CIS 330:

Lecture 2:
Permissions and Build Systems
Accessing a Unix environment

• Rm 100, Deschutes
• Remote logins (ssh, scp)
• Windows options
  – Cygwin / MSYS
  – Virtual machines

Who has home access to a Unix environment?

Who has Windows only and wants to pursue Cygwin/VM & needs help?
Accessing remote machines

• Windows->Unix
  – ??? (Hummingbird Exceed was the answer last time I used Windows)

• Unix->Unix
  – ssh: secure shell  
    ssh –l hank ix.cs.uoregon.edu
  – scp: secure copy  
    scp hank@ix.cs.uoregon.edu:~/file1
  • Also, ftp: file transfer protocol

Who is needing help with Unix environment on Windows? (only one response so far)
Unix systems

• Four basic use cases
  – Personal use machines
  – Servers
  – Embedded
  – Compute clusters

Are there more?
(this is off the top of my head)

In many of these scenarios, there is a system administrator who makes an “image” of the OS that they “clone” for each machine.

I have used Unix actively since 1994, but only did system administration 2005-2009 when I had a Linux box in my home.
Outline

• Review
• Permissions
• Project 1B Overview
• Build  
  (my hope is to make it halfway through build)
• Project 1C Overview
Outline

• Review
  – (INCLUDES vi)

• Permissions

• Project 1B Overview

• Build

• Project 1C Overview
Quiz

• What is the output of this sequence?

```
Last login: Wed Apr  2 14:42:31 on ttys006
C02LN00GFD58:~ hank$ mkdir CIS330
C02LN00GFD58:~ hank$ cd CIS330
C02LN00GFD58:CIS330 hank$ mkdir subDir
C02LN00GFD58:CIS330 hank$ touch subDir/a
C02LN00GFD58:CIS330 hank$ touch subDir/b
C02LN00GFD58:CIS330 hank$ touch a
C02LN00GFD58:CIS330 hank$ rm subDir/a
C02LN00GFD58:CIS330 hank$ ls ; ls subDir
```
Shells

• Shells are accessed through a terminal program
  – Typically exposed on all Linux
  – Mac: Applications->Utilities->Terminal
    • (I always post this on the dock immediately upon getting a new Mac)
Shells

• Shells are interpreters
  – Like Python

• You type a command, it carries out the command
Shells

• There are many types of shells
• Two most popular:
  – sh (= bash & ksh)
  – csh (= tcsh)
• They differ in syntax, particularly for
  – Environment variables
  – Iteration / loops / conditionals

The examples in this course will use syntax for sh
Environment Variables

• Environment variables: variables stored by shell interpreter
• Some environment variables create side effects in the shell
• Other environment variables can be just for your own private purposes
Environment Variables

New commands: export, echo, env
File Editors

• Existing file editors:
  – Vi
  – Emacs
  – Two or three hot new editors that everyone loves

• This has been the state of things for 25 years

I will teach “vi” in this course. You are welcome to use whatever editor you want.
Vi has two modes

• Command mode
  – When you type keystrokes, they are telling vi a command you want to perform, and the keystrokes don’t appear in the file

• Edit mode
  – When you type keystrokes, they appear in the file.
Transitioning between modes

• Command mode to edit mode
  – i: enter into edit mode at the current cursor position
  – a: enter into edit mode at the cursor position immediately to the right of the current position
  – I: enter into edit mode at the beginning of the current line
  – A: enter into edit mode at the end of the current line

There are other ways to enter edit mode as well
Transitioning between modes

• Edit mode to command mode
  – Press Escape
Useful commands

• yy: yank the current line and put it in a buffer
  – 2yy: yank the current line and the line below it
• p: paste the contents of the buffer
• x: delete the character at the current cursor
• “:100” go to line 100 in the file
• Arrows can be used to navigate the cursor position (while in command mode)
  – So do h, j, k, and l

We will discuss more tips for “vi” throughout the quarter. They will mostly be student-driven (Q&A time each class)
My first vi sequence

• At a shell, type: “vi cis330file”
• Press ‘i’ (to enter edit mode)
• Type “I am using vi and it is fun” (text appears on the screen)
• Press “Escape” (to enter command mode)
• Press “:wq” (command mode sequence for “write and quit”)
### vi / vim Graphical Cheat Sheet

#### Esc (Normal Mode)

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>toggle case</td>
</tr>
<tr>
<td>.</td>
<td>goto mark</td>
</tr>
<tr>
<td>1</td>
<td>toggle case</td>
</tr>
<tr>
<td>2</td>
<td>goto mark</td>
</tr>
<tr>
<td>!</td>
<td>external filter</td>
</tr>
<tr>
<td>@</td>
<td>play macro</td>
</tr>
<tr>
<td>#</td>
<td>prev ident</td>
</tr>
<tr>
<td>$</td>
<td>eol</td>
</tr>
<tr>
<td>%</td>
<td>goto match</td>
</tr>
<tr>
<td>^</td>
<td>&quot;soft&quot; bol</td>
</tr>
<tr>
<td>&amp;</td>
<td>repeat s</td>
</tr>
<tr>
<td>*</td>
<td>next ident</td>
</tr>
<tr>
<td>(</td>
<td>begin sentence</td>
</tr>
<tr>
<td>)</td>
<td>&quot;hard&quot; bol</td>
</tr>
<tr>
<td>0</td>
<td>end sentence</td>
</tr>
<tr>
<td>+</td>
<td>next line</td>
</tr>
</tbody>
</table>

#### Command Mode

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>ex mode</td>
</tr>
<tr>
<td>q</td>
<td>record macro</td>
</tr>
<tr>
<td>W</td>
<td>next word</td>
</tr>
<tr>
<td>E</td>
<td>end word</td>
</tr>
<tr>
<td>R</td>
<td>replace char</td>
</tr>
<tr>
<td>T</td>
<td>'till</td>
</tr>
<tr>
<td>Y</td>
<td>yank line</td>
</tr>
<tr>
<td>U</td>
<td>undo</td>
</tr>
<tr>
<td>I</td>
<td>insert at bol</td>
</tr>
<tr>
<td>O</td>
<td>insert above</td>
</tr>
<tr>
<td>P</td>
<td>paste before</td>
</tr>
<tr>
<td>{</td>
<td>begin parag.</td>
</tr>
<tr>
<td>}</td>
<td>end parag.</td>
</tr>
</tbody>
</table>

#### Motion

- **motion**: moves the cursor, or defines the range for an operator
- **command**: direct action command, if **red**, it enters insert mode
- **operator**: requires a motion afterwards, operates between cursor & destination
- **extra**: special functions, requires extra input

#### Main command line commands ('ex'):

- `:w` (save), `:q` (quit), `:q!` (quit w/o saving)
- `:e` (open file `f`),
- `:s/x/y/g` (replace 'x' by 'y' filewide),
- `:h` (help in vim), `:new` (new file in vim),

#### Other important commands:

- `CTRL-R`: redo (vim),
- `CTRL-F/-B`: page up/down,
- `CTRL-E/-Y`: scroll line up/down,
- `CTRL-V`: block-visual mode (vim only)

#### Visual mode:

- Move around and type operator to act on selected region (vim only)

#### Notes:

1. Use "x before a yank/paste/del command to use that register ('clipboard') (x=a..z,*), 
   e.g.: "ay$` to copy rest of line to reg 'a'
2. Type in a number before any action to repeat it that number of times 
   (e.g.: 2p, d2w, 5i, d4j)
3. Duplicate operator to act on current line (dd = delete line, >> = indent line)
4. ZZ to save & quit, ZQ to quit w/o saving
5. zt: scroll cursor to top, 
   zb: bottom, zz: center
6. gg: top of file (vim only), 
   gf: open file under cursor (vim only)

For a graphical vi/vim tutorial & more tips, go to [www.viemu.com](http://www.viemu.com) - home of ViEmu, vi/vim emulation for Microsoft Visual Studio
Project 1A

