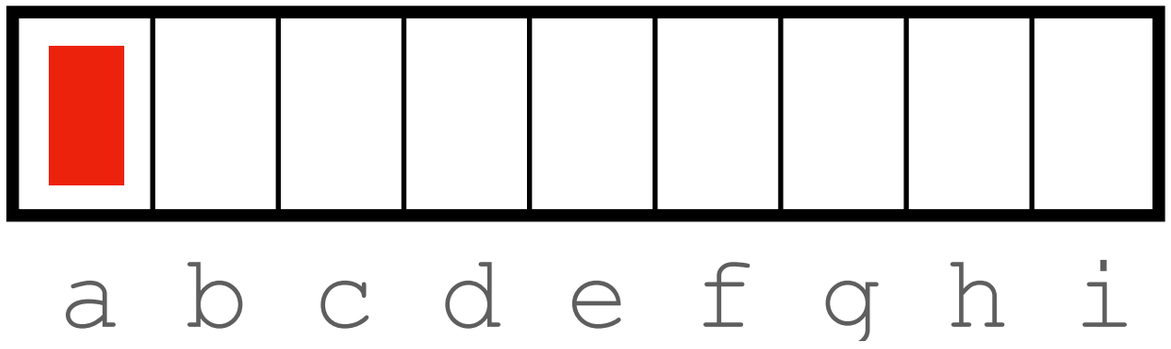
Additionally, `&` can be used to pass references to variables

eg. `int& red_box = black_box;`

- We read this as, ‘the address in memory that points to `red box` is equal to the address in memory that points to `black_box`’ (which we saw earlier was `a`)

- The variable stored in the location `a` will be effectively aliased by a new variable, pointed to by same memory location

- When passed as a function parameter, this is known as ‘passing by reference’

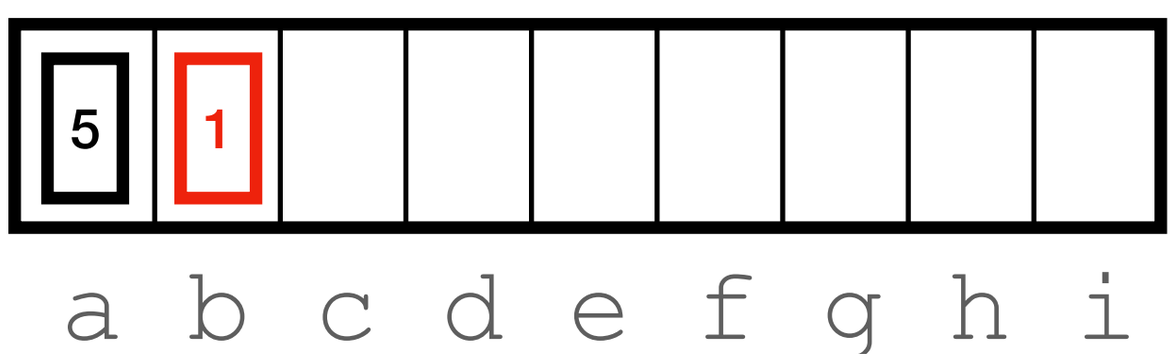


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```

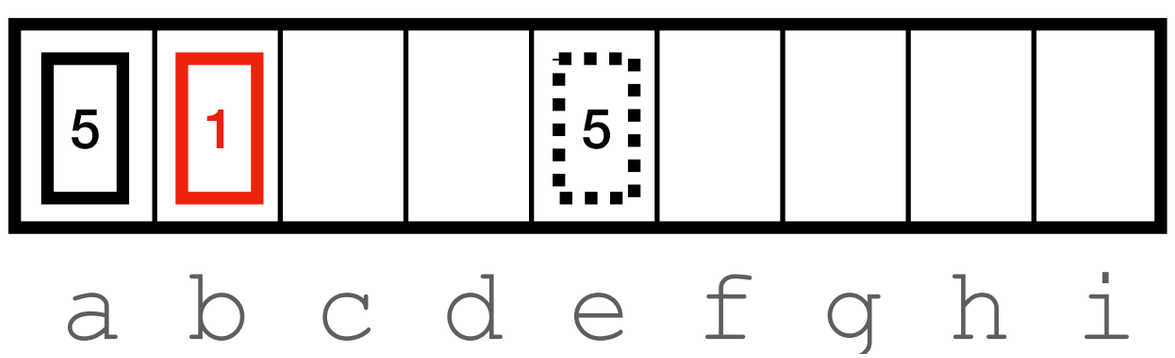


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;              //Read x, y as 'the variable
    y = temp;             pointed to by this address'
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```

