

Unit 1: C Programming and Intro to Linux

A crash course for C++/Windows
Users

C vs. C++: Differences

- C does not have classes/objects!
 - all code is in functions (subroutines).
- C structures can not have methods
- C I/O is based on library functions:
 - printf, scanf, fopen, fclose, fread, fwrite, ...

C vs. C++: Differences (cont.)

- C does not support any function overloading (you can't have 2 functions with the same name).
- C does not have `new` or `delete`, you use `malloc()` and `free()` library functions to handle dynamic memory allocation/deallocation.
- C does not have *reference variables*

C vs. C++: Similarities

- Built-in data types: `int`, `double`, `char`, etc.
- Preprocessor (handles `#include`, `#define`, etc.)
- Control structures: `if`, `while`, `do`, `for`, etc.
- Operators: `+` `-` `*` `/` `=` `==` `!=` `<` `>` `+=` `++` etc.

C vs. C++: Similarities (cont.)

- There must be a function named `main()` .
- function definitions are done the same way.
- Can split code in to files (object modules) and link modules together.

Evolution of C

- Traditional C: 1978 by K&R
- Standard C: 1989 (aka ANSI C)
- Standard C: 1995, amendments to C89 standard
- Standard C: 1999, is the new definitive C standard replace all the others.
- GCC is a C99 compliant compiler (mostly, I think :-)).

Standard C (C89)

- The addition of truly standard library
 - libc.a, libm.a, etc.
- New processor commands and features
- Function prototypes -- argument types specified in the function declaration
- New keywords: const, volatile, signed
- Wide chars, wide strings and multibyte characters.
- Clarifications to conversion rules, declarations and type checking

Standard C (C95)

- 3 new std lib headers: `iso646.h`, `wctype.h` and `wchar.h`
- New formatting codes for `printf/scanf`
- A large number of new functions.

Standard C (C99)

- Complex arithmetic
- Extensions to integer types, including the longer standard type (long long, long double)
- Boolean type (stdbool.h)
- Improved support for floating-point types, including math functions for all types
- C++ style comments (//)
- For-loop variable initialization
 - `for (int i = 0; ...; ...) { }`

Simple C Program

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return(0);
}
```

Another Program

```
#include <stdio.h>

void printhello(int n) {
    int i;
    for (i=0;i<n;i++)
        printf("Hello World\n");
}

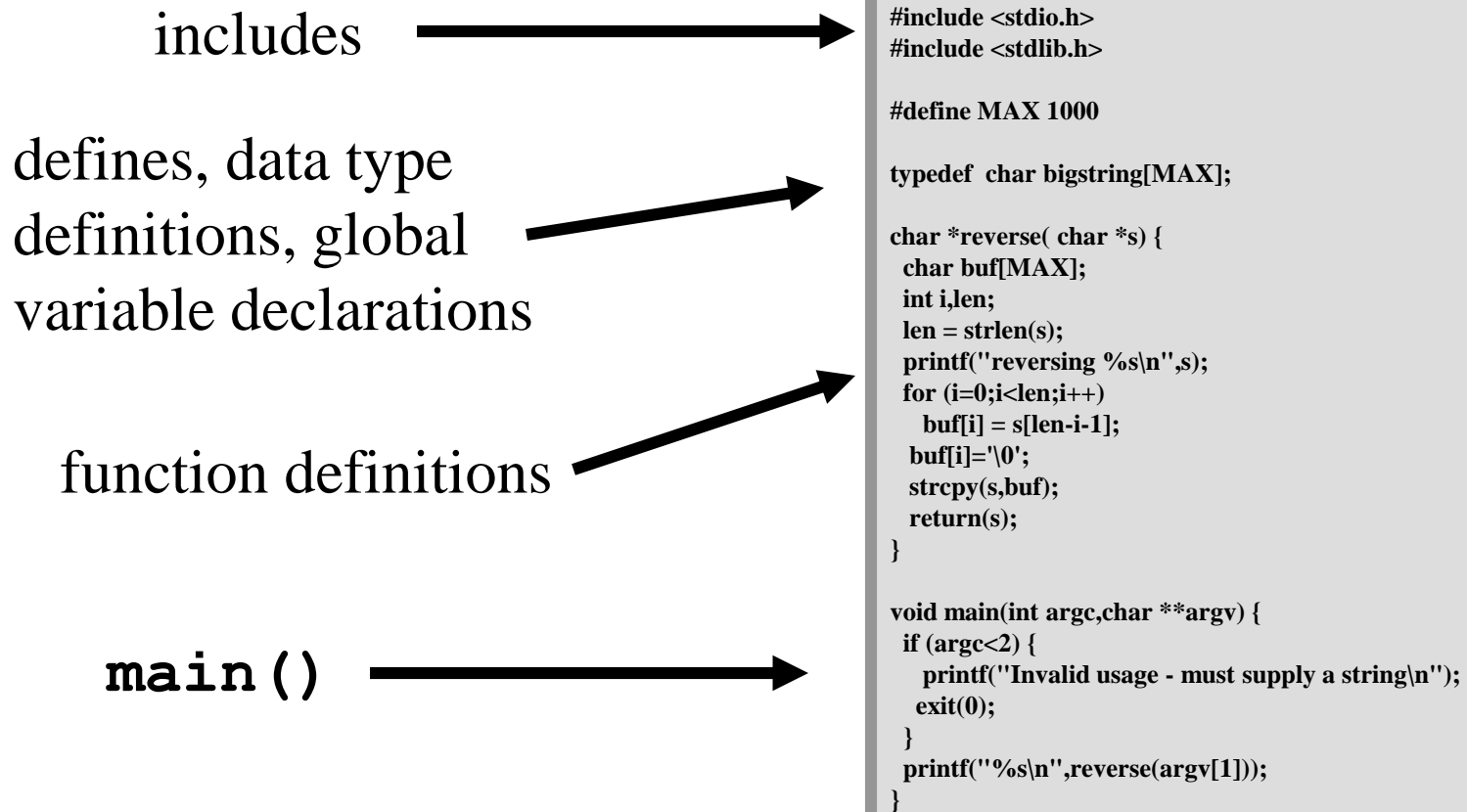
void main() {
    printhello(5);
}
```

Valid C Program?!

- Is the following program a valid C file?

```
int foo(int & x)
{
    return x * x;
}
int main()
{
    int y = 4;
    return foo(y);
}
```

Typical C Program



A Real C Example Program

- Program that accepts one command line argument.
- Treats the command line argument as a string, and reverses the letters in the string.
- Prints out the result (the reversed string).

reverse.c - part 1

```
#include <stdio.h>  /* printf */
#include <stdlib.h> /* malloc, free */

/* MAX is the size of the largest string
   we can handle */
#define MAX 1000

/* bigstring is a new data type */
typedef char bigstring[MAX];
```

reverse.c - part 2

```
/* reverses a string in place
   returns a pointer to the string
*/
char *reverse( char *s ) {
    bigstring buf;
    int i,len;
    len = strlen(s); /* find the length */
    for (i=0;i<len;i++)
        buf[i] = s[len-i-1];
    buf[i]='\0';      /* null terminate! */
    strcpy(s,buf);    /* put back in to s */
    return(s);
}
```


reverse.c - part 3

```
void main(int argc, char **argv) {  
    if (argc < 2) {  
        printf("Invalid usage - must supply a  
string\n");  
        exit(0);  
    }  
    printf("%s\n", reverse(argv[1]));  
}
```

Using dynamic allocation

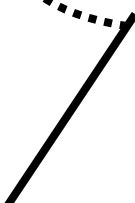
```
char *reverse( char *s) {  
    char *buf;  
    int i,len;  
    len = strlen(s);  
  
    /* allocate memory len + 1 for null term */  
    buf = (char *)malloc(len+1);  
    for (i=0;i<len;i++)  
        buf[i] = s[len-i-1];  
    buf[i]='\0';  
    strcpy(s,buf);  
    free(buf);  
    return(s);  
}
```

Compiling on Unix

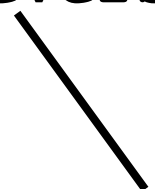
Traditionally the name of the C compiler that comes with Unix is “cc”.

We can use the Gnu compiler named “gcc”.

```
gcc -Wall -o reverse reverse.c
```



tells the compiler to
create executable file
with the name **reverse**



tells the compiler
the name of the input
file.

Running the program

```
>./reverse Hello  
olleH
```

```
>./reverse This is a long string  
sihT
```

```
>./reverse "This is a long string"  
gnirts gnol a si sihT
```

C Libraries

- Standard I/O: printf, scanf, fopen, fread, ...
- String functions: strcpy, strspn, strtok, ...
- Math: sin, cos, sqrt, exp, abs, pow, log, ...
- System Calls: fork, exec, signal, kill, ...

Quick I/O Primer - `printf`

```
int printf( const char *, . . . );
```

. . . means “variable number of arguments”.
The first argument is required (a string).

Given a simple string, `printf` just prints the string (to *standard output*).

Simple printf

```
printf("Hi Dr. J., I'm Dr. G!\n");
```

```
printf("I\t have\t\t tabs\n");
```

```
char s[100];
```

```
strcpy(s, "printf is fun!\n");
```

```
printf(s);
```

arguing with printf

You can tell `printf` to embed some values in the string – these values are determined at run-time.

```
printf("here is an integer: %d\n", i);
```

the `%d` is replaced by the value of the argument following the string (in this case `i`).

More integer arguments

```
printf("%d + %d = %d\n", x, y, x+y) ;
```

```
for (j=10; j>=0; j--) {  
    printf("%d\n", j);    /* countdown */  
}
```

```
printf("%d is my favorite number\n", 13) ;
```

printf is dumb

- `%d` is replaced by the value of the parameter when treated as a integer.
- If you give `printf` something that is not an integer – it doesn't know!

```
printf("Print an int %d\n", "Hi Dr. J.");
```

```
Print an int 134513884
```

Other formats

- `%d` is a format – it means “treat the parameter as a signed integer”
- `%u` means unsigned integer
- `%x` means print as hexadecimal
- `%s` means “treat it as a string”
- `%c` is for characters (`char`)
- `%f` is for floating point numbers
- `%%` means print a single ‘%’

Fun with printf

```
char *s = "Hi Dr. J";  
printf("The string \"%s\" is %d  
characters long\n", s, strlen(s));  
  
printf("The square root of 10 is  
%f\n", sqrt(10));
```

Controlling the output

- There are formatting options that you can use to control field width, precision, etc.

