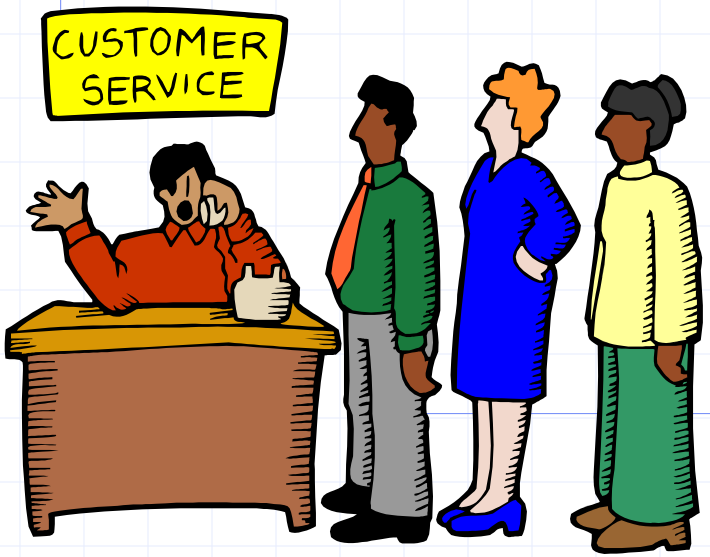
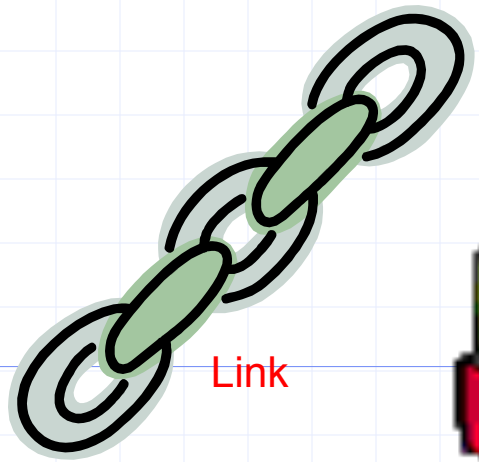
ESC101: Introduction to Computing

