Lecture 4:
Build Systems, Tar, Character Strings

Lecture 5:
Finish up memory overview
Build: The Actors

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

- Programs
  - Compiler
  - Linker
Analogy

Source 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;
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

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$  

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$
```

"-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 ... shouldn’t fail, but can make “undefined behavior” worse and no longer has to mimic your code

• “gcc –O3 –g”
  – This is fine, but –g may bloat executables (possible slowdown)
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

```c
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.
How To Interpret Previous Slide

• gcc –c t1.c
  – → use the gcc compiler to create the object code file t1.o from the source code file t1.c
  – → “-c” is the flag that tells gcc that it should create just object code, and not try to call a linker

• gcc –c t2.c
  – Same as above

• gcc –o both t2.o t1.o
  – → use the gcc compiler to invoke a linker that will take the object code files t2.o and t1.o and combine them to make the executable code “both”
Multi-file development: example

```bash
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
```
Multi-file development: example

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

```c
int main() {
    return doubler(5);
}
```

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 function prototypes 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 connections 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

Note the ‘#’ is the comment character

```bash
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)
```
nm: What’s in the file?

C02LN00GFD58:multiplier hank$ nm multiplier.a

multiplier.a(doubler.o):
00000000000000038  s  EH_frame0
0000000000000000  T  _doubler
00000000000000050  S  _doubler.eh

multiplier.a(tripler.o):
00000000000000030  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

```
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$
```

