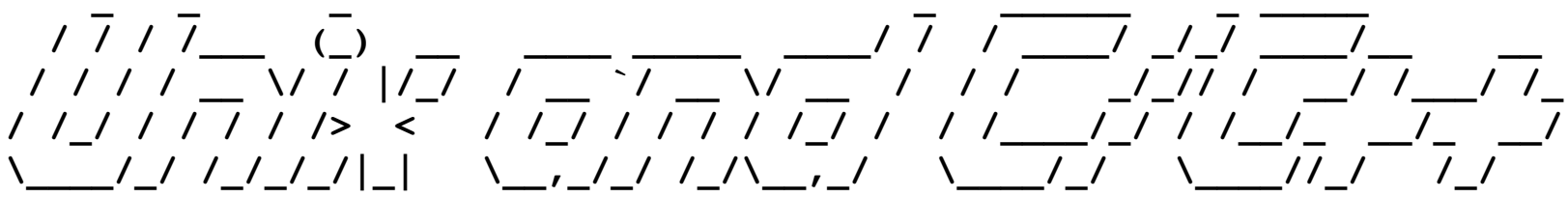
UNIVERSITY OF OREGON

CIS 330:



Lecture 4:
Build Systems, Tar, Character Strings

April 7th, 2016

Hank Childs, University of Oregon

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
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

Useful vi 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
- Navigation
  - ":100" go to line 100 in the file
  - '/': search forwards, '?': search backwards
- 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)

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.

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

Translating permissions to binary

#	Permission	rwx
7	full	111
6	read and write	110
5	read and execute	101
4	read only	100
3	write and execute	011
2	write only	010
1	execute only	001
0	none	000

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

We can have separate values (0-7) for user, group, and other

Image from wikipedia

Unix command: chmod

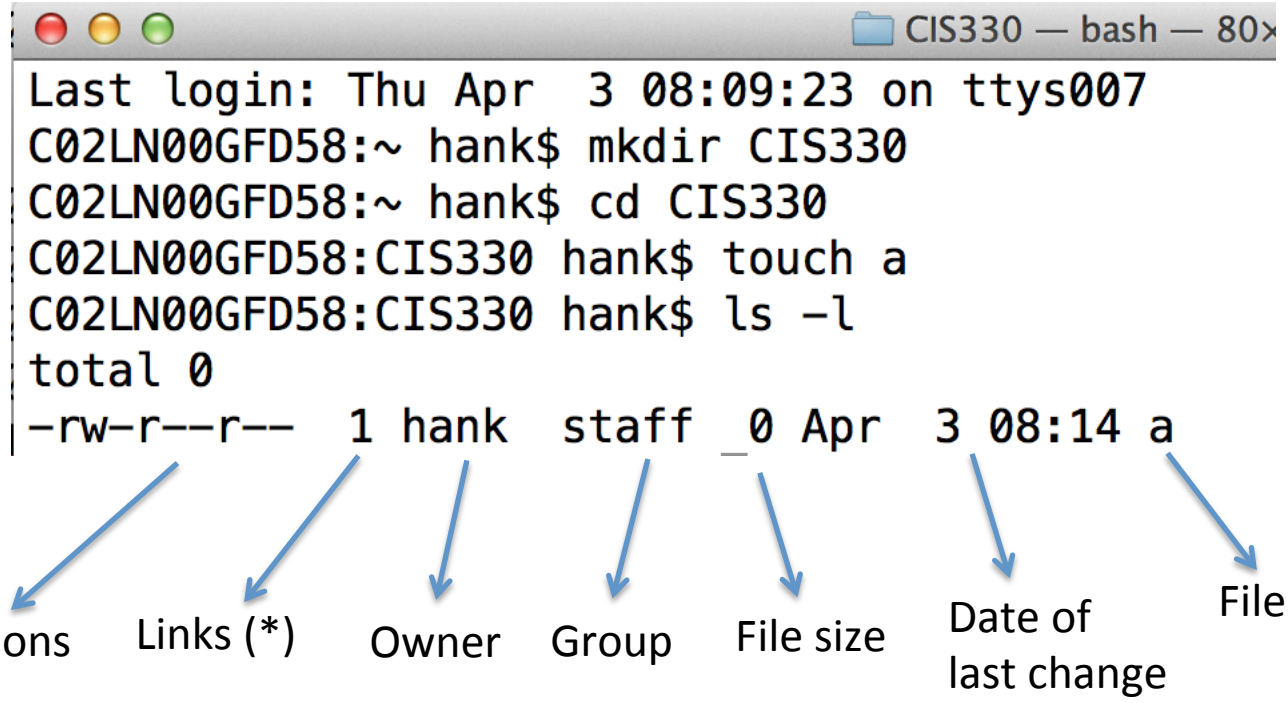
- chmod: change file mode

- 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)

ls -l

- Long listing of files



How to interpret this?

