

# Debugging and Building

Ramses van Zon

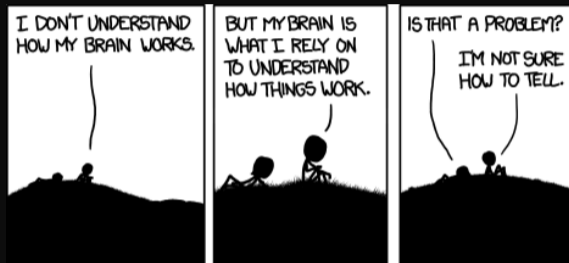
PHY1610H Winter 2024



# 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, straightforward 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 off 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 ( $\text{sqrt}(-0.0)$ ), 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

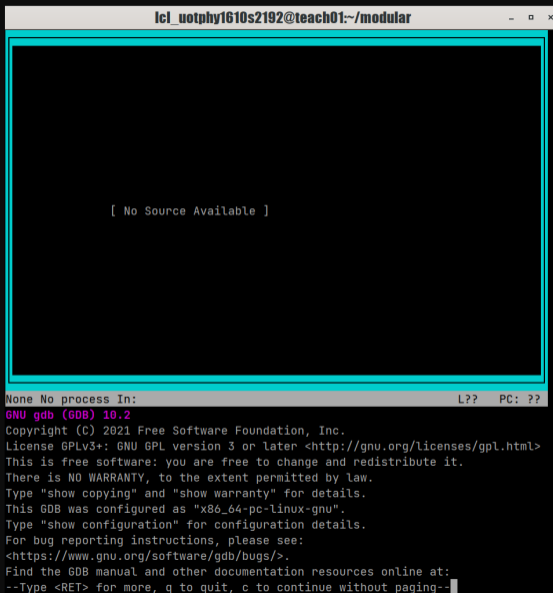
- Add required compilation flags, `-g`  
(both in compiling and linking!)
- Recommended: switch off optimization `-O0`  
(Recommended for production: `-O3 -march=native`)

## 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: Hydrogen Atom Ground State

```
$ module load gcc/13 rarray/2.7 gdb/13
$ g++ -g -O0 eigenval.cpp initmat.cpp \
  hydrogen.cpp outputarr.cpp -o hydrogen
$ 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 beginning (+args)
start	start	run from main
backtrace	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



# Code in multiple source file

# 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 relatively simple hydrogen code. It has to :

- 1 Create a matrix
- 2 Compute its eigenvalues
- 3 Output the result
- 4 And some code has to put it altogether.

At some point in the research project, any of these aspects may need to change independently. ...

# Managing complexity using modularity

- Modularity is extracting the different parts of the program that are responsible for different things into different files.
- 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.

We'll discuss modularity more in detail in the next lecture.

First, we need to discuss how to compile such multifile codes.



# Make



# 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** contains variables, rules and dependencies.
- The **Makefile** specifies executables, compiler flags, library locations, ...
- Build programs are a crucial component of *professional software development*.

[https://www.gnu.org/software/make/manual/html\\_node/index.html](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

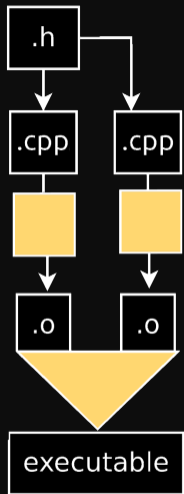
- A Makefile is a plain text file consisting of **rules**.
- Each rule begins with a textual dependency line which defines a **target** followed by a colon (:) and optionally an enumeration of **prerequisites** (files or other targets) on which the target depends.
- The **dependency line** is arranged 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.

```
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.
```

# Simple Makefile Example



Consider this set of commands:

```
$ g++ -std=c++17 -O2 outputarray.cpp -c -o outputarray.o
$ g++ -std=c++17 -O2 hydrogen.cpp -c -o hydrogen.o
$ g++ -O2 outputarray.o hydrogen.o -o hydrogen
```

Here, `-O2` stands for **Optimization level 2**.

This option makes the compiler create faster machine code.  
Do this unless you know why you shouldn't.

This can be encoded into this Makefile:

```
# Makefile
hydrogen: outputarray.o hydrogen.o
    g++ -O2 outputarray.o hydrogen.o -o hydrogen

outputarray.o: outputarray.cpp outputarray.h
    g++ -std=c++17 -O2 outputarray.cpp -c -o outputarray.o

hydrogen.o: hydrogen.cpp outputarray.h
    g++ -std=c++17 -O2 hydrogen.cpp -c -o hydrogen.o
```

which will build what is needed when running `make`.

# Rules - commands

- Each command is executed by a separate shell or command-line interpreter 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.

## Need multiple commands?

- The backslash \ can be used to have commands executed by the same shell, it represents line-continuation
- Commands can be separated by ;

# Macros & Variables

- Macros are usually referred to as variables when they hold simple string definitions, like `CXX = g++`.
- Macros in makefiles may be overridden by the command-line arguments passed to the Make utility (e.g. “make CXX=icpc”).
- Macros allow users to specify the programs invoked and other custom behavior during the build process.

For example, the macro `CXX` is used in makefiles to refer to the location of the C++ compiler

- To use variables, you need to use a dollar sign (\$) followed by the name of the variable in parenthesis (or curly braces).

## Examples

```
MACRO = definition
```

```
PACKAGE = package
VERSION = `date +"%Y.%m%d"`
ARCHIVE = $(PACKAGE)-$(VERSION)
```

```
dist:
```

```
# Notice that only now macros are
# expanded for shell to interpret:
# tar -cf ../package-`date +"%Y.%m%d"` .tar
tar -cf ../$(ARCHIVE).tar .
```

Note: Environment variables are also available as macros.

# Extended Example: Compilation and Linking

```
# Example Makefile for the `hydrogen` program (after modularization)
CXX=g++
CXXFLAGS=-std=c++17 -O2
LDFLAGS=-O2
all: hydrogen

hydrogen: hydrogen.o outputarray.o initmatrix.o eigenvalue.o
    $(CXX) $(LDFLAGS) -o hydrogen hydrogen.o outputarray.o initmatrix.o eigenvalue.o $(LDLIBS)

hydrogen.o: hydrogen.cpp outputarray.h initmatrix.h eigenvalue.h
    $(CXX) -c $(CXXFLAGS) -o hydrogen.o hydrogen.c

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

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

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

clean:
    $(RM) eigenvalue.o initmatrix.o outputarray.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 than 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 Example with Variables

```
# Example Makefile for the `hydrogen` program (after modularization)
CXX=g++
CXXFLAGS=-std=c++17 -O2
LDFLAGS=-O2
all: hydrogen

hydrogen: hydrogen.o outputarray.o initmatrix.o eigenvalue.o
    $(CXX) $(LDFLAGS) -o $@ $^ $(LDLIBS)

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

outputarray.o: outputarray.cpp outputarray.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 outputarray.o hydrogen.o
.PHONY: all clean
```