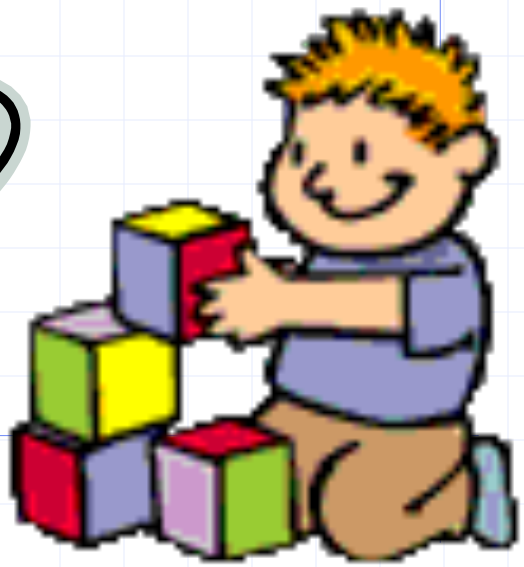
Data Structures

Queue





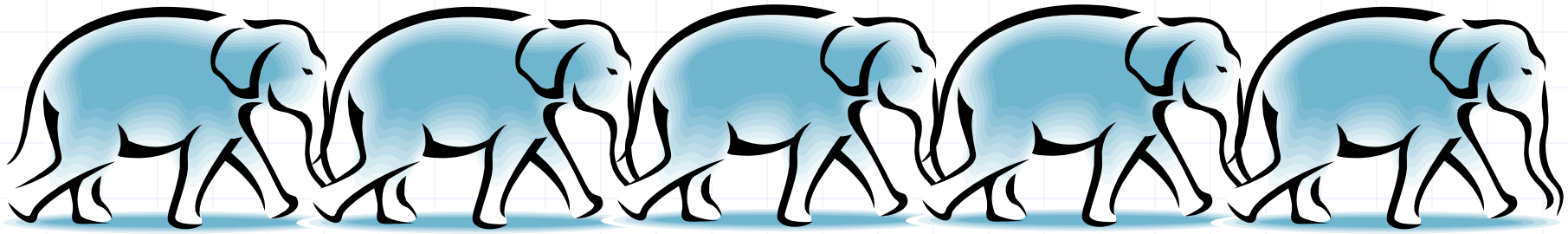
Link



Stack

Linked List

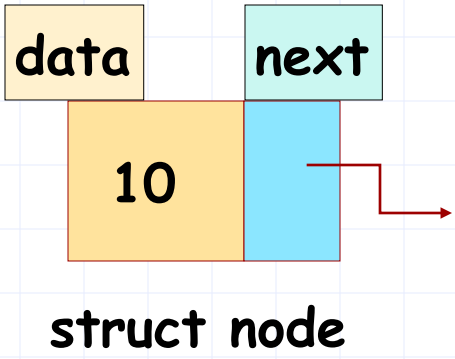
- A linear, dynamic data structure, consisting of nodes. Each node consists of two parts:
  - a "data" component, and
  - a "next" component, which is a pointer to the next node (the last node points to nothing).



Linked List : A Self-referential structure

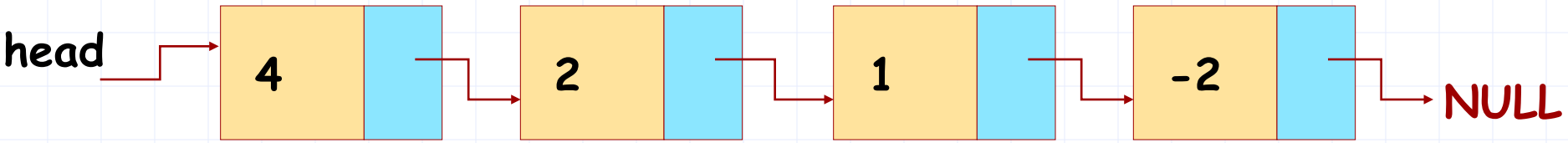
Example:

```
struct node {
      int data;
      struct node *next;
};
```



1. Defines the structure **struct node**, which will be used as a node in a "linked list" of nodes.
2. Note that the field **next** is of type **struct node ***
3. If it was of type struct node, it could not be permitted (recursive definition, of unknown or infinite size).

An example of a (singly) linked list structure is:

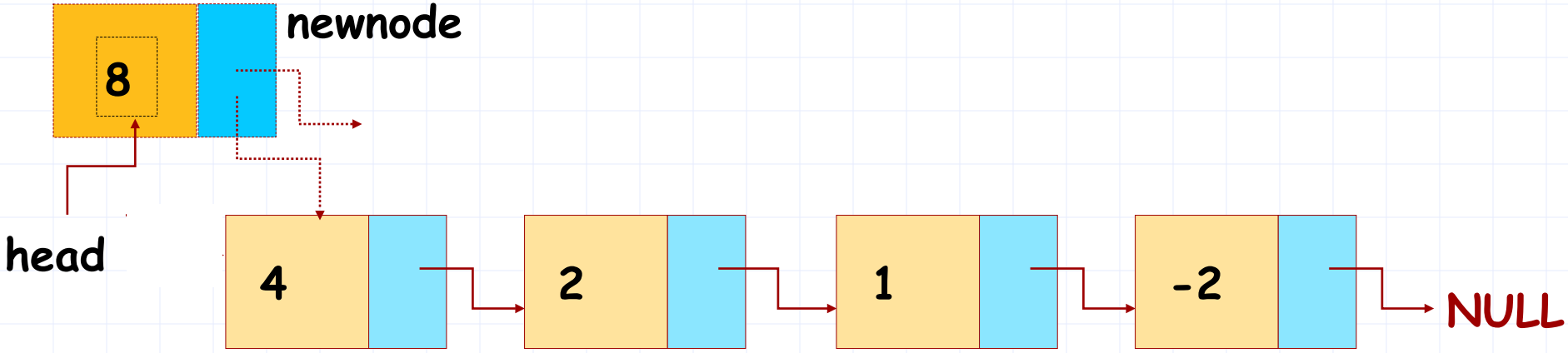


There is only one link (pointer) from each node, hence, it is also called "**singly linked list**".

Insert at Front

Inserting at the front of the list.

