Modular Programming

Ramses van Zon

PHY1610 Winter 2023

Modularity

Why does modularity matter?

Modularity? Why?

- Scientific software can be large, complex and subtle.

  E.g., sections for simulation parameters, system creation, initial conditions, output, time stepping, . . .

- 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 = 4913;
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] = H(i,j,n);
        }
    }
    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 this code?

```
// hydrogen.cpp
#include <iostream>
#include <fstream>
const int n = 4913;
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;
}
```

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

No, not by itself. A few *bad* things:

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

Modularity

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.**

That's why 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 and maintaining the code easier.

Up-front work

- 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.

```cpp
//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.

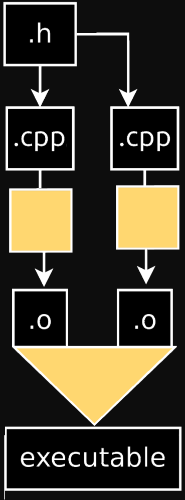
```
//outputarray.h
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
```

The source code with the definitions of the functions should 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:

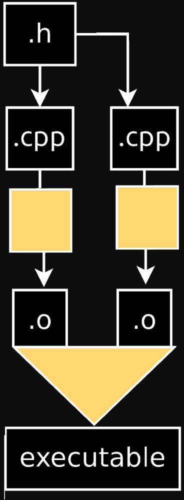
```
//hydrogen.cpp
#include "outputarray.h"

//...

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

Must now combine the pieces!

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**.
- No main function in outputarray.cpp, so it can't become executable.

  Instead `outputarray.cpp` is compiled into an **object file** using the "-c" flag
- It is advisable to separately 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++17 outputarray.cpp -c -o outputarray.o   // compile
$ g++ -std=c++17 hydrogen.cpp -c -o hydrogen.o         // compile
$ g++ outputarray.o hydrogen.o -o hydrogen  // link
```

- If you leave out one of the needed .o files you will get a fatal linking error: *"undefined reference to . . . ".*

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 new C++20 standard defines a way to define modules that does not rely on the preprocessor.

Support of C++20 modules by compilers are still not complete and buggy. And where implementations exist, important details like how you compile and link these modules, how you should name your modules files, and where the result of a module compilation goes, all varies wildly among compilers.

So unless you don’t want to compile your code, using C++20 modules at this point is more complex and less portable than sticking 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.

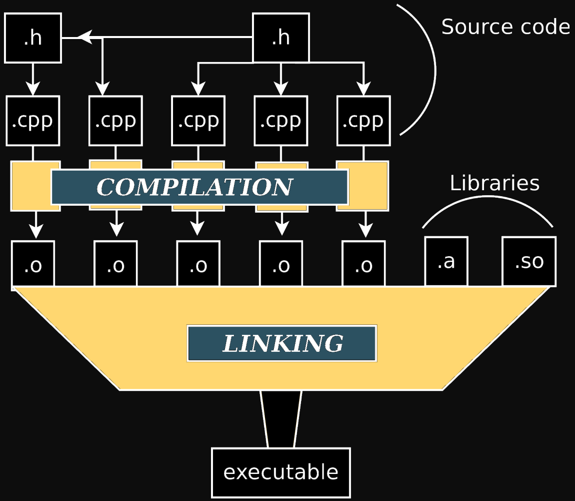
What goes into the implementation (source file)?

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 allows faster compilation

Consider a build tree with several source files, e.g.



1. Source file ***compilations*** can be done simultaneously.

   Parallel processing of code!

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

   Fast rebuilds

But:

- Now you need to keep track of what depends upon what.
- You need to retype in the entire compilation command every time you need to recompile.

This is where the `make` program comes in!

Make

Make

- `make` is a **build program** that is used to build programs from multiple `.cpp`, `.h`, `.o`, and other files.
- It is actually a very general framework that is used to compile code, of any type.
- `make` takes a Makefile as its input, which specifies what to do, and how.
- The Makefile contains variables, rules and dependencies.
- The Makefile specifies executables, compiler flags, library locations, . . .
- Build programs are a crucial component of *professional software development*.

  https://www.gnu.org/software/make/manual/html_node/index.html

Basic usage

- Make is invoked with a list of target files to build as *command-line arguments*:

```
$ make [TARGET ...]
```

- Without arguments, `make` builds the **first** target that appears in its makefile, which is traditionally a symbolic target named `all`.
- Make uses the rules in the Makefile to decide which targets needs to be (re)generated based on file modification times.
- This solves the problem of avoiding the building of files which are already up to date, as long as the timestamps are consistent and correct.

Rules

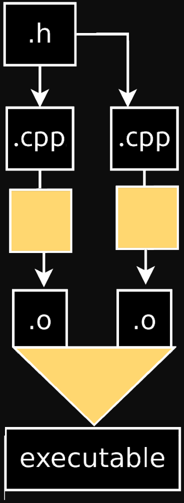
- A Makefile is a plain text file consisting of rules.
- Each rule begins with a textual dependency line which defines a target followed by a colon (:) and optionally an enumeration of prerequisites (files or other targets) on which the target depends.
- The dependency line is arranged so that the target (left of the colon) depends on the "prerequisites" (to its right)
- Each command-line must start with a TAB character to be recognized as a command.