• Practice using an editor
• Must be written using editor on Unix platform
  – I realize this is unenforceable.
  – If you want to do it with another mechanism, I can’t stop you
    • But realize this project is simply to prepare you for later projects
Project 1A

• Write $\geq500$ words using editor (vi, emacs, other)
• Topic: what you know about C programming language
• Can’t write 500 words?
  – Bonus topic: what you want from this course
• How will you know if it is 500 words?
  – Unix command: “wc” (word count)
Unix command: \texttt{wc} (word count)

\begin{verbatim}
  fawcett:~ childs$ vi hanks_essay
  fawcett:~ childs$ wc -w hanks_essay
        252 hanks_essay
  fawcett:~ childs$ wc hanks_essay
      63    252    1071 hanks_essay
  fawcett:~ childs$
\end{verbatim}

(63 = lines, 252 = words, 1071 = character)
CIS 330: Project #1A
Assigned: April 1st, 2015
Due April 6th, 2015
(which means submitted by 6am on April 7th, 2015)
Worth 1% of your grade

Assignment:
1) On a Unix platform (including Mac), use an editor (vi, emacs, other) to write
   500 word "essay"
   a. The purpose of the essay is to practice using an editor.
      i. Grammar will not be graded
   b. I would like to learn more about what you know about C and want
      from this class ... I recommend you each write about that.

Turn this in using Blackboard.
How to submit

• Blackboard
• If you run into trouble:
  – Email me your solution
What should you do if you run into trouble?

1) Start with Piazza
2) OH
3) Email me: hank@cs.uoregon.edu
Outline

• Review
• Permissions
• Project 1B Overview
• Build
• Project 1C Overview
Permissions: System Calls

• System calls: a request from a program to the OS to do something on its behalf
  – ... including accessing files and directories

• System calls:
  – Typically exposed through functions in C library
  – Unix utilities (cd, ls, touch) are programs that call these functions

Permissions in Unix are enforced via system calls.
Permissions: Unix Groups

• Groups are a mechanism for saying that a subset of Unix users are related
  – Last year, we had a “330_S14” unix group on ix
  – Members:
    • Me
    • 2 GTFs

The commands for creating a group tend to vary, and are often done by a system administrator

CIS uses “groupctl”
Permissions

- Permissions are properties associated with files and directories
  - System calls have built-in checks to permissions
    - Only succeed if proper permissions are in place

- Three classes of permissions:
  - User: access for whoever owns the file
    - You can prevent yourself from accessing a file!
      - (But you can always change it back)
  - Group: allow a Unix group to access a file
  - Other: allow anyone on the system to access a file
Executable files

• An executable file: a file that you can invoke from the command line
  – Scripts
  – Binary programs

• The concept of whether a file is executable is linked with file permissions
There are 9 file permission attributes

- Can user read?
- Can user write?
- Can user execute?
- Can group read?
- Can group write?
- Can group execute?
- Can other read?
- Can other write?
- Can other execute?

A bunch of bits ... we could represent this with binary

User = “owner”
Other = “not owner, not group”
Translating permissions to binary

<table>
<thead>
<tr>
<th>#</th>
<th>Permission</th>
<th>rwx</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>full</td>
<td>111</td>
</tr>
<tr>
<td>6</td>
<td>read and write</td>
<td>110</td>
</tr>
<tr>
<td>5</td>
<td>read and execute</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>read only</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>write and execute</td>
<td>011</td>
</tr>
<tr>
<td>2</td>
<td>write only</td>
<td>010</td>
</tr>
<tr>
<td>1</td>
<td>execute only</td>
<td>001</td>
</tr>
<tr>
<td>0</td>
<td>none</td>
<td>000</td>
</tr>
</tbody>
</table>

Which of these modes make sense? Which don’t?

