Project structure - working with multiple files

Declaration and definition

Recall the difference between declaration...

```
double max( double a, double b );
```

and definition...

```
double max( double a, double b ) {
    if (a>b) {
        return a;
    } else {
        return b;
    }
}
```

The rules for multiple files

- Every `.cpp` file that uses a function must contain a declaration for that function.
- Every function that is used must be defined in exactly one `.cpp` file or in a library.

This is why:

- `#include "anotherfile"` means perform all the C++ instructions contained in `anotherfile`.
- One writes header files (`.h`) which contain all the declarations needed in one place, ready to be `#included`.

Creating a header file

We want to be able to reuse the functions:

- `normcdf`—the cumulative distribution function of the normal distribution;
- `norminv`—the inverse of `normcdf`;
- and also a constant `PI`.

Let's create a new project called `MyLib`. Now create a header file:

- Right click on the folder **Header Files**.
- Create a header file called `matlib.h`.
- Type in the first line `#pragma once`

A complete header file

Starts `#pragma once`, followed by any required include statements, then the declarations for our functions (with comments):

```
#pragma once

const double PI = 3.14159265358979;

/*
 *  Computes the cumulative
 *  distribution function of the
 *  normal distribution
 */
double normcdf( double x );

/*
 *  Computes the inverse of normcdf
 */
double norminv( double x );
```

Some code that uses the functions

In a source file called `main.cpp` write the following code

```
#include <iostream>
#include "matlib.h"
using namespace std;

int main() {
    cout << "normcdf(1.96)="
         << normcdf(1.96) << "\n";
    cout << "norminv(0.975)="
         << norminv(0.975) << "\n";
    return 0;
}
```

Note angle brackets for standard libraries, quotation marks for your libraries.

Angle brackets really mean "this file won't have changed since I last compiled".

Attempt to build the code

- Try to build and run the code now.
- It will fail. This is because we haven't defined the functions.

```
main.obj : error LNK2019:
unresolved external symbol "double __cdecl
norminv(double)" (?norminv@@YANN@Z) referenced
in function _main
```

This is called a *linker error*. The phrase "unresolved external" is an unhelpful way of saying roughly either

- You forgot the definition altogether;
- Or the type information in the definition doesn't exactly match the type information in the declaration.

The build process

- The pre-processor performs simple text manipulation on the cpp files such as `#include` statements.
- The resulting cpp files are compiled.
- The compiled versions of all the files and all the libraries are *linked* together. Each use of a function is *linked* to the place where it is defined.

Define the functions

- Create a file called `matlib.cpp` to contain the function definitions.
- Start the file with the line `#include "matlib.h"`.
- Write the necessary code for the function definitions.

If you completed the homework, you could copy-and-paste the relevant bit of your solution. For the time being let's cheat for speed.

```
double norminv( double x ) {
    return 1234.0; // TODO fix this
}

double normcdf( double x ) {
    return 1234.0; // TODO fix this
}
```

Check everything builds

- You should now be able to build and run the code.
- Check that if you change the type in the definitions, you get build errors.

```
// change double to float
double norminv( float x ) {
    return 1234.0; // TODO fix this
}
```

Allowing the same file to be included twice

Delete the `#pragma once`. Now if you `#include matlib.h` twice, you get a build error. Either:

- Start every header file with `#pragma once`.
- For each header file pick a unique number (4569327457263475698023452376519876247) and then start and end every header file as follows:

```
#ifndef G4569327457263475698023452376519876247
#define G4569327457263475698023452376519876247
... rest of code ...
#endif
```

The first is easy but not officially part of the C++ language. The second is a pain but officially correct.

Tip: Recommended rules for header files

- Start every `.h` file with `#pragma once`.
- For every `.h` file there should be one `.cpp` file that defines everything it declares.
- The exception that proves the rule is you should have one header called `stdafx.h` that includes the standard libraries you're using.
- You should have a `main.cpp` file for testing too.
- The first line of the `.cpp` files should `#include` the corresponding `.h` file.
- NEVER type `using namespace` into a header file.

Information hiding

- Don't put helper functions in the header file. This means helper functions won't be part of your library.
- Users of your library will only see the `.h` files.
- Your users don't need to think about these functions.
- They won't phone you at 3 a.m.
- You can delete or change the functions.

Enhanced information hiding with static

- Try to declare global variables and functions that are not in your header file as `static`.
- This makes it *impossible* for someone else to write their own declarations and so use your functions.
- This avoids possible name clashes in large projects.

Example

We have created constants called `a0`, `a1`, `a2` etc. which make sense in the context of Moro's algorithm, but whose names we might want to reuse.

Another modifier: inline

- There is a small amount of overhead involved in calling a function.
- Marking a function as `inline` means "duplicate the code in this function whenever it is called rather than call the function".
- Functions which are `inline` might be marginally faster.
- You can't separate declaration and definition for `inline` functions.
- Too much inlining leads to bigger executables and hence *slower programs*.
- Our `hornerFunction` functions would be better inlined.
- It's just a hint. The compiler might ignore you.

Complete example

Unzip the project `FMLib.zip` from the website.

- The `hornerFunction` functions are *not* in the header file. We think users of our library won't want to know about them.
- The `hornerFunction` functions are `static`.
- The constants `a1`, `a2` etc. are all `static`.
- The `hornerFunction` functions are `inline`.

Using other libraries

How does Visual Studio find include files and libraries?

- Go to **Project→Properties** and look at **Configuration Properties→VC++ directories**.
- This lists the directories searched for `include` statements and the directories searched for libraries.
- If you decide to use a non-standard library you will need to tell it where the libraries `.h` files are saved. You do this by adding an entry to the "Include directories".
- You will need to say where the libraries binary file (`.lib`) is saved. Use "Library directories" for this.
- If you are writing a library, you should change the linker settings to create a `.lib` file and not a `.exe`.