```
printf("The square root of 10  
is %20.15f\n", sqrt(10));
```

```
The square root of 10 is  
3.162277660168380
```

Lining things up

```
int i;  
for (i=1;i<5;i++)  
    printf("%2d %f %20.15f\n",  
           i,sqrt(i),sqrt(i));
```

| | | |
|---|----------|----------------------|
| 1 | 1.000000 | 1.000000000000000000 |
| 2 | 1.414214 | 1.414213562373095 |
| 3 | 1.732051 | 1.732050807568877 |
| 4 | 2.000000 | 2.000000000000000000 |

Alas, we must move on

- There are more formats and format options for `printf`.
- The *man page* for `printf` includes a complete description (any decent C book will also).
- NOTE: to view the man page for `printf` you should type the following: `man 3 printf` (in linux)

Input - scanf

- `scanf` provides input from *standard input*.
- `scanf` is every bit as fun as `printf`!
- `scanf` is a little scary, you need to use pointers
- Actually, you don't really need pointers, just addresses.

Remember Memory?

- Every C variable is stored in memory.
- Every memory location has an *address*.
- In C you can use variables called *pointers* to refer to variables by their address in memory.

scanf

```
int scanf(const char *format, ...);
```

- Remember “...” means “variable number of arguments”
- Looks kinda like `printf`

What `scanf` does

- Uses format string to determine what kind of variable(s) it should read.
- The arguments are the *addresses* of the variables.
- The `&` operator here means “Take the address of”:

```
int x, y;  
scanf ("%d %d", &x, &y) ;
```

A simple example of `scanf`

```
float x;
```

```
printf("Enter a number\n");
```

```
scanf("%f", &x);
```

```
printf("The square root of %f is  
%f\n", x, sqrt(x));
```

scanf and strings

Using `%s` in a `scanf` string tells `scanf` to read the next *word* from input – NOT a line of input:

```
char s[100]; // ALLOC SPACE!!  
printf("Type in your name\n");  
scanf("%s", s); // note: s is a char *  
printf("Your name is %s\n", s);
```

man scanf

- Check out the man page for more details.

Reading a line

- You can use the function `fgets` to read an entire line:

```
char *fgets(char *s, int size,  
            FILE *stream);
```

`size` is the maximum # chars

`FILE` is a *file handle*

Using `fgets` to read from `stdin`

```
char s[101];
```

```
printf("Type in your name\n");  
fgets(s,100,stdin);
```

```
printf("Your name is %s\n",s);
```


Other I/O stuff

- `fopen, fclose`
- `fscanf, fprintf, fgets`
- `fread, fwrite`
- Check the man pages for the details.

String functions

```
char *strcpy(char *dest,  
             const char *src);
```

```
size_t strlen(const char *s);
```

```
char *strtok(char *s,  
            const char *delim);
```

Math library

- The math library is often provided as an external library (not as part of the *standard C library*).
- You must tell the compiler you want the math library:

```
gcc -o myprog myprog.c -lm
```



means “add in the math library”

Pop Quiz

- What happens when we compile, link and run the following code in lec02_01.c:

```
#include <stdio.h>
```

```
#include <math.h>
```

```
int main(void) {  
    printf("%.2f\n", sqrt(-1));  
}
```

using

```
gcc lec02_01.c
```

```
./a.out
```

Useful Predefined MACROS

- `__LINE__` : line # in source code file (%d)
- `__FILE__` : name of current source code file (%s)
- `__DATE__` : date “Mmm dd yyy” (%s)
- `__TIME__` : time of day, “hh:mm:ss” (%s)
- `__STDC__` : 1 if compiler is ISO compliant
- `__STDC_VERSION__` : integer (%s)

Editors: Vi(m), Emacs or Pico/Nano

- Emacs – it more than an editor, it's a full-on environment. Steep learning curve, but yields the greatest productivity for most developers.
- Vi – the grandfather of editors. Small memory foot print.
- Pico/Nano – easy to use but lacks a lot of advanced features developers want.

Pop Quiz (earn an A or an F)

- The best editor is:
 - A: vi (or derivatives)
 - B: emacs (or derivatives)
 - C: Notepad
 - D: Notepad++
 - E: Kate
 - F: GNU nano
 - G: Sublime Text
 - H: Vendor specific (Visual Studio / Xcode)

Review: C is a “subset” of C++

- Primary difference is the C has nothing to do with classes.
 - no constructors
 - no destructors
 - no inheritance
 - no operator overloading
 - no new or delete operator – memory allocated via system calls
 - no templates (no STL)
 - no String or Boolean (C89) types only base types: **char**, **short**, **int**, **float**, and **double**

O-O still possible with C

- Keyword **struct** is used to declare a record or “methodless” class, like

struct Student

```
{  
    char first_name[32];  
    char last_name[32];  
    unsigned int id;  
    double gpa;  
};
```

Using a struct

```
int main()
{
    struct Student Suzy;
    /* init data members by hand */
    /* strcpy is a standard libC function */
    strcpy( Suzy.last_name, "Chapstick");
    Suzy.id = 12345;
}
```

Variable Declarations

(pre-C99: for most modern C compilers, this won't be an issue, but it's good to know/recognize when reading older code)

```
int main()
{
    struct Student Suzy;
    strcpy( Suzy.last_name, "Chapstick");
    int i; /* WRONG!! */
    struct Student Sam; /* WRONG !*/
    Suszy.id = 12345;
}
```

All vars must be declared before the first executable statment in a function or block.

This has a significant impact on your “for” loops!

“for” Loops in C

- In C++ you will typically do:
 - `for(int i=0; i < num; i++)`
- In C you **MUST** do:
 - `int i; /* top of scope */`
 - `for(i = 0; i < num; i++)`
 - note, “i” exists outside of the for loop scope!
 - **NO LONGER TRUE IN C99!!!**

Comments in C

- The GNU C/C++ compiler (gcc) will support:
/* this is a comment */
// this is a comment as well
/*
this is a comment also */
/*
this is a comment too
*/
- nested comments are not allowed in GNU C/C++ compiler

Memory Allocation

- Use C system calls:
 - **void *malloc(size_t size)** returns a pointer to a chunk of memory which is the size in bytes requested
 - **void *calloc(size_t nmemb, size_t size)** same as malloc but puts zeros in all bytes and asks for the number of elements and size of an element.
 - **void free(void *ptr)** deallocates a chunk of memory. Acts much like the delete operator.
 - **void *realloc(void *ptr, size_t size)** changes the size of the memory chunk point to by **ptr** to **size** bytes.
 - prototypes found in the **stdlib.h** header file.

Example Memory Allocation

```
void main()
{
    char *cptr;
    double *dblptr;
    struct Student *stuptr;
    /* equiv to cptr = new char[100]; */
    cptr = (char *) malloc(100);
    /* equiv to dblptr = new double[100]; */
    dblptr = (double *) malloc(sizeof(double) * 100);
    /* equiv to stuptr = new Student[100]; */
    stuptr = (struct Student *) malloc( sizeof( struct
    Student) * 100);
}
```

Input & Output

- Cannot use << , >>, **cout**, or **cin** in C.
- Instead use **scanf** for input and **printf** output.
- These system calls take a variable number of arguments.
 - first argument is format string
 - remaining args are variables to write from in printf or read into for scanf.
 - **printf** format string, all characters are printed themselves except those following a %, which means *substitute the value of the next argument here and treat as a particular type as determined by the characters following the %*.

Special Format Characters

- `printf`
 - `%d`: signed integer
 - `%lld`: signed long long integer (64 bits)
 - `%u`: unsigned integer
 - `%x`: integer in hexadecimal format
 - `%f`: double
 - `%Lf`: long double
 - `%s` – a string
- `scanf` is the same except that `%f` is for float and `%lf` is a double
- clang will "help" you if you use the wrong format

I/O Examples

- `printf`:
 - `printf("Hello World! \n");`
 - `printf("X = %u \n", x);`
 - `printf("X = %u, Y = %f and Z = %s \n", x,y,z);`
- `scanf`
 - `scanf("%d", &i);`
 - `scanf("%d %d %d", &i, &j, &k);`
 - `scanf("%lf", &my_double);`
- prototypes found in **`stdio.h`** header file.

Well Used Header Files (based on Linux)

- ***stdio.h*** – printf/scanf/ FILE type
- ***stdlib.h*** – convert routines like string-to-XX, (strtol, strtod, stro), rand num gen, calloc and malloc
- ***unistd.h*** – system calls like fork, exec, read, write
- ***math.h/float.h*** – math routines
- ***errno.h*** – standard error numbers for return values and error handling routines like perror, system call numbers (in Linux).

C only passes arguments by value!

- In C++ you can do the following:

```
void square_it( int &x )  
{  
    x = x * x; // in calling scope x is now x^2  
}
```

- In C you MUST do:

```
void square_it( int *x )  
{  
    *x = (*x) * (*x); // caller must pass a pointer to var!!  
}
```

- Thus, you must be GOOD at POINTERS
- You will get even better when you write assembly language code!

Pointers, Arrays & Memory Addresses

- What is the difference between:
`char array[4][4];`
- and
`char **array;`
`int i;`
`array = (char **)malloc(4 * sizeof(char *));`
`for(i = 0; i < 4; i++)`
`array[i] = (char *)malloc(4);`
- Is `&array[2][2]` the same in both cases??
- Note, `&` operator is only to take the address of a variable (i.e., symbol in a program).

Pop Quiz

- What is the difference between:

char array[4][4]; and

char **array;

int i;

array = (char **)malloc(4 * sizeof(char *));

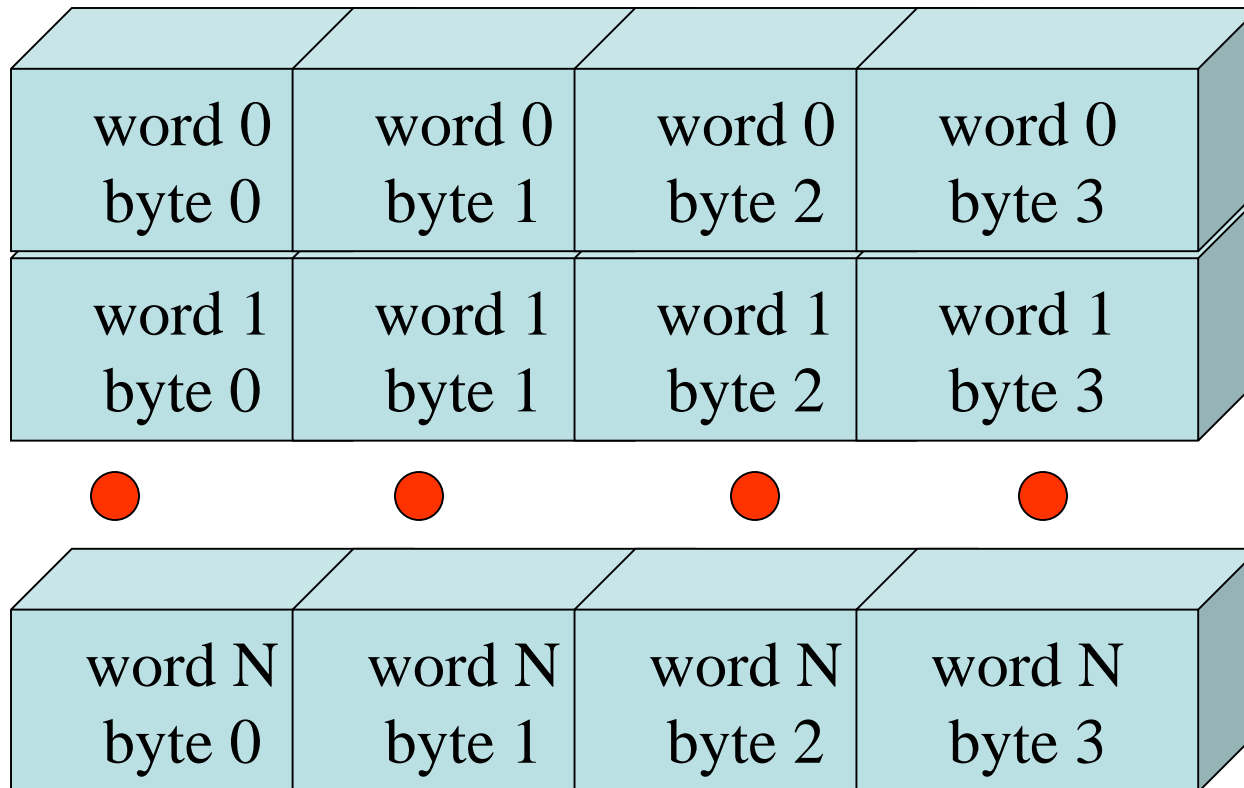
for(i = 0; i < 4; i++)

array[i] = (char *)malloc(4);

- Is **&array[2][2]** the same in both cases??

Memory

- a linear series of address starting a zero
- addressed in 32 or 64 bit chunks called words



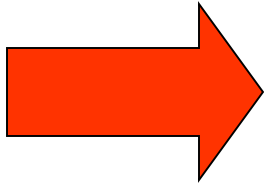
•byte order is Big-endian

•could be Little-endian (i.e., 3,2,1,0)

Arrays: char a[4][4]

symbol “a” or

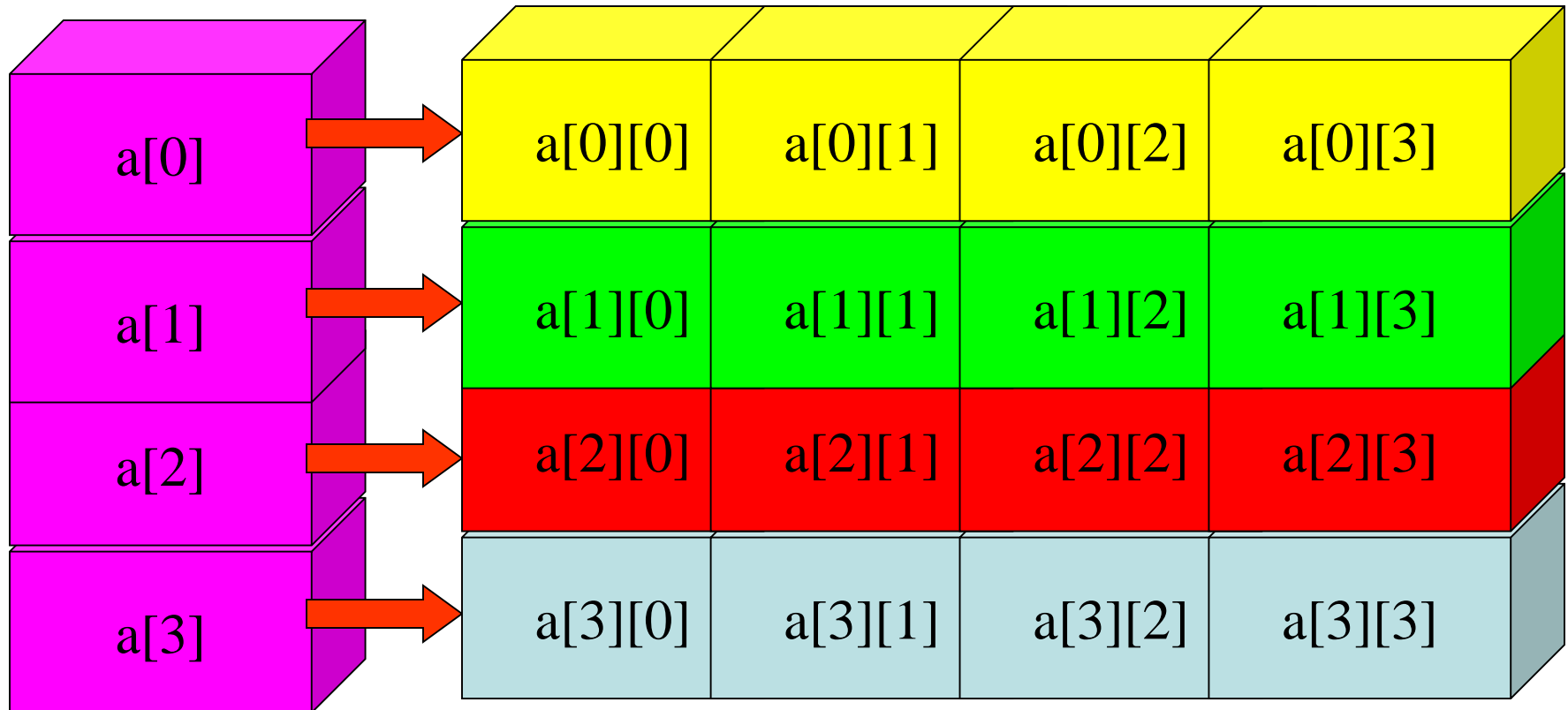
a[0][0] starts here



| | | | |
|--------------------|--------------------|--------------------|--------------------|
| a[0][0] addr 0 | a[0][1] addr 1 | a[0][2] addr 2 | a[0][3] addr 3 |
| a[1][0] addr 4 | a[1][1] addr 5 | a[1][2] addr 6 | a[1][3] addr 7 |
| a[2][0] addr 8 | a[2][1] addr 9 | a[2][2] addr 10 | a[2][3] addr 11 |
| a[3][0] addr 12 | a[3][1] addr 13 | a[3][2] addr 14 | a[3][3] addr 15 |

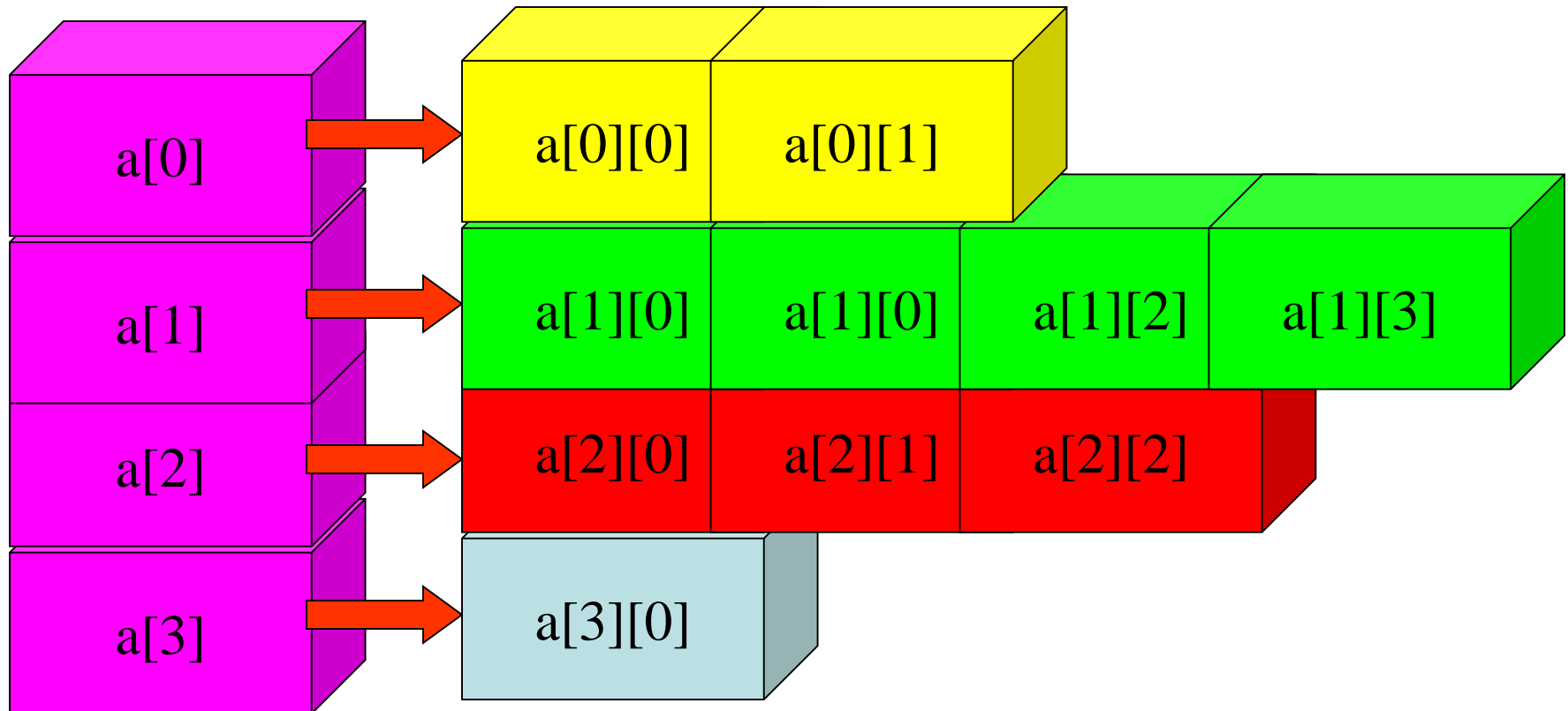
Pointers: char **a

- here the first layer is an array of pointers
- each pointer then points to a string or char *



Dynamic Arrays

- each “row” can have an independent number of elements from the other rows.
- implemented as a `char **` data structure



Pop Quiz

- What is the difference between:

char array[4][4]; and

char **array;

int i;

array = (char **)malloc(4 * sizeof(char *));

for(i = 0; i < 4; i++)

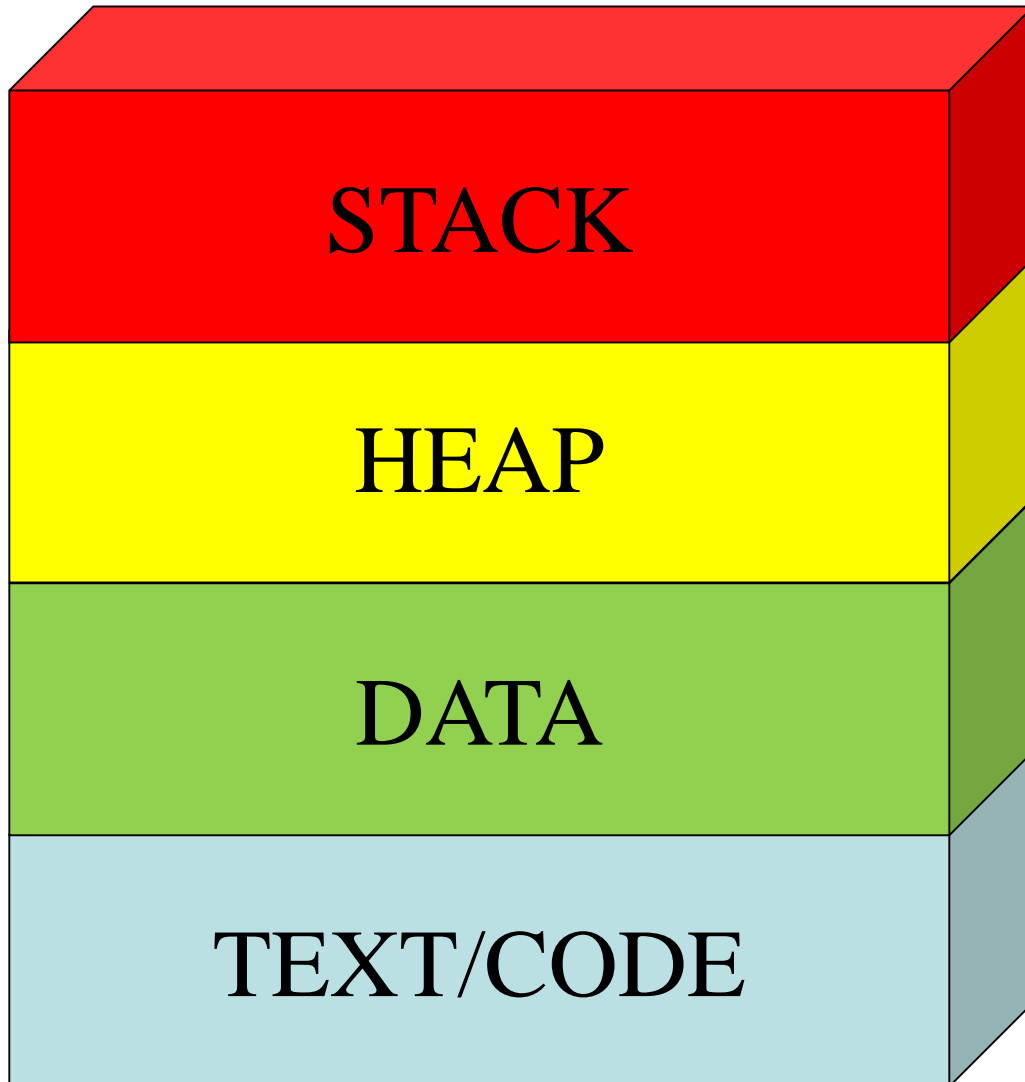
array[i] = (char *)malloc(4);

- Is **&array[2][2]** the same in both cases??

Function Pointers!

- Guess what “code” is really just “data”!
 - Jon Von Neumann’s idea of the Stored Program Computer is still with us today...50 years later
- A computer program is divided largely into 3 areas or segments.
 - text or code segment
 - heap (static/global and dynamic memory)
 - stack (where your local variables are stored)
- C/C++ supports the ability to have a pointer that points to a function that is in the text segment of a program.

Typical Program Segments



- Stack will typically grow down (in memory address) towards the Heap
- Heap will grow up towards Stack
- Data contains initialized and uninitialized program variables
- Text generally does not change except when dynamic classes or libs being loaded

Pop Quiz

- In which segment of memory is `term` located:

```
long term[50];
```

```
int main(void) {  
    printf("%ld\n", term[49]);  
}
```

Function Pointer Syntax

```
/* declare a function pointer type */  
typedef int (*function_pointer_t)( int );  
int square_it( int x) {return x*x;}  
int main() {  
    function_pointer_t sq_it_ptr=NULL;  
    sq_it_ptr = (function_ptr_t) &square_it;  
    return sq_it_ptr( 2 );  
}
```

Hacking your Stack!

- If “code” is really “data”, then can you copy it?
- Can you modify or “write” to your text segment?
 - memory protection in Unix/Linux prevents this.
 - will talk more about later.
- But nothing prevents us from running code off the stack!

Unix Accounts

- To access a Unix system you need to have an *account*.
- Unix account includes:
 - username and password
 - userid and groupid
 - home directory
 - shell

Crash Course in Unix

For more info check out the Unix man pages

-or-

Unix in a Nutshell (an O'Reilly book)

-or-

Linux User's Guide from class webpage

-or-

Google

username

- A username is (typically) a sequence of alphanumeric characters of length no more than 8.
- username is the primary identifying attribute of your account.
- username is (usually) used as an email address
- the name of your home directory is usually related to your username.

password

- a password is a secret string that only the user knows (not even the system knows!)
- When you enter your password the system encrypts it and compares to a stored string.
- passwords are (usually) no more than 8 characters long.
- It's a good idea to include numbers and/or special characters (don't use an english word!)

userid

- a userid is a number (an integer) that identifies a Unix account. Each userid is unique.
- It's easier (and more efficient) for the system to use a number than a string like the username.
- You don't need to know your userid!

Unix Groups and groupid

- Unix includes the notion of a "group" of users.
- A Unix group can share files and active processes.
- Each account is assigned a "primary" group.
- The groupid is a number that corresponds to this primary group.
- A single account can belong to many groups (but has only one primary group).

Pop Quiz

- To log on to a Unix system you need your:
 - A: userid
 - B: username
 - C: primary group
 - D: username and primary group
 - E: userid and primary group

Home Directory

- A home directory is a place in the file system where files related to an account are stored.
- A *directory* is like a Windows folder (more on this later).
- Many Unix commands and applications make use of the account home directory (as a place to look for customization files).

Shell

- A Shell is a unix program that provides an interactive session - a text-based user interface.
- When you log in to a Unix system, the program you initially interact with is your shell.
- There are a number of popular shells that are available.

Logging In

- To log in to a Unix machine you can either:
 - sit at the *console* (the computer itself)
 - access via the net (using telnet, rsh, ssh, kermi, or some other remote access client).
- The system prompts you for your username and password.
- Usernames and passwords are case sensitive!

Session Startup

- Once you log in, your shell will be started and it will display a prompt.
- When the shell is started it looks in your home directory for some customization files.
 - You can change the shell prompt, your PATH, and a bunch of other things by creating customization files.

Pop Quiz

- Which flavor of *nix are you using for this class?
 - A: Ubuntu
 - B: Slackware
 - C: Red Hat
 - D: Debian
 - E: ALT Linux
 - F: Gentoo
 - G: Arch
 - H: Centos
 - I: Fedora
 - J: BSD or derivatives
 - K: WSL
 - L: MacOS
 - M: I have not yet set up a *nix environment for use in this class
 - N: Other

