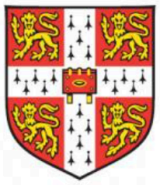

UNIVERSITY OF CAMBRIDGE

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