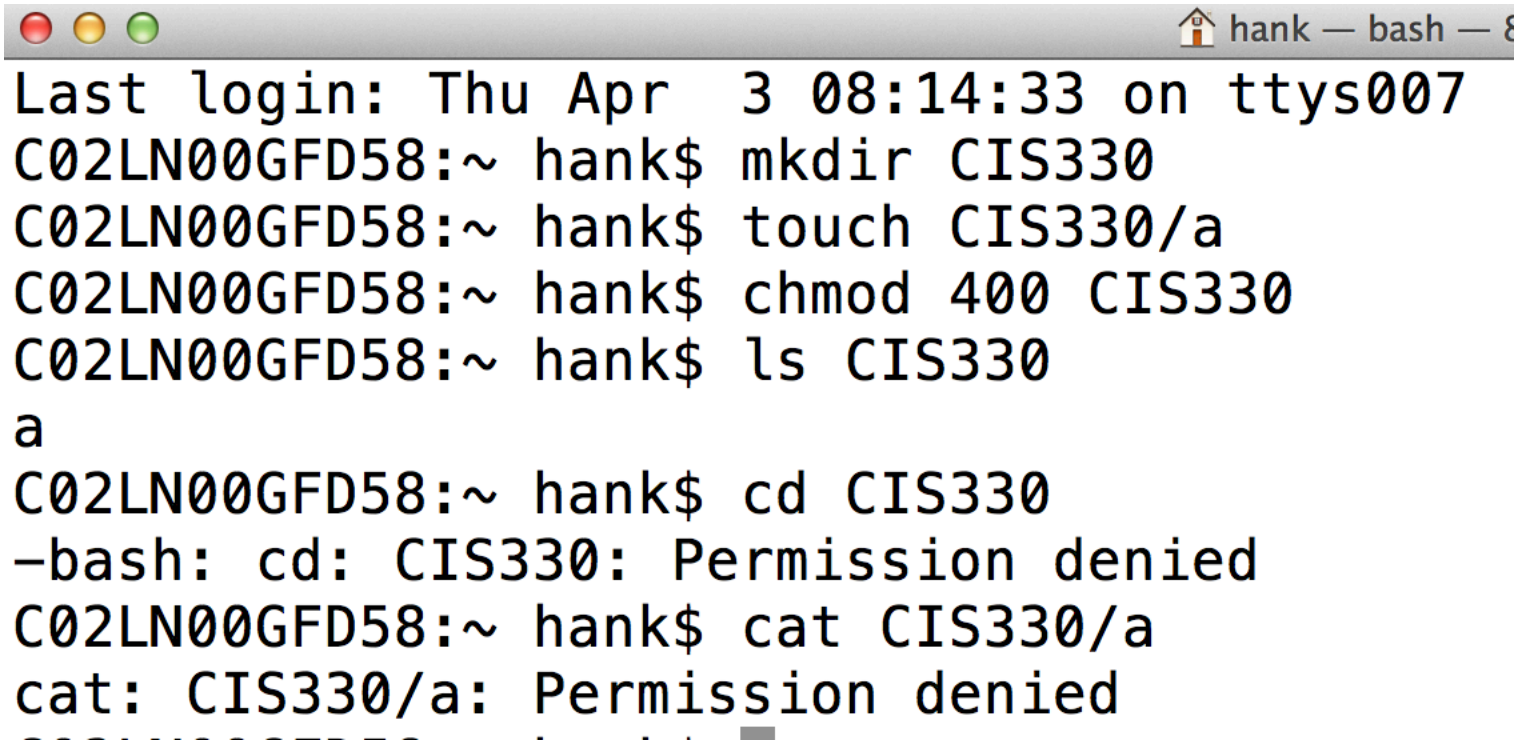
Permissions and Directories

- 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



Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

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 == foo
    - $2 == bar

Project 1B

- Summary: write a script that will create a specific directory structure, with files in the directories, and specific permissions.