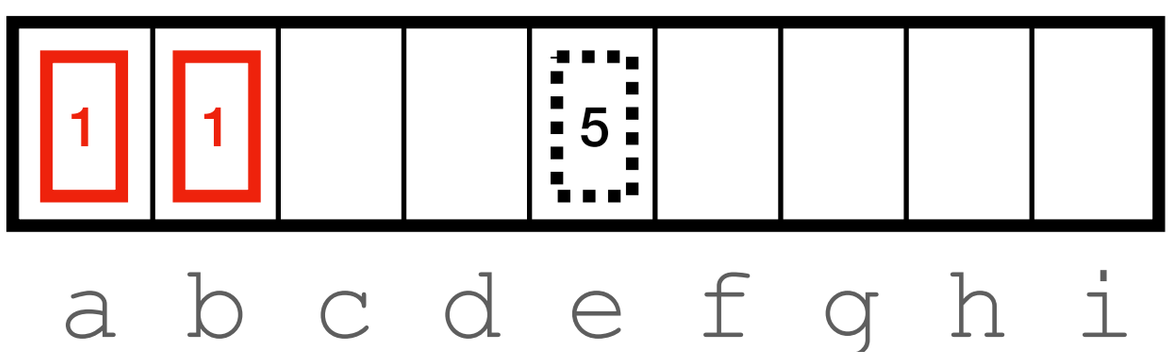


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;              //Read x, y as 'the variable
    y = temp;             pointed to by this address'
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```

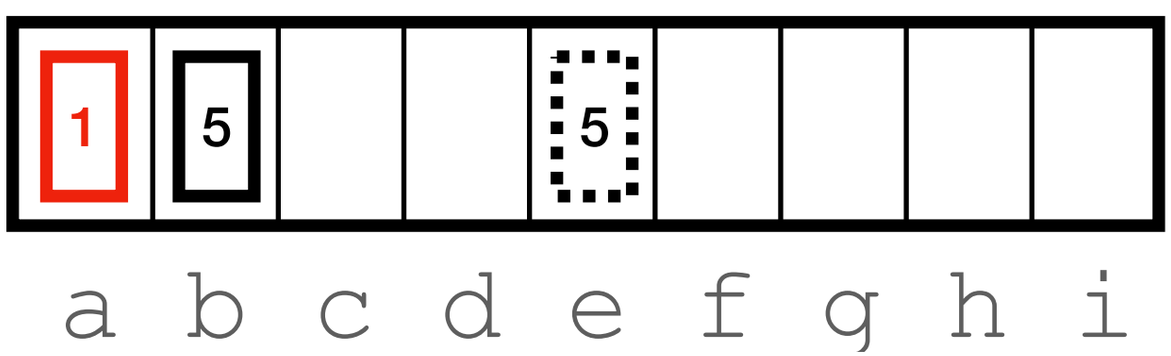


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;              //Read x, y as 'the variable
    y = temp;             pointed to by this address'
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```



Advanced Programming

Review of C++

- Pointers and references in particular can take a lot of repetition to get your head around

- Advise keeping good notes that you can refer to while you are getting familiar with the concept (and for when you forget)

- Passing by reference vs by value becomes *very important* with large datasets

- If you have gigabytes or terabytes of data eg. on a time-evolving 3D simulation grid, you do NOT want to be copying unnecessarily

- Passing by reference is the most memory- and time-efficient, especially for large simulations

Advanced Programming

Review of C++

- In our example, we see that the function `calculateAverage()` is passed a `vector` *by reference*

