Libraries and Testing

Ramses van Zon

PHY1610, Winter 2026

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: `ap1.cpp` contains the main function and `helper.cpp` + `helper.h` are a module.

```
# makefile for 'ap1'
CXX=g++
CXXFLAGS=-O3 -march=native -Wall -std=c++23

ap1: ap1.o helper.o
  $(CXX) -o ap1 ap1.o helper.o

ap1.o: ap1.cpp helper.h
  $(CXX) $(CXXFLAGS) -c -o ap1.o ap1.cpp

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

- To reuse the module, copy `helper.cpp/.h`
- What if we could use it in another project called ap2 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 'ap2'
CXX=g++
CXXFLAGS=-O3 -march=native -Wall -std=c++23
CPPFLAGS=-I/base/include

ap2: ap2.o
  $(CXX) -o ap2 ap2.o /base/lib/helper.o

ap2.o: ap2.cpp
  $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o ap2.o ap2.cpp
```

Libraries, continued

What we created is a "poor man's" library.

Real libraries are similar; they have

- to be installed (and perhaps built first)
- header files (`.h`) in some folder
- library (i.e. object code) in another folder.

Note: 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 covering creating your own libraries here.

```
# makefile for 'ap2'
CXX=g++
CXXFLAGS=-O3 -march=native -Wall -std=c++23
CPPFLAGS=-I/base/include

ap2: ap2.o
	$(CXX) -o ap2 ap2.o /base/lib/libhelper.a

ap2.o: ap2.cpp
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o ap2.o ap2.cpp
```

```
# makefile for 'ap2' using a library
CXX=g++
CXXFLAGS=-O3 -march=native -Wall -std=c++23
CPPFLAGS=-I/base/include
LDFLAGS=-L/base/lib
LDLIBS=-lhelper

ap2: ap2.o
	$(CXX) $(LDFLAGS) -o ap2 ap2.o $(LDLIBS)

ap2.o: ap2.cpp
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o ap2.o ap2.cpp
```

Libraries, once more

Adding an all and clean rule and extracting the common path, the Makefile will look like this:

```
# makefile for 'ap2'
CXX=g++
HELPERBASE=/base/
HELPERINC=$(HELPERBASE)include
HELPERLIB=$(HELPERBASE)lib
CPPFLAGS=-I$(HELPERINC)
CXXFLAGS=-O3 -march=native -Wall -std=c++23
LDFLAGS=-L$(HELPERLIB)
LDLIBS=-lhelper
all: ap2

ap2: ap2.o
	$(CXX) $(LDFLAGS) -o ap2 ap2.o $(LDLIBS)

ap2.o: ap2.cpp
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o ap2.o ap2.cpp

clean:
	$(RM) ap2.o
.PHONY: all clean
```

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 Trillium clusters.
- If you compile your own libraries in non-standard locations, you do need `-I` and `-L` options.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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 "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 your own 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
```

  Enter 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"      # shell looks for executables here
```

- **Read the documentation** that came with the library!

SciNet ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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 need to deal with pointers and type casts.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

GSL installation

```
$ module load gcc/14
```

```
$ mkdir $HOME/buildgsl
```

```
$ cd $HOME/buildgsl
```

```
$ wget ftp://ftp.gnu.org/gnu/gsl/gsl-2.8.tar.gz
```

```
$ tar xzf gsl-2.8.tar.gz
```

```
$ cd gsl-2.8
```

```
$ ./configure CFLAGS="-O2 -march=native -ffp-contract=off" --prefix=$HOME/gsl
```

```
$ make -j4
```

```
$ make check
```

```
$ make install
```

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

GSL root finding example

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

```
// grx.cpp
#include <print>
#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);
 fwrapper.function = examplefunction;
 fwrapper.params = &args;
 gsl_root_fsolver_set(solver,&fwrapper,x_lo,x_hi);

 std::println("iter lower upper root err");

 int status = 1;
 for (int i=0; status != 0 && i < 100; ++i) {
   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::println("{} {} {} {} {}",
                i, x_lo, x_hi, x_rt, x_hi-x_lo);
   status=gsl_root_test_interval(x_lo,x_hi,0,1e-3);
 }

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

Compilation and linkage

- The algorithms come from the GSL.
- The rest is just wrappers, setting up parameters and calling functions.
- There are pointers and casts because it's a C library.
- ***How shoule we compile this?***

```
$ module load gcc/14
$ GSLPREFIX=$HOME/gsl
$ g++ -I$GSLPREFIX/include -O3 -march=native -Wall -c -o grx.o grx.cpp
$ g++ -L$GSLPREFIX/lib -O3 -o grx grx.o -lgsl -lgslcblas
$ ./grx
```

Output

```
$ ./grx
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++
GSLPREFIX?=.
GSLINC=$(GSLPREFIX)/include
GSLLIB=$(GSLPREFIX)/lib
CXXFLAGS=-O3 -march=native -Wall -std=c++23
CPPFLAGS=-I$(GSLINC)
LDFLAGS=-L$(GSLLIB)
LDLIBS=-lgsl -lgslcblas

all: grx

grx.o: grx.cpp
	$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c -o grx.o grx.cpp

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

clean:
	$(RM) grx.o

.PHONY: all clean
```

Compilation on Teach cluster:

```
$ module load gcc/14
$ make GSLPREFIX=$HOME/gsl
```

Note:

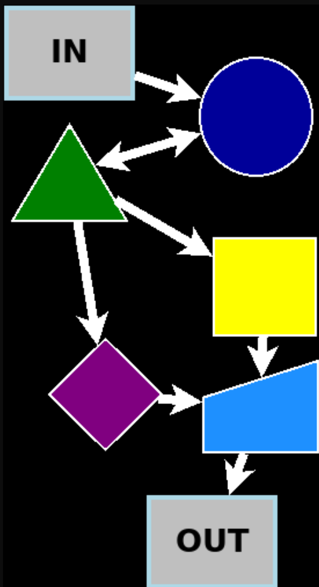
- In the Makefile, if no GSLPREFIX is defined, it will set it to the current directory.
- We could make all -I and -L arguments superfluous by setting

```
export GSLPREFIX=$HOME/gsl
export CPATH="$CPATH:$GSLPREFIX/include"
export LIBRARY_PATH="$LIBRARY_PATH:$GSLPREFIX/lib"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$GSLPREFIX/lib
```

(You could put this in your `$HOME/.bashrc`)

Testing

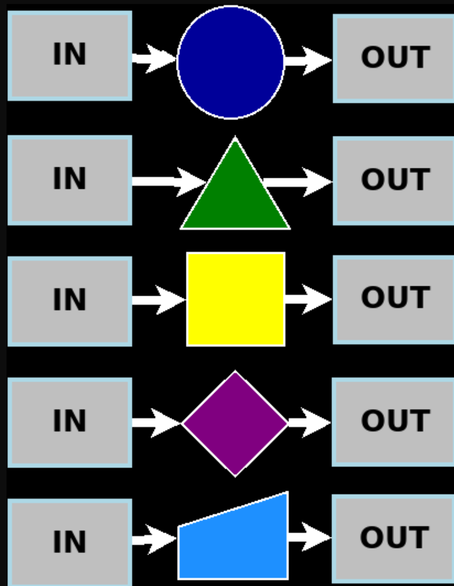
Integrated testing



- 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.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

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.
- 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.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Remember the example from lecture 5/6

```
# Example Makefile for the `hydrogen` program
CXX=g++
CXXFLAGS=-std=c++23 -march=native -O3 -Wall
LDFLAGS=-O3
OBJS=hydrogen.o output.o init.o eigen.o
all: hydrogen

hydrogen: $(OBJ)
	$(CXX) $(LDFLAGS) -o hydrogen $(OBJ)
hydrogen.o: hydrogen.cpp output.h init.h eigen.h
	$(CXX) -c $(CXXFLAGS) -o hydrogen.o hydrogen.cpp
output.o: output.cpp output.h
	$(CXX) -c $(CXXFLAGS) -o output.o output.cpp
init.o: init.cpp initmatix.h
	$(CXX) -c $(CXXFLAGS) -o init.o init.cpp
eigen.o: eigen.cpp eigen.h
	$(CXX) -c $(CXXFLAGS) -o eigen.o eigen.cpp

clean:
	$(RM) $(OBJS)

.PHONY: all clean
```

The main code may look something like this:

```
// hydrogen.cpp
#include <print>
#include <rarray>
#include "eigen.h"
#include "output.h"
#include "init.h"
int main() {
    const int n = 4913;
    rarray<double,2> m = initMatrix(n);
    rarray<double,1> a;
    double e;
    groundState(m, e, a);
    println("Ground state energy={}", e);
    writeText("data.txt", a);
    writeBinary("data.bin", a);
}
```

How would we create an integrated test?

Example: Integrated test for hydrogen

1. Create reference output

```
$ g++ -std=c++23 -O3 -march=native -o hydrogen0 hydrogen0.cpp
$ ./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
```

4. Store your reference

```
$ git add data0.txt data0.bin cout0.txt
$ git commit -m 'Added original output 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
.PHONY: integratedtest
```

7. `make integratedtest`

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

```cpp
// 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 rarray<double,1>& a);
// The writeText function writes the 1d rarray
// 'a' to the file 'name' in ASCII format
void writeText(const std::string& name,
               const rarray<double,1>& 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:

```cpp
// output_t.cpp
#include "output.h"
#include <print>
#include <fstream>
int main() {
   std::println("A UNIT TEST FOR 'writeText');
   // test file writing:
   auto a = ra::make_rarray<double>({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" || s[1]!="2" || s[2]!="3") {
       std::println("TEST FAILED");
       return 1;
   } else {
       std::println("TEST PASSED");
       return 0;
   }
```

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

Add to makefile:

```
check: run_output_test integratedtest

run_output_test: output_t
	./output_t

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

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

.PHONY: run_output_test integratedtest check
```

To run:

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

Important notes:

- Unit tests are separate from the application!
- The unit test should be isolated and only depend on output.h and output.o.
- It's a separate program, which may require its own data initialization.
- The 'check' rule (sometimes 'test' rule) runs all tests.
- All tests for one module 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 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

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

#include <catch2/catch_all.hpp>

TEST_CASE("writeText test")
{
  // create file:
  auto a = ra::make_rarray<double>({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/14.3 catch2/3.3.1

$ g++ -std=c++23 -O3 -march=native -c output_t.cpp
$ g++ -O3 -march=native -o output_t output_t.o \
    output.o -lCatch2Main -lCatch2

$ ./output_t -s
```

```
Randomness seeded to: 3824212292
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
output_t 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_t.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_test: output_t
	./output_t -s
output_t: output_t.o output.o
	$(CXX) $(LDFLAGS) -o $@ $^ -lCatch2Main -lCatch2
output_t.o: output_t.cpp output.h
	$(CXX) $(CXXFLAGS) -c -o $@ $<
.PHONY: run_output_test
```

- An overall '`check`' or '`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.

SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO