Libraries and Testing

Ramses van Zon

PHY1610, Winter 2025

Libraries

Code is bad

There is a big different in the way scientists view code and the way software developer view it.

Scientists	Software developer
Code is an asset.	**Code is a liability.**

- Every line of code you write has potential issues now or in the future and needs to be maintained.
- Scientists will often come up with quick and dirty solutions to get results, which causes headaches later in the development process: Technical Debt.
- Furthermore there is a lot of code that has already been written and that can be reused, so you might be reinventing the wheel.

The solution is to code less!

Reuse and recycle code that is out there by using **libraries**.

Libraries are modules

- So let's start with a modular code.
- Several object files for different modules that need to be linked together.
- Example: `thisapp.cpp` contains the main function and `helper.cpp/helper.h` are a module.

```
# makefile for 'thisapp'
CXX=g++
CXXFLAGS=-O3 -Wall -std=c++17
all: thisapp

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

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

thisapp: thisapp.o helper.o
  $(CXX) -o thisapp thisapp.o helper.o
```

- To reuse the module, copy `helper.cpp/.h`
- What if we could use it in another project called without recompiling `helper.cpp`?
- Install .o and .h to separate directories:
  `helper.h -> /base/include/helper.h`
  `helper.o -> /base/lib/helper.o`
- Must let compiler know where they are: Add `-I` flag for include directories.

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -O3 -Wall -std=c++17
all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) -o newapp newapp.o /base/lib/helper.o
```

Libraries, continued

What we just did is a poor man's library building.

Real libraries are similar; they have

- to be installed (and perhaps built first)
- header files (`.h` or `.hpp`) in some folder
- library files (object code) in a related folder.

Library filenames start with `lib` & end in `.a`/`.so`.

To avoid explict paths in makefile rules, we specify:

- the path to the library's object using the `-L` option in the `LDFLAGS` variable;
- the object code using `-lNAME` (a lower case `l`!) stored in variable `LDLIBS`.

We're not getting into creating your own libraries here, which requires some system-dependent specialized linking commands.

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -O3 -Wall -std=c++17
all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) -o newapp newapp.o /base/lib/libhelper.a
```

```
# makefile for 'newapp'
CXX=g++
CXXFLAGS=-I/base/include -O3 -Wall -std=c++17
LDFLAGS=-L/base/lib
LDLIBS=-lhelper
all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) $(LDFLAGS) -o newapp newapp.o $(LDLIBS)
```

Libraries, once more

Adding a clean rule and extracting the common path, the Makefile for newapp will look like this:

```
# makefile for 'newapp'
CXX=g++
HELPERBASE?=/base/
HELPERINC=$(HELPERBASE)include
HELPERLIB=$(HELPERBASE)lib
CXXFLAGS=-I$(HELPERINC) -O3 -Wall -std=c++17
LDFLAGS=-L$(HELPERLIB)
LDLIBS=-lhelper

all: newapp

newapp.o: newapp.cpp
  $(CXX) $(CXXFLAGS) -c -o newapp.o newapp.cpp

newapp: newapp.o
  $(CXX) $(LDFLAGS) -o newapp newapp.o $(LDLIBS)

clean:
  $(RM) newapp.o
```

Note:

- C++ standard libaries (`vector`,`cmath`,`...`) do not need any `-l...`'s.
- There are standard directories for libraries that needn't be specified in `-I` or `-L` options (`/usr/include`,`...`)
- Libraries installed through a package manager end up in standard paths; they just need `-l...` options in LDLIBS.
- You also do not need `-I` or `-L` for libraries accessed using the 'module load' command on the Teach or Niagara clusters.
- If you compile your own libraries in non-standard locations, you do need `-I` and `-L` options.

Installing libraries from source

What to do when your package manager does not have that library, or you do not have permission to install packages in the standard paths?

Or, what if you are on SciNet systems (where you do not have permissions to install using the package manager) and there isn't a module for that library already?

Compile from source code with a "base" or "prefix" directory.