Your Home Directory

- Every Unix process* has a notion of the “current working directory”.
- Your shell (which is a process) starts with the current working directory set to your home directory.

*A process is an instance of a *program* that is currently running.

Interacting with the Shell

- The shell prints a prompt and waits for you to type in a command.
- The shell can deal with a couple of types of commands:
 - Shell internals - commands that the shell handles directly.
 - External programs - the shell runs a program for you.

Files and File Names

- A file is a basic unit of storage (usually storage on a disk).
- Every file has a name.
- Unix file names can contain any characters (although some make it difficult to access the file).
- Unix file names can be long!
 - how long depends on your specific flavor of Unix

File Contents

- Each file can hold some raw data.
- Unix does not impose any structure on files
 - files can hold any sequence of bytes.
- Many programs *interpret* the contents of a file as having some special structure
 - text file, sequence of integers, database records, etc.

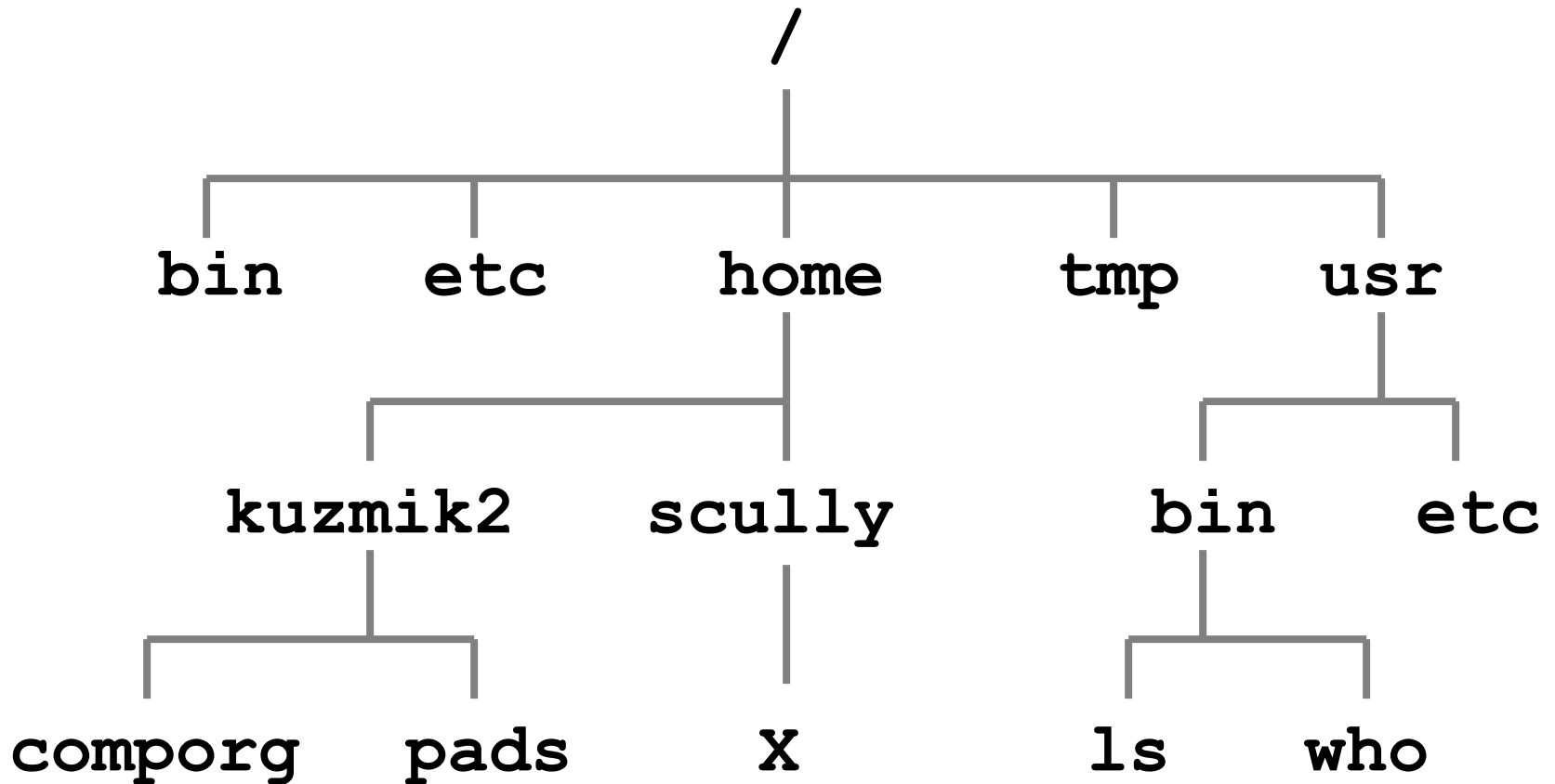
Directories

- A directory is a special kind of file - Unix uses a directory to hold information about other files.
- We often think of a directory as a container that holds other files (or directories).
- Mac and Windows users: A directory is the same idea as a *folder*.
- *Folders are used as a GUI interface to directories and not unique to Unix/Linux/FreeBSD*

More about File Names

- Review: every file has a name.
- Each file *in* the same directory must have a unique name.
- Files that are in different directories can have the same name.

The Filesystem



Unix Filesystem

- The filesystem is a hierarchical system of organizing files and directories.
- The top level in the hierarchy is called the "root" and *holds* all files and directories.
- The name of the root directory is /

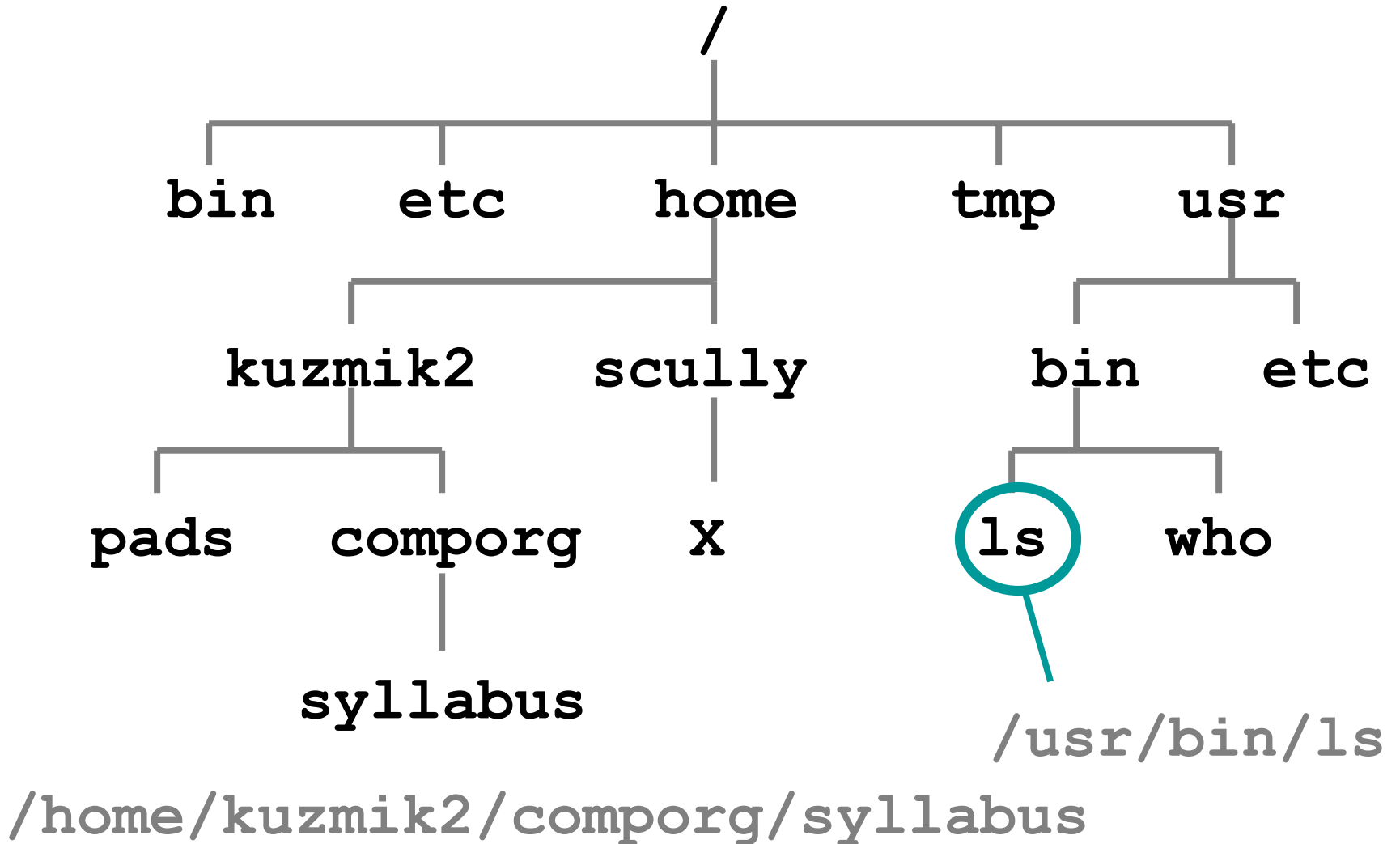
Pathnames

- The *pathname* of a file includes the file name and the name of the directory that holds the file, and the name of the directory that holds the directory that holds the file, and the name of the ... up to the root
- The pathname of every file in a Unix *filesystem* is unique.

Pathnames (cont.)

- To create a pathname you start at the root (so you start with "/"), then follow the path down the hierarchy (including each directory name) and you end with the filename.
- In between every directory name you put a "/".

Pathname Examples



Absolute Pathnames

- The pathnames described in the previous slides start at the *root*.
- These pathnames are called "absolute pathnames".
- We can also talk about the pathname of a file *relative* to a directory.

Relative Pathnames

- If we are *in* the directory `/home/kuzmik2`, the relative pathname of the file **`syllabus`** in the directory `/home/kuzmik2/comporg/` is:

`comporg/syllabus`

- Most Unix commands deal with pathnames!
- We will usually use relative pathnames when specifying files.

Example: The ls command

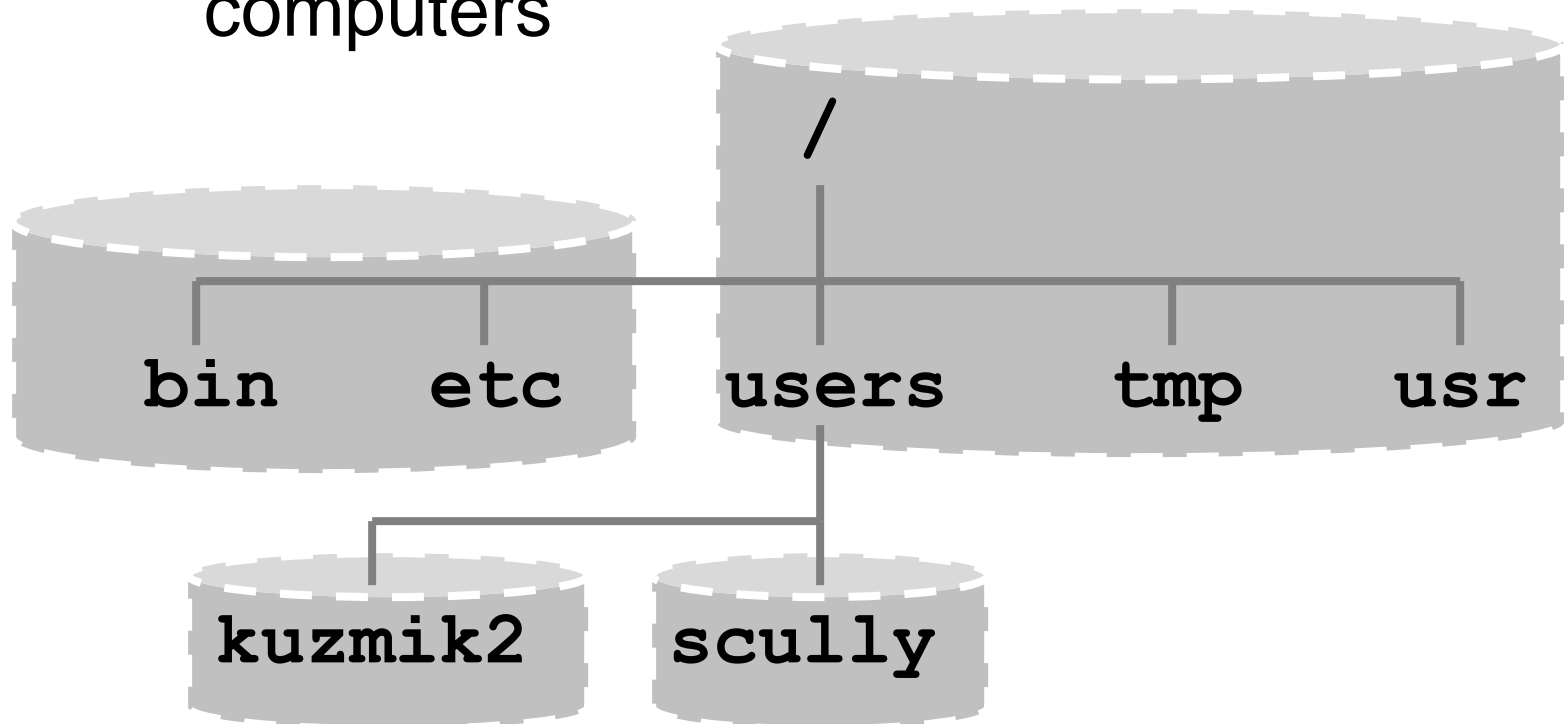
- Exercise: login to a unix account and type the command "ls".
- The names of the files are shown (displayed) as relative pathnames.
- Try this:

ls /usr

- `ls` should display the name of each file in the directory `/usr`.

Disk vs. Filesystem

- The entire hierarchy can actually include many disk drives.
 - some directories can be on other computers



The current directory and *parent* directory

- There is a special relative pathname for the current directory:

▪

- There is a special relative pathname for the parent directory:

▪▪

Your home directory

- There is also a special pathname for the current user's home directory:

~

- Try this:

```
touch /home/yourusername/afile
```

```
ls -l /home/yourusername
```

```
touch -l ~/anotherfile
```

```
ls ~
```

Pop Quiz

- Which of these is an *absolute* path:
 - A: usr/bin/matlab
 - B: ./usr/bin/matlab
 - C: ~/matlab
 - D: /home/kuzmik2/matlab

Some Simple Commands

- Here are some simple commands to get you started:
 - **ls** lists file names (like DOS dir command).
 - **who** lists users currently logged in.
 - **date** shows the current time and date.
 - **pwd** print working directory

The `ls` command

- The `ls` command displays the names of some files.
- If you give it the name of a directory as a *command line parameter* it will list all the files in the named directory.

ls Command Line Options

- We can modify the output format of the **ls** program with a *command line option*.
 - The **ls** command support a bunch of options:
 - **l** *long* format (include file times, owner and permissions)
 - **a** *all* (shows hidden* files as well as regular files)
 - **F** include special char to indicate file types.
- *hidden files have names that start with "."

Moving Around in the Filesystem

- The `cd` command can change the current working directory:

`cd` *change directory*

- The general form is:

`cd [directoryname]`

cd

- With no parameter, the `cd` command changes the current directory to your home directory.
- You can also give `cd` a relative or absolute pathname:

```
cd /usr
```

```
cd ..
```

Some more commands and command line options

- **ls -R** will list everything in a directory and in all the subdirectories recursively (the entire hierarchy).
 - you might want to know that Ctrl-C will cancel a command (stop the command)!
- **pwd**: print working directory
- **df**: shows what disk holds a directory.

Copying Files

- The **cp** command copies files:
cp [options] source dest
- The source is the name of the file you want to copy.
- dest is the name of the new file.
- source and dest can be relative or absolute.

Another form of `cp`

- If you specify a dest that is a directory, `cp` will put a copy of the source in the directory.
- The filename will be the same as the filename of the source file.

```
cp [options] source destdir
```

Deleting (removing) Files

- The `rm` command deletes files:
`rm [options] names...`
- `rm` stands for "remove".
- You can remove many files at once:

```
rm foo /tmp/blah /users/clinton/intern
```

File attributes

- Every file has some attributes:
 - Access Times:
 - when the file was created
 - when the file was last changed
 - when the file was last read
 - Size
 - Owners (user and group)
 - Permissions

File Time Attributes

- Time Attributes:
 - when the file was last changed: `ls -l`
 - when the file was created*: `ls -lc`
 - when the file was last accessed: `ls -ul`

*actually it's the time the file status last changed (ctime). This can come from `chmod`. Depends on OS! Check your man page!

File Owners

- Each file is owned by a user.
- You can find out the username of the file's owner with the `-l` option to `ls`,
- Each file is also owned by a Unix group.
- `ls -lg` also shows the group that owns the file.

File Permissions

- Each file has a set of permissions that control who can mess with the file.
- There are three kinds of permissions:
 - read abbreviated **r**
 - write abbreviated **w**
 - execute abbreviated **x**
- There are separate permissions for the file owner, group owner and everyone else.

ls -l

```
> ls -l foo
```

```
-rw-rw----
```

permissions

```
1 kuzmik2 grads 13 Jan 10 23:05 foo
```

owner

group

size

time

name

ls -l and permissions

- **rwx** **rwx** **rwx**
/ **Owner** **Group** **Others**

Type of file:

- means plain file**
- d means directory**

rwX

- Files:
 - **r**: allowed to read.
 - **w**: allowed to write.
 - **x**: allowed to execute
- Directories:
 - **r**: allowed to see the names of the files.
 - **w**: allowed to add and remove files.
 - **x**: allowed to enter the directory

Changing Permissions

- The **chmod** command changes the permissions associated with a file or directory.
- There are a number of forms of **chmod**, this is the simplest:

chmod mode file

chmod mode file

- Mode has the following form*:

[u go a] [+ - =] [rwx]

u=user g=group o=other a=all
+ add permission - remove permission = set
permission

*The form is really more complicated, but this simple version will do enough for now.

chmod examples

