Modular Programming

Ramses van Zon

PHY1610 Winter 2026

Modular Programming

Why does modular programming matter?

- 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
#include <print>
#include <fstream>
#include <rarray>
const int n = 4913;
rarray<double,2> m(n,n); rarray<double,1> a(n);
double b = 0.0;
void pw() {
    auto q = make_rarray(n, 0.0);
    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 + (z?(e/z):0.0);
}
```

```cpp
double H(int i, int j, int n);
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();
    if (en() <= b) throw "Method failed";
    std::println("Ground state energy={}", en());
    std::ofstream f("data.txt");
    std::println(f, "{}", a);
    std::ofstream g("data.bin", std::ios::binary);
    g.write((char*)(&a[0]), sizeof(a[0]));
}
// ...
```

What is wrong with this code?

```
#include <print>
#include <fstream>
#include <rarray>
const int n = 4913;
rarray<double,2> m(n,n); rarray<double,1> a(n);
double b = 0.0;
void pw() {
    auto q = make_rarray(n, 0.0);
    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 + (z?(e/z):0.0);
}
```

The `hydrogen.cpp` code uses functions.
Is that not modular?
No, not by itself. A few *bad* things:

- Global variables `n,m,a,b`.
- 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.
- Hard-coded filenames and parameters.

What's this "rarray"?

Rarray provides multidimensional arrays.
https://raw.githubusercontent.com/vanzonr/rarray/main/rarray

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.

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 <print>
#include <fstream>
#include <rarray>
#include <string>


void writeBinary(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(&x[0]), n*sizeof(x[0]));
}
void writeText(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream f(s);
   std::println(f, "{}", a);
}
//...
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 <print>
#include <fstream>
#include <rarray>
#include <string>


void writeBinary(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(&x[0]), n*sizeof(x[0]));
}
void writeText(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream f(s);
   std::println(f, "{}", a);
}
//...
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 <print>
#include <fstream>
#include <rarray>
#include <string>
void writeBinary(const std::string& s, int n, const rarray<double,1>& x);
void writeText(const std::string& s, int n, const rarray<double,1>& x);
void writeBinary(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(&x[0]), n*sizeof(x[0]));
}
void writeText(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream f(s);
   std::println(f, "{}", a);
}
//...
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 <print>
#include <fstream>
#include <rarray>
#include <string>
void writeBinary(const std::string& s, int n, const rarray<double,1>& x);
void writeText(const std::string& s, int n, const rarray<double,1>& x);
void writeBinary(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(&x[0]), n*sizeof(x[0]));
}
void writeText(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream f(s);
   std::println(f, "{}", a);
}
//...
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 <print>
#include <fstream>
#include <rarray>
#include <string>
void writeBinary(const std::string& s, int n, const rarray<double,1>& x);
void writeText(const std::string& s, int n, const rarray<double,1>& x);
//...
int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
}
void writeBinary(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(&x[0]), n*sizeof(x[0]));
}
void writeText(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream f(s);
   std::println(f, "{}", a);
}
```

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 <print>
#include <fstream>
#include <rarray>
#include <string>
void writeBinary(const std::string& s, int n, const rarray<double,1>& x);
void writeText(const std::string& s, int n, const rarray<double,1>& x);
//...
int main() {
   //...
   writeText("data.txt", n, a);
   writeBinary("data.bin", n, a);
}
void writeBinary(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(&x[0]), n*sizeof(x[0]));
}
void writeText(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream f(s);
   std::println(f, "{}", a);
}
```

Creating the module

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

```
//output.h
#include <rarray>
#include <string>
void writeBinary(const std::string& s, int n, const rarray<double,1>& x);
void writeText(const std::string& s, int n, const rarray<double,1>& x);
```

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

