Modular Programming (PHY1610 Lecture 3)

Ramses van Zon Marcelo Ponce

Winter 2021

Why does modularity matter?

Modularity? Who cares?

- Scientific software can be large, complex and subtle.
- If each section uses the internal details of other sections, you must understand the entire code at once to understand what the code in a particular section is doing.

  (This is why global variables are bad bad bad!)
- Interactions grow as (number of lines of code)^2.

Example: Monolythic code for hydrogen's ground state

```cpp
// hydrogen.cpp
#include <iostream>
#include <fstream>
const int n = 100;
double m[n][n], a[n], b = 0.0;
void pw() {
    double q[n] = {0};
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            q[i] += m[i][j]*a[j];
    for (int i = 0; i < n; i++)
        a[i] = q[i];
}
double en() {
    double e = 0.0, z = 0.0;
    for (int i = 0; i < n; i++) {
        z += a[i]*a[i];
        for (int j = 0; j < n; j++)
            e += a[i]*m[i][j]*a[j];
    }
    return b + e/z;
}
```

```cpp
int main() {
    for (int i = 0; i < n; i++) {
        a[i] = 1.0;
        for (int j = 0; j < n; j++) {
            m[i][j] = ...;
        }
    }
    for (int i = 0; i < n; i++)
        if (m[i][i] > b)
            b = m[i][i];
    for (int i = 0; i < n; i++)
        m[i][i] -= b;
    for (int p = 0; p < 20; p++)
        pw();
    std::cout<<"Ground state energy="<<en()<<"\n";
    std::ofstream f("data.txt");
    for (int i = 0; i < n; i++)
        f << a[i] << std::endl;
    std::ofstream g("data.bin", std::ios::binary);
    g.write((char*)(a), sizeof(a));
    return 0;
}
```

What is wrong with that code?

The `hydrogen.cpp` code uses functions. Is that not modular?

A few bad things:

- Global variables that all of the code can modify.
- All code in one file.
- No comments.
- Not clear what part does what, or what part needs which variables.
- Cryptic variable and function names.
- Hard-coded filenames and parameters.
- Automatic arrays.

Who cares, you might say, as long as it runs? But:

- **Code is not written for a computer but for humans.**
- **Code almost never a one-off.**

What to do: Use modularity

- You must enforce **boundaries** between sections of code so that you have self-contained modules of functionality.
- This is not just for your own sanity. There are added benefits:
- Each section can then be tested individually, which is significantly easier.
- Makes rebuilding software more efficient.
- Makes version control more powerful.
- Makes changing the code easier.

But it's more work up-front

- Think about the **blocks of functionality** that you are going to need.
- **How** are the routines within these blocks going to be **used**?
- Think about **what** you might want to use these routines **for**; only then design the interface.
- The interfaces to your routines may change a bit in the early stages of your code development, but if it changes a lot you should stop and rethink things – you're not using the functionality the way you expected to.
- More work up-front but results in higher productivity in the long run.

Developing good infrastructure is always time well spent.

A simple example of modularization

The code writes out the array in binary and text formats. Let's start with putting those parts in functions.

```
//hydrogen.cpp
#include <string>


void writeBinary(const std::string& s, int n, const double x[]) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(x), n*sizeof(x[0]));
   g.close();
}
void writeText(const std::string& s, int n, const double x[]) {
   std::ofstream f(s);
   for (int i=0; i<n; i++)
      f << a[i] << std::endl;
   f.close();
}
//...
int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
   //...
}
```

A simple example of modularization

The code writes out the array in binary and text formats. Let's extract function declarations.

```
//hydrogen.cpp
#include <string>


void writeBinary(const std::string& s, int n, const double x[]) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(x), n*sizeof(x[0]));
   g.close();
}
void writeText(const std::string& s, int n, const double x[]) {
   std::ofstream f(s);
   for (int i=0; i<n; i++)
      f << a[i] << std::endl;
   f.close();
}
//...
int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
   //...
}
```

A simple example of modularization

The code writes out the array in binary and text formats. Let's extract declarations.

```
//hydrogen.cpp
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
void writeBinary(const std::string& s, int n, const double x[]) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(x), n*sizeof(x[0]));
   g.close();
}
void writeText(const std::string& s, int n, const double x[]) {
   std::ofstream f(s);
   for (int i=0; i<n; i++)
      f << a[i] << std::endl;
   f.close();
}
//...
int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
   //...
}
```