We can have separate values (0-7) for user, group, and other.
Unix command: chmod

• chmod: change file mode

• chmod 750 <filename>
  – User gets 7 (rwx)
  – Group gets 5 (rx)
  – Other gets 0 (no access)

Lots of options to chmod (usage shown here is most common)
Manpage for chmod

- “man chmod”

CHMOD(1)
BSD General Commands Manual

NAME
chmod -- change file modes or Access Control Lists

SYNOPSIS
chmod [-fv] [-R [-H | -L | -P]] mode file ...
chmod [-fv] [-R [-H | -L | -P]] [-a | +a | =a] ACE file ...
chmod [-fhv] [-R [-H | -L | -P]] [-E] file ...
chmod [-fhv] [-R [-H | -L | -P]] [-C] file ...
chmod [-fhv] [-R [-H | -L | -P]] [-N] file ...

DESCRIPTION
The chmod utility modifies the file mode bits of the listed files as specified by the mode operand. It may also be used to modify the Access Control Lists (ACLs) associated with the listed files.

The generic options are as follows:

-f  Do not display a diagnostic message if chmod could not modify the mode for file.
Unix commands for groups

• chgrp: changes the group for a file or directory
  – chgrp <group> <filename>

• groups: lists groups you are in
`ls -l`

- **Long listing of files**

```
Last login: Thu Apr  3 08:09:23 on ttys007
C02LN00GFD58:~ hank$ mkdir CIS330
C02LN00GFD58:~ hank$ cd CIS330
C02LN00GFD58:CIS330 hank$ touch a
C02LN00GFD58:CIS330 hank$ ls -l

total 0
-rw-r--r--   1 hank staff   0 Apr  3 08:14 a
```

- **Permissions**
- **Links (**)
- **Owner**
- **Group**
- **File size**
- **Date of last change**
- **Filename**

**How to interpret this?**
Permissions and Directories

• You can only enter a directory if you have “execute” permissions to the directory.

• Quiz: a directory has permissions “400”. What can you do with this directory?

Answer: it depends on what permissions a system call requires.
Directories with read, but no execute

Last login: Thu Apr  3 08:14:33 on ttys007
C02LN00GFD58:~ hank$ mkdir CIS330
C02LN00GFD58:~ hank$ touch CIS330/a
C02LN00GFD58:~ hank$ chmod 400 CIS330
C02LN00GFD58:~ hank$ ls CIS330
  a
C02LN00GFD58:~ hank$ cd CIS330
  -bash: cd: CIS330: Permission denied
C02LN00GFD58:~ hank$ cat CIS330/a
  cat: CIS330/a: Permission denied
Outline

- Review
- Permissions
- Project 1B Overview
- Build
- Project 1C Overview
Unix scripts

• Scripts
  – Use an editor (vi/emacs/other) to create a file that contains a bunch of Unix commands
  – Give the file execute permissions
  – Run it like you would any program!!
Unix scripts

• Arguments
  – Assume you have a script named “myscript”
  – If you invoke it as “myscript foo bar”
  – Then
    • $\# == 2$
    • $1 == \text{foo}$
    • $2 == \text{bar}$
Project 1B

• Summary: write a script that will create a specific directory structure, with files in the directories, and specific permissions.
CIS 330: Project #1B
Assigned: April 3rd, 2015
Due April 8th, 2015
(which means submitted by 6am on April 9th, 2014)
Worth 2% of your grade

Assignment: Create a script that will create a directory structure, and files within that directory structure, all with the specified file permissions. The script should be named “proj1b”. (A consistent name will help with grading.)

Note: you are only allowed to use the following commands: mkdir, touch, cd, chmod, mv, cp. (You do not need to use all of these commands to successfully complete the assignment.)
Project 1B

The directory structure should be:

- **Root dir**
  - **Dir1** Permissions: 770
    - **File1** Permissions: 400
    - **Dir3** Permissions: 000
      - **Dir4** Permissions: 750
        - **File4** Permissions: 666
        - **File3** Permissions: 200
  - **Dir2** Permissions: 775
    - **File2** Permissions: 640

**Key**
- **Files:** Name of file Permissions
- **Directories:** Name of directory Permissions
Outline

• Review
• Permissions
• Project 1B Overview
• Build
• Project 1C Overview
**Build: The Actors**

- **File types**
  - Source code
  - Object code
  - Executable code

- **Programs**
  - Compiler
  - Linker

Source code (e.g., C code) → Compiler → Object code → Linker → Executable code
Compilers, Object Code, and Linkers

• Compilers transform source code to object code
  – Confusing: most compilers also secretly have access to linkers and apply the linker for you.

