# Testing and Debugging

Ramses van Zon

PHY1610H Winter 2023

# Motivation

# Three bits of reality about scientific software:

- **Scientific software can be large, complex and subtle.**
- **Scientific software is constantly evolving.**
- **Code will be handed down, shared, reused.**

## Example of this complexity

Consider the sample code to simulate a damped wave equation in one dimension. It had to

1. Read parameters;
2. Set initial conditions;
3. Compute the evolution of the wave in time;
4. Output the result.

At some point in the research project, initial conditions may need to change, or the output, or the algorithm to compute the time evolution, . . .

# Managing complexity using modularity

- Modularity is extracing the different parts of the program that are responsible for different things.
- Each of these should be fairly independent.
- Implementation changes of one module should not affect other modules.
- Each part can be maintained by a different person.
- Once a part is working well, it can be used as an appliance.

# Questions

1. How do we ensure a module works correctly?

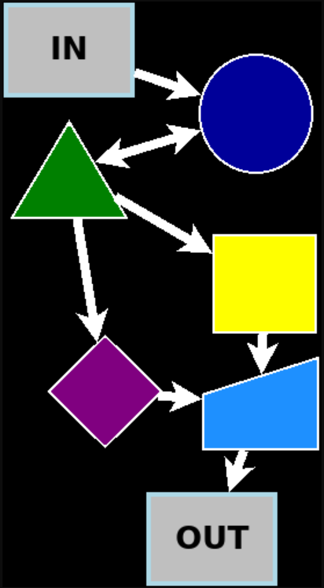
   **Unit testing**

2. What if we find that it doesn't?

   **Debugging**

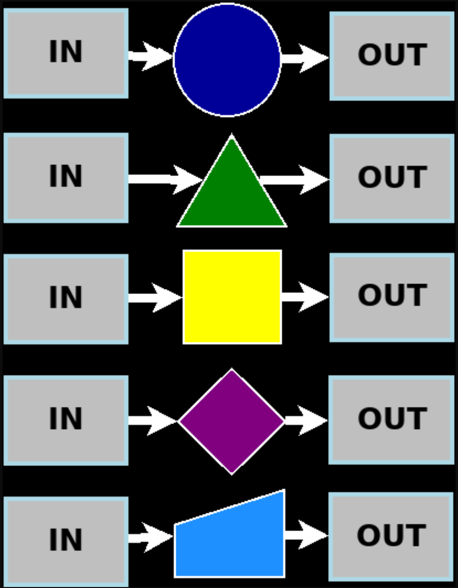
# Unit testing

# 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.

# Example from lecture 5 (modular)

```cpp
// hydrogen.cpp
#include <iostream>
#include <rarray>
#include "eigenval.h"
#include "outputarr.h"
#include "initmat.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 -O2 -g
LDFLAGS=-g
all: hydrogen
hydrogen.o: hydrogen.cpp eigenval.h outputarr.h initm
eigenval.o: eigenval.cpp eigenval.h
outputarr.o: outputarr.cpp outputarr.h
initmat.o: initmat.cpp initmat.h
hydrogen: hydrogen.o eigenval.o outputarr.o initmat.o
    $(CXX) $(LDFLAGS) -o $@ $^
clean:
    $(RM) hydrogen.o eigenval.o outputarr.o initmat.o
```

How would we create an integrated test?

# Example: Integrated test for hydrogen

1. Create reference output

```
$ g++ -std=c++17 -O2 -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 outputarr module (1/2)

```
// outputarr.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 "outputarr.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 outputarr module (2/2)

Add to makefile:

```
...
test: run_outputarr_test integratedtest

run_outputarr_test:
   ./outputarr_test

outputarr_test: outputarr_test.o outputarr.o
   $(CXX) $(LDFLAGS) -o $@ $^

outputarr_test.o: outputarr_test.cpp outputarr.h
   $(CXX) $(CXXFLAGS) -c -o $@ $<
```