Project 1B

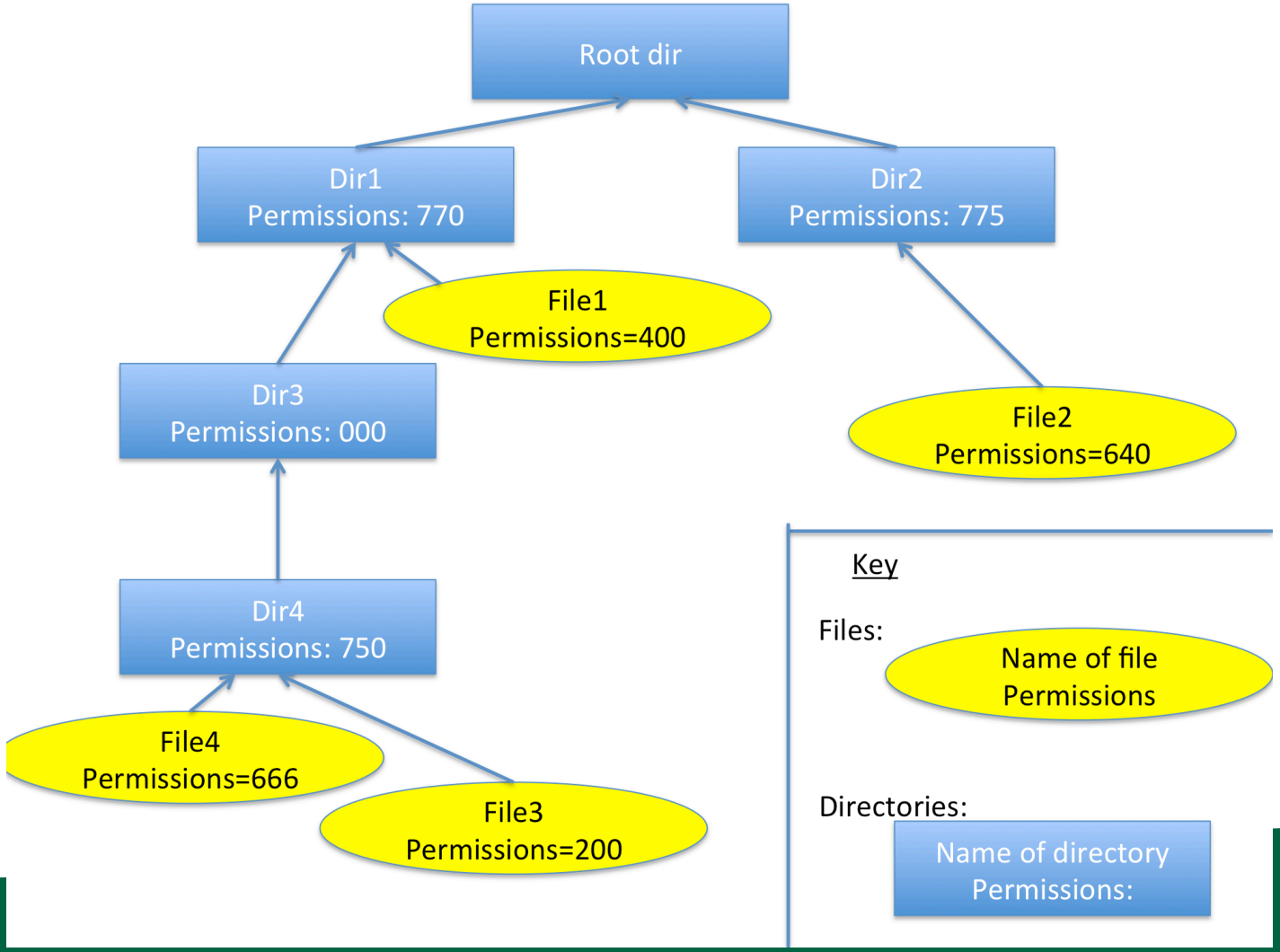
CIS 330: Project #1B
Assigned: April 3rd, 2015
Due April 8th, 2015
(which means submitted by 6am on April 9th, 2014)
Worth 2% of your grade