• Object code: statements in machine code
  – not executable
  – intended to be part of a program

• Linker: turns object code into executable programs
GNU Compilers

• GNU compilers: open source
  – gcc: GNU compiler for C
  – g++: GNU compiler for C++

C++ is superset of C. With very few exceptions, every C program should compile with a C++ compiler.
C++ comments

• “//” : everything following on this line is a comment and should be ignored

• Examples:

  // we set pi below
  float pi = 3.14159; // approximation of pi

Can you think of a valid C syntax that will not compile in C++?

float radians=degrees/*approx. of pi*//3.14159;
Our first gcc program

```c
#include <stdio.h>
int main()
{
    printf("hello world!\n");
}
```

Invoke gcc compiler

Name of file to compile

Default name for output programs
Our first gcc program: named output

C02LN00GFD58:CIS330 hank$ cat t.c
#include <stdio.h>
int main()
{
    printf("hello world!\n");
}
C02LN00GFD58:CIS330 hank$ gcc t.c
C02LN00GFD58:CIS330 hank$ ./a.out
hello world!
C02LN00GFD58:CIS330 hank$ gcc -o helloworld t.c
C02LN00GFD58:CIS330 hank$ ./helloworld
hello world!
C02LN00GFD58:CIS330 hank$ ls -l helloworld
-rwxr-xr-x 1 hank staff 8496 Apr 3 15:15 helloworld
C02LN00GFD58:CIS330 hank$
gcc flags: debug and optimization

- “gcc –g”: debug symbols
  - Debug symbols place information in the object files so that debuggers (gdb) can:
    - set breakpoints
    - provide context information when there is a crash
- “gcc –O2”: optimization
  - Add optimizations ... never fails
- “gcc –O3”: provide more optimizations
  - Add optimizations ... sometimes fails
- “gcc –O3 –g”
  - Debugging symbols slow down execution ... and sometimes compiler won’t do it anyways...
Debug Symbols

- live code

```c
int main()
{
    int sum = 0;
    int i;
    for (i = 0; i < 10; i++)
        sum += i;
    return sum;
}
```

- `gcc -S t.c` # look at t.s
- `gcc -S -g t.c` # look at t.s

- (-S flag: compile to assembly instead of object code)
Object Code Symbols

- Symbols associate names with variables and functions in object code.
- Necessary for:
  - debugging
  - large programs
Large code development

Why could this be a good idea?
Multi-file development: example

fawcett:330 childs$ cat t1.c
int doubler(int x)
{
    return 2*x;
}
fawcett:330 childs$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childs$ gcc -c t1.c
fawcett:330 childs$ gcc -c t2.c
fawcett:330 childs$ gcc -o both t2.o t1.o
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $?
10

*cat* is a Unix command that prints the contents of a file

$? is a shell construct that has the return value of the last executed program
Multi-file development: example

```c
fawcett:330 childds$ cat t1.c
int doubler(int x)
{
    return 2*x;
}
fawcett:330 childds$ cat t2.c
int main()
{
    return doubler(5);
}
fawcett:330 childds$ gcc -c t1.c
fawcett:330 childds$ gcc -c t2.c
fawcett:330 childds$ gcc -o both t2.o t1.o
fawcett:330 childds$ ./both
fawcett:330 childds$ echo $?
10
fawcett:330 childds$ gcc -o both t2.o
Undefined symbols:
    "_doubler", referenced from:
        _main in t2.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
fawcett:330 childds$ gcc -o both t1.o
Undefined symbols:
    "_main", referenced from:
        start in crt1.10.6.o
ld: symbol(s) not found
collect2: ld returned 1 exit status
```
Multi-file development: example

```c
int doubler(int x)
{
    return 2*x;
}
```

```bash
fawcett:330 childs$ gcc -o both t1.o t2.o
fawcett:330 childs$ ./both
```

Linker order matters for some linkers (not Macs). Some linkers need the .o with “main” first and then extract the symbols they need as they go. Other linkers make multiple passes.
Libraries

• Library: collection of “implementations” (functions!) with a well defined interface
• Interface comes through “header” files.
• In C, header files contain functions and variables.
  – Accessed through “#include <file.h>”
Libraries

• Why are libraries a good thing?