- This means that no copies of `values` will be made in memory

- We have also used `const`, which additionally ensures that the values will not be changed

```
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

- `<vector>` is included as an external library (https://en.cppreference.com/w/cpp/container/vector)
- This is a *class template* (see later) for sequence containers that allow for *dynamic size arrays*
- Vectors are useful if you want to store variables, but you do not know how many there will be before you run the code
- Eg. output from a simulation that depends on runtime parameters
- Some particularly useful functions are `size` (in example, this measures the size of the vector), `push_back` (add an element to the end of the vector), `insert`

Advanced Programming

Review of C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <string>

double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}

int main() {
    std::ifstream file("data.csv"); // Replace "data.csv" with the path to

    if (!file) {
        std::cerr << "Failed to open the file." << std::endl;
        return 1;
    }
```

```
    std::vector<double> columnData;
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream ss(line);
        std::string cell;

        // Split the line into cells using comma as the delimiter
        std::getline(ss, cell, ','); // Skip the first column
        std::getline(ss, cell, ','); // Read the second column

        // Convert the cell value to double and store it in the vector
        double value;
        std::istringstream(cell) >> value;
        columnData.push_back(value);
    }

    file.close();

    if (columnData.empty()) {
        std::cerr << "No data found in the second column." << std::endl;
        return 1;
    }

    double average = calculateAverage(columnData);
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

Advanced Programming

Review of C++

- We also see a `for` loop
- This is a type of *control flow statement*
- This allows us to iterate a particular operation using an index

```cpp
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

- We define an index `i` (for the purposes of this example, read `size_t` as `int`)

- The index begins at `0`, and the second argument provides the total number of points - here, the loop will iterate from `0` to `values.size() - 1`

- `i++` indicates at the end of each time the statement in `{ }` is executed, `i` is increased by 1, i.e. `i = i+1`

- The statement in `{ }` will be executed until the max value of `i` is reached, then the loop will be exited

```
for (size_t i = 0; i < values.size(); i++) {
    sum += values[i];
}
```

Advanced Programming

Review of C++

- A more concise method is to use a range-based for loop
- This functionality has been available since C++11
- Here, `double value : values` indicates that the `for` loop should be executed for every `value` within the `values` vector

```
for (double value : values) {
    sum += value;
}
```

Advanced Programming

Review of C++

- We can now determine exactly what this function does
  - It is passed the reference to a vector of values
  - It defines a `double sum = 0.0`
  - It iterates through vector, adding the values together
  - The average value is returned

```
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <string>

double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}

int main() {
    std::ifstream file("data.csv"); // Replace "data.csv" with the path to

    if (!file) {
        std::cerr << "Failed to open the file." << std::endl;
        return 1;
    }
```

Advanced Programming

Review of C++

- The file is read in the `main()` function
- First, `std::ifstream file("data.csv")` opens file (input file stream)
- Second part is an error message - these are very important for smooth debugging

```cpp
if (!file) {
    std::cerr << "Failed to open the file." << std::endl;
    return 1;
}
```

Advanced Programming

Review of C++

- `if` is a conditional statement
  - Checks whether a given statement is true
  - If yes, executes the commands in `{ }`
- Often combined with an `else` statement with what to do if the statement is *not* true
- `if (!file)` checks if the file was *not* found, `{ }` outputs an error message

```
    std::vector<double> columnData;
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream ss(line);
        std::string cell;

        // Split the line into cells using comma as the delimiter
        std::getline(ss, cell, ','); // Skip the first column
        std::getline(ss, cell, ','); // Read the second column

        // Convert the cell value to double and store it in the vector
        double value;
        std::istringstream(cell) >> value;
        columnData.push_back(value);
    }

    file.close();

    if (columnData.empty()) {
        std::cerr << "No data found in the second column." << std::endl;
        return 1;
    }

    double average = calculateAverage(columnData);
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

Advanced Programming

Review of C++

- We have another type of control flow statement, a `while` loop
- These loop through a block of code contained in `{ }` as long as the condition specified is `true`

```
while (std::getline(file, line)) {
```

- Here, we use the `std::getline` function (https://cplusplus.com/reference/string/string/getline/) to extract characters from `file` and store them as a string in `line`
- `while` loop is executed while there is still a row of data in the file

Advanced Programming

Review of C++

- `std::istringstream` reads a string to a stream
- `std::getline` is also used with a comma delimiter to read the value of the second column in each row
- Values are appended onto the vector `columnData` using `push_back`
- File is closed with `file.close()`

Practical Task: Compile and run the `data_reader.cpp` example

Advanced Programming

Review of C++

- We have now been introduced to some key C++ concepts
  - Defining additional functions
  - Passing by value/reference
  - Control statements: `if, else, for, while`
  - Opening and reading data files
  - Manipulating strings and streams
  - Using vectors
  - Writing error messages

Advanced Programming

Review of C++

- What would we do to `data_reader.cpp` if we wanted to calculate the same average, but for a different data file?
- Using what we currently know, we would either have to:
  - Write all the same code out again with a different file name
  - Recompile the code with a different file name and run again
  - Change the name of our data file to fit with the name in the code
- Might work for a few files, but not straightforward with many files
- A more efficient option is to use a more advanced, central C++ concept: a *class* (a user-defined data type)
- We can define a class that takes a file name as a *parameter*

Advanced Programming

Review of C++

- Looking at the new `main()` function in `data_reader_class.cpp`, we see an unfamiliar data type - `CSVReader`

- This is a user-defined *class*

