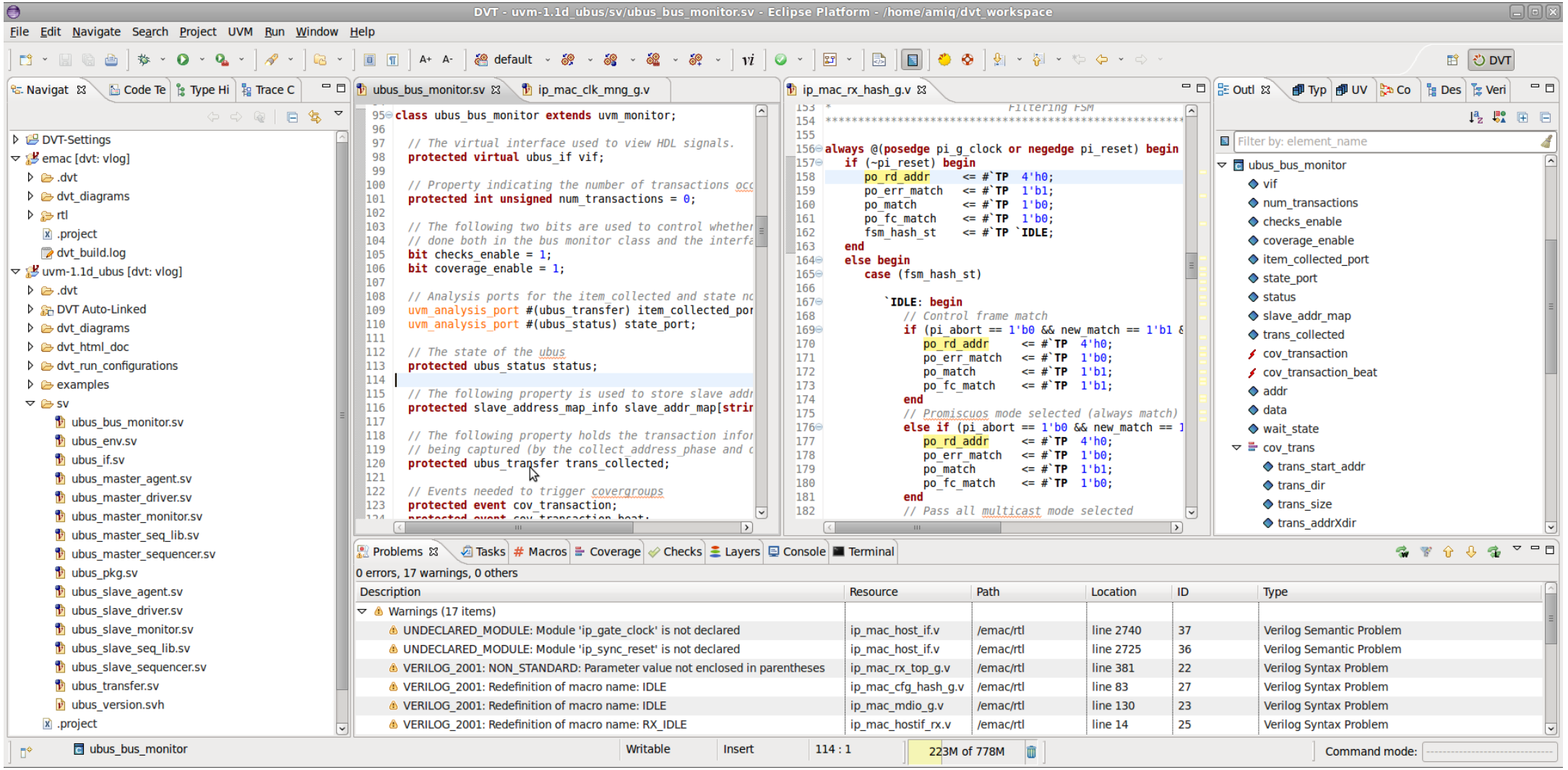
The CS-220 Development Environment

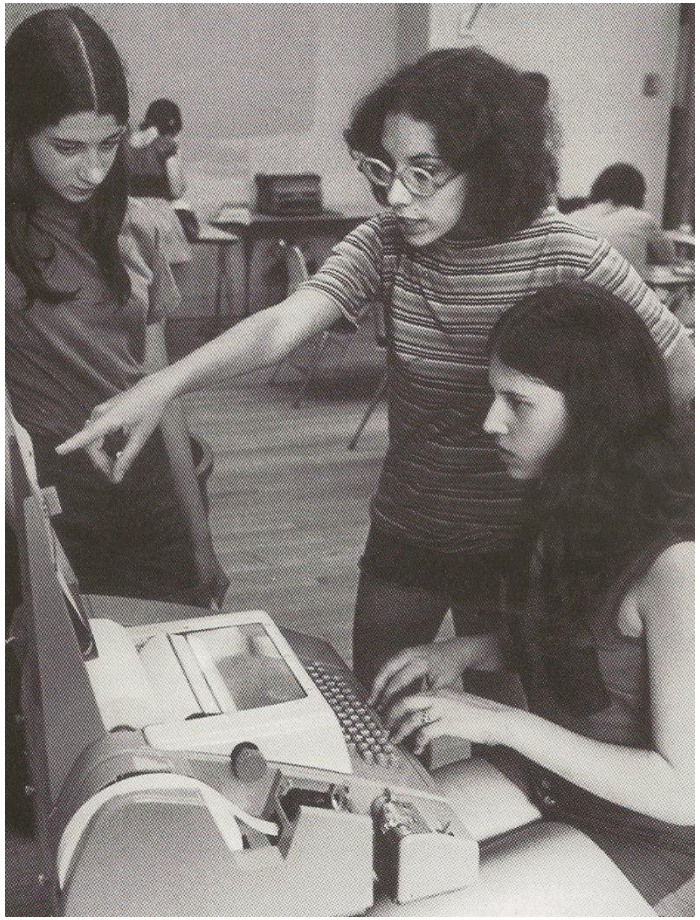
(No relevant sections in text)

Picking the right tool for the job

Integrated Development Environment



Command Line Mentality



Old fashioned but…

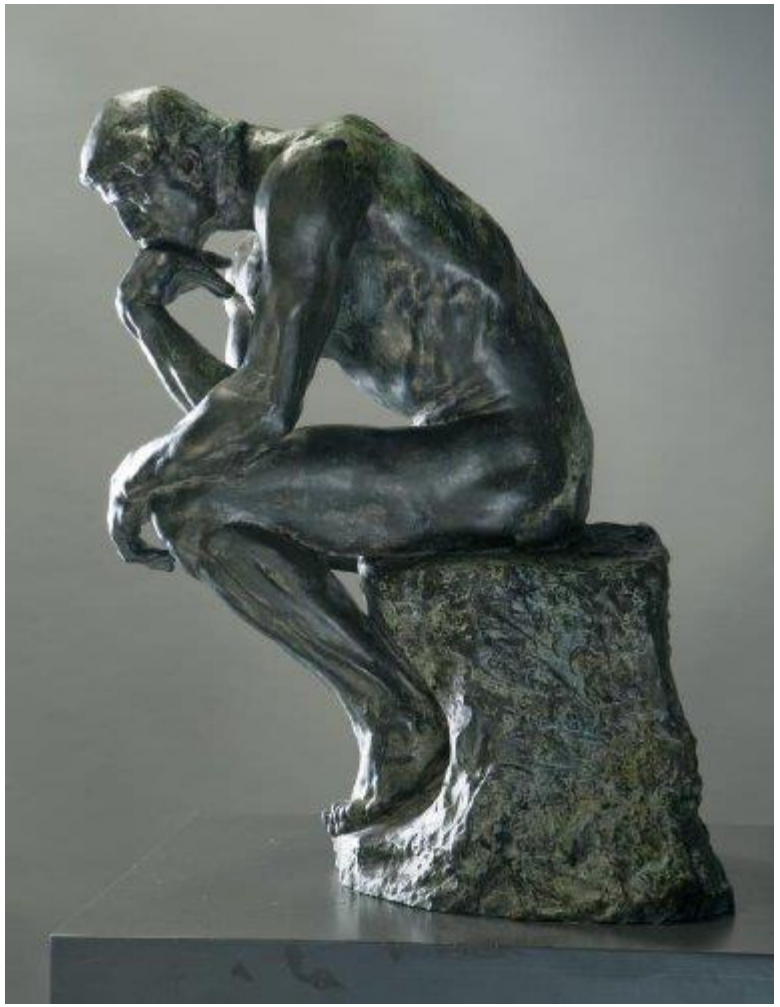
…surprisingly efficient…

… (except for editing).

Command Line Mentality

- Artisan vs. Factory Worker
- Learn to use lots of tools
- May be slower
- Take more knowledge
- Require more effort
- Tools you can take with you
- Tools universally available
- Applicable to wide variety of projects

Picking the Right Tool for the Job



Operating System: UNIX

- Oldest “modern” operating system
- Base for Linux, HP/UX, Solaris, AIX, Android, etc.
- Most widely used
- Available on Lab accounts
- Free version available on Windows : CYGWIN (https://www.cygwin.com/)
- Large library of free software - GNU compilers, editors, debuggers, etc. (http://en.wikipedia.org/wiki/GNU)
- Assume basic knowledge of UNIX (http://www.ee.surrey.ac.uk/Teaching/Unix)

WARNING: Not All UNIX is the same!

You may use:
Cygwin, virtual box, linux laptops, PODS linux, etc.
for labs, homework, projects, study, etc.

- Be aware – each UNIX installation is slightly different!
- If you have a technical problem, you are on your own!
- Grading will be performed on CS "LDAP" machines!
  - "It worked on my laptop" is not a valid excuse

- Test on LDAP machines!
- Use the lab(s) when no labs are running
- Access remotely with PUTTY