• Answers:
  – separation
    • I.e., divide and conquer
      – increases productivity
    • I.e., simplicity
    • I.e., prevents tendrils between modules that shouldn’t exist
  – encapsulation (hides details of the implementation)
    • “A little knowledge is a dangerous thing”...
  • Products
    – I can sell you a library and don’t have to give you the source code.
Libraries

• Why are libraries a bad thing?

• Answers:
  – separation
    • I.e., makes connections between modules harder
      – (were the library interfaces chosen correctly?)
  – complexity
    • need to incorporate libraries into code compilation
Includes and Libraries

• gcc support for libraries
  – “-I”: path to headers for library
  – “-L”: path to library location
  – “-Lname”: link in library libname
Library types

• Two types:
  – static and shared

• Static: all information is taken from library and put into final binary at link time.
  – library is never needed again

• Shared: at link time, library is checked for needed information.
  – library is loaded when program runs

More about shared and static later ... for today, assume static
Making a static library

Note the ‘#’ is the comment character

C02LN000GFD58:multiplier hank$ cat doubler.h # here's the header file
int doubler(int);
int tripler(int);
C02LN000GFD58:multiplier hank$ cat doubler.c # here's one of the c files
int doubler(int x) {return 2*x;}
C02LN000GFD58:multiplier hank$ cat tripler.c # here's the other c files
int tripler(int x) {return 3*x;}
C02LN000GFD58:multiplier hank$ gcc -c doubler.c # make an object file
C02LN000GFD58:multiplier hank$ ls doubler.o # we now have a .o
  doubler.o
C02LN000GFD58:multiplier hank$ gcc -c tripler.c
C02LN000GFD58:multiplier hank$ ar r multiplier.a doubler.o tripler.o
C02LN000GFD58:multiplier hank$ (should have called this libmultiplier.a)
What’s in the file?

C02LN00GFD58:multiplier hank$ nm multiplier.a

multiplier.a(doubler.o):
000000000000000038 s EH_frame0
0000000000000000 T _doubler
00000000000000050 S _doubler.eh

multiplier.a(tripler.o):
000000000000000030 s EH_frame0
0000000000000000 T _tripler
00000000000000048 S _tripler.eh
C02LN00GFD58:multiplier hank$
Typical library installations

• Convention
  – Header files are placed in “include” directory
  – Library files are placed in “lib” directory

• Many standard libraries are installed in /usr
  – /usr/include
  – /usr/lib

• Compilers automatically look in /usr/include and /usr/lib (and other places)
Installing the library

```
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier/include
C02LN00GFD58:multiplier hank$ cp multiplier.h ~/multiplier/include/
C02LN00GFD58:multiplier hank$ mkdir ~/multiplier/lib
C02LN00GFD58:multiplier hank$ cp doubler.c multiplier.a tripler.c
C02LN00GFD58:multiplier hank$ cp doubler.o multiplier.h tripler.o
C02LN00GFD58:multiplier hank$ cp multiplier.a ~/multiplier/
C02LN00GFD58:multiplier hank$ mv multiplier.a libmultiplier.a
C02LN00GFD58:multiplier hank$ cp libmultiplier.a ~/multiplier/lib/
C02LN00GFD58:multiplier hank$
```

“mv”: unix command for renaming a file
Example: compiling with a library

```c
#include <multiplier.h>
#include <stdio.h>
int main()
{
    printf("Twice 6 is %d, triple 6 is %d\n", doubler(6), tripler(6));
}
```

```bash
C02LN00GFD58:CIS330 hank$ cat t.c
#include <multiplier.h>
#include <stdio.h>
int main()
{
    printf("Twice 6 is %d, triple 6 is %d\n", doubler(6), tripler(6));
}
C02LN00GFD58:CIS330 hank$ gcc -o mult_example t.c -I/Users/hank/multiplier/inclu
de -L/Users/hank/multiplier/lib -lmultiplier
C02LN00GFD58:CIS330 hank$ ./mult_example
Twice 6 is 12, triple 6 is 18
C02LN00GFD58:CIS330 hank$
```