1. Create a new node of type struct node. Set its data field to the value given.
2. "Add" it to the front of the list:
   Make its next pointer point to target of head.
3. Adjust head correctly to point to newnode.



```
struct node * make_node(int val) {
      struct node *nd;
      nd = calloc(1, sizeof(struct node));
      nd->data = val;
      return nd;
}
```

```
/* Allocates new node
pointer and sets the
data field to val, next
field initialized to NULL
*/
```

```
struct node *insert_front(int val, struct node *head) {
     struct node *newnode= make_node(val);
     newnode->next = head;
     head = newnode;
     return head;
}
```

```
/* Inserts a node with data field val at the head
    of the list currently pointed to by head.
   Returns pointer to the head of new list.
   Works even when the original list is empty,
     i.e. head == NULL */
```

Exercise

- Write a program to read in two polynomials and add them.
- Input

  1st Poly terms consisting of *e* exponent and *c* coefficient as integers in descending order -1 -1 indicating end of input

  2nd Poly terms consisting of *e* exponent and *c* coefficient as integers in descending order -1 -1 indicating end of input

  Example Input (In descending order)

  2 2 1 2 0 1 -1 -1

  4 2 3 1 -1 -1

  Output (In ascending order)

  0 1 1 2 2 2 3 1 4 2

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

struct term
{
	int exp;
	int coeff;
	struct term * next;
};

struct term *make_term(int exp, int coeff)
{
	struct term *t = (struct term *) calloc(sizeof(struct term),1);
	t->exp = exp; t->coeff = coeff;
	t->next = NULL;
	return t;
}

void print_poly(struct term *p)
{
	while(p)
	{
		printf("%d %d ",p->exp, p->coeff);
		p = p->next;
	}
}
```

```
void free_poly(struct term *p)
{
        struct term *t;
        while(p)
        {
                t = p;
                p = p->next;
                free(t);
        }
}
```

```
int main()
{
	struct term *p1=NULL, *p2=NULL;
	struct term *curr;
	int exp,coeff;
	scanf("%d %d",&exp,&coeff);
	while(exp!=-1 && coeff !=-1)
	{
		curr = make_term(exp, coeff);
		curr->next = p1;
		p1 = curr;
		scanf("%d %d",&exp, &coeff);
	}
	scanf("%d %d",&exp,&coeff);
	while(exp!=-1 && coeff !=-1)
	{
		curr = make_term(exp, coeff);
		curr->next = p2;
		p2 = curr;
		scanf("%d %d",&exp, &coeff);
	}
	print_poly(p1); printf("\n");
	print_poly(p2); printf("\n");
	polyadd( p1, p2);

