Unit1
C Programming and Intro to Linux

A crash course for
C++/Windows Users

For your reference, not going to spend a lot of time here

C BASICS

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*

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 (//)

Typical C Program

includes

defines, data type definitions, global variable declarations

function definitions

`main()`

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

#define MAX 1000

typedef  char bigstring[MAX];

char *reverse( char *s) {
  char buf[MAX];
  int i,len;
  len = strlen(s);
  printf("reversing %s\n",s);
  for (i=0;i<len;i++)
     buf[i] = s[len-i-1];
   buf[i]='\0';
   strcpy(s,buf);
   return(s);
}

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 am a string\n");

printf("I\thave\ttabs\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=99;j>=0;j--)
  printf("%d bottles of beer on the wall\n",
  j);


printf("%d is my favorite number\n",17);
```

`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 '`%`'

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

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
```

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`**

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);
```

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 a 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"

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)

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

```
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!!!

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 dlbptr = new double[100]; */
    dblptr = (double *) malloc(sizeof(double) * 100);
    /* equiv to stuptr = new Student[100]; */
    stuptr = (struct Student *) malloc( sizeof( struct Student) *
    100);
}
```

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).

This section is for your reference, not going to spend any time here

UNIX BASICS

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

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.

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

*except NUL and / but your filesystem may have additional restrictions

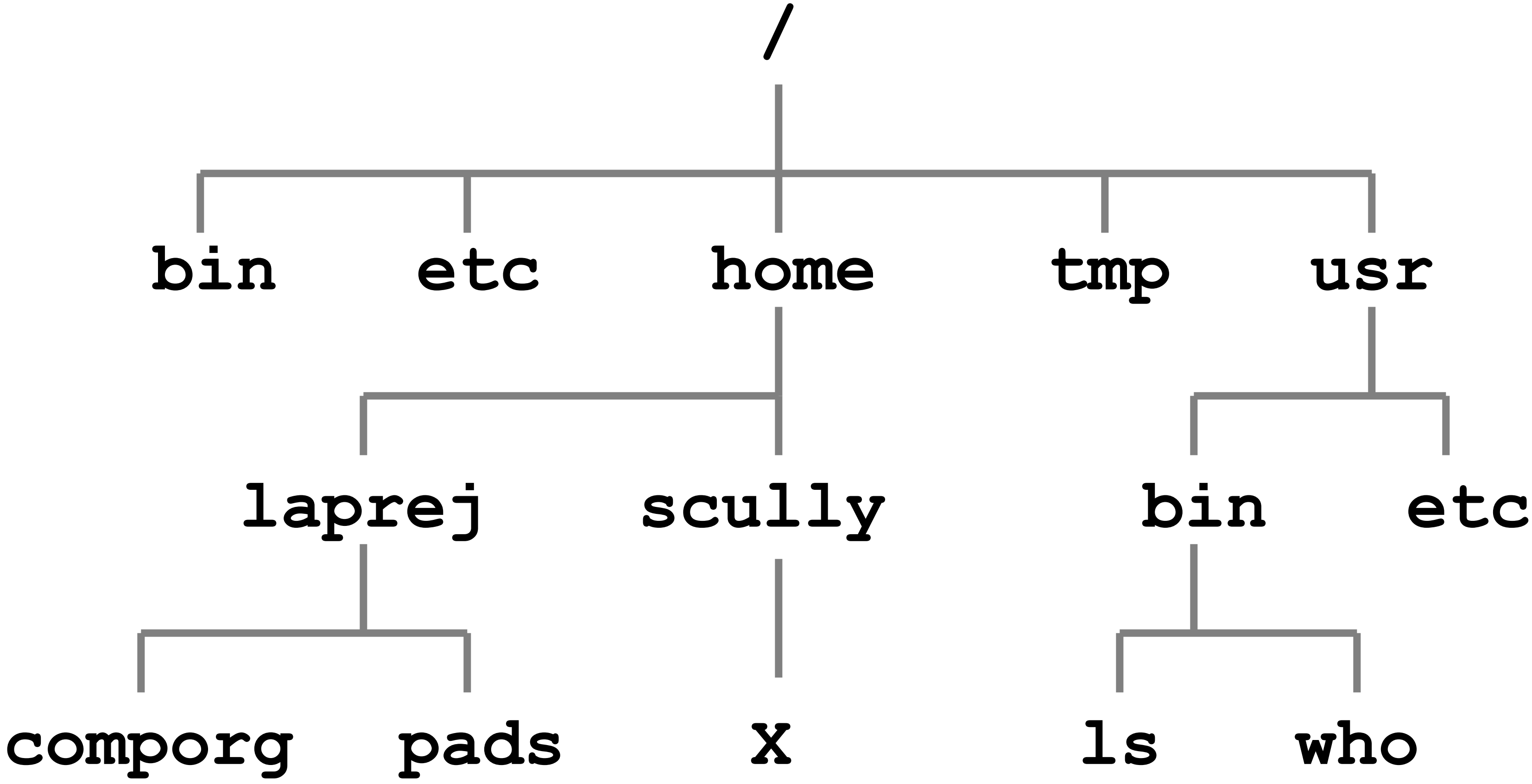
File Contents

- 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*

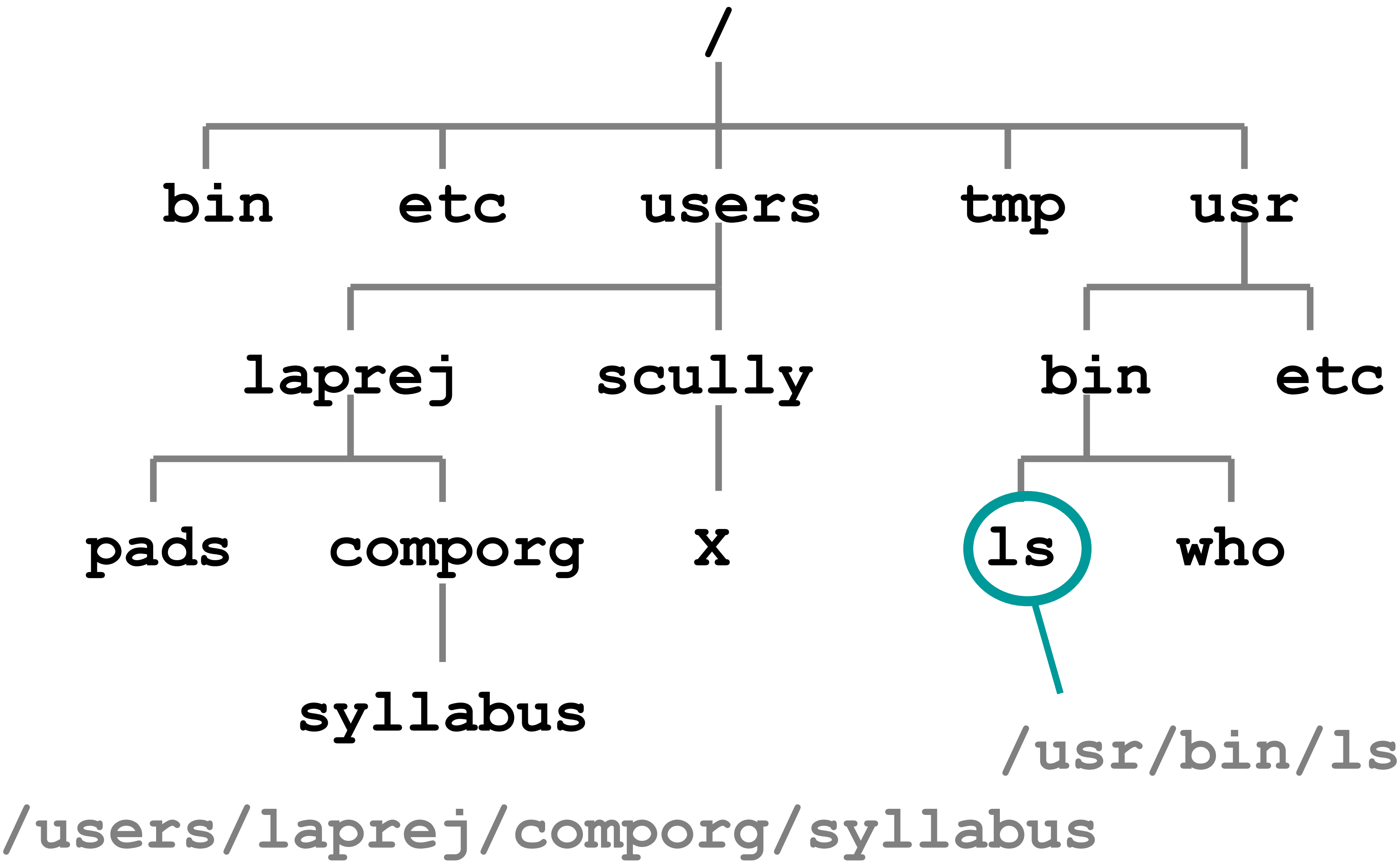
The Filesystem



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 **/**

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/laprej, the relative pathname of the file **`syllabus`** in the directory **`/home/laprej/comporg/`** is:

  `comporg/syllabus`

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

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 relative pathname for the current user's home directory:

`~`

- Try this:

```
touch /home/yourusername/afile
ls -l /home/yourusername
touch -l ~/anotherfile
ls ~
```

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` *__c__hange __d__irectory*

- 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 ..`

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.
- -r is a common option for "recursive", needed if the source is a directory

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 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` and permissions

`-rwxrwxrwx`

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*:

`[ugoa][+-=][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
rwxrwx--x   1 laprej grads …

> chmod g-x foo
> ls -al foo
-rwxrw---x   1 laprej 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.

Shells

Also known as: Unix Command Interpreter

Shell as a user interface

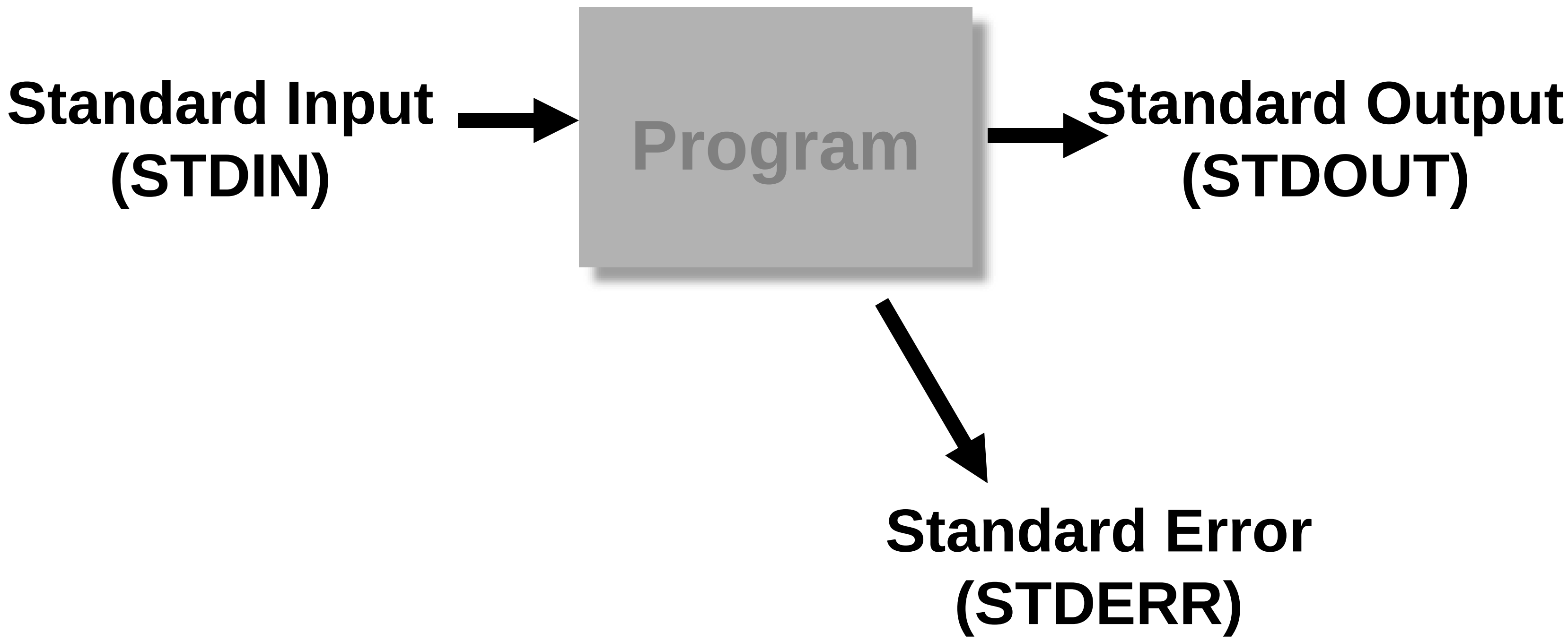
- A shell is a command interpreter that turns text that you type (at the command line) in to actions:
  - runs a program, perhaps the **`ls`** program.
  - allows you to edit a *command line*.
  - can establish alternative sources of input and destinations for output for programs.

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?

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`**.

* 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"

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

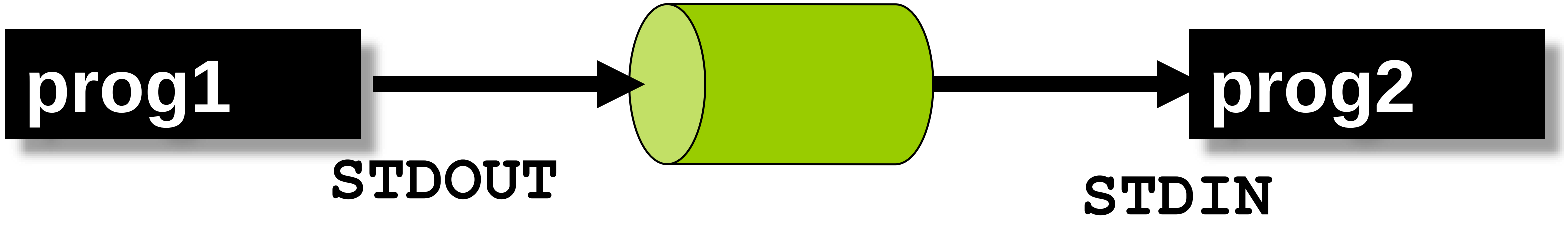
- To tell the shell to print standard error AND standard out to the same file, use the "&>" phrase:

```
./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
```

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/laprej/comporg echo $PATH
/home/laprej/bin:/usr/bin:/bin:/usr/local/bin
  :/usr/sbin:/usr/bin/X11:/usr/games:/usr/loc
  al/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**)

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 \"
```

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<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);
        fclose( myfile );
        /* could check for error here, but usually not needed */
    }
```

sscanf

```
#include<stdio.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<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 );
        fclose( myfile );
        /* could check for error here, but usually not needed */
    }
```

Pipes

- They to are realized as a file descriptor which links either ouput 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

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).

opendir/closedir

```
#include <dirent.h>
DIR *opendir(const char *dirname);

int closedir(DIR *dirp);
```

opendir() opens a directory, just like **fopen()** but for directories

closedir() is like **fclose()**

readdir

```
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

Gets the next entry in the directory, which can be "." or ".." so be careful!

Directory example part 1

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

int main(){
  DIR* dirp;
  struct dirent *dp;
  char dirname[255];

  if((dirp = opendir(".")) == NULL){
    printf("Couldn't open dir\n");
    return -1;
  }
```

Directory example part 2

```
  printf("The contents of %s are:\n",
            getcwd(dirname,255));
  while (dp != NULL){
    if((dp = readdir(dirp)) != NULL){
      printf("\t%s\n",dp->d_name);
    }
  }

  closedir(dirp);
  return 0;
}
```

stat

```
#include <sys/stat.h>
int stat(const char *restrict path,
struct stat *restrict buf);
```

stat structure documented in **man 2 stat** (if you just do **man stat** you will probably get the page in section 1 which is for the stat program)

stat example

```
while (dp != NULL){
  if((dp = readdir(dirp)) != NULL){
    if(stat(dp->d_name, &sb) == -1){
      printf("\t !!! Failed to stat %s\n",dp-
>d_name);
      continue;
    }
    if(sb.st_mode & S_IFDIR){
      printf("\t D %s\n",dp->d_name);
    }
    else if(sb.st_mode & S_IFDIR){
      printf("\t F %s\n",dp->d_name);
    }
    else{
      printf("\t ? %s\n",dp->d_name);
    }
```

stat example v2

```
while (dp != NULL){
  if((dp = readdir(dirp)) != NULL){
    if(stat(dp->d_name, &sb) == -1){
      printf("\t !!! Failed to stat %s\n",dp-
>d_name);
      continue;
    }
    if(S_ISDIR(sb.st_mode)){
      printf("\t D %s\n",dp->d_name);
    }
    else if(S_ISREG(sb.st_mode)){
      printf("\t F %s\n",dp->d_name);
    }
    else{
      printf("\t ? %s\n",dp->d_name);
    }
```

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 Inheritence

- 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!)

fork bomb!

```
#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());
}
```

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.

execl()

```
int execl(char *path,
          char *arg0,
          char *arg1, …,
          char *argn,
              (char *) 0);

execl("/home/laprej/reverse",
        "reverse", "Hello!",NULL);
```

A complete `execl` 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));
  execl("/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 **`exec`**s "**`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()
>
```