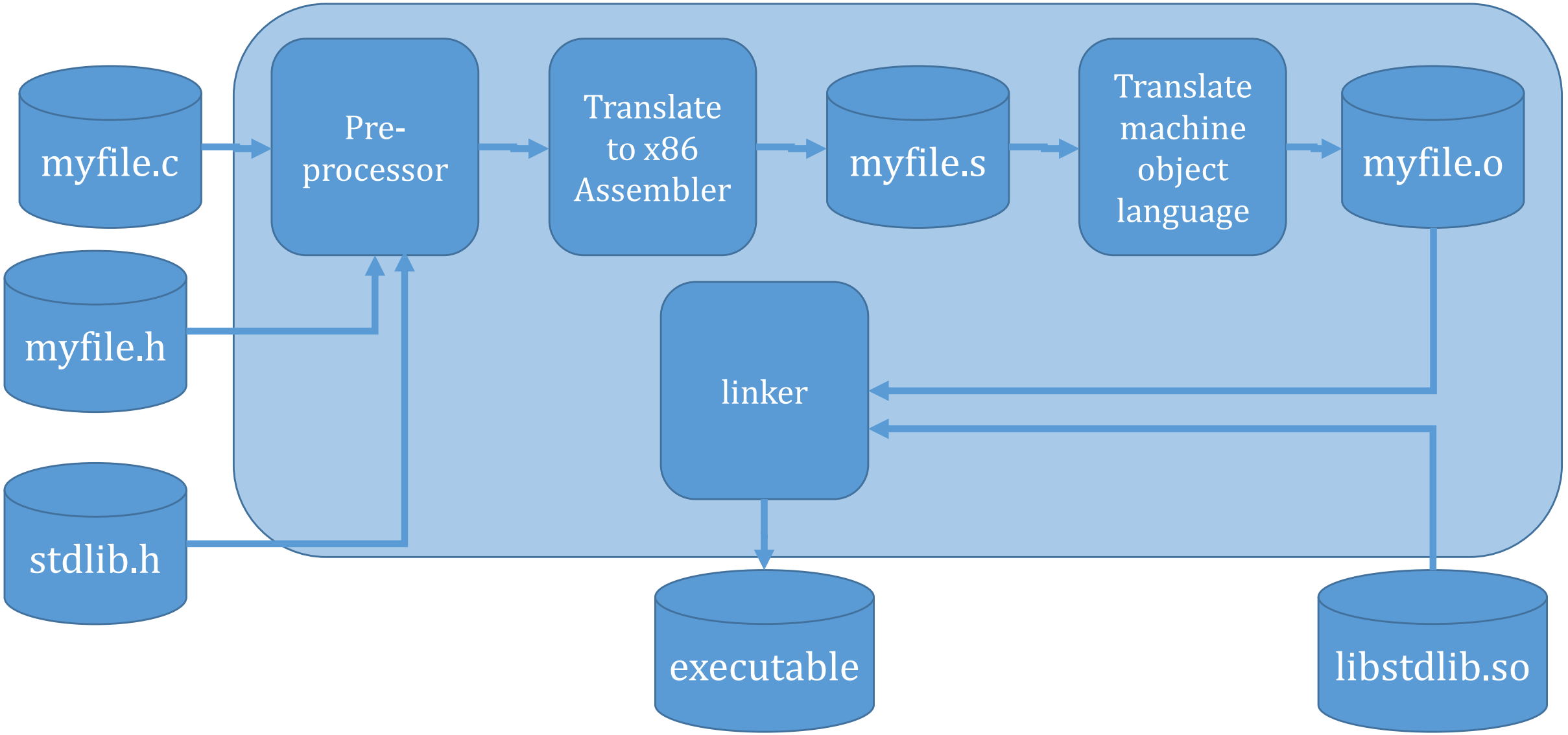
Basic Commands

- Editor – gedit (not available everywhere)
- Compiler / Linker – gcc
- Build manager – make
- Debugger - gdb

Editor - gedit

- Hybrid command line / Graphical User Interface (GUI)
- First invocation from command line starts GUI
  - Suggest "gedit myfile.c &" first time – otherwise, hangs up terminal window until gedit is closed
- Once gedit is running, successive "gedit" commands send messages to existing GUI
- Recommend Edit/Preferences/
  - View - ☑ Display line numbers
  - View - ☑ Highlight matching brackets
  - Editor - ☐ Create a backup copy of files before saving

The compile / link process



GCC (Gnu C Compiler)

- gcc <options> <source>
- Basic options:
  - -o <output file name / command name>
  - -g [include debug information in output file]
  - -Wall [turn on all warning messages]
- For example: gcc –g –Wall –o executable myfile.c
- For more detail, gcc –– help or man gcc
- Complete documentation: https://gcc.gnu.org/onlinedocs/

Makefile

- File that tells "make" what to do and how to do it
- Composed of a list of "make rules"
- A rule has three parts:
  - Target – the file that this rule produces
  - Pre-requisite files – A list of file used to make the target
  - Recipe – Unix command(s) to produce target from the pre-requisite files
- For example:

```
mymain : mymain.c mymain.h comp.c comp.h
	gcc -g -Wall -o mymain mymain.c comp.c
```

Required Tab

Example Makefile

```
mycmd : mymain.c mymain.h comp.c comp.h
	gcc -g -Wall -o mycmd mymain.c comp.c

test : mycmd
	mycmd “test string”
	mycmd “test string 2”

clean:
	-rm mycmd
```