```
> ls -al foo
```

```
rw-rwx--x    1 kuzmik2 grads ...
```

```
> chmod g-x foo
```

```
> ls -al foo
```

```
-rw-rw---x    1 kuzmik2 grads
```

```
> chmod u-r .
```

```
> ls -al foo
```

```
ls: .: Permission denied
```

Other filesystem and file commands

- **mkdir** make directory
- **rmdir** remove directory
- **touch** change file timestamp (can also create a blank file)
- **cat** concatenate files and print out to terminal.

Pop Quiz

- `test/some_file` has `rw-rw-rw-`
Both the file and directory are yours.
What are the minimum permissions needed to see the file with `ls test`?
 - A: `dr-----`
 - B: `d--x-----`
 - C: `dr-x-----`
 - D: `drwx-----`

Shells

Also known as: Unix Command
Interpreter

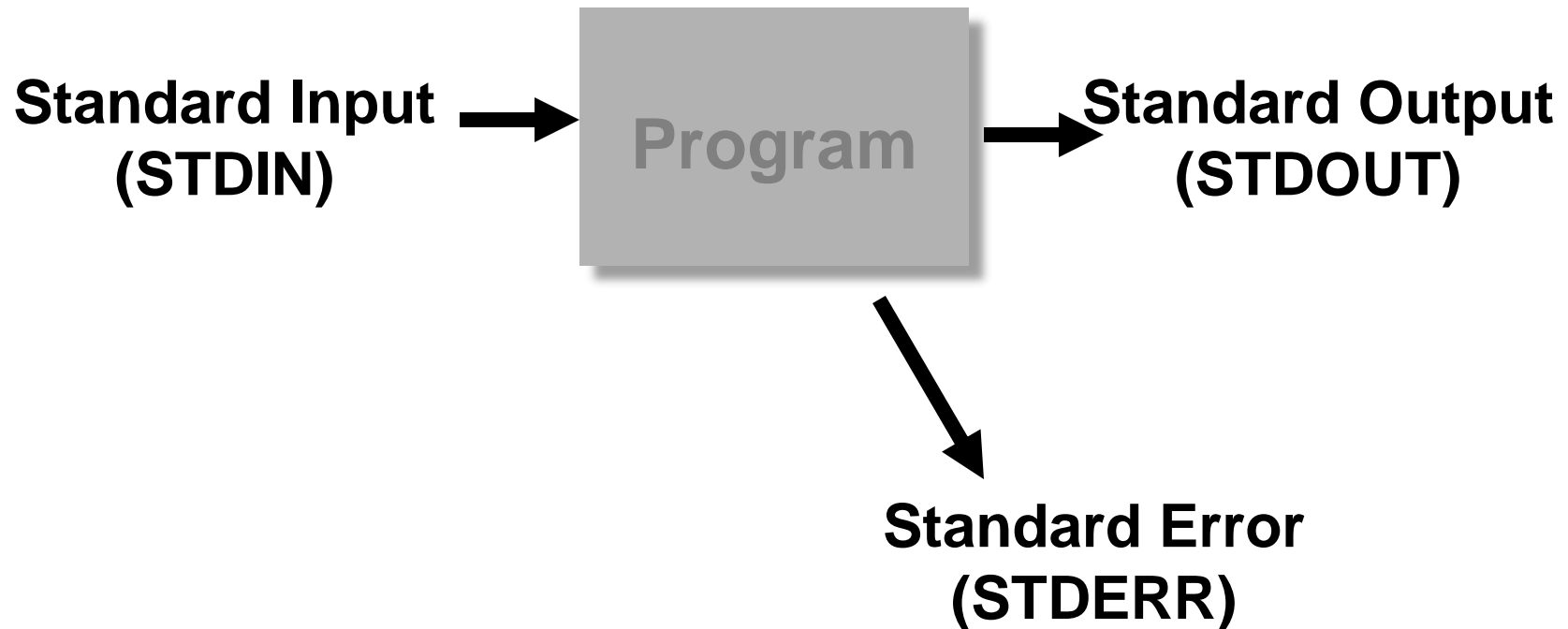
Shell as a user interface

- A shell is a command interpreter that turns text that you type (at the command line) in to actions:
 - runs a program, perhaps the `ls` program.
 - allows you to edit a *command line*.
 - can establish alternative sources of input and destinations for output for programs.

Running a Program

- You type in the name of a program and some command line options:
 - The shell reads this line, finds the program and runs it, feeding it the options you specified.
 - The shell establishes 3 I/O *channels*:
 - Standard Input
 - Standard Output
 - Standard Error

Programs and Standard I/O



Unix Commands

- Most Unix commands (programs):
 - read something from standard input.
 - send something to standard output (typically depends on what the input is!).
 - send error messages to standard error.

Defaults for I/O

- When a shell runs a program for you:
 - standard input is your keyboard.
 - standard output is your screen/window.
 - standard error is your screen/window.

Terminating Standard Input

- If standard input is your keyboard, you can type stuff in that goes to a program.
- To end the input you press Ctrl-D (^D) on a line by itself, this ends the input *stream*.
- The shell is a program that reads from standard input.
- What happens when you give the shell ^D?

Popular Shells

| | |
|--------------------|----------------------------|
| sh | Bourne Shell |
| ksh | Korn Shell |
| cs h | C Shell |
| bash | Bourne-Again Shell |
| fish | Friendly Interactive Shell |
| zsh | Z Shell |

Pop Quiz

- Which shell does your main *nix installation run?
 - A: sh
 - B: ksh
 - C: csh
 - D: bash
 - E: fish
 - F: zsh

Customization

- Each shell supports some customization.
 - User prompt
 - Where to find mail
 - Shortcuts
- The customization takes place in *startup* files – files that are read by the shell when it starts up

Startup files

sh, ksh:

`/etc/profile` (system defaults)

`~/.profile`

bash:

`~/.bash_profile`

`~/.bashrc`

`~/.bash_logout`

csh:

`~/.cshrc`

`~/.login`

`~/.logout`

Wildcards (metacharacters) for filename abbreviation

- When you type in a command line the shell treats some characters as special.
- These special characters make it easy to specify filenames.
- The shell processes what you give it, using the special characters to replace your command line with one that includes a bunch of file names.

The special character *

- * matches anything.
- If you give the shell * by itself (as a command line argument) the shell will remove the * and replace it with all the filenames in the current directory.
- “**a*b**” matches all files in the current directory that start with **a** and end with **b**.

Understanding *

- The **echo** command prints out whatever you give it:
 `> echo hi`
 hi
- Try this:
 `> echo *`

* and ls

- Things to try:

```
ls *
```

```
ls -al *
```

```
ls a*
```

```
ls *b
```

Input Redirection

- The shell can attach things other than your keyboard to standard input.
 - A file (the contents of the file are fed to a program as if you typed it).
 - A pipe (the output of another program is fed as input as if you typed it).

Output Redirection

- The shell can attach things other than your screen to standard output (or stderr).
 - A file (the output of a program is stored in file).
 - A pipe (the output of a program is fed as input to another program).

How to tell the shell to redirect things

- To tell the shell to store the output of your program in a file, follow the command line for the program with the “>” character followed by the filename:

`ls > lsout`

the command above will create a file named **`lsout`** and put the output of the **`ls`** command in the file.

Input redirection

- To tell the shell to get standard input from a file, use the “<” character:

sort < nums

- The command above would sort the lines in the file `nums` and send the result to `stdout`.

You can do both!

```
sort < nums > sortednums
```

```
tr a-z A-Z < letter > rudeletter
```

Note: “tr” command is translate.

Here it replaces all letters
“a-z” with “A-Z”

Pop Quiz

- Running `gcc main.c > compile.log` puts all output into `compile.log` instead of printing it to the terminal
 - A: True
 - B: False

More Output redirection

- To tell the shell to print standard error to a file, use the “2>” phrase:

```
gcc buggy_file.c 2> compile.log
```

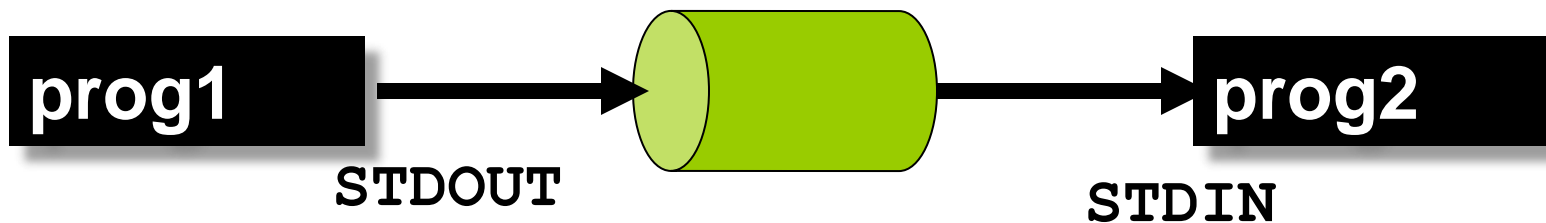
- The command above would send any error messages during the compile to compile.log

Even More Output redirection

- To tell the shell to print standard error AND standard out to the same file, use the “&>” phrase:
`./kind_of_works.out &> run.log`
- The command above would send any output and errors to run.log

Pipes

- A pipe is a holder for a stream of data.
- A pipe can be used to hold the output of one program and feed it to the input of another.



Asking for a pipe

- Separate 2 commands with the “|” character.
- The shell does all the work!

```
ls | sort
```

```
ls | sort > sortedls
```

Shell Variables

- The shell keeps track of a set of parameter names and values.
- Some of these parameters determine the behavior of the shell.
- We can access these variables:
 - set new values for some to customize the shell.
 - find out the value of some to help accomplish a task.

Example Shell Variables

sh / ksh / bash

| | |
|-----------------|--|
| PWD | <i>current working directory</i> |
| PATH | <i>list of places to look for commands</i> |
| HOME | <i>home directory of user</i> |
| MAIL | <i>where your email is stored</i> |
| TERM | <i>what kind of terminal you have</i> |
| HISTFILE | <i>where your command history is saved</i> |

Displaying Shell Variables

- Prefix the name of a shell variable with "\$".
- The **echo** command will do:

```
echo $HOME
```

```
echo $PATH
```

- You can use these variables on any command line:

```
ls -al $HOME
```

Setting Shell Variables

- You can change the value of a shell variable with an assignment command (this is a shell *builtin* command):

```
HOME=/etc
```

```
PATH=/usr/bin:/usr/etc:/sbin
```

```
NEWVAR="blah blah blah"
```

```
PATH=/usr/bin/foo:$PATH
```


`set` command (shell builtin)

- The `set` command with no parameters will print out a list of all the shell variables.
- You'll probably get a pretty long list...
- Depending on your shell, you might get other stuff as well...

The **PATH**

- Each time you give the shell a command line it does the following:
 - Checks to see if the command is a shell built-in.
 - If not - tries to find a program whose name (the filename) is the same as the command.
- The **PATH** variable tells the shell where to look for programs (non built-in commands).

echo \$PATH

```
===== [foo.cs.rpi.edu] - 22:43:17 =====  
/home/kuzmik2/comporg echo $PATH  
/home/kuzmik2/bin:/usr/bin:/bin:/usr/local/bin:  
usr/sbin:/usr/bin/X11:/usr/games:/usr/local/  
packages/netscape
```

- The **PATH** is a list of ":" delimited directories.
- The **PATH** is a list and a *search order*.
- You can add stuff to your PATH by changing the shell startup file (**~/ .bashrc**)

Pop Quiz

- The current directory “.” is not included by default in \$PATH because...
 - A: They forgot to make it a default
 - B: OS developers love seeing you get frustrated when you forget to type ./
 - C: It's a security risk
 - D: It's assumed you want the shell to look in the current directory without checking \$PATH

Job Control

- The shell allows you to manage *jobs*
 - place *jobs* in the *background*
 - move a job to the foreground
 - suspend a job
 - kill a job

Background jobs

- If you follow a command line with "&", the shell will run the *job* in the background.
 - you don't need to wait for the job to complete, you can type in a new command right away.
 - you can have a bunch of jobs running at once.
 - you can do all this with a single terminal (window).

```
ls -lR > saved_ls &
```

Listing jobs

- The command *jobs* will list all background jobs:

```
> jobs
```

```
[1] Running      ls -lR > saved_ls &
```

```
>
```

- The shell assigns a number to each job (this one is job number 1).

Suspending and Killing the Foreground Job

- You can suspend the foreground job by pressing `^Z` (Ctrl-Z).
 - Suspend means the job is stopped, but not dead.
 - The job will show up in the `jobs` output.
- You can *kill* the foreground job by pressing `^C` (Ctrl-C).
- If `^C` does not work, use `^Z` to get back to your terminal prompt and issue:
`$> kill -9 %1`

Quoting - the problem

- We've already seen that some characters mean something special when typed on the command line: `*` (also `?`, `[]`)
- What if we don't want the shell to treat these as special - we really mean `*`, not all the files in the current directory:

```
echo here is a star *
```

Quoting - the solution

- To turn off special meaning - surround a string with double quotes:

➤ `echo here is a star "*"`

➤ `here is a star *`

Quoting Exceptions

- Some *special* characters are **not** ignored even if inside double quotes:
- \$ (prefix for variable names)
- " the quote character itself
- \ slash is always something special (\n)
 - you can use \\$ to mean \$ or \" to mean "

```
>echo "This is a quote \" "  
>This is a "
```

Single quotes

- You can use single quotes just like double quotes.
 - Nothing (except ') is treated special.

