Portland State
Computer Science

CS 333

Intro to Operating Systems
Command-Line Arguments

```
int main(int argc, char *argv[], char **envp)
{
    …
    return(0);
}
```

The `envp` parameter to `main()` is actually optional and rarely used.

Examples can be found in
`~rchaney/Classes/cs333/src/argc_argv`

- **Every** C program must have a function called `main()`, which is the point where execution of the program starts.
- When the program is executed, the command line arguments (**the separate words parsed by the shell**) are made available via two arguments to the function `main()`.
- The shell is what does the meta-character evaluation. Your C program (probably) never sees the * or ? a user may place on the command line.

Portland State
Computer Science

Command Line Examples

A couple examples of command lines:

```
ls -l -a
```

Run the `ls` command, passing 2 command line arguments, `-l` and `-a`.

```
head -n 5 /etc/passwd
```

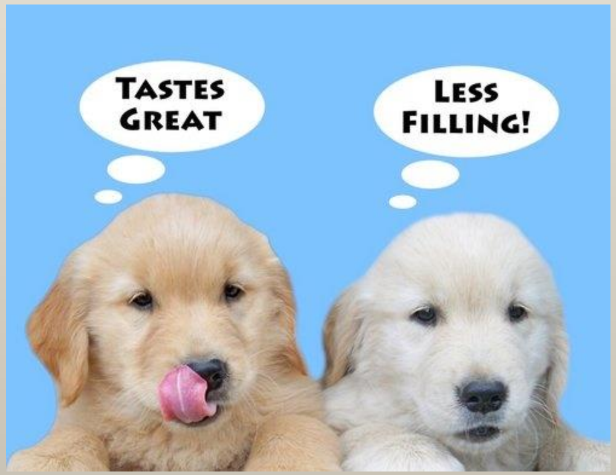
Run the `head` command, passing 2 command line argument2, `-n`. The `-n` command line argument has an option, the 5.
The second command line argument is the file name `/etc/passwd`.

Portland State Computer Science

It’s really by convention that we continue call them `argc` and `argv`.

We could call then *yin* and *yang* or *Coke* and *Pepsi*. Or *TastesGreat* and *LessFilling*.

But, we **will** continue to call them `argc` and `argv`.



- The first argument, `int argc`, indicates **how many** command-line arguments are on the command-line.
- The second argument, `char *argv[]`, is an **array of pointers** to the command-line arguments, each of which is a **null-terminated character string**.

`argv[0]` is (almost) always the name of the program.

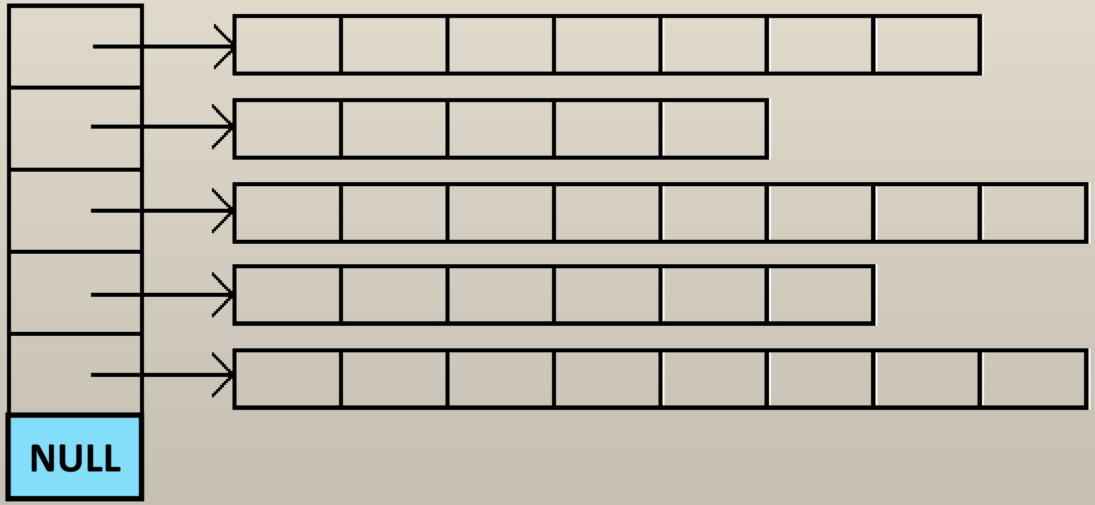
```
for (i = 0; i < argc; i++)
{
   printf("\tThe value of argv[%d] is: %s\n", i, argv[i]);
}
```

Portland State Computer Science

The **argv** parameter is exactly like a **ragged array**, except that it has an additional trailing **NULL** pointer at the end. Also known as a jagged array.

```
for (i = 1; NULL != argv[i]; i++)
{
   printf("\tThe value of argv[%d] is: %s\n", i, argv[i]);
}
```

The above code simply loops through `argv` and prints each item, except for `argv[0]`, the name of the program.



A regular array of C strings

a	b	c	**NULL**		
d	**NULL**				
e	f	g	h	i	**NULL**
j	k	l	**NULL**		
m	n	**NULL**			
o	p	q	r	**NULL**	
s	t	u	v	w	**NULL**

Every row has the same number of columns

A ragged array of C strings

a	b	c	**NULL**		
d	**NULL**				
e	f	g	h	i	**NULL**
j	k	l	**NULL**		
m	n	**NULL**			
o	p	q	r	**NULL**	
s	t	u	v	w	**NULL**

Each row has only the **necessary** number of columns

Environment Variables

Every UNIX process runs in a specific environment.