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:

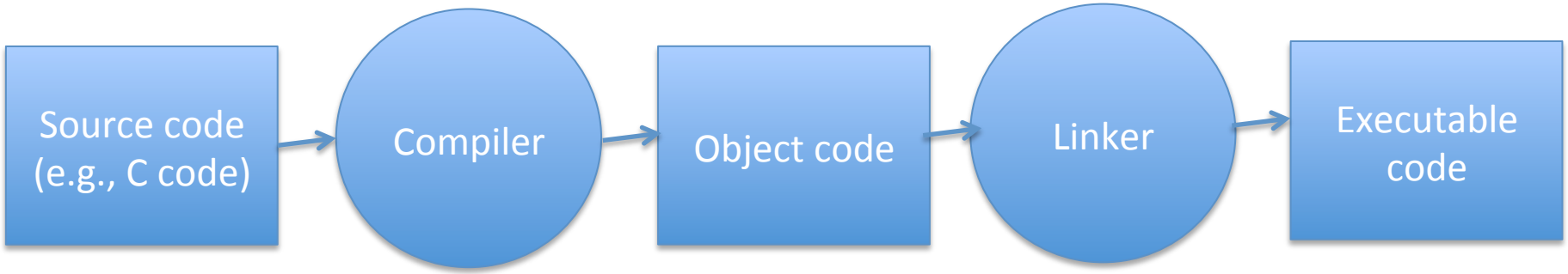


Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

Build: The Actors

- File types
  - Source code
  - Object code
  - Executable code
- Programs
  - Compiler
  - Linker



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;
```

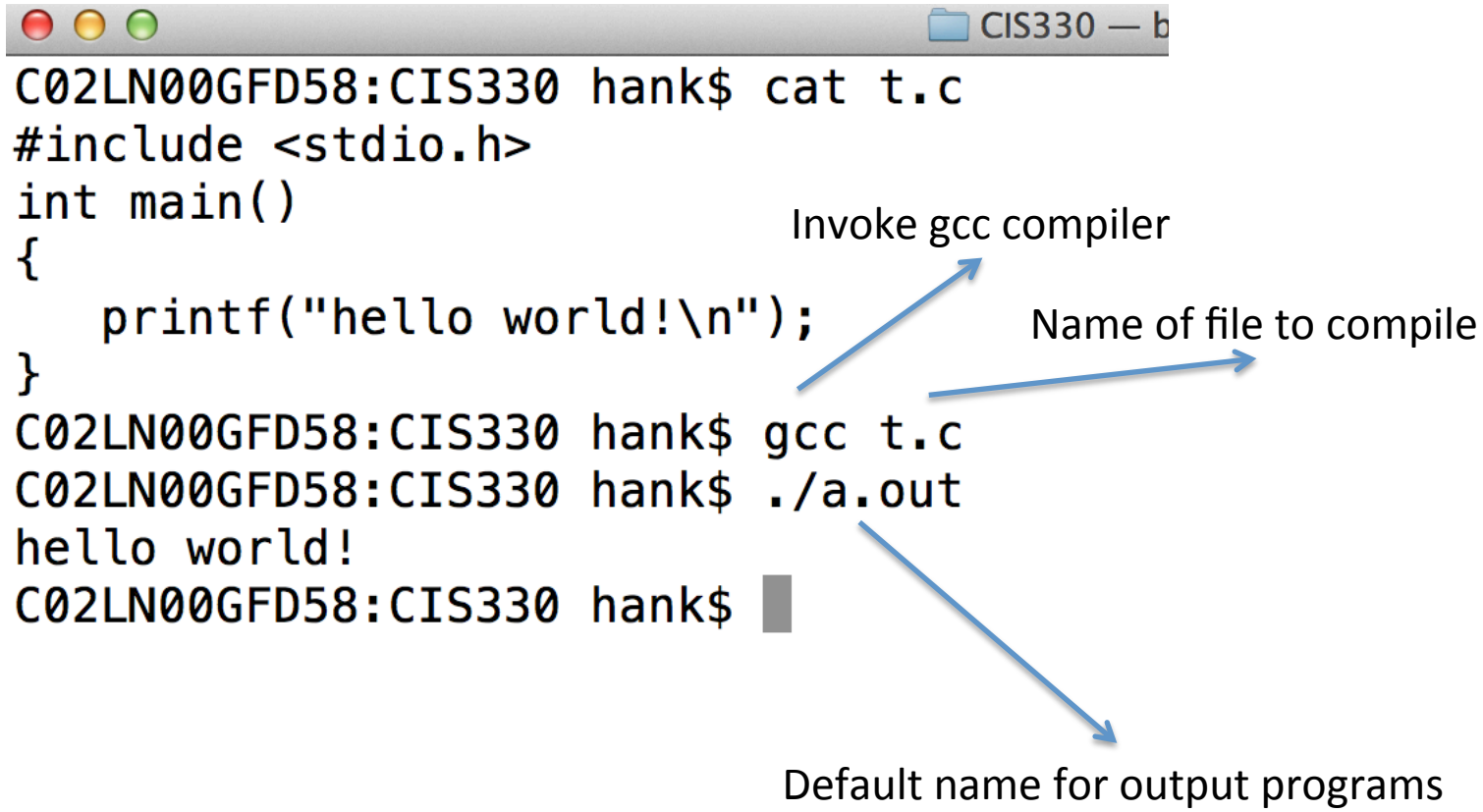
A comment on case (i.e., uppercase vs lowercase)

- Case is important in Unix
  - But Mac is tolerant
- gcc t.c

  →invokes C compiler
- gcc t.C

  → invokes C++ compiler

Our first gcc program



Our first gcc program: named output

CIS330 — bash — 80×24

