Part VI

# The compilation process

# Outline

# Global variables

- It is possible to declare (and set) variables outside of any function
- They are then globally available

```
int a = 3;

void setA(int x){
  a = x;
}

int main(void){
  std::cout << "a = " << a << std::endl;
  setA(8);
  std::cout << "a = " << a << std::endl;
}
```

- will output `a = 3` and `a = 8`.

# Global variables ctd

- In the above, `a` is a global variable.
- In any function, the same value is available for read/write access
- Global variables can be dangerous, because it is usually not obvious simply from a function's prototype whether it alters global variables
- It can then be hard to debug a function if it may have effects outside of those suggested by its parameters and return value
- Using global variables for anything other than constant variables is usually a bad idea (or suggests bad design)
- You could legitimately use them for storing user-input simulation parameters, such as the end-time for a simulation.

# Outline

## Pre-processor

- The first step of compilation is pre-processing
- Basic text/file processing (very simple, very powerful, very dangerous)
- We've already seen one pre-processing command:
  `#include <iostream>`
  includes the contents of the C++ file `iostream` in the current file
- The compiler can only really deal with one file at a time, so use `#include` to include pre-defined function prototypes in your code.
- Other files, including your own, can be included here, by `#include "MyFunctionDefns.H"`
- `#include "MyIncludeFile.H"` is used for your own header files, `#include <iostream>` is used for system/library header files.
- Note that the pre-processor always starts from the top of a file and works downwards

# Pre-processing #define

- `#define PI 3.1415926535`
  defines the symbol `PI`.
- Throughout the remaining code, `PI` is replaced by the exact string given
- This only happens when `PI` is a separate "token"
  i.e. not part of a variable/type name and separated from other tokens by white-space or an operator.

```
#define PI 3.1415926535
double circleArea(double r){
  return PI*r*r;
}

int main(void){
  int PIN;
  double r = 2;
  std::cout << "PI*r^2 = " << circleArea(r) << std::endl;
}
```

# After pre-processing

- After pre-processing, the previous code gives:

```
double circleArea(double r){
  return 3.1415926535*r*r;
}
```

- This is exactly what the compiler sees.
- No variable "PI" is defined or allocated.
- The pre-processor knows (virtually) nothing about C++, so it can easily give strange behaviour

# A slightly more useful example

- A slightly more useful example of macros could be

```
#define DO_FOREVER for(; ;)

DO_FOREVER
{
  // Some code here
  if(stopCondition){
    break;
  }
}
```

- although even this has questionable value.

# Optional compilation

- It is also possible to have conditional compilation
- This is quite useful for switching between blocks of code at compile time.
- or for disabling computationally expensive checks when compiling optimized.

```
#define DEBUG
// Code here
#ifdef DEBUG
std::cout << "Current value of x is " << x << std::endl;
#endif

#if 0
 // Old code...
#else
 // New code... (hopefully equivalent to previous)
#endif
```

## Optional compilation ctd

- Similarly, you can use

```
#ifndef FAST_CODE
// Simple, but slow code
#else
// Possibly less clear, but faster code.
#endif
```

- and this can be enabled/disabled at compile-time by adding the option `-DFAST_CODE`
- The compiler will define the pre-processing symbol `FAST_CODE`.

# Compile-time errors

- The pre-processor can also be used to trigger compile-time errors:

```
#if DIMN == 1
// Code for 1D here
#elif DIMN == 2
// Code for 2D here
#else
#error Code only implemented for DIMN = 1, 2
#endif
```

- If `DIMN == 3` when compiling, then the compiler will abort and display the error message above.
- Similarly `#warning` will cause the compiler to print just a warning, which will stand out because of course your code compiles without any other warnings.

# Macros

- Pre-processing macros can take arguments:

```
#define DEBUG(x) std::cout << #x << " = " << x << " at line "
                           << __LINE__ << std::endl

int myVar;
// Complex code
DEBUG(myVar)
// More complex code
```

will produce output "myVar = 42 at line 723".

- `#x` forms a string from a macro parameter
- `__LINE__` is the current line-number
- `__FILE__` is the current filename

Note that there is no space after `DEBUG`. Putting one here would define a macro starting with `(x)` rather than a macro taking a parameter `x`.

# Dangers of macros

- Macros do simple text-replacement, so can be very dangerous:

  ```
  #define SQUARE(a) a*a
  int b = SQUARE(c+3);
  ```

  expands to

  ```
  int b = c+3*c+3;
  ```

  which is wrong.

- Furthermore:

  ```
  #define SQUARE(a) (a)*(a)
  int b = SQUARE(c++);
  ```

  expands to `(c++)*(c++)` which will increment `c` twice, rather than the once that was hoped for.

- In short, macros are very dangerous, and should not be used, unless you really know what you are doing.

# Assertions

- When programming, we often make assumptions that certain conditions hold at, for example, the beginning of a function
- For example, that an array index is not out of bounds, that a pointer is not `nullptr`, etc.
- In theory, if coded correctly, such conditions should never arise
- However, if they do, they can cause odd side-effects and produce errors apparently unconnected with the original error
- To check for such errors, you can use `assert` to ensure that required conditions hold:

```
#include <assert.h>

double f(int a){
  assert( a > 5 );
  return sqrt(a - 5);
}
```

- If `a <= 5` on entry to the function, the program will abort.

# Outline

# Project organization

- When developing a large project, it is very useful to divide up functions into groups, depending on their intended use
- So, for a given group of functions, might have `IdealGas.H` containing function prototypes and `IdealGas.C` containing function definitions.
- A separate group of functions in `Simulation.C` that need to know about those defined in `IdealGas.C`, just needs to `#include "IdealGas.H"`
- So, `Simulation.C` knows all about the functions from `IdealGas.C`.
- But, functions in `Simulation.C` need to be able to call functions in `IdealGas.C`
- How do we link the two files together?

# Linking

Suppose we have the following:

MyFunc.H:

```
int doubleIt(int);
```

MyFunc.C:

```
#include "MyFunc.H"
int doubleIt(int x){
  return 2*x;
}
```

Main.C:

```
#include <iostream>
#include "MyFunc.H"

int main(void){
  std::cout <<
    doubleIt(5) <<
    std::endl;
  return 0;
}
```

# Linking

- First compile *object* files:
  - `g++ -c MyFunc.C -o MyFunc.o`
  - `g++ -c Main.C -o Main.o`

  (`-c` means to produce an object file)
- The object files contain the definitions of the functions.
- Then link these object files into a final executable:
  - `g++ Main.o MyFunc.o -o myProg`
- The final command *links* the files together; the linker tries to find all functions that are referenced in each file, but whose definitions do not occur in that file.
- If some functions are not found at this stage, then a linker error results.

# Linking error

If we omit one of the object files, we get a linker error:

```
g++ Main.o -o MyProgram
Main.o: In function ‘main’:
Main.C:(.text+0xa): undefined reference to ‘doubleIt(int)’
collect2: ld returned 1 exit status
```

If we had created a function with the same name and arguments in two separate object files, we might get a different linker error:

```
MyFunc2.o: In function ‘doubleIt(int)’:
MyFunc2.C:(.text+0x0): multiple definition of ‘doubleIt(int)’
MyFunc.o:MyFunc.C:(.text+0x0): first defined here
```

# Extern

- To allow global variables to be seen in all source-files, we can use:

```
extern int a;
```

- This can be put in a header file in the same way as a function prototype
- As long as the variable is defined in exactly one file:

```
int a = 77;
```

  the linker will not raise an error.
- Remember that global variables are usually evil, and making them available in multiple source-files is usually worse.

# Libraries

As with any major programming language, there are libraries of functions written by various developers. Examples are:

- BLAS - Optimized Vector/Matrix operations
- Boost - Advanced C++ utilities

These usually come in the form of various include files (`.h`,`.hpp`, or `.H`), which contain function prototypes which must be included in any source-file that uses them, and library files `.so`

# Linking to system libraries

In order to link to system libraries, do the following:

```
g++ Main.o MyFunc.o -o MyProgram -lblas
```

to link in the library in `/usr/lib/libblas.so`.
To link in a library not in a system directory, use:

```
g++ Main.o MyFunc.o -o MyProgram -L/opt/blas/lib -lblas
```

if `/opt/blas/lib` contains `libblas.so`
Note: the `.so` extension stands for "shared-object".

# Compiler options (gcc)

Other possible compiler options are:

- `-O1`, `-O2`, `-O3` levels of optimization
  `-O2` is usually sufficient, although `-O3` may be necessary for inter-procedural optimization.
- `-g` Compile with debugging symbols
- `-I<path>` Specify directory in which to look for include files.
- `-ansi` Turn off implementation specific features not in the C++ standard.
- `-Wall` Turn on compiler warnings
- `-Werror` Make warnings into errors (i.e. fail to compile)
- `-pedantic` Issue all warnings mandated by the standard
- `-o` *filename* Output to this file

# Other compilers

- The compiler flags listed in this section are for `gcc`
- Other compilers use similar options for include directories, libraries, and simple optimization
- Differences occur for more advanced optimization options, warning specifications, standard compatibility, etc.
- Read your compiler's documentation to find out more.

# Multiple include files

- For large projects, there will be many include files, with complex interdependencies.
- However, it is an error in C++ to define classes or functions more than once.
- It is possible that multiple header files will try to include the same header file themselves.
- For example,
  MyIncludeFile_1.H:

  ```
  #include <vector>
  ```

  MyIncludeFile_2.H:

  ```
  #include <vector>
  #include "MyIncludeFile_1.H"
  ```

  could cause problems if `vector` defined some functions/classes.

# Include guards

However, looking at `vector` reveals the following:

```
#ifndef _GLIBCXX_VECTOR
#define _GLIBCXX_VECTOR 1
// ... Code goes here ...
#endif  /* _GLIBCXX_VECTOR */
```

This will prevent the code in between the "include-guards" from being included more than once.