	free_poly(p1);
	free_poly(p2);
	return 0;
}
```

```
void polyadd( struct term *p1, struct term *p2)
{
	while( p1 && p2 )
	{
		if(p1->exp == p2->exp)
		{
			printf("%d %d ",p1->exp, p1->coeff+p2->coeff);
			p1 = p1->next; p2 = p2->next;
		}
		else if(p1->exp > p2->exp)
		{
			printf("%d %d ",p2->exp, p2->coeff);
			p2 = p2->next;
		}
		else {
			printf("%d %d ",p1->exp, p1->coeff);
			p1 = p1->next;
		}
	}
	while( p1 ) {
		printf("%d %d ",p1->exp, p1->coeff);
		p1 = p1->next;
	}
	while( p2 ) {
		printf("%d %d ",p2->exp, p2->coeff);
		p2 = p2->next;
	}
}
```

ESC101: Introduction to Computing

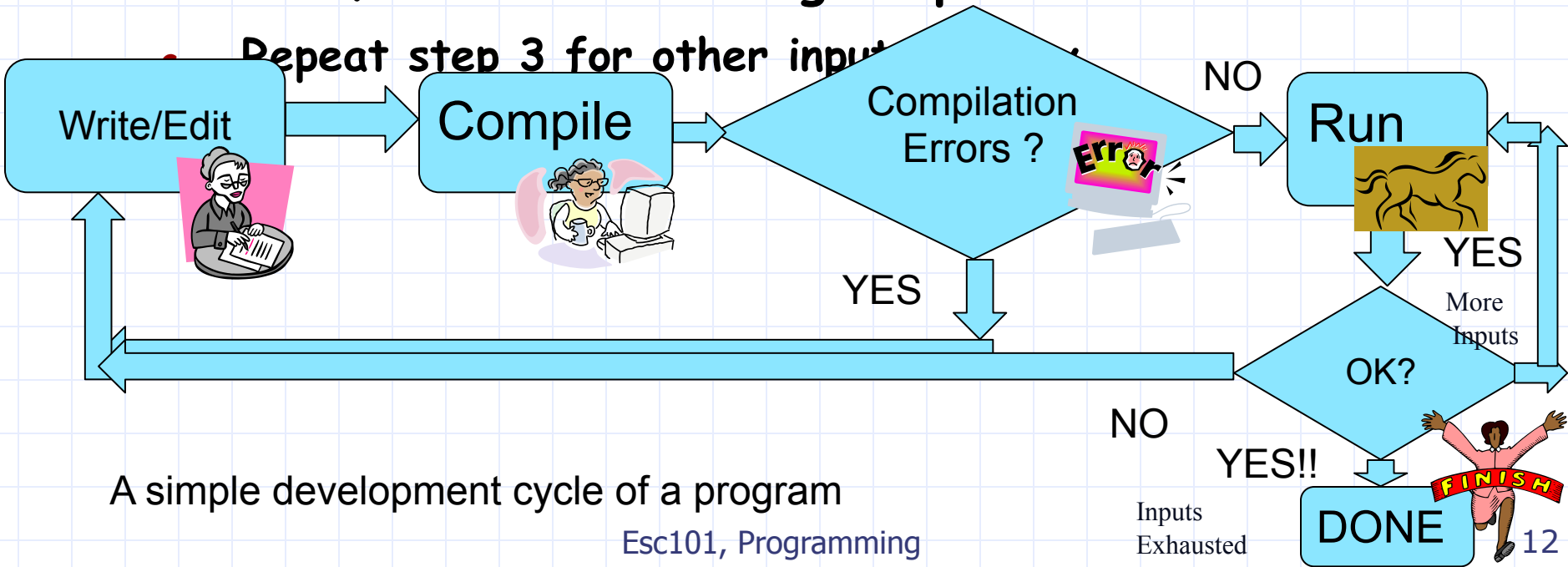
Command Line

&

File Handling

The Programming Cycle

1. Write your program or **edit** (i.e., change or modify) your program.
2. **Compile** your program. If compilation fails, return to editing step.
3. **Run** your program on an input. If output is not correct, return to editing step.
   - Repeat step 3 for other input



A simple development cycle of a program

Edit

- First login to the system.
- Now open an editor. An editor is a system program that lets you type in text, modify and update it.
  - Some popular editors are: vim, emacs, gedit
  - Use an editor that provides syntax highlighting and auto-indent
- Type in your code in the editor. Save what you type into a file.
  - Give meaningful names to your files.

Compile

- After editing, you have to COMPILE the program.
- The computer cannot execute a C program or the individual statements of a C program directly.
  - For example, in C you can write

```
g = a + b * c
```

  - The microprocessor cannot execute this statement. It translates it into an equivalent piece of code consisting of even more basic statements.
- Some error checking is also done as part of compilation process.

How do you compile?

- On Unix/Linux Konsole you can **COMPILE** the program using the gcc command.

```
gcc sample.c
```

- If there are no errors, then the system silently shows the prompt ($).
- If there are errors, the system will list the errors and line numbers. Then you can edit (change) your file, fix the errors and recompile.
  - Warnings may also be produced.

Compile…

- As long as there are compilation errors, the EXECUTABLE file is not created.
- If there are no errors then gcc places the machine program in an executable format for your machine and calls it a.out
- The file a.out is placed in your current working directory.

Simple! Program

- Lets compile some of the simplest C programs.
- Login, then open an editor and type in the following lines. Save the program as sample.c

```
# include <stdio.h>
int main ()  {
      printf("Welcome to C");
      return 0;
}
```

sample.c: The program prints the message "Welcome to C"

Compile and Run

- Now compile the program. System compiles without errors.