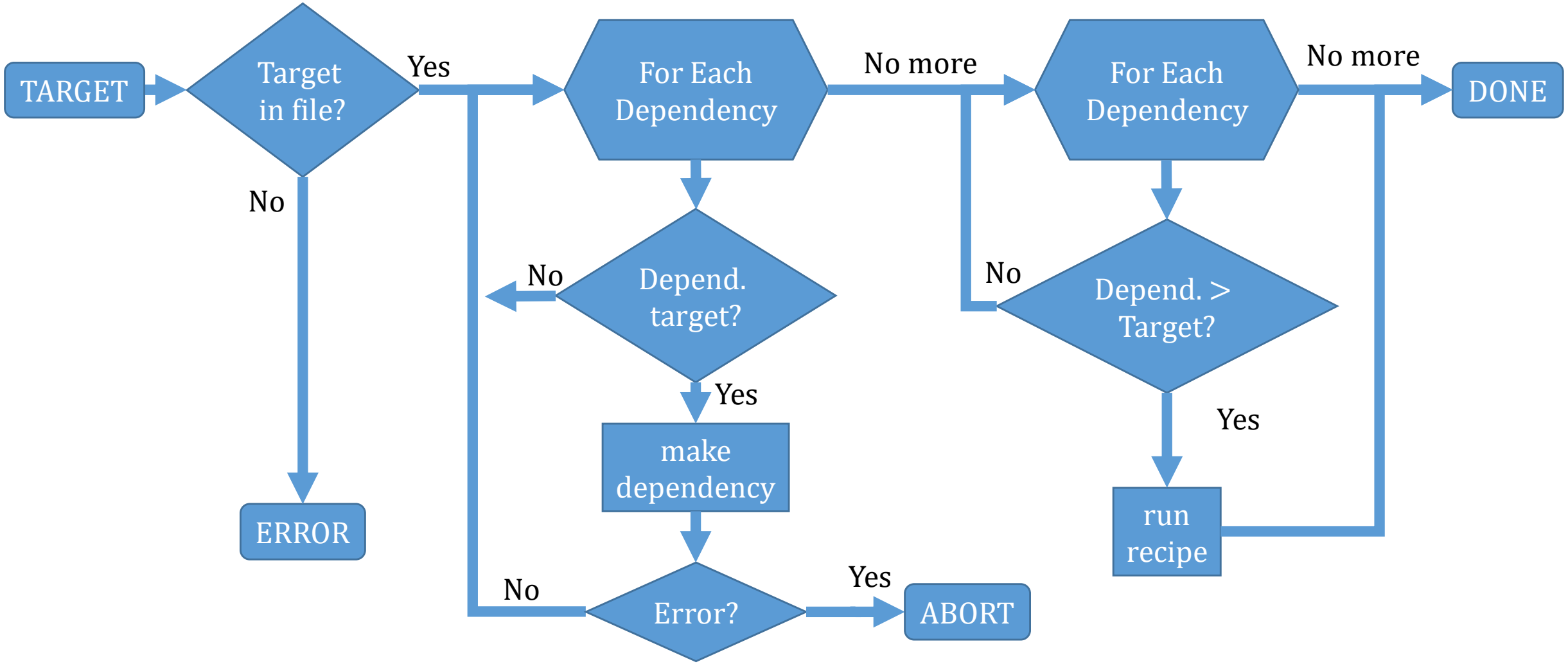
Invoking make

- Command : make <target>
- If <target> not specified, look for target “all”
- If <target> not specified and no “all” target, make first target in the make file

Make processing (simplified)

- Find the rule for the target specified
- Make all dependencies for that rule
  - At least those that show up as targets
- If any dependencies are newer than target file, invoke recipe

Make process (flowchart)