Common installation procedure (but read documentation!):

```
$ ./configure --help
$ ./configure --prefix=<BASE>
$ make -j 4
$ make install
```

```
$ mkdir builddir && cd builddir
$ cmake .. -DCMAKE_INSTALL_PREFIX=<BASE>
$ make -j 4
$ make install
```

You choose the `<BASE>`, but it should be a directory that you have write permission to, e.g., a subdirectory of your **$HOME**. These are "non-standard" installation directories.

If the documentation says to do **sudo**, **it is wrong** except for system-wide installations on personal computers.

Using Libraries

- Include its header file(s) in your code.
- Link with `-lLIBNAME`.
- Non-standard installation directory? You need `-I<BASE>/include` and `-L<BASE>/lib` options.
- Alternatively, you can omit these for g++ under linux by setting some environment variables:

```
export CPATH="$CPATH:<BASE>/include"                     # compiler looks here for include files
export LIBRARY_PATH="$LIBRARY_PATH:<BASE>/lib"           # and here for library files
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:<BASE>/lib" # runtime linker looks here
```

  Eenter these commands on the linux prompt before `make` or add to your `~/.bashrc`.

- The `LD_LIBRARY_PATH` is necessary to run application linked against dynamic libraries (`.so`).
- If the library installs binary applications (i.e. commands) as well, you'll also need to set

```
export PATH="$PATH:<BASE>/bin"     # linux shell looks for executables here
```

- **Read the documentation** that came with the library (before searching the web)!

Library Example: GNU Scientific Library

GNU Scientific Library (GSL)

Is a C library containing many useful scientific routines, such as:

- Root finding
- Minimization
- Sorting
- Integration, differentiation, interpolation, approximation
- Statistics, histograms, fitting
- Monte Carlo integration, simulated annealing
- ODEs
- Polynomials, permutations
- Special functions
- Vectors, matrices

Note: C library means we'll likely need to deal with some pointers and casts.

GSL root finding example

Suppose we want to find where $f(x) = a\cos(\sin(v + wx)) + bx - cx^2$ is zero (a *“root”*).

```cpp
// gslrx.cpp
#include <iostream>
#include <gsl/gsl_roots.h>

struct Params {
 double v, w, a, b, c;
};
double examplefunction(double x, void* param){
 Params* p = reinterpret_cast<Params*>param;
 return p->a*cos(sin(p->v+p->w*x))+p->b*x-p->c*x*x;
}

int main() {
 double x_lo = -4.0;
 double x_hi = 5.0;
 Params args = {0.3, 2/3.0, 2.0, 1/1.3, 1/30.0};
 gsl_root_fsolver* solver;
 gsl_function      fwrapper;
 solver = gsl_root_fsolver_alloc(
              gsl_root_fsolver_brent);
```

```cpp
 fwrapper.function = examplefunction;
 fwrapper.params = &args;
 gsl_root_fsolver_set(solver,&fwrapper,x_lo,x_hi);

 std::cout << "iter lower upper root err\n";

 int status = 1;
 for (int iter=0; status and iter < 100; ++iter) {
   gsl_root_fsolver_iterate(solver);
   double x_rt = gsl_root_fsolver_root(solver);
   double x_lo = gsl_root_fsolver_x_lower(solver);
   double x_hi = gsl_root_fsolver_x_upper(solver);
   std::cout << iter <<" "<< x_lo <<" "<< x_hi
             <<" "<< x_rt <<" "<<x_hi-x_lo<<"\n";
   status=gsl_root_test_interval(x_lo,x_hi,0,1e-3);
 }

 gsl_root_fsolver_free(solver);
 return status;
}
```

Compilation and linkage

- Lots of `gsl...` stuff.
- All of the algorithms come from the GSL.
- The rest is just wrappers, setting up parameters and calling the appropriate functions.
- There are pointers and typecasts, because we're dealing with a C library.
- ***How to compile on the command line?***

```
$ module load gcc/13 gsl/2.7.1
$ GSLINC=$MODULE_GSL_PREFIX/include
$ GSLLIB=$MODULE_GSL_PREFIX/lib
$ g++ -c -I$GSLINC gslrx.cpp -o gslrx.o
$ g++ gslrx.o -o gslrx -L$GSLLIB -lgsl -lgslcblas
$ ./gslrx
```