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

Example:

```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/include -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 compilation

Basic idea: all details for compilation are captured in a 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$
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
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
c02lnn00gfd58: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 program 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.
File I/O
File I/O: streams and file descriptors

• Two ways to access files:
  – File descriptors:
    • Lower level interface to files and devices
      – Provides controls to specific devices
    • Type: small integers (typically 20 total)
  – Streams:
    • Higher level interface to files and devices
      – Provides uniform interface; easy to deal with, but less powerful
    • Type: FILE *

Streams are more portable, and more accessible to beginning programmers. (I teach streams here.)
File I/O

• Process for reading or writing
  – Open a file
    • Tells Unix you intend to do file I/O
    • Function returns a “FILE *
      – Used to identify the file from this point forward
    • Checks to see if permissions are valid
  – Read from the file / write to the file
  – Close the file
Opening a file

• FILE *handle = fopen(filename, mode);

  The argument mode points to a string beginning with one of the following sequences (Additional characters may follow these sequences.):

  \`r\'\' Open text file for reading. The stream is positioned at the beginning of the file.

  \`r+\'\' Open for reading and writing. The stream is positioned at the beginning of the file.

  \`a+\'\' Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

  \`w+\'\' Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file. Subsequent writes to the file will always end up at the then current end of file, irrespective of any intervening fseek(3) or similar.

Example: FILE *h = fopen("/tmp/330", "wb");

Close when you are done with “fclose”

Note: #include <stdio.h>
Reading / Writing

FREAD(3) BSD Library Functions Manual FREAD(3)

NAME
fread, fwrite -- binary stream input/output

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <stdio.h>

size_t
fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream);

size_t
fwrite(const void *restrict ptr, size_t size, size_t nitems,
       FILE *restrict stream);

DESCRIPTION
The function fread() reads nitems objects, each size bytes long, from the stream
pointed to by stream, storing them at the location given by ptr.

The function fwrite() writes nitems objects, each size bytes long, to the stream
pointed to by stream, obtaining them from the location given by ptr.

RETURN VALUES
The functions fread() and fwrite() advance the file position indicator for the
stream by the number of bytes read or written. They return the number of objects
read or written. If an error occurs, or the end-of-file is reached, the return
value is a short object count (or zero).
Example

C02LN00GFD58:330 hank$ cat rw.c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char *hello = "hello world: file edition\n";
    FILE *f = fopen("330", "w");
    fwrite(hello, sizeof(char), strlen(hello), f);
    fclose(f);
}
C02LN00GFD58:330 hank$ gcc rw.c
C02LN00GFD58:330 hank$ ./a.out
C02LN00GFD58:330 hank$ cat 330
hello world: file edition
File Position Indicator

- File position indicator: the current location in the file
- If I read one byte, the one byte you get is where the file position indicator is pointing.
  - And the file position indicator updates to point at the next byte
  - But it can be changed...
The `fseek()` function sets the file position indicator for the stream pointed to by `stream`. The new position, measured in bytes, is obtained by adding `offset` bytes to the position specified by `whence`. If `whence` is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the offset is relative to the start of the file, the current position indicator, or end-of-file, respectively. A successful call to the `fseek()` function clears the end-of-file indicator for the stream and undoes any effects of the `ungetc(3)` and `ungetwc(3)` functions on the same stream.
The `ftell()` function obtains the current value of the file position indicator for the stream pointed to by `stream`. 
We have everything we need to make a copy command...

- fopen
- fread
- fwrite
- fseek
- ftell

Can we do this together as a class?
argc & argv

- two arguments to every C program
- argc: how many command line arguments
- argv: an array containing each of the arguments
- "./a.out hank childs"
- \( \rightarrow \) argc == 3
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) 
{
    FILE *f_in, *f_out;
    int buff_size;
    char *buffer;

    if (argc != 3)
    {
        printf("Usage: %s <file1> <file2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    f_in = fopen(argv[1], "r");
    fseek(f_in, 0, SEEK_END);
    buff_size = ftell(f_in);
    fseek(f_in, 0, SEEK_SET);

    buffer = malloc(buff_size);
    fread(buffer, sizeof(char), buff_size, f_in);

    printf("Copying %d bytes from %s to %s\n", buff_size, argv[1], argv[2]);

    f_out = fopen(argv[2], "w");
    fwrite(buffer, sizeof(char), buff_size, f_out);

    fclose(f_in);
    fclose(f_out);

    return 0;
}
Return values in shells

C02LN00GFD58:330 hank$ ./a.out copy.c copy2.c
Copying 697 bytes from copy.c to copy2.c
C02LN00GFD58:330 hank$ echo $? 
0
C02LN00GFD58:330 hank$ ./a.out copy.c
Usage: ./a.out <file1> <file2>
C02LN00GFD58:330 hank$ echo $? 
1

$? is the return value of the last executed command
Printing to terminal and reading from terminal

• In Unix, printing to terminal and reading from terminal is done with file I/O

• Keyboard and screen are files in the file system!
  – (at least they were ...)


Standard Streams

• Wikipedia: “preconnected input and output channels between a computer program and its environment (typically a text terminal) when it begins execution”

• Three standard streams:
  – stdin (standard input)
  – stdout (standard output)
  – stderr (standard error)

What mechanisms in C allow you to access standard streams?
printf

• Print to stdout
  – printf(“hello world\n”);
  – printf(“Integers are like this %d\n”, 6);
  – printf(“Two floats: %f, %f”, 3.5, 7.0);
fprintf

• Just like printf, but to streams
  • fprintf(stdout, “helloworld\n”);
    – → same as printf
  • fprintf(stderr, “helloworld\n”);
    – prints to “standard error”
  • fprintf(f_out, “helloworld\n”);
    – prints to the file pointed to by FILE *f_out.
buffering and printf

• Important: printf is buffered
• So:
  – printf puts string in buffer
  – other things happen
  – buffer is eventually printed
• But what about a crash?
  – printf puts string in buffer
  – other things happen ... including a crash
  – buffer is never printed!

Solutions: (1) fflush, (2) fprintf(stderr) always flushed
Outline

• Review
• File I/O
• Project 2B
• Redirection
• Pipes
Project 2B

Worth 4% of your grade

Assignment: Write a program that reads the file “2E_binary_file”. This file contains a two-dimensional array of integers, that is 10x10. You are to read in the 5x5 bottom left corner of the array. That is, the values 0-4, 10-14, 20-24, 30-34, and 40-44. You may only read 25 integers total. Do not read all 100 and throw some out. You will then write out the new 5x5 array. Please write this as strings, one integer per line (25 lines total). You should be able to “cat” the file afterwards and see the values.

Use Unix file streams for this project (i.e., fopen, fread, fseek, fprintf). Your program will be checked for good programming practices. Close your file streams, use memory correctly, etc. I am not referring to style, variable initialization, etc.

Also, add support for command line arguments (argc and argv).

Your program should run as:

./<prog_name> <input_name> <output_name>

(The input_name will be 2E_binary_file, unless you change it.)

Finally, note that I am handing you a binary file. I think we are all little endian, and so it will be fine. But, if it is big endian, then we will have a problem. You can check if it is little endian by printing the first two values of the file. They should be “0” and “1”.

Please submit a tarball with (1) a Makefile (should be simple), (2) your source code, and (3) the output ASCII file from running your program, with the name “ASCII_output”.)
Outline

• Review
• File I/O
• Project 2B
• Redirection
• Pipes
Unix shells allows you to manipulate standard streams.

- “>” redirect output of program to a file
- Example:
  - `ls > output`
  - `echo "this is a file" > output2`
  - `cat file1 file2 > file3`
Unix shells allows you to manipulate standard streams.

- “<” redirect file to input of program
- Example:
  - `python < myscript.py`
    - Note: python quits when it reads a special character called EOF (End of File)
    - You can type this character by typing Ctrl-D
    - This is why Python quits when you type Ctrl-D
      - (many other programs too)
Unix shells allows you to manipulate standard streams.

- “>>” concatenate output of program to end of existing file
  - (or create file if it doesn’t exist)

- Example:
  - echo “I am starting the file” > file1
  - echo “I am adding to the file” >> file1
  - cat file1
    - I am starting the file
    - I am adding to the file
What's happening here?