A simple example of modularization

The code writes out the array in binary and text formats. Let's extract declarations.

```
//hydrogen.cpp
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
void writeBinary(const std::string& s, int n, const double x[]) {

    // bunch of commands

}
void writeText(const std::string& s, int n, const double x[]) {

   // bunch of commands


}
//...
int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
   //...
}
```

A simple example of modularization

The code writes out the array in binary and text formats. Function definitions can be moved.

```
//hydrogen.cpp
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
void writeBinary(const std::string& s, int n, const double x[]) {

    // bunch of commands

}
void writeText(const std::string& s, int n, const double x[]) {

   // bunch of commands


}
//...
int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
   //...
}
```

A simple example of modularization

The code writes out the array in binary and text formats. Function definitions can be moved.

```
//hydrogen.cpp
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);

//...

int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
   //...
}

void writeBinary(const std::string& s, int n, const double x[]) {
    // bunch of commands
}

void writeText(const std::string& s, int n, const double x[]) {
   // bunch of commands
}
```

A simple example of modularization

The code writes out the array in binary and text formats. We're ready to make a module now!

```
//hydrogen.cpp
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);

//...

int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
   //...
}

void writeBinary(const std::string& s, int n, const double x[]) {
    // bunch of commands
}

void writeText(const std::string& s, int n, const double x[]) {
   // bunch of commands
}
```

Creating the module

To create our own module, put the declarations for the functions in their own 'header' file.

```
//outputarray.h
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
```

The source code with the definitions of the functions shoule be put into its own separate file.

```
//outputarray.cpp
#include "outputarray.h"
#include <fstream>
void writeBinary(const std::string& s, int n, const double x[]) {
    // bunch of commands
}

void writeText(const std::string& s, int n, const double x[]) {
   // bunch of commands
}
```

Using the module

The original code that uses these would look like:

```
//hydrogen.cpp
#include "outputarray.h"

//...

int main() {
   //...
   writeText("data.txt", data, n);
   writeBinary("data.bin", data, n);
   //...
}
```

Compiling + Linking = Building

So how to compile this code?

- Before the full program can be compiled, all the **source** files (hydrogen.cpp, outputarray.cpp) must be **compiled**.
- outputarray.cpp doesn't contain a main function, so it can't be an executable (no program to run).

  Instead `outputarray.cpp` is compiled into a ".o" file, an **object file**, using the "-c" flag
- It is customary and advisable to compile all the code pieces into object files.
- After all the object files are generated, they are **linked** together to create the working executable.

```
$ g++ -std=c++14 outputarray.cpp -c -o outputarray.o    // compile
$ g++ -std=c++14 hydrogen.cpp -c -o hydrogen.o          // compile
$ g++ -std=c++14 outputarray.o hydrogen.o -o hydrogen  // link
```

- If you leave out one of the needed .o files you will get a fatal linking error: *"symbol not found"*.

Interface v. Implementation

By creating a header file, we separated the interface from the implementation.

- The implementation - the actual code for writeBinary and writeText - goes in the .cpp (or .cc or .cxx) or 'source' file. This is compiled on its own, separately from any program that uses its functions.
- The interface - what the calling code needs to know - goes in the .h or 'header' files. This is also called the API (Application Programming Interface).

```
//outputarray.h
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
```

This distinction is crucial for writing modular code.

Interface v. implementation

So, to review:

- When hydrogen.cpp is being compiled, the header file outputarray.h is included to tell the compiler that there exists out there somewhere functions of the form

```
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
```

- This allows the compiler to check the number and type of arguments and the return type for those functions (the interface).
- The compiler does not need to know the details of the implementation, since it's not compiling the implementation (the source code of the routine).
- The programmer of hydrogen.cpp also does not need to know the implementation, and is free to assume that writeBinary and writeText have been programmed correctly.

Guards against multiple inclusion

Protect your header files!

- Header files can include other header files.
- It can be hard to figure out which header files are already included in the program.
- Including a header file twice will lead to doubly-defined entities, which results in a compiler error.
- The solution is to add a 'preprocessor guard' to every header file:

```
//outputarray.h
#ifndef OUTPUTARRAY_H
#define OUTPUTARRAY_H
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
#endif
```

We'll expect to see these in your homework.