To run:

```
$ make test
g++ ...
g++ ...
./outputarr_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 outputarr.h and outputarr.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.

# 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, if you're going to use a framework, use Catch2.

# Example of Boost.Test

```cpp
// output_bt.cpp
#include "outputarr.h"
#include <fstream>
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE output_bt
#include <boost/test/unit_test.hpp>
BOOST_AUTO_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
    BOOST_CHECK(s[0]=="1");
    BOOST_CHECK(s[1]=="2");
    BOOST_CHECK(s[2]=="3");
}
```

```
$ module load gcc/12 boost

$ g++ -std=c++17 -g -O2 -c output_bt.cpp
$ g++ -g -O2 -o output_bt output_bt.o outputarr.o
   -lboost_unit_test_framework

$ ./output_bt --log_level=all
```

```
Running 1 test case...
Entering test module "output_bt"
output_bt.cpp(7): Entering test case "writeText_test"
output_bt.cpp(18): info: check s[0]=="1" has passed
output_bt.cpp(19): info: check s[1]=="2" has passed
output_bt.cpp(20): info: check s[2]=="3" has passed
output_bt.cpp(7):
Leaving test module "output_bt"; testing time: 521us
*** No errors detected
```

# Example of Catch2

```cpp
// output_c2.cpp
#include "outputarr.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/12 catch2/3.3.1

$ g++ -std=c++17 -g -O2 -c output_c2.cpp
$ g++ -g -O2 -o output_c2 output_c2.o outputarr.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.

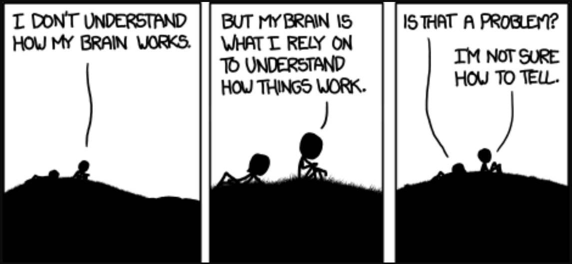
```
run_output_c2:
   ./output_c2 -s
output_c2: output_c2.o outputarr.o
   $(CXX) $(LDFLAGS) -o $@ $^ -lCatch2Main -lCatch2
output_c2.o: output_c2.cpp outputarr.h
   $(CXX) $(CXXFLAGS) -c -o $@ $<
.PHONY: run_output_c2
```

- 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.

# Debugging

# What if your program or test isn't running correctly. . .

- Nonsense. All programs execute "correctly'".
- We just told it to do the wrong thing.
- Debugging is the *art* of reconciling your mental model of what the code is doing with what you actually told it to do.




https://imgs.xkcd.com/comics/debugger.png

**Debugger: program to help detect errors in other programs.**

# Tips to avoid debugging

- Write better code.
  - simple, clear, straightfoward code.
  - modularity (avoid global variables and 10,000 line functions).
  - avoid "cute tricks'', (**no** obfuscated C code winners – IOCCC).
- Don't write code, use existing libraries.
- Write (simple) tests for each module.
- Use version control and small commits.
- Switch on the `-Wall` flag, inspect all warnings, fix them or understand them all.
- Use defensive programming:

  Check arguments, use assert (which can be switched of with `-DNDEBUG` compilation flag)
  E.g.:

```
#include <cassert>
#include <cmath>
double mysqrt(double x) {
    assert(x>=0);
    return sqrt(x);
}
```

# Despite that, still errors?

Some common issues:

| | |
|---|---|
| Arithmetic | Corner cases (`sqrt(-0.0)`), infinities |
| Memory access | Index out of range, uninitialized pointers |
| Logic | Infinite loop, corner cases |
| Misuse | Wrong input, ignored error, no initialization |
| Syntax | Wrong operators/arguments |
| Resource starvation | Memory leak, quota overflow |
| Parallel | Race conditions, deadlock |

# Debugging workflows

- As soon as you are convinced there is a real problem, create the simplest situation in which it repeatedly occurs.
- Take a scientific approach: model, hypothesis, experiment, conclusion.
- Try a smaller problem size, turning off different physical effects with options, etc, until you have a simple, fast, repeatable example.
- Try to narrow it down to a particular module/function/class.
- Integrated calculation: Write out intermediate results, inspect them.