```cpp
int main() {
    std::string filename = "data.csv"; // Replace "data.csv" with the path t

    CSVReader reader(filename);
    if (!reader.readData()) {
        return 1;
    }

    double average = reader.calculateAverage();
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

```cpp
class CSVReader {
private:
    std::string filename;
    std::vector<double> columnData;

public:
    CSVReader(const std::string& filename) : filename(filename) {}

    bool readData() {
        std::ifstream file(filename);
        if (!file) {
            std::cerr << "Failed to open the file." << std::endl;
            return false;
        }

        std::string line;
        while (std::getline(file, line)) {
            std::istringstream ss(line);
            std::string cell;

            // Skip the first column
            if (std::getline(ss, cell, ',') && std::getline(ss, cell, ',')) {
                double value;
                std::istringstream(cell) >> value;
                columnData.push_back(value);
            }
        }

        file.close();

        if (columnData.empty()) {
            std::cerr << "No data found in the second column." << std::endl;
            return false;
        }

        return true;
```

Advanced Programming

Review of C++

- First, focus on the section underneath keyword `public`
- This defines functions that we can access from outside the class - here, we have `readData()` and `calculateAverage()`
- These are called in `main()` by:

```
CSVReader reader(filename);
if (!reader.readData()) {
    return 1;
}

double average = reader.calculateAverage();
```

- Any function defined within a class is called a *member function*

Advanced Programming

Review of C++

- At the top of the section `public`, we also have

```
public:
    CSVReader(const std::string& filename) : filename(filename) {}
```

- This is called the *constructor*
- The name of the constructor, `CSVReader`, must match the name of the class itself
- `(const std::string& filename)` specifies the parameters that the constructor takes

Advanced Programming

Review of C++

- At the top of the section `public`, we also have

```
public:
    CSVReader(const std::string& filename) : filename(filename) {}
```

- This is called the *constructor*
  - The name of the constructor, `CSVReader`, must match the name of the class itself
  - `(const std::string& filename)` specifies the parameters that the constructor takes
  - The member variables must be initialised - this is done here by `: filename(filename)`
  - Eg. Here, if we did not initialise, `reader.readData()` would return an error

Advanced Programming

Review of C++

- In the section underneath keyword `private`, we see the declaration of two variables

```
class CSVReader {
private:
    std::string filename;
    std::vector<double> columnData;
```

- These are used by the functions in `public`, but *cannot* be accessed outside the class - eg. `reader.columnData` is not valid

- Any variable defined within a class is called a *member variable*

Advanced Programming

Review of C++

- Another useful property of classes is *inheritance*
- This allows a class to *inherit* (access) public members from a base class

```
class CSVReader {
public:
    // Public members and functions of the CSVReader
};

class HigherLevelReader : public CSVReader {
    // Additional members and functions specific to t
};

int main() {
    HigherLevelReader obj;

    // Accessing public members from the base class
    obj.somePublicMemberOfCSVReader;
    obj.somePublicFunctionOfCSVReader();

    return 0;
}
```

Advanced Programming

Review of C++

- One specific kind of class is a `friend class`
- These can access *private and protected* members of other classes in which it is declared as a friend
- This is not possible via inheritance

Advanced Programming

Review of C++

```cpp
class CSVReader {
private:
    int privateData;

public:
    CSVReader(int data) : privateData(data) {}

    friend class CSVReaderFriend;
};

class CSVReaderFriend {
public:
    void accessPrivateData(const CSVReader& obj) {
        int data = obj.privateData;  // Friend class
        // Perform operations using private data
    }
};

int main() {
    CSVReader obj(42);
    CSVReaderFriend friendObj;

    friendObj.accessPrivateData(obj);  // Friend clas

    return 0;
}
```

Advanced Programming

Review of C++

- Note: there is another user-defined data type - `struct`
- All members are `public` by default, vs `private` by default with classes
- They are defined similarly to classes, eg.