```
> echo 'This is a quote \" '
```

```
> This is a quote \"
```

Backquotes are different!

- If you surround a string with backquotes the string is replaced with the result of running the command in backquotes:

```
> echo `ls`
```

```
foo fee file?
```

```
> PS1=`date`
```

```
Tue Jan 25 00:32:04 EST 2000
```

← new prompt!

Programming tools

- Text editors
 - Nano
 - Vi (vim)
 - GNU emacs, Xemacs
- Compilers: gcc / clang
- Debuggers: gdb, lldb, ddd => data display debugger
- Build tools: Make, autoconf, libtool

What are stdin, stdout, stderr?

- File descriptors...or more precisely a pointer to type FILE.
- These FILE descriptors are setup when your program is run.
- So, then what about regular user files...

File I/O Operations

- `fopen` -- opens a file
- `fclose` -- close a file
- `fprintf` -- “printf” to a file.
- `fscanf` -- read input from a file.
- ...and many other routines..

fopen

```
#include<stdio.h>
```

```
void main()
```

```
{
```

```
    FILE *myfile;
```

```
    myfile = fopen("myfile.txt", "w");
```

```
}
```

- 2nd arg is mode:
 - w -- create/truncate file for writing
 - w+ -- create/truncate for writing and reading
 - r -- open for reading
 - r+ -- open for reading and writing

fclose

```
#include<stdio.h>
#include<errno.h>
void main()
{
    FILE *myfile;
    if(NULL == (myfile = fopen("myfile.txt", "w")))
    {
        perror("fopen failed in main");
        exit(-1);
    }
    fclose(myfile);
    /* could check for error here, but usually not needed
*/
}
```

fscanf

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
void main()
{
    FILE *myfile;
    int i, j, k;
    char buffer[80];
    if( NULL == (myfile = fopen("myfile.txt", "r")))
    {
        perror("fopen failed in main");
        exit(-1);
    }
    fscanf(myfile, "%d %d %d %s", &i, &j, &k, buffer);
    fclose(myfile);
    /* could check for error here, but usually not needed */
}
```

sscanf

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
void main()
{
    FILE *myfile;
    int i, j, k;
    char buffer[1024];
    char name[80];
    if( NULL == (myfile = fopen("myfile.txt", "w")))
    {
        perror("fopen failed in main");
        exit(-1);
    }
    fgets(buffer, 1024, myfile);
    sscanf(buffer, "%d %d %d %s", &i, &j, &k, name);
    fclose(myfile);
    /* could check for error here, but usually not needed */
}
```

fprintf

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
void main()
{
    FILE *myfile;
    int i, j, k;
    char buffer[80];
    if(NULL == (myfile = fopen("myfile.txt", "w+")))
    {
        perror("fopen failed in main");
        exit(-1);
    }
    fscanf(myfile, "%d %d %d %s", &i, &j, &k, buffer);
    fprintf(myfile, "%d %d %d %s", i, j, k, buffer);
    printf("%d %d %d %s", i, j, k, buffer);
    fclose(myfile);
    /* could check for error here, but usually not needed */
}
```

Pop Quiz

- What happens when program `fprintf.c` from the previous slide is run? Assume that `myfile.txt` is in the same directory as `fprintf.c` and contains:

```
2021 24 09
```

```
CompOrg is about computer organization
```

- A: Segmentation fault
- B: Nothing gets printed
- C: It prints `fopen failed in main`
- D: It prints `2021 24 9 CompOrg`
- E It prints three integers followed by a string but you can't say what the exact values are

Pipes

- They too are realized as a file descriptor which links either output to input or input to output.
 - recall doing shell commands of the form:
 - `> ls -al | grep "Jan 1" | more`
 - “|” is implemented as a libc call to “popen”

Operating Systems: Unix/Linux

O.S. Responsibilities

- Manages Resources:
 - I/O devices (disk, keyboard, mouse, terminal)
 - Memory
- Manages Processes:
 - process creation, termination
 - inter-process communication
 - multi-tasking (scheduling processes)

Posix - Portable Operating System Interface

- Posix is a popular standard for Unix-like operating systems.
- Posix is actually a *collection* of standards that cover system calls, libraries, applications and more...
- Posix 1003.1 defines the C language interface to a Unix-like kernel.

Posix and Unix

- Most current Unix-like operating systems are *Posix compliant* (or nearly so).
Linux, BSD, Mac OS X
- We won't do anything fancy enough that we need to worry about specific versions/flavors of Unix (any Unix will do).

Posix 1003.1

- process primitives
 - creating and managing processes
- managing process environment
 - user ids, groups, process ids, etc.
- file and directory I/O
- terminal I/O
- system databases (passwords, etc)

System Calls

- A *system call* is an interface to the kernel that makes some request for a service.
- The actual implementation (how a program actually contacts the operating system) depends on the specific version of Unix and the processor.
- The C interface to system calls is standard (so we can write an program and it will work anywhere).

Unix Processes

- Every process has the following attributes:
 - a *process id* (a small integer)
 - a *user id* (a small integer)
 - a *group id* (a small integer)
 - a *current working directory*.
 - a chunk of memory that hold name/value pairs as text strings (the *environment variables*).
 - a bunch of other things...

Creating a Process

- The only way to create a new process is to issue the `fork()` system call.
- `fork()` *splits* the current process in to 2 processes, one is called the *parent* and the other is called the *child*.

Parent and Child Processes

- The child process is a *copy* of the parent process.
- Same program.
- Same place in the program (almost – we'll see in a second).
- The child process gets a new process ID.

Process Inheritance

- The child process *inherits* many attributes from the parent, including:
 - current working directory
 - user id
 - group id

The `fork()` system call

```
#include <unistd.h>
pid_t fork(void);
```

`fork()` returns a process id (a small integer).

`fork()` returns twice!

In the parent – `fork` returns the id of the child process.

In the child – `fork` returns a 0.

Example

```
#include <unistd.h>
#include <stdio.h>

void main(void) {
    if (fork())
        printf("I am the parent\n");
    else
        printf("I am the child\n");
    printf("I am the walrus\n");
}
```

Bad Example (don't try this!)

```
#include <unistd.h>
#include <stdio.h>

void main(void) {
    while (fork()) {
        printf("I am the parent %d\n"
               ,getpid());
    }
    printf("I am the child %d\n"
           ,getpid());
}
```

fork bomb!

I told you so...

- Try pressing Ctrl-C to stop the program.
- It might be too late.
- If this is your own machine – try rebooting.
- If this is a campus machine – run for your life. If they catch you – deny everything.

Switching Programs

- `fork()` is the only way to create a new process.
- This would be almost useless if there was not a way to switch what *program* is associated with a process.
- The `exec()` system call is used to start a new program.

exec

- There are actually a number of exec functions:

`execlp execl execle execvp execv execve`

- The difference between functions is the parameters... (how the new program is identified and some attributes that should be set).

The exec family

- When you call a member of the exec family you give it the pathname of the executable file that you want to run.
- If all goes well, exec will never return!
- The process *becomes* the new program.

exec1()

```
int exec1(char *path,  
          char *arg0,  
          char *arg1, ...,  
          char *argn,  
          (char *) 0);  
  
exec1("/home/kuzmik2/reverse",  
      "reverse", "Hello!", NULL);
```

A complete `exec1` example

```
#include <unistd.h>    /* exec, getcwd */
#include <stdio.h>      /* printf */

/* Exec example code */
/* This program simply execs "/bin/ls" */

void main(void) {
    char buf[1000];

    printf("Here are the files in %s:\n",
           getcwd(buf,1000));
    exec1("/bin/ls","ls","-al",NULL);
    printf("If exec works, this line won't be
    printed\n");
}
```

`fork()` and `exec()` together

- Program does the following:
 - `fork()` - results in 2 processes
 - parent prints out it's `PID` and waits for child process to finish (to exit).
 - child prints out it's `PID` and then `execs` "`ls`" and exits.

execandfork.c part 1

```
#include <unistd.h>    /* exec, getcwd */
#include <stdio.h>      /* printf */
#include <sys/types.h>  /* need for wait */
#include <sys/wait.h>   /* wait() */
```

execandfork.c part 2

```
void child(void) {  
    int pid = getpid();  
  
    printf("Child process PID is %d\n",pid);  
    printf("Child now ready to exec ls\n");  
    execl("/bin/ls","ls",NULL);  
}
```

execandfork.c part 3

```
void parent(void) {  
    int pid = getpid();  
    int stat;  
  
    printf("Parent process PID is %d\n",pid);  
    printf("Parent waiting for child\n");  
    wait(&stat);  
    printf("Child is done. Parent now  
    transporting to the surface\n");  
}
```

execandfork.c part 4

```
void main(void) {  
    printf("In main - starting things with a  
fork()\n");  
    if (fork()) {  
        parent();  
    } else {  
        child();  
    }  
    printf("Done in main()\n");  
}
```

execandfork.c output

```
> ./execandfork
```

```
In main - starting things with a fork()
```

```
Parent process PID is 759
```

```
Parent process is waiting for child
```

```
Child process PID is 760
```

```
Child now ready to exec ls
```

```
exec          execandfork      fork
```

```
exec.c        execandfork.c    fork.c
```

```
Child is done. Parent now transporting to  
the surface
```

```
Done in main()
```

```
>
```


Pop Quiz

- What happens when the parent exits before all of its children exit?
 - A: The OS will not allow the parent to exit and will keep running it until all of its children are finished.
 - B: The parent will exit and any unfinished children will turn into zombies.
 - C: All children will be automatically terminated by the OS when the parent exits.
 - D: It cannot possibly happen because there is no way the parent can return from `main()` or call `exit()` earlier in the code than children return or call `exit()`.