Output

```
$ ./gslrx
iter lower      upper    root      err
0    -4        -1.27657 -1.27657  2.72343
1    -1.95919  -1.27657 -1.95919  0.682622
2    -1.75011  -1.27657 -1.75011  0.473542
3    -1.75011  -1.74893 -1.74893  0.0011793
$
```

GSL Makefile usage

```
CXX=g++
GSL_MODULE_PREFIX?=.
GSLINC?=$(MODULE_GSL_PREFIX)/include
GSLLIB?=$(MODULE_GSL_PREFIX)/lib
CXXFLAGS=-I$(GSLINC) -O3 -Wall -std=c++17
LDFLAGS=-L$(GSLLIB)
LDLIBS=-lgsl -lgslcblas

all: gslrx
.PHONY: all clean

gslrx.o: gslrx.cpp
  $(CXX) $(CXXFLAGS) -c -o gslrx.o gslrx.cpp

gslrx: gslrx.o
  $(CXX) $(LDFLAGS) -o gslrx gslrx.o $(LDLIBS)

clean: ; $(RM) gslrx.o
```

Compilation on Teach cluster:

```
$ module load gcc/13 gsl/2.7.1
$ make
```

Compilation on your own computer:

```
$ export GSLINC=... # whereever headers are
$ export GSLLIB=... # whereever libs are
$ make
```

or

```
$ export MODULE_GSL_PREFIX=... # with include & lib
$ make
```

You can also set make variables like this:

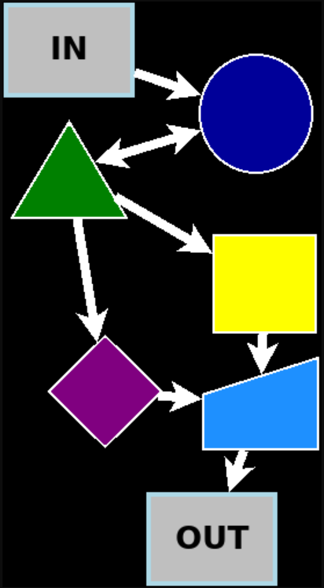
```
$ make MODULE_GSL_PREFIX=... # with include & lib
```

Don’t Reinvent the Wheel

- There are many possible algorithms to implement for root finding.
- But they are all pretty standard.
- Surely, someone must have done this already? Correct!
- The GNU Scientific Library is one such library.
- Don’t implement this yourself if there is a library that does it for you.
- Even existing solutions like the once in the GSL, can’t really be used until you understand the algorthims on a high level.

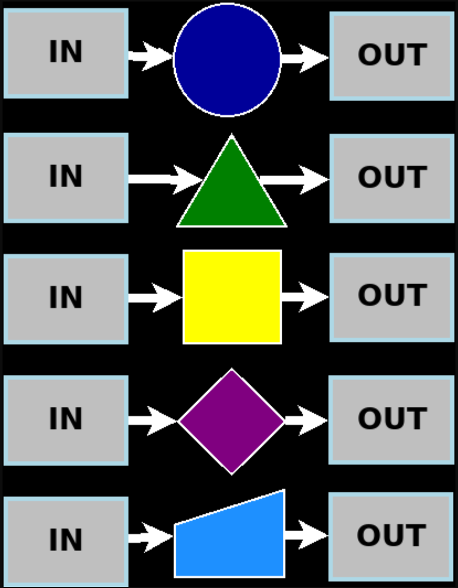
Testing

Integrated testing



- Especially with new software, or old software that was modified, you'll want to verify that it ***works as a whole***.
- Test the application with a smaller test case for which you know that output.
- This can strictly only prove incorrectness (no tests can prove correctness).
- But if no errors are found, it increases your level of confidence in the software.