```
//output.cpp
#include "output.h"
#include <print>
#include <fstream>
void writeBinary(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream g(s, std::ios::binary);
   g.write((char*)(&x[0]), n*sizeof(x[0]));
}
void writeText(const std::string& s, int n, const rarray<double,1>& x) {
   std::ofstream f(s);
   std::println(f, "{}", a);
}
```

Using the module

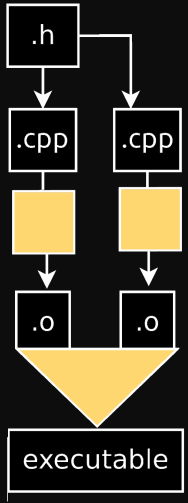
The original code that uses these would look like:

```
//hydrogen.cpp
#include <print>
#include <rarray>
#include "output.h"

//...

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

Compiling + Linking = Building



So how to compile this code?

- Before the full program can be compiled, all the **source** files (hydrogen.cpp, output.cpp) must be **compiled**.
- No main function in output.cpp, so it can't become executable.

  Instead output.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.

```
$ module load gcc/14.2 rarray/2.8.2 # on the teach cluster
$ g++ -std=c++23 -march=native -Wall -O3 -c -o hydrogen.o hydrogen.cpp
$ g++ -std=c++23 -march=native -Wall -O3 -c -o output.o   output.cpp
$ g++ -g -O3 -o hydrogen output.o hydrogen.o
```

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

Compiler options (a.k.a flags)

`-std=c++23`

Tells the compiler to adhere to this standard of the C++ language. The default standard is likely something close to C++17, so we need this flag.

`-march=native`

The optimal code also depends on the CPU you want to compile for. Different CPUs support different machine instruction sets.

By default, the compiler generates machine code that conforms to rather low baseline, which will run on older computers as well.

The `-march=native` tells the compiler to use all instruction available on the computer.

`-Wall`

Lets the compiler give you warnings about common mistakes, i.e., about pieces of the code that look like they may not do what you want.

Always check and try to fix these warnings!

`-O3`

The translation from C++ to an object file is not unique, but finding the most optimal machine code takes time.

So, by default, many compilers do not try to find this optimal code, but they will if you specify -O1, -O2 or -O3.

Interface v. Implementation

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

```
void writeBinary(const std::string& s, int n, const rarray<double,1>& x);
void writeText(const std::string& s, int n, const rarray<double,1>& 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 function).
- 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.
- This separation of interface and implementation is essential to modular programming.

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:

```
//output.h
#ifndef OUTPUT_H
#define OUTPUT_H
#include <rarray>
#include <string>
void writeBinary(const std::string& s, int n, const rarray<double,1>& x);
void writeText(const std::string& s, int n, const rarray<double,1>& x);
#endif
```

We expect to see these in your homework.

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`.

C++20 modules

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

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

See: Colloquium on “C++20 Modules”, SciNet YouTube Channel

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

- 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.

Managing Compilation Complexity

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 uses variables, rules and dependencies to declare how to build the app or library.
- The Makefile also specifies compiler commands, compiler flags, library locations, etc.
- Build programs like this 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.

Example

```
# Makefile for the monolythic hydrogen code
hydrogen: hydrogen.cpp
    g++ -std=c++23 hydrogen.cpp -o hydrogen
```

```
$ make
g++ -std=c++23 hydrogen.cpp -o hydrogen
$ ./hydrogen
Ground state energy=-0.379419
```

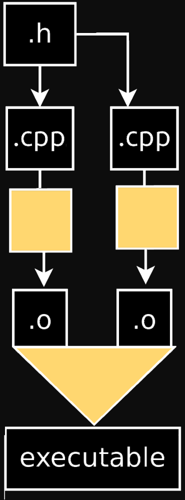
General format:

```
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.
```

Makefile Example



Consider this set of commands:

```
$ g++ -std=c++23 -march=native -Wall -O3  -c -o hydrogen.o hydrogen.cpp
$ g++ -std=c++23 -march=native -Wall -O3  -c -o output.o output.cpp
$ g++ -O3 -o hydrogen output.o hydrogen.o
```

This can be encoded into this Makefile:

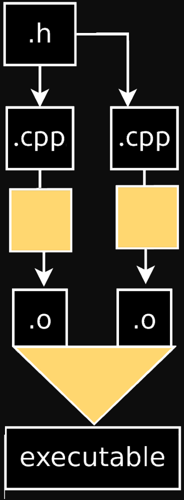
```
# Makefile
hydrogen: output.o hydrogen.o
    g++ -O3 -o hydrogen output.o hydrogen.o

output.o: output.cpp output.h
    g++ -std=c++23 -march=native -Wall -O3  -c -o output.o output.cpp

hydrogen.o: hydrogen.cpp output.h
    g++ -std=c++23 -march=native -Wall -O3  -c -o hydrogen.o hydrogen.cpp
```

which will build what is needed when running make.

Rules - commands




```
# Makefile
hydrogen: output.o hydrogen.o
	g++ -O3 -o hydrogen output.o hydrogen.o

output.o: output.cpp output.h
	g++ -std=c++23 -march=native -Wall -O3  -c -o output.o output.cpp

hydrogen.o: hydrogen.cpp output.h
	g++ -std=c++23 -march=native -Wall -O3  -c -o hydrogen.o hydrogen.cpp
```

- Each command is executed by a separate shell 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.

Macros & Variables

- Macros are the variables or function of Makefile
- Variables can hold simple string definitions, like `CXX = g++`.
  The macro `CXX` is typically used in makefiles to refer to the location of the C++ compiler

```
# Makefile
CXX=g++
hydrogen: output.o hydrogen.o
    $(CXX) -O3 -o hydrogen output.o hydrogen.o

output.o: output.cpp output.h
    $(CXX) -std=c++23 -march=native -Wall -O3  -c -o output.o output.cpp

hydrogen.o: hydrogen.cpp output.h
    $(CXX) -std=c++23 -march=native -Wall -O3  -c -o hydrogen.o hydrogen.cpp
```

- Macros in makefiles may be overridden by the command-line arguments passed to the Make utility (e.g. "`make CXX=clang++`").
- To use macros, you need to use a dollar sign ($) followed by the name of the variable in parentheses.
- Environment variables are also available as macros.

Extended Makefile Example

```
# Example Makefile for the `hydrogen` program (after modularization)
CXX=g++
CXXFLAGS=-std=c++23 -march=native -O3 -Wall
LDFLAGS=-O3

all: hydrogen

hydrogen: hydrogen.o output.o initmatrix.o eigenvalue.o
    $(CXX) $(LDFLAGS) -o hydrogen hydrogen.o output.o initmatrix.o eigenvalue.o
hydrogen.o: hydrogen.cpp output.h initmatrix.h eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o hydrogen.o hydrogen.cpp
output.o: output.cpp output.h
    $(CXX) -c $(CXXFLAGS) -o output.o output.cpp
initmatrix.o: initmatrix.cpp initmatix.h
    $(CXX) -c $(CXXFLAGS) -o initmatrix.o initmatrix.cpp
eigenvalue.o: eigenvalue.cpp eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o eigenvalue.o eigenvalue.cpp

clean:
    $(RM) eigenvalue.o initmatrix.o output.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 Makefile Example with Variables

```
# Example Makefile for the `hydrogen` program (after modularization)
CXX=g++
CXXFLAGS=-std=c++23 -O3 -Wall
LDFLAGS=-O3

all: hydrogen

hydrogen: hydrogen.o output.o initmatrix.o eigenvalue.o
    $(CXX) $(LDFLAGS) -o $@ $^
hydrogen.o: hydrogen.cpp output.h initmatrix.h eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o $@ $<
output.o: output.cpp output.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 output.o hydrogen.o

.PHONY: all clean
```