```
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$
```

"-o" sets name of output

Output name is different

Output has execute permissions

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

```
int main()
{
    int sum = 0;
    int i;

    for (i = 0 ; i < 10 ; i++)
        sum += i;
    return sum;
}
```

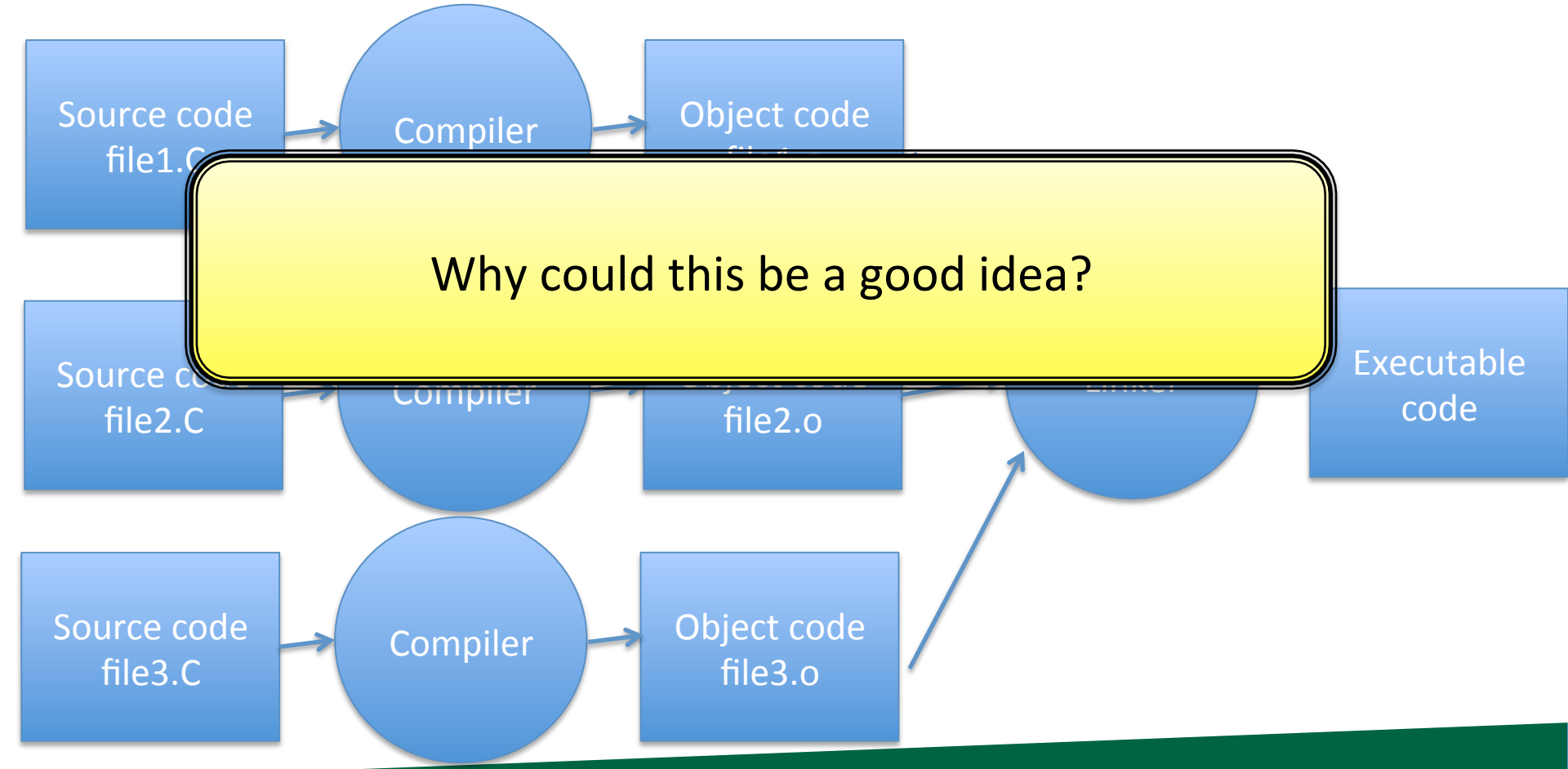
- gcc –S t.c # look at t.s
- gcc –S –g t.c # look at t.s

- (-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



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

