Modular Programming (PHY1610 Lecture 3)

Ramses van Zon

SciNet HPC Consortium

January 12, 2017

Modularity

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

```
// hydrogen.cc
#include <rarray>
#include <rarrayio>
#include <iostream>
#include <fstream>
const int n = 100;
rmatrix<double> m(n,n);
rvector<double> a(n);
void pw() {
  rvector<double> q(n);
  q.fill(0.0);
  for (int i=0;i<n;i++)
      for (int j=0;j<n;j++)
          q[i] += m[i][j]*a[i];
  a = q.copy();
}
double en() {
  rvector<double> q(n);
  q.fill(0.0);
  for (int i=0;i<n;i++)
      for (int j=0;j<n;j++)
          q[i] += m[i][j]*a[i];
  double e = 0.0;
  for (int i=0;i<n;i++)
      e += a[i] * q[i];
  return e;
}
```

```
int main() {
  a.fill(1);
  for (int i=0;i<n;i++)
      for (int j=0;j<n;j++)
          m[i][j] = ...;
  double b = 0;
  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<10;p++)
      pw();
  std::cout<<"Ground state energy is "<<en()<<std::endl;
  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.data()),a.size()*sizeof(a[0]));
  return 0;
}
```

It uses functions. Is that not modular?
What is bad about this code?

SciNet Physics UNIVERSITY OF TORONTO

Why does modularity matter?

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-codes filenames.

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.
- Like documentation, thinking about the overall design, enforcing boundaries between modules, and testing, is more work up-front but results in higher productivity in the long run.

Developing good infrastructure is always time well spent.

A simple example

Imagine we're writing out the vector in binary and ASCII formats:

```
//hydrogen.cc

void toBin(const char* s, double *x, int n)
{
   std::ofstream g(s,std::ios::binary);
   g.write((char*)(x.data()),n*sizeof(x[0]));
   g.close();
}

void toAsc(const char* s, double *x, int n)
{
   std::ofstream f(s);
   for (int i=0; i<n; i++)
      f << a[i] << std::endl;
   f.close();
}

//...

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

A simple example, continued

```
//hydrogen.cc

//Prototypes.
void toBin(const char* s, double *x, int n);
void toAsc(const char* s, double *x, int n);

//...

int main() {
   //...
   toBin("data.bin", data, n);
   toAsc("data.txt", data, n);
   //...
}

void toBin(const char* s, double *x, int n)
{
   // bunch of commands
}

void toAsc(const char* s, double *x, int n)
{
   // bunch of commands
}
```

Creating modules

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

The source code for the functions can then be put into its own file.

```
//outputarray.h
void toBin(const char* s, double *x, int n);
void toAsc(const char* s, double *x, int n);
```

```
//hydrogen.cc
#include "outputarray.h"
//...
int main() {
   //...
   toBin("data.bin", data, n);
   toAsc("data.txt", data, n);
   //...
}
```

Interface v. implementation

By creating a header file we can separate the interface from the implementation.

- The implementation - the actual code for toBin and toAsc - goes in the .cc (or .cpp 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
void toBin(const char* s, double *x, int n);
void toAsc(const char* s, double *x, int n);
```

This distinction is crucial for writing modular code.

Interface v. implementation

So, to review:

- When hydrogen.cc 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 toBin(const char* s, double *x, int n);
void toAsc(const char* s, double *x, int n);
```

- 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.cc also does not need to know the implementation, and is free to assume that toBin and toAsc have been programmed correctly.

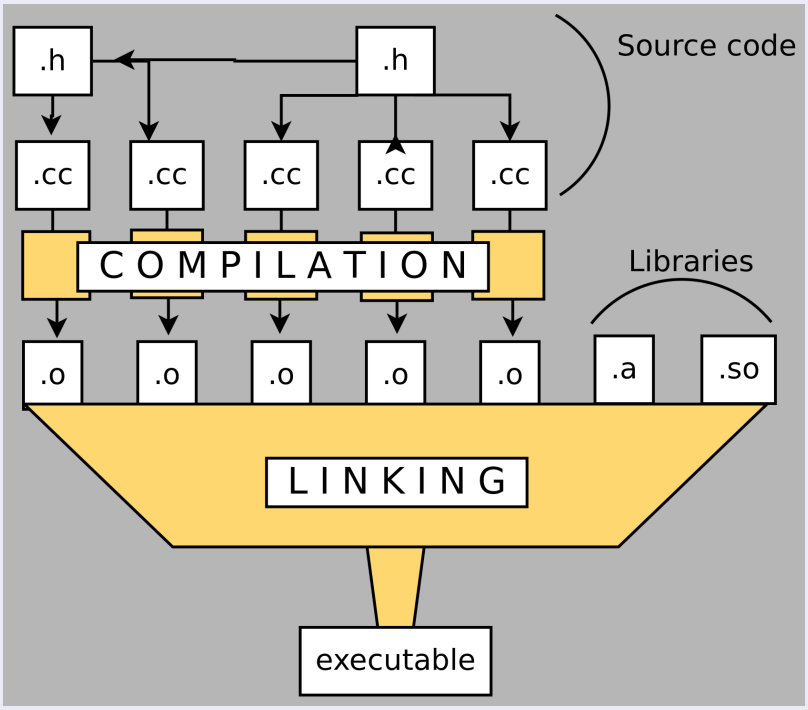
Compiling + Linking = Building

So how to we compile this code?

- Before the full program can be compiled, all the **source** files (hydrogen.cc, outputarray.cc) must be **compiled**.
- outputarray.cc doesn't contain a main fucntion, so it can't be an executable (no program to run). Instead outputarray.cc is compiled into a ".o" file, an **object file**, using the "-c" flag ("compile only")
- 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.
- If you leave out one of the needed .o files you will get a fatal linking error: **"symbol not found".**

```
$ g++ outputarray.cc -c -o outputarray.o
$ g++ hydrogen.cc -c -o hydrogen.o
$ g++ outputarray.o hydrogen.o -o hydrogen
```

Compiling + Linking = Building



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
void toBin(const char* s, double *x, int n);
void toAsc(const char* s, double *x, int n);
#endif
```

We expect to see these in your homework.

A note about preprocessors

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`.

```
//outputarray.h
#ifndef OUTPUTARRAY_H
#define OUTPUTARRAY_H
void toBin(const char* s, double *x, int n);
void toAsc(const char* s, double *x, int n);
#endif
```

What goes into the header file?

So what should one expect in a header file?

- At the very least, the **function prototypes**.
- 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 prototype 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 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 prototyped 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.

Benefits

Benefit 1: Clarity

Not modular:

```
$ ls
hydrogen.cc

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

Modular:

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

$
```

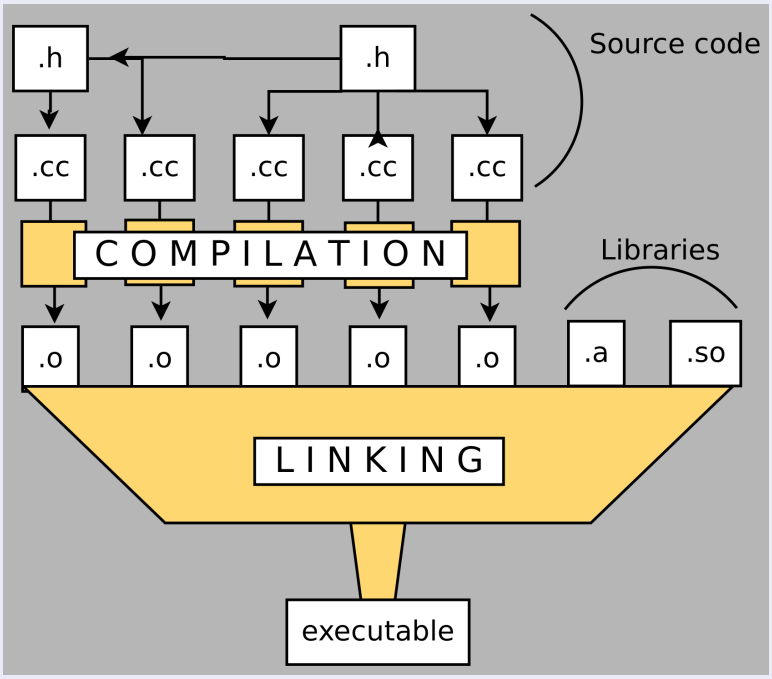
Having each file have a single responsibility, makes it easier to understand:

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

Clearer means easier to maintain and adapt.

Benefit 2: Faster compilation

Consider the build tree again:



1. Each compilation of an .cc file 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 a future lecture.

Benefit 3: Better version control

- As we will see, version control systems can keep **track of changes in files.**
- By splitting our code up into several files, we can more easily see what changes were made in what functional piece of the code.
- 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.

Benefit 4: Easier testing and reuse

- Each module should have a separate test suite. If the code is properly modular, those module test should not need any of the other .cc 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.