```
$ gcc sample.c
$
```

- Compilation creates the executable file a.out by default.
- Now run it. Then looks like this:

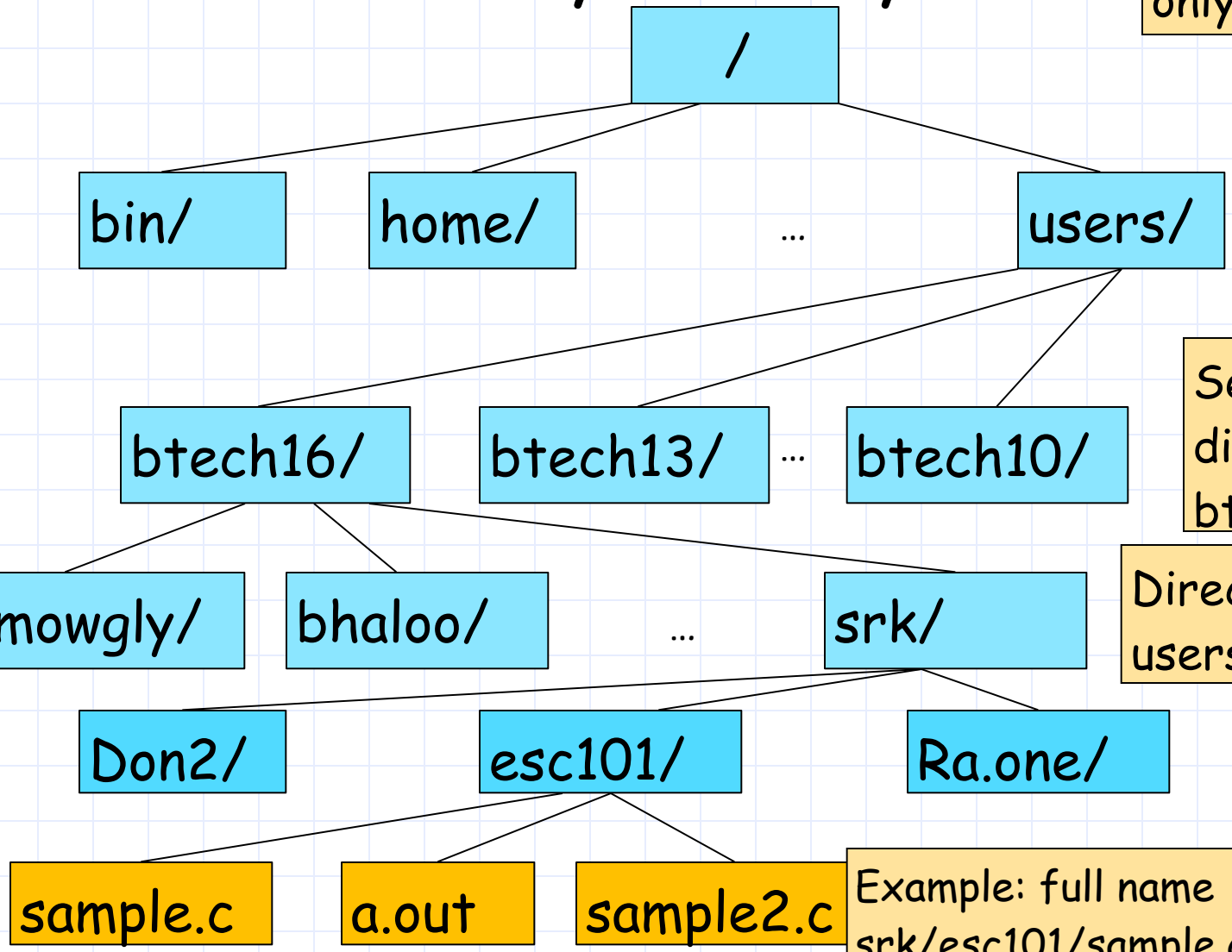
```
$ ./a.out
Welcome to C$
```

Introduction to Files and Directory

- Compiling using gcc by default produces the file a.out in your current working directory.
- Let us understand the notion of directory and current working directory.
- The unit of data in a system is a file.
- Files are organized into directories, also called folders. Each directory may have many files inside it and also many directories inside it.