```
TARGET: prerequisites1 prerequisite2 ...
     [command 1]
     :
     [command n]
```

Unfortunately, as you can't easily see in your editor whether you have a TAB character or a set of spaces. If you have spaces instead of a TAB, `make` will print the unhelpful error:

```
Makefile:3: *** missing separator.  Stop.
```

Simple Makefile Example



Consider this set of commands:

```
$ g++ -std=c++17 -O2 outputarray.cpp -c -o outputarray.o
$ g++ -std=c++17 -O2 hydrogen.cpp -c -o hydrogen.o
$ g++ -O2 outputarray.o hydrogen.o -o hydrogen
```

Here, -O2 stands for **Optimization level 2**.
This option makes the compiler create faster machine code.
Do this unless you know why you shouldn't.

This can be encoded into this Makefile:

```
# Makefile
hydrogen:
    g++ -O2 outputarray.o hydrogen.o -o hydrogen

outputarray.o: outputarray.cpp outputarray.h
    g++ -std=c++17 -O2 outputarray.cpp -c -o outputarray.o

hydrogen.o: hydrogen.cpp outputarray.h
    g++ -std=c++17 -O2 hydrogen.cpp -c -o hydrogen.o
```

which will build what is needed when running `make`.

Rules - commands

- Each command is executed by a separate shell or command-line interpreter instance.
- Comments are included using #
- A rule may have no command lines defined.
  The dependency line can consist solely of components that refer to targets.
  This means either there is nothing to do, or there is a predefined rule.
- The Makefile dependencies are declarative.
  They define the build tree.
  Their order does not matter.

Need multiple commands?

- The backslash \ can be used to have commands executed by the same shell, it represents line-continuation
- Commands can be separated by ;

Macros & Variables

- Macros are usually referred to as variables when they hold simple string definitions, like `CXX = g++`.
- Macros in makefiles may be overridden by the command-line arguments passed to the Make utility (e.g. "`make CXX=icpc`").
- Macros allow users to specify the programs invoked and other custom behavior during the build process.

  For example, the macro `CXX` is used in makefiles to refer to the location of the C++ compiler
- To use variables, you need to use a dollar sign ($) followed by the name of the variable in parenthesis (or curly braces).

Examples

```
MACRO = definition
```

```
PACKAGE  =  package
VERSION  =  `date +"\%Y.\%m\%d"`
ARCHIVE  =  $(PACKAGE)-$(VERSION)

dist:
    # Notice that only now macros are
    # expanded for shell to interpret:
    # tar -cf ../package-`date +"\%Y\%m\%d"`.tar
    tar -cf ../$(ARCHIVE).tar .
```

Note: Environment variables are also available as macros.

Extended Example: Compilation and Linking

```
# Example Makefile for the `hydrogen` program (after modularization)
CXX=g++
CXXFLAGS=-std=c++17 -O2
LDFLAGS=-O2
all: hydrogen

hydrogen: hydrogen.o outputarray.o initmatrix.o eigenvalue.o
    $(CXX) $(LDFLAGS) -o hydrogen hydrogen.o outputarray.o initmatrix.o eigenvalue.o $(LDLIBS)

hydrogen.o: hydrogen.c outputarray.h initmatrix.h eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o hydrogen.o hydrogen.c

outputarray.o: outputarray.cpp outputarray.h
    $(CXX) -c $(CXXFLAGS) -o outputarray.o outputarray.c

initmatrix.o: initmatrix.cpp initmatix.h
    $(CXX) -c $(CXXFLAGS) -o initmatrix.o initmatrix.c

eigenvalue.o: eigenvalue.cpp eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o eigenvalue.o eigenvalue.c
clean:
    $(RM) eigenvalue.o initmatrix.o outputarray.o hydrogen.o
.PHONY: all clean
```

Compilation and Linking

What happens when you type `make`?

- `make` will only recompile those dependencies that have source files that are newer then the library, thus only the code you are working on is modified.
- If a target is not a file, you should declare it 'PHONY'.
  Otherwise, should a file by that name exist, `make` thinks it's done already.
- It's good practice to put a clean rule in your Makefile that allows the whole compilation to restart.
- Several rules could be processed at the same time; you can tell `make` to try and use multiple processes when the dependencies allow it, but specifying a `-j` option, e.g.

```
$ make -j 4
```

Special Variables

- `$@`: the target filename
- `$*`: the target filename without the file extension
- `$<`: the first prerequisite filename
- `$^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.
- `$+`: similar to `$^`, but includes duplicates
- `$?`: the names of all prerequisites that are newer than the target, separated by spaces

Extended Example with Variables

```
# Example Makefile for the `hydrogen` program (after modularization)
CXX=g++
CXXFLAGS=-std=c++17 -O2
LDFLAGS=-O2
all: hydrogen

hydrogen: hydrogen.o outputarray.o initmatrix.o eigenvalue.o
    $(CXX) $(LDFLAGS) -o $@ $^ $(LDLIBS)

hydrogen.o: hydrogen.c outputarray.h initmatrix.h eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o $@ $<

outputarray.o: outputarray.cpp outputarray.h
    $(CXX) -c $(CXXFLAGS) -o $@ $<

initmatrix.o: initmatrix.cpp initmatix.h
    $(CXX) -c $(CXXFLAGS) -o $@ $<

eigenvalue.o: eigenvalue.cpp eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o $@ $<
clean:
    $(RM) eigenvalue.o initmatrix.o outputarray.o hydrogen.o
.PHONY: all clean
```