Example Make internals

```
mycmd : mymain.c mymain.h comp.c comp.h
        gcc –o mycmd mymain.c comp.c

test : mycmd
        mycmd “test string”
        mycmd “test string 2”

clean:
        -rm mycmd
```

```
~> make test
// dependency mycmd is a target
// make mycmd
        // dependency mymain.c not a target
        // dependency mymain.h not a target
        // dependency comp.c not a target
        // dependency comp.h not a target
        // mymain.c older than mycmd
        // mymain.h older than mycmd
        // comp.c older than mycmd
        // comp.h newer than mycmd
        // run gcc –o mycmd mymain.c comp.c
// mycmd is newer than “test”
// run mycmd “test string”
// run mycmd “test string 2”
```

Phony Targets

- Targets for which there is no corresponding file
  - For example: clean & test
  - When recipe is invoked, it does not create the dependency file!
- Recipe invoked each time target is specified
  - If make cannot find the target file, it assumes its date is ancient
  - Therefore, the phony “target file” is ALWAYS “older” than dependencies
  - Note that dependencies are still made… “make test” will “make mycmd”
- make mycmd
  - will run gcc –o mycmd only if mycmd is older than its dependencies
- make test
  - will run mycmd “test string” and mycmd “test string 2” EVERY time

Make Errors

- What are errors in make?
  - Compile Errors
  - Test Failed
  - etc.
- How does make know there was a problem?
  - If recipe returns a non-zero return code, there was a problem
  - Unless recipe is preceded by "-" which indicates ignore non-zero rc
    - For example make clean often performs commands like "rm mycomp.o"
    - If mycomp.o does not exist, rm returns non-zero rc
    - therefore, typically we use "-rm mycomp.o" for make clean
  - Useful to stop make when there is a problem

Example make file with multiple tests

```
test : test1 test2 test3

test1 : mycmd
        mycmd testfile1.txt

test2 : mycmd
        mycmd testfile2.txt

test3 : mycmd
        mycmd testfile3.txt

mycmd : mycmd.c mycmd.h comp.c comp.h
        gcc -g -o mycmd mycmd.c comp.c
```

- Run “make” or “make test” to run all tests
  - make will build mycmd if you have changed source files
  - make will stop with an error if compile failed
  - make will run test1
  - make will stop with an error if test1 fails
  - make will run test2
  - make will stop with an error if test2 fails
  - make will run test3
  - make will stop with an error if test3 fails
- If test2 fails
  - change the code to fix the problem
  - run “make test2” to see if that problem is fixed
  - then run “make test” to run all tests

Typical Targets

- all - all executables required
- clean - remove all built files
- test – test all executables built
- install - any commands required to make executables generally available

Make Concepts

- Build a product from components
  - Standardize the build process
  - Make it repeatable - make builds the same way every time
- Rebuild only components that have changed
  - As long as dependencies are correct, make will build exactly what is needed – and no more
- Package complex commands using simple targets
  - clean, test, debug – all may be sophisticated commands
  - invoke as “make test”

Debugging – GDB (Gnu DeBugger)

- gdb <executable>
  - Starts debugger
  - Loads executable
  - Prompts for gdb commands
- Basic GDB commands:
  - h[elp] – help on gdb commands
  - b[reak] <location> – set a breakpoint at the specified location
    - Location can be either a line number in a file or a function name
  - run <command arguments> - invoke command and run to next breakpoint
  - c[ontinue] – continue to next breakpoint
  - s[tep] – run next program instruction, stepping into function invocations
  - n[ext] – run next program instruction, skipping over function invocations
  - p[rint] <variable or expression> - print the current value of variable
  - x <options> <location> - print (examine) memory at locations
  - q[uit] – exit out of gdb

GDB hints

- gdb –x gdbcmds.txt mycommand
  - Runs command from gdbcmds.txt before prompting
  - Useful when you have lots of breakpoints and/or complicated parameters
- Null gdb command same as “repeat previous”
  - E.g. “step” <enter> <enter> to step 3 instructions
- If you see a prompt like:

  Breakpoint 1, 0x0000000100401d8 in main ()

  You forgot to compile with –g
- Prompt shows the line of C code ABOUT to execute
- Documentation: http://www.gnu.org/software/gdb/documentation/