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

// Define a struct for representing a student
struct Student {
    int id;
    string name;
    int age;
};
```

- In practise, classes are used much more often

Advanced Programming

Review of C++

- Another key concept in C++ is *templating*
- This allows us to define a function *without having to specify the type of the variables it acts on*
- Some example syntax:

```cpp
// Templated function that swaps two values of any type
template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

- Note that this is the same syntax used for `vector` - these are class templates

Advanced Programming

Review of C++

Some miscellaneous advanced features:

- When defining the `main()` function, we will often see the definition

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

- This is the method used to pass *command line arguments* to `main()` - whatever you put on the command line will become an input
- `argc` - argument count (number of strings pointed to by `argv`)
- `argv` - argument vector (these names are convention, can be given other names)
- Run programs like this using eg. `./Hello_World param1 param2`
- `::` denotes the *scope* (as we have seen before, in i.e. the namespace)
- This also applies to *classes*, eg. `classname::functionname`

Advanced Programming

Review of C++

Now that we understand a lot of C++ key features, how do we navigate a large, complex research code?

- Will use the example of adaptive mesh refinement numerical relativity code, GRChombo (https://github.com/GRChombo/GRChombo)

- Other codes large codes include eg. CosmoLattice, SOFTSUSY, N-Body code GADGET

- In short:

  - Read any documentation

  - Make use of example code

  - Find `main()` and follow functions from there

Advanced Programming

Debugging

Advanced Programming

Debugging: Types of Bugs

- Debugging is the process of identifying and fixing errors in code
- Almost impossible to avoid, so must know how to identify and fix them
- Common types of error in C++ code are:
  - Compile-time errors
    - Syntax error (eg. missing `;` or `)`) or type error (eg. `int x = “hello”;`)
    - These are usually flagged during compilation
    - The program will not compile until the error is fixed
    - Usually easy to identify

Advanced Programming

Debugging: Types of Bugs

- Linker error
  - Occurs during the ‘linking’ stage of compilation, program will not compile
  - Can occur if eg.
    - Included header file is not found
    - `main()` is incorrectly written as eg. `Main()`
    - A function or class is declared, but never defined, eg.

```
int func();

int main() { int x = func(); }
```

Advanced Programming

Debugging: Types of Bugs

- Runtime error
  - The code will compile and run, but will crash or give an unexpected result, such as:
    - Segmentation fault - the program tries to access a memory location that is not allowed, eg. accessing `array[10]` when we have only defined `array[5]`
    - `NaN` - this stands for ‘not a number’ and occurs most commonly when the code has attempted to divide a number by zero
  - Can be tricky to find, experience often helps

Advanced Programming

Debugging: Types of Bugs

- Logic or other semantic error (type of runtime error)

  - Again, the code will usually compile and run, but will give a result that we know is wrong, eg.:

```
int add(int x, int y){ return x-y; }
int main() {
      cout << “5+3=“ << add(5,3) << ‘\n’;
      return 0;
 }
```

  - These can be very difficult to identify, and can cause subtly incorrect results

  - Can mitigate against these by comparing with output from another code, or using own physics/maths knowledge

Advanced Programming

Debugging: Types of Bugs

Example: Compile `data_reader_class_buggy.cpp`

First, some setup (some of this will be used later) - this is to enable us to easily edit files within the container:

- Pull the latest commit from the course repository
- Open Docker on your machine (laptop)
- Install ‘Remote Development’ extension pack in VSCode

Advanced Programming

Debugging: Types of Bugs

Running the Docker container:

- Rebuild the Docker image in VSCode (View -> Command Palette -> Docker Images: Build Image)
- ‘Run interactive’ Docker container

Next, we want to be able to use VSCode tools from within our container:

- View -> Command Palette… -> Attach to running container
- Open in new window `/usr/src/dockertest1/data_reader_class_buggy.cpp`

Advanced Programming

Debugging: Types of Bugs

- From within the container, we can now compile the example
- We first see a *syntax error* at compile time - how do we trace it?

```
root@5b69181e1e32:/usr/src/dockertest1# g++ data_reader_class_buggy.cpp data_reader_class_buggy
data_reader_class_buggy.cpp: In function 'int main()':
data_reader_class_buggy.cpp:51:5: error: expected ',' or ';' before 'reader'
   51 |     reader.readData();
      |     ^~~~~~
```

Example: Fix the syntax error, recompile and run

- The code now complies, but running gives a runtime error

```
root@5b69181e1e32:/usr/src/dockertest1# ./data_reader_class_buggy
Average value of the second column: -nan
```

- How do we determine the cause of this bug?

