C++: Practical session 6

1 Preprocessors and macros

The following macro could be used in place of `assert`:

```
#define ASSERT(x) if( !(x) )\
{\
  std::cout <<  "Test " << #x << " failed at line " \
            << __LINE__ \
            << " in " << __FILE__ << std::endl; \
}
```

Points to note:

1. The \ is a line-continuation character that tells the pre-processor that the macro definition has not yet finished.

2. `#x` expands to a stringified form of the macro argument `x`

3. `__LINE__` is replaced by the line number where the macro is expanded

4. `__FILE__` is replaced by the file name where the macro is

Try using the above macro in a simple program:

```
int a = 5;
ASSERT(a == 5);
ASSERT(a == 6); // Should fail
```

How could it be improved?
To apply just the preprocessor to a file, use:

```
g++ -E MyProgram.C -o MyProgram.preproc
```

Open the pre-processed code in a text-editor. Note the amount of code produced by `<iostream>`.
Why are the brackets necessary around `!(x)` ?

1.1 More macros

Comment on the following macros. You may wish to try using them in a program to see how they work.

1. `#define POW2(i) 1 << i`
   Try using this as `int i = POW2(3);` and outputting `std::cout << POW2(8);`

2. `#define MIN(a,b) (((a) < (b)) ?  (a) :  (b) )`
   Try using with `MIN(a++, b++)` and displaying `a` and `b` afterwards.

3. `#define DISPLAY(x) std::cout << "At line " << __LINE__ << " " << #x << " = " << x << std::endl;`

4. `#define fabs(x) ( (x > 0) ?  (x) :  (-(x)) )`

5.
   ```
   #define DEFINE_MIN3(T)\
   T myMin(T a, T b, T c){\
      if( a < b && a < c ){\
       return a;\
      }\
      else if( b < c ){\
        return b;\
      }\
      else{\
        return c;\
      }\
   }\

   DEFINE_MIN3(int)
   DEFINE_MIN3(float)
   DEFINE_MIN3(double)
   ```

   which generates three overloaded functions (same name, different arguments), called `myMin` which can be used as:

   ```
   int a = myMin(3,1,8);
   ```

Use `gcc -E` to see exactly what the above macros all produce after pre-processing.

2 Object files and linking

Take the code that you wrote to solve an ODE, and put the Euler solver into a separate file. You should do this in the following stages:

1. Turn the Euler solver into a function with signature:

   ```
   double eulerStep(double x, double dt);
   ```

2. Create a header file `Euler.H` containing the above signature.

3. Remove the `eulerStep` function into a separate file `Euler.C`.

4. Create a header file `MyFunc.H` containing the prototype for the derivative function `double f(double x);`, which should be contained in `MyFunc.C`

5. `#include` the header file for `f` in `Euler.C` and likewise for `Euler.H` in `Main.C` (which contains the definition of `f`).

6. Compile the three files `MyFunc.C`, `Main.C` and `Euler.C` into object files and link them together.

2.1 Extensions

1. Implement the RK2 scheme in a separate file.

2. Rewrite the Euler/RK2 functions so that they take a function object instead of assuming that the function is called `f`. You can now dispose of the `MyFunc.H` header file from within Euler.C (although it is still needed in Main.C).