# Ways to debug

**To figure out what is going wrong, and where in the code, we can**

- Put strategic print statements in the code.
- Use a debugger.

# What's wrong with using print statements?

## Strategy

- Constant cycle:
  - strategically add print statements
  - compile
  - run
  - analyze output
  - repeat
- Removing the extra code after the bug is fixed
- Repeat for each bug

## Problems with this approach

A bug is always unexpected, so you don't know where to put those strategic print statements.

As a result, this approach:

- is time consuming
- is error prone (print statements can have bugs)
- changes memory layout, output format, timing

  ...

**There's a better way!**

# Debuggers

are programs that can show what happens in a program at runtime.

## Features

1. Crash inspection
2. Function call stack
3. Step through code
4. Automated interruption
5. Variable checking and setting

## Use a graphical/IDE debugger or not?

- Local work station: graphical/IDE is convenient
- Remotely (SciNet): can be slow or hard to set up.
- In any case, graphical and text-based debuggers use the same concepts.

# Debuggers

## Preparing the executable for debugging

- Add required compilation flags, `-g` (sometimes `-g -gstabs`)
  (both in compiling and linking!)
- Optional: switch off optimization `-O0`

## Command-line based symbolic debugger: gdb

- Free, GNU license, symbolic debugger.
- Available on many systems.
- Been around for a while, but still developed and up-to-date
- Command-line based, does not show code listing by default, unless you use the `-tui` option.

# Example

```
$ module load gcc/12 rarray/2.4 gdb/10
$ gdb -tui hydrogen
```

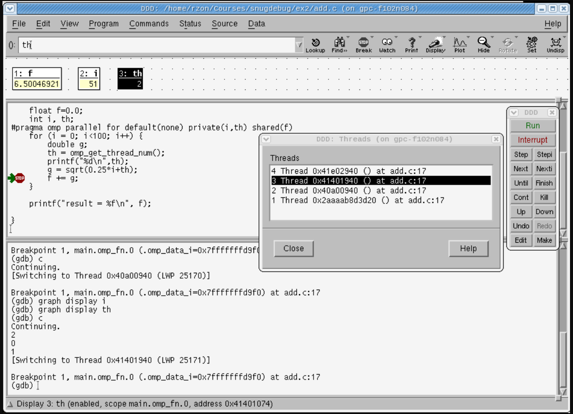
```
lcl_uotphy1610s2192@teach01:~/modular

                    [ No Source Available ]

None No process In:                                         L??   PC: ??
GNU gdb (GDB) 10.2
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
--Type <RET> for more, q to quit, c to continue without paging--
```

# GDB command summary

| | | |
|---|---|---|
| `help` | `h` | print description of command |
| `run` | `r` | run from the start (+args) |
| `backtrace/where` | `ba` | function call stack |
| `break` | `b` | set breakpoint |
| `delete` | `d` | delete breakpoint |
| `continue` | `c` | continue |
| `list` | `l` | print part of the code |
| `step` | `s` | step into function |
| `next` | `n` | continue until next line |
| `print` | `p` | print variable |
| `display` | `disp` | print variable at every prompt |
| `finish` | `fin` | continue until function end |
| `set variable` | `set var` | change variable |
| `down` | `do` | go to called function |
| `until` | `unt` | continue until line/function |
| `up` | `up` | go to caller |
| `watch` | `wa` | stop if variable changes |
| `quit` | `q` | quit gdb |

# Graphical debuggers

DDD: free, bit old, can do serial and threaded debugging.




DDT: commercial, on SciNet, good for parallel debugging (including mpi and cuda)