- Having files and directories inside directories gives it a hierarchical structure.

Directory Hierarchy

- The root directory has the symbol /



Root directory usually has no files, only directories.

First level directories: /bin, /users

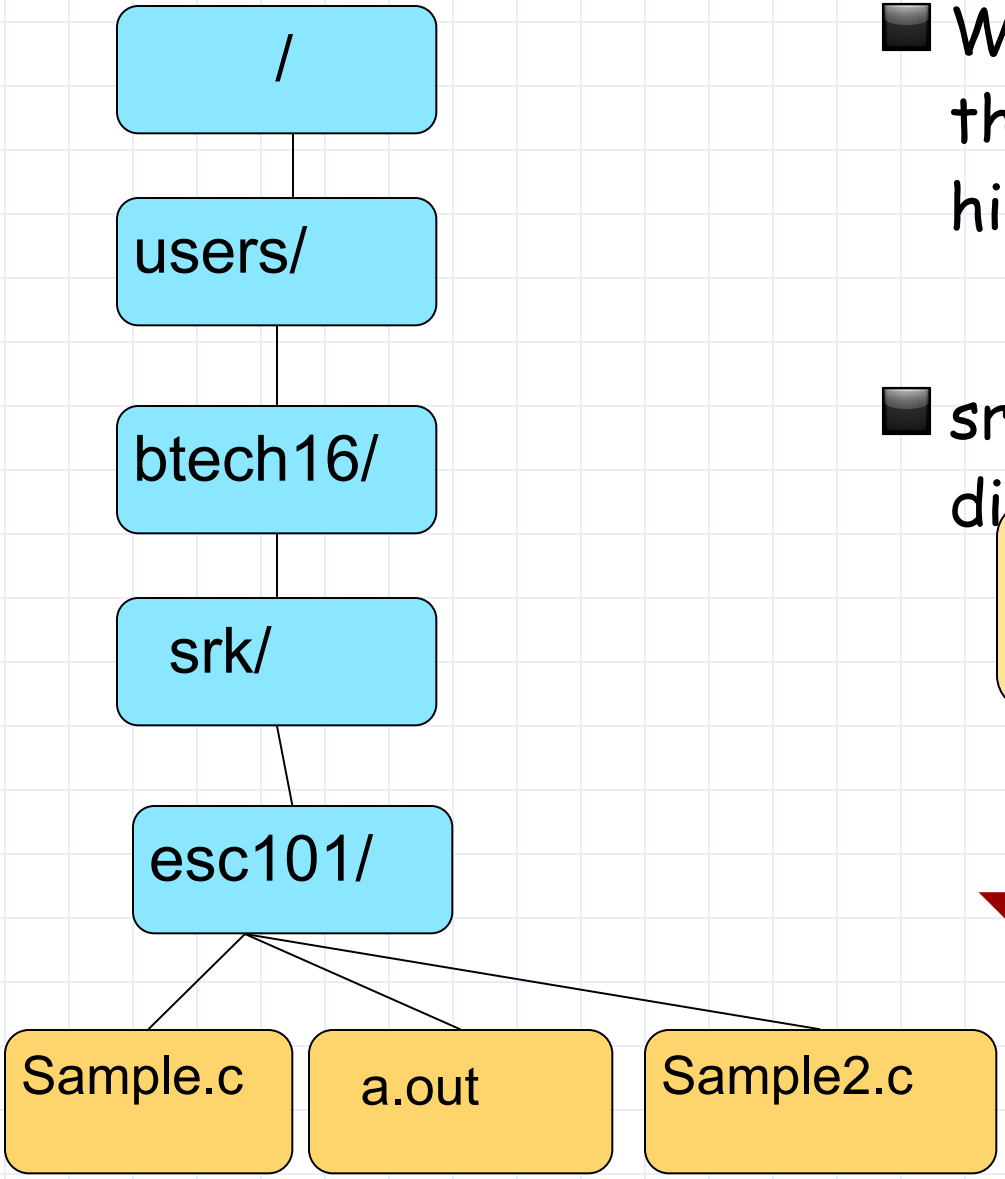
Second level directory /users/btech10/,...

Directory per user / users/btech16/srk/

Directories of user srk

Example: full name /users/btech16/srk/esc101/sample.c

Directory commands



- When user srk logs in, the system places him in his home directory:

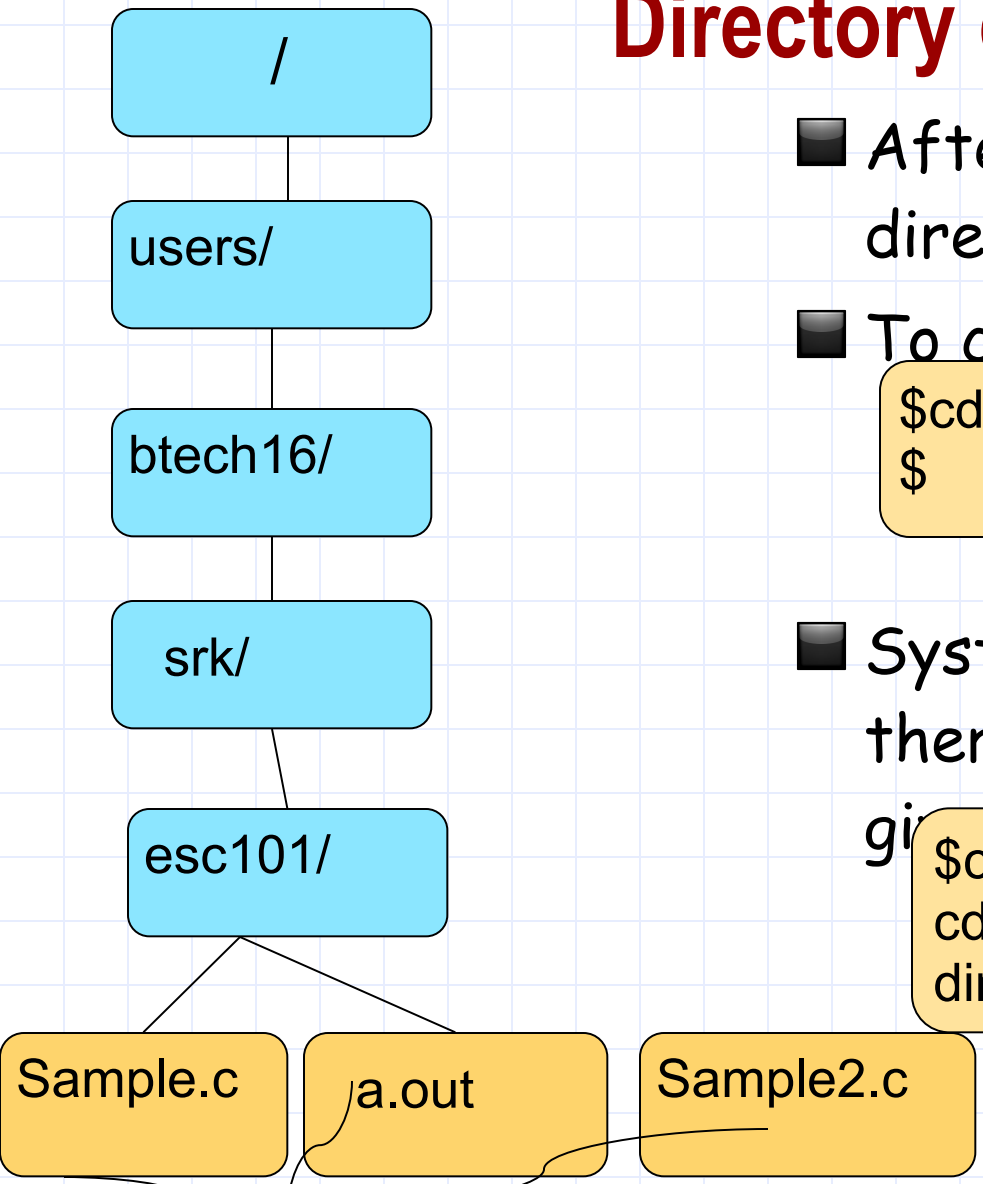
  /users/btech16/srk/

- srk can find his current directory by typing