Advanced Programming

Debugging: Methods

- There are two primary methods of debugging:
  - Print debugging - coder adds print statements at strategic points to check whether the code runs to those points
  - Using a debugger program - coder steps through the source code to find sources of error
- Even better than debugging is to add in your own error messages in advance to anticipate potential problems
  - Note that we included these in the `data_reader_class.cpp` example - would this help us catch the runtime error above?

Advanced Programming

Debugging: Methods

- Another very useful method is ‘rubber duck debugging’
- Essentially, this means using your own brain rather than outsourcing the solution to the computer
- Go through the section of code that you suspect contains the bug, line by line, explaining what each line does
- ‘Rubber duck’ refers to any object that you can pretend you are explaining to
- Unless you are lucky, real humans tend not to want to listen to the minutiae of your code implementation 😇 (unless you are really stuck, which is where RSEs come in)

Advanced Programming

Debugging: Print Debugging

Advantages of print debugging:

- Easy to add print statements into code
- No need to learn to use other tools
- Especially useful in serial codes - it is clear where the program has run to
- If you have an idea of what might be going wrong, you can print out specific variables to double check

Advanced Programming

Debugging: Print Debugging

Disadvantages of print debugging:

- In compiled codes, must recompile and run each time you want to move the print statements
- Can take time if you have to queue your simulation on a large machine
- Doesn’t always catch errors
- Need to go back through at the end and remove the statements that you have added

Advanced Programming

Debugging: Print Debugging

Example: Compile `data_reader_class_buggy.cpp` *with additional print statement(s) to help locate the bug*

- We notice that `calculateAverage()` is the function that returns the error
- Might want to print out the variables that it uses in its calculation
- See that `columnData.size()` returns zero

- If we are still stuck, we can use a debugger program…

Advanced Programming

Debugging: Debugger Programs

Advantages of debugger programs:

- Can obtain detailed information about variables and memory locations while it executes
- Can step through code line by line
- Can stop execution at any point using a *breakpoint*
- Some integrated with code editors for easy use
- Some debuggers designed for parallel code

Advanced Programming

Debugging: Debugger Programs

Disadvantages of debugger programs:

- Have to learn to use (especially if using from command line)
- Setup and use can be complex
- Doesn’t always catch errors

- Two common examples are GDB and Valgrind

Advanced Programming

Debugging: Debugger Programs

Example - Debugging in Docker Container in VSCode:

- Click ‘Run and Debug’ in tab on left
- Choose preferred compiler from drop down menu and ‘debug active file’
- This allows us to use the debugger we installed in the container, GDB, to examine the code
- At first, the code runs and no obvious bug is identified - we must add *breakpoints* to examine variables which we suspect
- In VSCode, click on the left of a line number where you would like the execution to stop
- Click ‘Run and Debug’ again - the code will stop at this point, and we can look at the variables in the panel on the left

Advanced Programming

Debugging: GDB

- We can also run GDB from the command line:
  - `cd /usr/src/dockertest1/`
  - `gdb data class reader buggy`
- We then add breakpoints and control the program flow using commands (full docs https://www.sourceware.org/gdb/)
  - The sample session (https://sourceware.org/gdb/current/onlinedocs/gdb.html/Sample-Session.html#Sample-Session) is a useful starting point

Advanced Programming

Debugging: GDB

```
(gdb) break data_reader_class_buggy.cpp:47
Breakpoint 1 at 0x26b7: file /usr/src/dockertest1/data_reader_class_buggy.cpp, line 48.
(gdb) run
Starting program: /usr/src/dockertest1/data_reader_class_buggy
warning: Error disabling address space randomization: Operation not permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at /usr/src/dockertest1/data_reader_class_buggy.cpp:48
48          CSVReader reader(filename);
(gdb)
```

Advanced Programming

Debugging: Valgrind

Advanced Programming

Debugging: Valgrind

Advanced Programming

Profiling

Advanced Programming

Profiling: Introduction to gprof

Advanced Programming

Profiling: Valgrind for Profiling

Advanced Programming

Profiling: Other Advanced Tools

Parallelisation

Visualisation