Unit testing



- An integrated test essentially gives you one data point.
- If you've modularized the code into *n* parts, you should have at least *n* data points to know that the parts aren't failing.
- Because each module has one responsibility, you can write a test for each module.
- If the test for a module fails, you only need to inspect that module, not the whole code of the application.
- Note that if you did not modularize, everything is connected, you could not have *n* tests. And when the integrated test fails, the error could be anywhere in the code.

Remember the example from lecture 5

```cpp
// hydrogen.cpp
#include <iostream>
#include <rarray>
#include "eigenval.h"
#include "output.h"
#include "init.h"
int main() {
    const int n = 4913;
    rmatrix<double> m = initMatrix(n);
    rvector<double> a;
    double e;
    groundState(m, e, a);
    std::cout<<"Ground state energy="<<e<<"\n";
    writeText("data.txt", a);
    writeBinary("data.bin", a);
}
```

```makefile
# Makefile
CXXFLAGS=-std=c++17 -O3 -Wall -g
LDFLAGS=-g
all: hydrogen
hydrogen.o: hydrogen.cpp eigenval.h output.h init.h
eigenval.o: eigenval.cpp eigenval.h
output.o: output.cpp output.h
init.o: init.cpp init.h
hydrogen: hydrogen.o eigenval.o output.o init.o
	$(CXX) $(LDFLAGS) -o $@ $^
clean:
	$(RM) hydrogen.o eigenval.o output.o init.o
```

By the way: Makefile 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

Furthermore, there are built-in make rules, such as making a .o from a .cpp is done with

```
        $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o $@ $<
```

Remember the example from lecture 5

```cpp
// hydrogen.cpp
#include <iostream>
#include <rarray>
#include "eigenval.h"
#include "output.h"
#include "init.h"
int main() {
    const int n = 4913;
    rmatrix<double> m = initMatrix(n);
    rvector<double> a;
    double e;
    groundState(m, e, a);
    std::cout<<"Ground state energy="<<e<<"\n";
    writeText("data.txt", a);
    writeBinary("data.bin", a);
}
```

```makefile
# Makefile
CXXFLAGS=-std=c++17 -O3 -Wall -g
LDFLAGS=-g
all: hydrogen
hydrogen.o: hydrogen.cpp eigenval.h output.h init.h
eigenval.o: eigenval.cpp eigenval.h
output.o: output.cpp output.h
init.o: init.cpp init.h
hydrogen: hydrogen.o eigenval.o output.o init.o
	$(CXX) $(LDFLAGS) -o $@ $^
clean:
	$(RM) hydrogen.o eigenval.o output.o init.o
```

How would we create an integrated test?

Example: Integrated test for hydrogen

1. Create reference output

```
$ g++ -std=c++17 -O3 -g -o hydrogen0 hydrogen0.cpp
$ # or 'make' and 'mv hydrogen hydrogen0'
$ ./hydrogen0 > cout0.txt
$ mv data.txt data0.txt
$ mv data.bin data0.bin
```

2. Run the new modular code

```
$ make hydrogen
$ ./hydrogen > cout.txt
```

3. Compare the outputs

```
$ diff cout.txt cout0.txt
$ diff data.txt data0.txt
$ cmp data.bin data0.bin
```

Automate everything!

4. Store your reference

```
$ git add data0.txt data0.bin cout0.txt
$ git commit -m 'Added original output as reference'
```

5. Add a `integratedtest` rule to the Makefile

```
cout.txt: hydrogen
    hydrogen >  cout.txt
integratedtest: data0.txt data0.bin cout0.txt \
                data.txt data.bin cout.txt
    diff cout.txt cout0.txt
    diff data.txt data0.txt
    cmp data.bin data0.bin
```

6. Always `git commit`

```
$ git add Makefile
$ git commit -m 'Added integratedtest to Makefile'
```

7. `make integratedtest`

Example: Unit test for output module (1/2)

```
// output.h
#ifndef OUTPUTARRH
#define OUTPUTARRH
#include <string>
#include <rarray>
// The writeBinary function writes the 1d rarray
// 'a' to the file 'name' in binary format
void writeBinary(const std::string& name,
                 const rvector<double>& a);
// The writeText function writes the 1d rarray
// 'a' to the file 'name' in ASCII format
void writeText(const std::string& name,
               const rvector<double>& a);
#endif
```

Both `writeBinary` and `writeText` should have at least one unit test.