```
$pwd
/users/btech16/srk
```

  - pwd stands for print working directory

Directory commands



- After login srk is in home directory /users/btech16/srk
- To change directory to esc101

```
$cd esc101/
$
```

- System returns silently. If there is spelling error, system gi...

```
$cd esc101a/
cd: esc101a: no such file or
directory
```

These are files. Files are at the bottom of the directory hierarchy. Files do not contain Files or directories. Only directories contain files or other directories (or both).

Arguments on the Command Line

- Typically when using commands we provide arguments to the command in the same line.
  - cd my_dir
  - gcc my_file.c
  - cp file1.c file2.c
- In each case, stuff in red is the command line argument
- In the third example, cp is the command name and file1.c and file2.c are its two arguments.

Batch mode vs. Interactive mode

- Interactive mode:
  1. first you enter command (say mkdir)
  2. then you get prompted and you enter an arg (the directory name, say esc101)
  3. mkdir creates the directory esc101, and asks if you want to create more directories. If you say yes, it goes to step 2. Else, it exits.
- This is cumbersome.
- Batch Mode: If the arguments are standard, we prefer entering them along with the command (Also called command-line mode):
  - mkdir esc101 phy102 chm_lab
  - 3 Directories created: esc101, phy102 and chm_lab

Arguments on the Command Line

- Typically when using commands we provide arguments to the command in the same line.
  - cd my_dir
  - gcc my_file.c
  - cp file1.c file2.c
- In each case, stuff in red is the command line argument
- In the third example, cp is the command name and file1.c and file2.c are its two arguments.

Command Line Args in C

- Write a program to read a name from command line, and say "Hello" to it.
- Some Example Interaction (Output in red):