```
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
```

```
fawcett:330 childs$ 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 childs$ 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

```
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
fawcett:330 childs$ gcc -c t2
fawcett:330 childs$ gcc -o bo
fawcett:330 childs$ ./both
fawcett:330 childs$ echo $?
10
```

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

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
    - when you say "#include <file.h>, then it looks for file.h in the directories -I points at
  - "-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

multiplier — bash — 80×24

```
C02LN00GFD58:multiplier hank$ cat multiplier.h # here's the header file
int doubler(int);
int tripler(int);
C02LN00GFD58:multiplier hank$ cat doubler.c # here's one of the c files
int doubler(int x) {return 2*x;}
C02LN00GFD58:multiplier hank$ cat tripler.c # here's the other c files
int tripler(int x) {return 3*x;}
C02LN00GFD58:multiplier hank$ gcc -c doubler.c # make an object file
C02LN00GFD58:multiplier hank$ ls doubler.o # we now have a .o
doubler.o
C02LN00GFD58:multiplier hank$ gcc -c tripler.c
C02LN00GFD58:multiplier hank$ ar r multiplier.a doubler.o tripler.o
C02LN00GFD58:multiplier hank$
```

(should have called this libmultiplier.a)

Note the '#' is the comment character

What's in the file?

multiplier — bash — 80×24

```
C02LN00GFD58:multiplier hank$ nm multiplier.a

multiplier.a(doubler.o):
0000000000000038 s EH_frame0
0000000000000000 T _doubler
0000000000000050 S _doubler.eh

multiplier.a(tripler.o):
0000000000000030 s EH_frame0
0000000000000000 T _tripler
0000000000000048 S _tripler.eh
C02LN00GFD58:multiplier hank$
```

Typical library installations

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

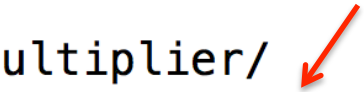
- Many standard libraries are installed in /usr
  - /usr/include
  - /usr/lib
- Compilers automatically look in /usr/include and /usr/lib (and other places)

Installing the library

multiplier — bash — 80×24

```
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
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$
```

(fixing my mistake)



"mv": unix command for renaming a file

Example: compiling with a library

CIS330 — bash — 80×24

```
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

code — bash — 80×26