- An environment consists of **a table of environment variables**, each with an assigned value.
- When you log in certain login files are executed, which initialize the table holding the environment variables for the process.
- When this file passes the process to the shell, the table becomes accessible to the shell.
- When a parent process starts up a child process, the **child process is given a copy of the parent's environment table**.
- Environment variable names are generally given in **upper case,** by convention.

Portland State
Computer Science

Environment Variables

The environment with which a process starts is **inherited** from the shell/process in which it was started.

You can easily see what your shell environment is by issuing the command `printenv` or `env` from your shell.

Your environment variables contain a large number **interesting** and **useful** information.

Portland State
Computer Science

Environment Variables

```
// This is the one passed to main as char **envp
for (i = 0; NULL != envp[i]; i++)
{
   printf("\tThe value of envp[%d] is: %s\n", i, envp[i]);
}

////////////////////////////////////////////////////////////

// This is the newer, better, cooler way to handle environment
//  variables
#include <unistd.h> // POSIX stuff.
extern char **environ;

for (i = 0; NULL != environ[i]; i++)
{
   printf("\tThe value of environ[%d] is: %s\n", i, environ[i]);
}
```

Environment Variables

```
// When using envp (from main), this will not be
//  found in your environment. The envp data are static.
// If you use the environ external variable, you
//  will find these.
putenv("ENVIRONMENT_TEST=test_value");
putenv("HOME=test_value");

new_env = getenv("ENVIRONMENT_TEST");
new_env = getenv("HOME");
```

Portland State
Computer Science

Processing the Command Line

Processing the command line yourself can be challenging.

1. Are there command line options?
2. Do the options have arguments?
3. Can no-argument options be grouped?
4. Can the options be given in any order?
5. Are there things on the command line other than options with/without arguments?

Luckily, there's an app for that!

Portland State Computer Science

The getopt() Library Function

```
#include <unistd.h>

int getopt(int argc
      , char * const argv[]
      , const char * optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

The getopt() function makes your life better.

These are magic global getopt() variables.

The getopt() library function **parses** the command-line.

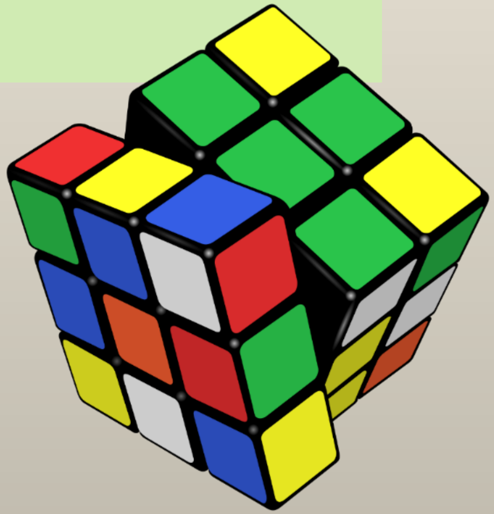
Its arguments argc and argv are the argument count and array as passed to the main() function on program invocation.

- An element of argv that starts with '-' (and is not exactly "-" or "--") is an option element.

- **`optstring`** is a string containing the legitimate option characters.
  - If such a character is **followed by a single colon**, the option **requires** an argument, so `getopt()` places a pointer to the following text in the same `argv`-element, or the text of the following `argv`-element, in `optarg`.
  - **Two colons** mean an option takes an **optional arg**; if there is text in the current `argv`-element (i.e., in the same word as the option name itself, for example, `"-oarg"`), then it is returned in `optarg`, otherwise `optarg` is set to zero.
- The variable **`optind`** is the index of the next element to be processed in `argv`. The system initializes this value to `1`.

- By default, `getopt()` **permutes** the contents of `argv` as it scans, so that eventually **all the non-options are at the end**.
- If `getopt()` does not recognize an option character, it prints an error message to `stderr`, stores the character in `optopt`, and returns `'?'`.

Permute: to rearrange.



Portland State
Computer Science

```
while ((opt = getopt(argc, argv, "os:i:")) != -1) {
  switch (opt) {
  case 'o':
    o_opt++; // Increment the variable each time the -o option is seen.
    printf("The -o option has been seen: %d\n", o_opt);
    break;
  case 's':
    strcpy(s_opt, optarg);
    printf("The -s option has been seen with argument %s\n", s_opt);
    break;
  case 'i':
    i_opt = (int) strtol(optarg, NULL, 10);
    printf("The -i option has been seen with argument %d\n", i_opt);
    break;
  default:
    printf("something strange has happened\n");
    break;
  }
}
```

The **colon** in the list of command line options means an argument is **required** for that option.

Magic variable that contains the string for the argument to a command line option.

Examples of **valid** command lines for this program are:

```
prog -s str -i17
prog -i5 -o -sStr
prog -osStr
```

Example of **invalid** command lines are:

```
prog -s
prog -s str -i
```

Portland State
Computer Science

What might remain on the command line after `getopt()` is done chewing on it?

Magic variable that contains the index from `argv` that is just **past** the last command line option (and argument) that was processed by `getopt()`.

```
if (optind < argc)
{
   int j;

   fprintf(stderr, "\nThis is what remains on the command line:\n");
   for(j = optind; j < argc; j++) {
     printf("\t%s\n", argv[j]);
   }
 }
```

Examples of other stuff on the command line are:

```
    prog -osStr stuff1 stuff2 moreStuff
    prog someStuff -s str -i17
    prog -i5 oddStuff -o -sStr
```

Portland State Computer Science

Man Page for `getopt`

- The man page for `getopt` (`man 3 getopt`) not only contains an excellent description of how `getopt` works, but it also contains **a terrific example** of it use.
- I often start a new program by copying and pasting the example from the `getopt` man page into my code.
- Appendix B from TLPI also has a description for how `getopt` works.
- Learning to use `getopt` will make your life better and easier.