```
$ ./a.out ABC
Hello ABC
$ ./a.out World
Hello World
$ ./a.out ESC101
Hello ESC101
```

Note that the program really has no sense of what is a name. It just prints the argument provided.

Command Line Args = Args to main

- So far we used the following signature for main

  int main()

- But main can take arguments. The modified prototype of main is

  int main(int argc, char **argv)

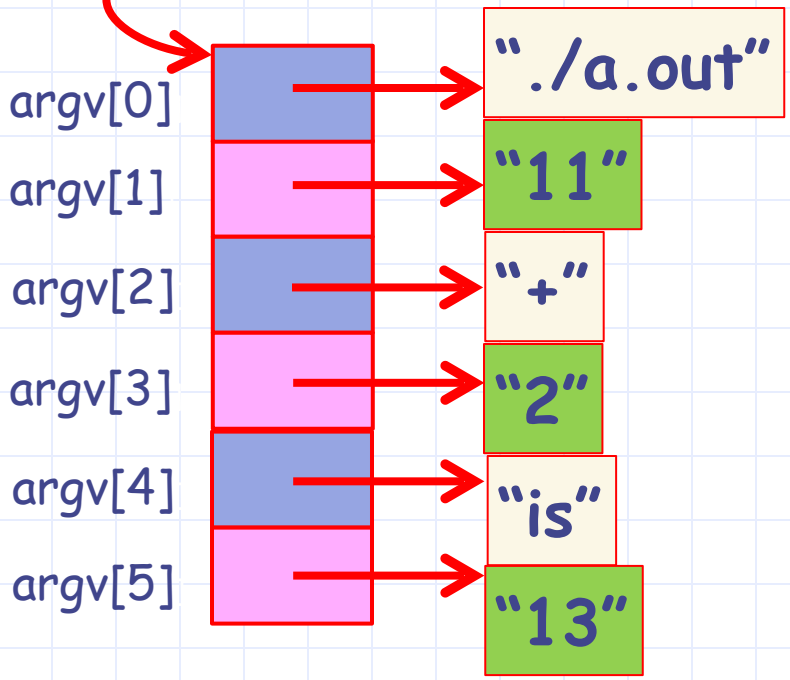
  - Argument Count (argc): An int that tells the number of arguments passed on command line
  - Argument Values (argv): Array of strings. argv[i] is the i-th argument as string.

Args to main

./a.out 11 + 2 is 13

argc = 6 ./a.out is included in arguments

argv



Note that everything is treated as string, even the numbers!

Example

NOTE: char **argv is same as char *argv[]

```
#include<stdio.h>

int main(int argc, char *argv[]) {
    if (argc<2)
        printf ("Too few args!\n");
    else if (argc == 2)
        printf ("Hello %s\n",argv[1]);
    else
        printf("Too many args!\n");
    return 0;
}
```

```
$ ./a.out
Too few args!
```

```
$ ./a.out ABC
Hello ABC
$ ./a.out World
Hello World
$ ./a.out ESC101
Hello ESC101
```

```
$ ./a.out Hey There
Too many args!
```

What about Other Types?

- Write a program that takes two numbers (integers) on command line and prints their sum.
- Problem:
  - Everything on command line is read as string!
  - How do I convert string to int?
- Solution: Library functions in stdlib.h
  - atoi: takes a string and converts to int

```
atoi("1234") is 1234, atoi("123ab") is 123, atoi("ab") is 0
```

  - atof: converts a string to double
- Other variations : atol, atoll

Adding 2 Numbers

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc, char *argv[]) {
    if (argc != 3)
        printf ("Bad args!\n");
    else {
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        printf ("%d\n",a+b);
    }
    return 0;
}
```

```
$ ./a.out
Bad args!
$ ./a.out 3 4
7
$ ./a.out 3 -4
-1
$ ./a.out 3 four
3
$ ./a.out 3 4 5
Bad args!
```

Command Line Sorting

```
int main(int argc, char *argv[]) {
   int *ar, n;
  n = argc - 1;
   ar = (int *)malloc(sizeof(int) * n);
   for (i=0; i<n; i++)
      ar[i] = atoi(argv[i+1]);

   merge_sort(ar, n); // or any other sort

   for (i=0; i<n; i++)
      printf("%d ",ar[i]);
   return 0;
}
```

```
void merge_sort (
        int *arr, int n)
{
    ...
}
```

```
$ ./a.out 1 4 2 5 3 9 -1 6 -10 10
-10 -1 1 2 3 4 5 6 9 10
```