```
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

code — bash — 80×26

```
C02LN00GFD58:code hank$ cat Makefile
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
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

CIS 330: Project #1C
Assigned: April 7th, 2016
Due April 12th, 2016
(which means submitted by 6am on April 13th, 2016)
Worth 2% of your grade

Assignment: Download the file "Proj1C.tar". This file contains a C-based project. You will build a Makefile for the project, and also extend the project.

Project 1C

== Build a Makefile for math330 ==

Your Makefile should:

(1) create an include directory
(2) copy the Header file to the include directory
(3) create a lib directory
(4) compile the .c files in trig and exp as object files (.o's)
(5) make a library
(6) install the library to the lib directory
(7) compile the "cli" program against the include and library directory

== Extend the math330 library ==

You should:

(1) add 3 new functions: arccos, arcsin, and arctan (each in their own file)
(2) Extend the "cli" program to support these functions
(3) Extend your Makefile to support the new functions

Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

Unix command: tar

- Anyone know what tar stands for?

tar = tape archiver

IBM tape library

Unix command: tar

- Problem: you have many files and you want to…
  - move them to another machine
  - give a copy to a friend
  - etc.
- Tar: take many files and make one file
  - Originally so one file can be written to tape drive
- Serves same purpose as “.zip” files.

Unix command: tar

- tar cvf 330.tar file1 file2 file3
  - puts 3 files (file1, file2, file3) into a new file called 330.tar
- scp 330.tar @ix:~
- ssh ix
- tar xvf 330.tar
- ls

  file1 file2 file

Outline

- Review
- Project 1B Overview
- Build
- Project 1C Overview
- Tar
- Character Strings

ASCII Character Set

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

There have been various extensions to ASCII …
now more than 128 characters
Many special characters are handled outside this convention

image source: granneman.com

signed vs unsigned chars

- signed char (“char”):
  - valid values: -128 to 127
  - size: 1 byte
  - used to represent characters with ASCII
    - values -128 to -1 are not valid
- unsigned char:
  - valid values: 0 to 255
  - size: 1 byte
  - used to represent data

character strings

- A character “string” is:
  - an array of type “char”
  - that is terminated by the NULL character
- Example:

  char str[12] = “hello world”;
  - str[11] = ‘\0’ (the compiler did this automatically)
- The C library has multiple functions for handling strings

Character strings example

```
128-223-223-72-wireless:330 hank$ cat string.c
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char *str2 = str+6;

    printf("str is \"%s\" and str2 is \"%s\"\n",
           str, str2);

    str[5] = '\0';

    printf("Now str is \"%s\" and str2 is \"%s\"\n",
           str, str2);
}
128-223-223-72-wireless:330 hank$ gcc string.c
128-223-223-72-wireless:330 hank$ ./a.out
str is "hello world" and str2 is "world"
Now str is "hello" and str2 is "world"
```

Useful C library string functions

- strcpy: string copy
- strncpy: string copy, but just first N characters
- strlen: length of a string

```
128-223-223-72-wireless:330 hank$ cat strcpy.c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[6], str3[7];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
128-223-223-72-wireless:330 hank$ gcc strcpy.c
128-223-223-72-wireless:330 hank$ ./a.out
world,hello
```

Useful C library string functions

- strcpy: string copy
- strncpy: string copy, but just first N characters
- strlen: length of a string

```
128-223-223-72-wireless:330 hank$ cat strcpy.c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[7], str3[6];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
128-223-223-72-wireless:330 hank$ gcc strcpy.c
128-223-223-72-wireless:330 hank$ ./a.out
world,hello world
```

What happened here?

More useful C library string functions

Functions

Copying:

memcpy	Copy block of memory (function)
memmove	Move block of memory (function)
strcpy	Copy string (function)
strncpy	Copy characters from string (function)

Concatenation:

strcat	Concatenate strings (function)
strncat	Append characters from string (function)

Comparison:

memcmp	Compare two blocks of memory (function
strcmp	Compare two strings (function)
strcoll	Compare two strings using locale (functior
strncmp	Compare characters of two strings (functi
strxfrm	Transform string using locale (function)

Searching:

memchr	Locate character in block of memory (function)
strchr	Locate first occurrence of character in string (fu
strcspn	Get span until character in string (function)
strpbrk	Locate characters in string (function)
strrchr	Locate last occurrence of character in string (fu
strspn	Get span of character set in string (function)
strstr	Locate substring (function)
strtok	Split string into tokens (function)

Other:

memset	Fill block of memory (function)
strerror	Get pointer to error message string (function)
strlen	Get string length (function)

Macros

NULL	Null pointer (macro)

Types

size_t	Unsigned integral type (type)

source: cplusplus.com