- gcc support for libraries
  - “-I”: path to headers for library
  - “-L”: path to library location
  - “-lname”: link in library libname
Makefiles

• There is a Unix command called “make”
• make takes an input file called a “Makefile”
• A Makefile allows you to specify rules
  – “if timestamp of A, B, or C is newer than D, then carry out this action” (to make a new version of D)
• make’s functionality is broader than just compiling things, but it is mostly used for computation

Basic idea: all details for compilation are captured in a configuration file … you just invoke “make” from a shell
Makefiles

• Reasons Makefiles are great:
  – Difficult to type all the compilation commands at a prompt
  – Typical develop cycle requires frequent compilation
  – When sharing code, an expert developer can encapsulate the details of the compilation, and a new developer doesn’t need to know the details ... just “make”
Makefile syntax

• Makefiles are set up as a series of rules
• Rules have the format:
  target: dependencies
  [tab] system command
Makefile example: multiplier lib

C02LN00GFD58:code hank$ cat Makefile
lib: doubler.o tripler.o
      ar r libmultiplier.a doubler.o tripler.o
      cp libmultiplier.a ~/multiplier/lib
      cp multiplier.h ~/multiplier/include

doubler.o: doubler.c
     gcc -c doubler.c

 tripler.o: tripler.c
     gcc -c tripler.c
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$ make
      ar r libmultiplier.a doubler.o tripler.o
      cp libmultiplier.a ~/multiplier/lib
      cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
C02LN00GFD58:code hank$ touch doubler.c
C02LN00GFD58:code hank$ make
     gcc -c doubler.c
     ar r libmultiplier.a doubler.o tripler.o
     cp libmultiplier.a ~/multiplier/lib
     cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
Fancy makefile example: multiplier lib

```bash
CC=gcc
CFLAGS=-g
INSTALL_DIR=~/.multiplier

AR=ar
AR_FLAGS=r

SOURCES=doubler.c tripler.c
OBJECTS=$(SOURCES:.c=.o)

lib: $(OBJECTS)
    $(AR) $(AR_FLAGS) libmultiplier.a $(OBJECTS)
    cp libmultiplier.a $(INSTALL_DIR)/lib
    cp multiplier.h $(INSTALL_DIR)/include

c.o:
    $(CC) $(CFLAGS) -c $<
```

C02LN00GFD58:code hank$ touch doubler.c
C02LN00GFD58:code hank$ make
gcc -g -c doubler.c
ar r libmultiplier.a doubler.o tripler.o
cp libmultiplier.a ~/multiplier/lib
cp multiplier.h ~/multiplier/include
C02LN00GFD58:code hank$
Configuration management tools

• Problem:
  – Unix platforms vary
    • Where is libX installed?
    • Is OpenGL supported?

• Idea:
  – Write problem that answers these questions, then adapts build system
    • Example: put “-L/path/to/libX –lX” in the link line
    • Other fixes as well
Two popular configuration management tools

• Autoconf
  – Unix-based
  – Game plan:
    • You write scripts to test availability on system
    • Generates Makefiles based on results

• Cmake
  – Unix and Windows
  – Game plan:
    • You write .cmake files that test for package locations
    • Generates Makefiles based on results

CMake has been gaining momentum in recent years, because it is one of the best solutions for cross-platform support.
Outline

• Review
• Permissions
• Project 1B Overview
• Build
• Project 1C Overview
NOTE

• NOTE: I don’t expect to make it this far.
• If you missed class and are looking at this, note that project 1C will likely not be posted until Weds.
Project 1C

• Prompt will be posted over the weekend
• Your goal will be to generate a Makefile for a real software system.
If extra time...

• Show the difference in signatures between C and C++ (mangling)