```
C02LN00GFD58:330 hank$ mkdir tmp
C02LN00GFD58:330 hank$ cd tmp
C02LN00GFD58:tmp hank$ touch f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out
ls: f2: No such file or directory
C02LN00GFD58:tmp hank$ cat out
f1
```

ls is outputting its error messages to stderr
Redirecting stderr in a shell

C02LN00GFD58:Documents hank$ cd ~/330
C02LN00GFD58:330 hank$ mkdir tmp
C02LN00GFD58:330 hank$ cd tmp
C02LN00GFD58:tmp hank$ touch f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out
ls: f2: No such file or directory
C02LN00GFD58:tmp hank$ cat out f1
C02LN00GFD58:tmp hank$ ls f1 f2 > out 2>out_error
C02LN00GFD58:tmp hank$ cat out_error
ls: f2: No such file or directory
Redirecting stderr to stdout

Convenient when you want both to go to the same stream
Outline

• Review
• File I/O
• Project 2B
• Redirection
• Pipes
C functions: fork and pipe

• fork: duplicates current program into a separate instance
  – Two running programs!
  – Only differentiated by return value of fork (which is original and which is new)

• pipe: mechanism for connecting file descriptors between two forked programs

Through fork and pipe, you can connect two running programs. One writes to a file descriptor, and the other reads the output from its file descriptor.

Only used on special occasions. (And one of those occasions is with the shell.)
pipes in Unix shells

- represented with “|”
- output of one program becomes input to another program

```c
#include <stdio.h>

int main() {
    int ch = getc(stdin);
    while (ch != EOF) {
        printf("%c%c", ch, ch);
        ch = getc(stdin);
    }
}
```

```bash
C02LN00GFD58:tmp hank$ gcc -o printer printer.c
C02LN00GFD58:tmp hank$ gcc -o doubler doubler.c
C02LN00GFD58:tmp hank$ ./printer | ./doubler
HHeelllloo  wwooorrlldd
```

```bash
C02LN00GFD58:tmp hank$
```
Very useful programs

- grep: keep lines that match pattern, discard lines that don’t match pattern
Very useful programs

• **sed**: replace pattern 1 with pattern 2
  – `sed s/pattern1/pattern2/g`
    • `s` means substitute
    • `g` means “global” ... every instance on the line

  sed is also available in “vi”
  :%s/pattern1/pattern2/g (% means all lines)
  :103,133s/p1/p2/g (lines 103-133)
Wildcards

• ‘*’ is a wildcard with unix shells

```
fawcett:tmp child$ ls
Abe Chavarria Hebb Macy Smith
Alajaji Chen Jia Maguire Steelhammer
Alamoudi Clark Kine Michlanski Szczepanski
Anastas Collier Lee Moreno Totten
Andrade Costello Legge Olson Vega-Fujioka
Ballarche Donnelly Li Owen Wang
Brennan Etzel Lin Pogrebinsky Whiteley
Brockway Friedrich Liu Qin Woodruff
Brogan Garvin Lopes Rhodes Xu
Brooks Gonzales Luo Roberts Yaconelli
Bruce Guo Lynch Rodriguez Young
Carlton Hampton Lyon Roush Zhang
Chalmers Harris Machado Rozenboim de
```

'?' is a wildcard that matches exactly one character
Other useful shell things

• ‘tab’: auto-complete
• esc=: show options for auto-complete
• Ctrl-A: go to beginning of line
• Ctrl-E: go to end of line
• Ctrl-R: search through history for command