About the preprocessor

What do you mean by "preprocessor"?

- Before the compiler actually compiles the code, a "preprocessor" is run on the code.
- For our purposes, the preprocessor is essentially just a text-substitution tool.
- Every line that starts with "#" is interpreted by the preprocessor.
- The most common directives a beginner encounters are `#include`, `#ifndef`, `#define`, and `#endif`.

Future Feature

The brand new C++20 standard defines a better way to build modules than to rely on the preprocessor.

Implementations by compilers are not fully there yet, or are experimental. They also still vary a lot from compiler to compiler, to the extent that it changes how you build software.

So for now, let's stick with the `#include` technique.

What goes into the interface (i.e. the header file)?

So what should one expect in a header file?

- At the very least, the **function declarations**.
- There may also be **constants** that the calling function and the routine need to agree on (error codes, for example) or **definitions of data structures**, classes, etc.
- **Comments**, which give a description of the module and its functions.

Further guidelines:

- There should really only be one header file per module. In theory there can be multiple source files.
- Not necessarily every function declaration is in the header file, just the public ones. Routines internal to the module are not in the public header file.

What goes into the implementation (source file)?

What should one expect in a source file?

- Everything which is defined in the .h file which requires code that is not in the .h file. Particularly, **function definitions**.
- **Internal routines** which are used by the routines declared in the .h file.
- To ensure consistency, include the corresponding .h file at the top of the file.
- Everything that needs to be compiled and linked to code that uses the .h file.

Modularization Benefit 1: Clarity

File list without modules:

```
$ ls
hydrogen.cpp

$ cat hydrogen.cpp
#include <iostream>
// ...
int main() {
  // monolythic code
}
$
```

Modular file list looks like:

```
$ ls
hydrogen.cpp     outputarray.cpp  outputarray.h
initmatrix.cpp   initmatrix.h     eigenvalues.cpp
eigenvalues.h    Makefile

$
```

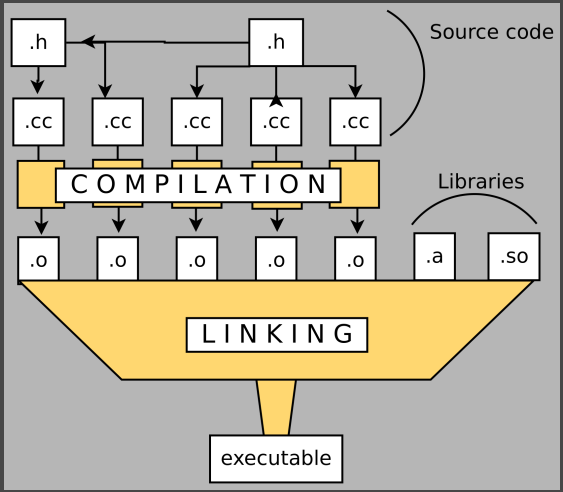
Having each file have a single responsibility, makes it easier to understand:

- for you collaborators.
- and for your future self!

Clearer means easier to maintain and adapt.

Modularization Benefit 2: Faster compilation

Consider a build tree with several source files, like this:



1. Each source file compilation into an .o file can be done simultaneously.

   Parallel processing of code!

2. If only one .cc file has changed, only that file needs to be recompiled.

   Fast rebuilds

The compilation process itself gets more complex, but we'll see how you use **make** to automate that in the next lecture.

Modularization Benefit 3: Better version control

- As we will show, version control systems can keep **track of changes in files.**
- By splitting our code up into several files, one can more easily see what changes were made in what functional piece of the code (most code changes will onlu involve one or two files).
- Also easier to restore a functionality by restoring just that file.
- By the way, you would not put your executable or object files under version control; they can be generated from the existing files.

We'll discuss version control with **git** in a separate lecture.

Modularization Benefit 4: Easier testing and reuse

- Each module should have a separate test suite. If the code is properly modular, those module tests should not need any of the other .cpp files.
- Testing will give confidence in your module, and will tell you which modules have stopped working properly.
- Once your tests are okay, you now have a piece of code that you could easily use in other applications as well, and which you can comfortably share.

We'll discuss **unit testing** in a separate lecture as well.