But let's start with one unit test for writeText.

It could look like this:

```
#include "output.h"
#include <iostream>
#include <fstream>
int main() {
   std::cout << "A UNIT TEST FOR 'writeText'\n";
   // test file writing:
   rvector<double> a(3);
   a = 1, 2, 3;
   writeText("testoutputarr.txt", a);
   // read it back
   std::ifstream in("testoutputarr.txt");
   std::string s[3];
   in >> s[0] >> s[1] >> s[2];
   // check
   if (s[0]!="1" or s[1]!="2" or s[2]!="3") {
       std::cout << "TEST FAILED\n";
       return 1;
   }  else {
       std::cout << "TEST PASSED\n";
       return 0;
   }
```

Example: Unit test for output module (2/2)

Add to makefile:

```
...
test: run_output_test integratedtest

run_output_test:
   ./output_test

output_test: output_test.o output.o
   $(CXX) $(LDFLAGS) -o $@ $^

output_test.o: output_test.cpp output.h
   $(CXX) $(CXXFLAGS) -c -o $@ $<
```

To run:

```
$ make test
g++ ...
g++ ...
./output_test
A UNIT TEST FOR 'writeText'
TEST PASSED
$ echo $?
0
```

Important things to note

- Unit tests are separate from the application!
- The test only depends on output.h and output.o. (test isolation)
- It's a separate program, which requires its own data initialization and checking.
- The 'test' rule runs all tests
- All tests for one module are ideally in one file.
- To automate, we need a consistent way to report errors, a way to run only some tests, etc.: frameworks.

Unit testing frameworks

- There's a lot of extra coding here just to run the tests.
- The tests need to be maintained as well.
- Especially when your project contains a lot of tests,
  use a unit testing framework.

Examples:

- Boost.Test (from the Boost library suite)
- Google C++ Testing Framework (a.k.a googletest)
- Catch2

These are typically combinations of macros, a driver main function that can select which tests to run, etc.

- For the assignment, you should be using Catch2.

Example of Catch2

```cpp
// output_c2.cpp
#include "output.h"
#include <fstream>

#include <catch2/catch_all.hpp>

TEST_CASE("writeText test")
{
    // create file:
    rvector<double> a(3);
    a = 1,2,3;
    writeText("testoutputarr.txt", a);
    // read back:
    std::ifstream in("testoutputarr.txt");
    std::string s[3];
    in >> s[0] >> s[1] >> s[2];
    // check
    REQUIRE(s[0]=="1");
    REQUIRE(s[1]=="2");
    REQUIRE(s[2]=="3");
}
```

```
$ module load gcc/13 catch2/3.3.1

$ g++ -std=c++17 -g -O3 -Wall -c output_c2.cpp
$ g++ -g -O3 -o output_c2 output_c2.o output.o
   -lCatch2Main -lCatch2

$ ./output_c2 -s
```

```
Randomness seeded to: 3824212292
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
output_c2 is a Catch2 v3.3.1 host application.
Run with -? for options
-------------------------------------------------------
writeText test
...
All tests passed (3 assertions in 1 test case)
```

Guidelines for testing

- Each module should have a separate test suite
  (e.g. output_c2.cpp should also have a test for `writeBinary`).
- If the code is properly modular, those module test should not need any of the other .cpp files.
- Each module should have a named target in the Makefile that runs its test suite.

```
run_output_c2:
   ./output_c2 -s
output_c2: output_c2.o outputarr.o
   $(CXX) $(LDFLAGS) -o $@ $^ -lCatch2Main -lCatch2
output_c2.o: output_c2.cpp outputarr.h
   $(CXX) $(CXXFLAGS) -c -o $@ $<
.PHONY: run_output_c2
```

- An overall 'test' target should run all test suites and any integrated tests.
- Testing gives confidence in your module, and tells 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.