Modular Programming

Ramses van Zon

PHY1610 Winter 2024

Modularity

Why does modularity 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
// 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.

```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[]);
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.

```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[]);
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!

```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
}
```

Creating the module

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

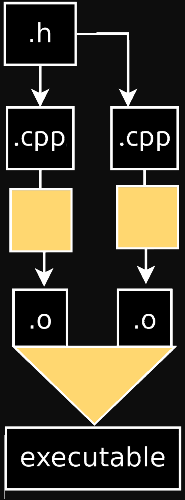
```
//outputarray.h
#include <string>
void writeBinary(const std::string& s, int n, const double x[]);
void writeText(const std::string& s, int n, const double x[]);
```

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:

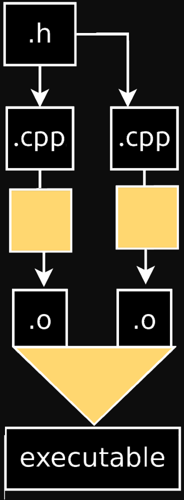
```
//hydrogen.cpp
#include "outputarray.h"

//...

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)?

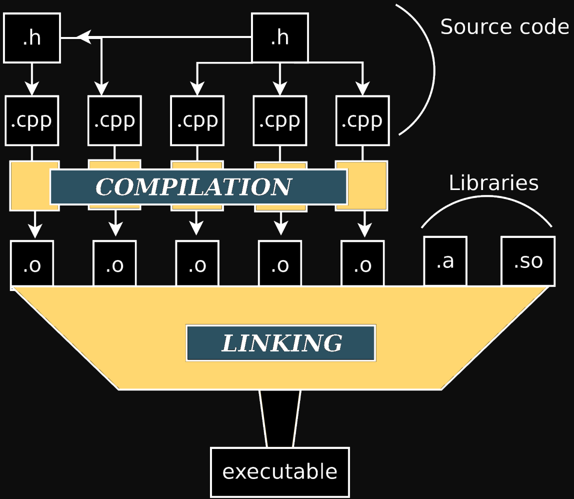
What should one expect in a source file?

- 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.

Modularity allows faster compilation

Modularity 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!

Assignment 2

Your assignment

For this week's assignment, you will be:

- modularizing an existing, monolythic code
- automating its build process with a Makefule
- adding some functionality as a module

The monolythic code to start with compute a 1D version of Conway's Game of Life.

The Code 1/5

```cpp
// gameof1d.cpp
//
// This code computes the evolution of a one-dimensional variant of Conway's Game of Life,
// as conceived by Jonathan K. Millen and published in BYTE magazine in December, 1978.
//
// This system is a linear set of cells that are either "alive" or "dead".
// Time in this system progresses in discrete steps.
//
// The state of each cell at the next time step depends on its present
// state and that of its neighbors, in the following way:
//
//   - First, count the number of alive neighbors at most 2 cells away
//   - An alive cell stays alive if that count is 2 or 4, else it dies
//   - A dead cell becomes alive if that count is 2 or 3, else it stays dead.
//
// Since the first two and the last two cells would not have enough neighbours to apply
// this rule, we use cells on the other side as neighbours, i.e., 'periodic boundaries'.
//
// The initial state of this system is constructed with a given fraction of alive
// cells which are (nearly) equally spaced among dead cells.
//
// The code computes the time evolution for 120 steps, and for each step, prints out
// a line with a representation of the state and fraction of alive cells.
//
```

The Code 2/5

```cpp
// We use bool to store the state of each cell, but for convenience define the following
const bool alive = true;
const bool dead = false;
using Cells = std::unique_ptr<bool[]>;
// Determine next state of a single cell based on the state of all the cells
bool next_cell_state(const Cells& cell_state, int cell_index, int num_cells) {
    // the modulus operator (%) enforces periodic boundary conditions
    int alive_neighbours = 0;
    for (int j = 1; j <= 2; j++) {
        if (cell_state[(cell_index+j+num_cells)%num_cells] == alive)
            alive_neighbours++;
        if (cell_state[(cell_index-j+num_cells)%num_cells] == alive)
            alive_neighbours++;
    }
    if (cell_state[cell_index] == alive  and
        (alive_neighbours == 2 or alive_neighbours == 4))
        return alive;
    else if (cell_state[cell_index] == dead  and
               (alive_neighbours == 2 or alive_neighbours == 3))
        return alive;
    else
        return dead;
}
```

The Code 3/5

```
int main(int argc, char* argv[]) {
    // Set default simulation parameters then accept command line arguments
    int num_cells = 70;
    int num_steps = 120;
    double target_alive_fraction = 0.35;
    try {
        if (argc > 1)
            num_cells = std::stoi(argv[1]);
        if (argc > 2)
            num_steps = std::stoi(argv[2]);
        if (argc > 3)
            target_alive_fraction = std::stod(argv[3]);
    } catch(...) {
        std::cout <<
            "Computes a 1d version of Conway's game of life\n\n"
            "Usage:\n"
            "  gameof1d [-h | --help] | [NUMCELLS [NUMSTEPS [FRACTION]]]\n\n";
        if (std::string(argv[1]) != "-h" and std::string(argv[1]) !="--help") {
            std::cerr << "Error in arguments!\n";
            return 1;
        } else
            return 0;
    }
```

The Code 4/5

```
    // Simulation creation
    Cells cell(std::make_unique<bool[]>(num_cells));
    // Fill cells such that the fraction of alive cells is approximately target_alive_fraction.
    double fill = 0.0;
    for (int i = 0; i < num_cells; i++) {
        fill += target_alive_fraction;
        if (fill >= 1.0) {
            cell[i] = alive;
            fill -= 1.0;
        } else
            cell[i] = dead;
    }
    // Output time step, state of cells, and fraction of alive cells
    const char on_char = 'I', off_char = '-';
    double alive_fraction = 0.0;
    for (int i = 0; i < num_cells; i++)
        if (cell[i] == alive)
            alive_fraction++;
    alive_fraction /= num_cells;
    std::cout << 0 << "\t";
    for (int i = 0; i < num_cells; i++)
        if (cell[i] == alive)
            std::cout << on_char;
```

The Code 5/5

```
        else
            std::cout << off_char;
    std::cout << " " << alive_fraction << "\n";
    // Evolution loop
    for (int t = 0; t < num_steps; t++) {
        // Update cells
        Cells newcell = std::make_unique<bool[]>(num_cells);
        for (int i = 0; i < num_cells; i++)
            newcell[i] = next_cell_state(cell, i, num_cells);
        std::swap(cell, newcell);  // swap without a copy
        // Output time step, state of cells, and fraction of alive cells
        alive_fraction = 0.0;
        for (int i = 0; i < num_cells; i++)
            alive_fraction += cell[i];
        alive_fraction /= num_cells;
        std::cout << t+1 << "\t";
        for (int i = 0; i < num_cells; i++)
            if (cell[i] == alive)
                std::cout << on_char;
            else
                std::cout << off_char;
        std::cout << " " << alive_fraction << "\n";
    }
} // end main
```

To Do