CS 332/532 Systems Programming

Lecture 35

-REVIEW 1/3-

Professor : Mahmut Unan – UAB CS

Final Exam (1G)

April 27th, 2022, Wednesday

08:00am – 10:30am

We will start at 09:00 am

- Multiple Choice
- True/False
- Short Answer

Grades

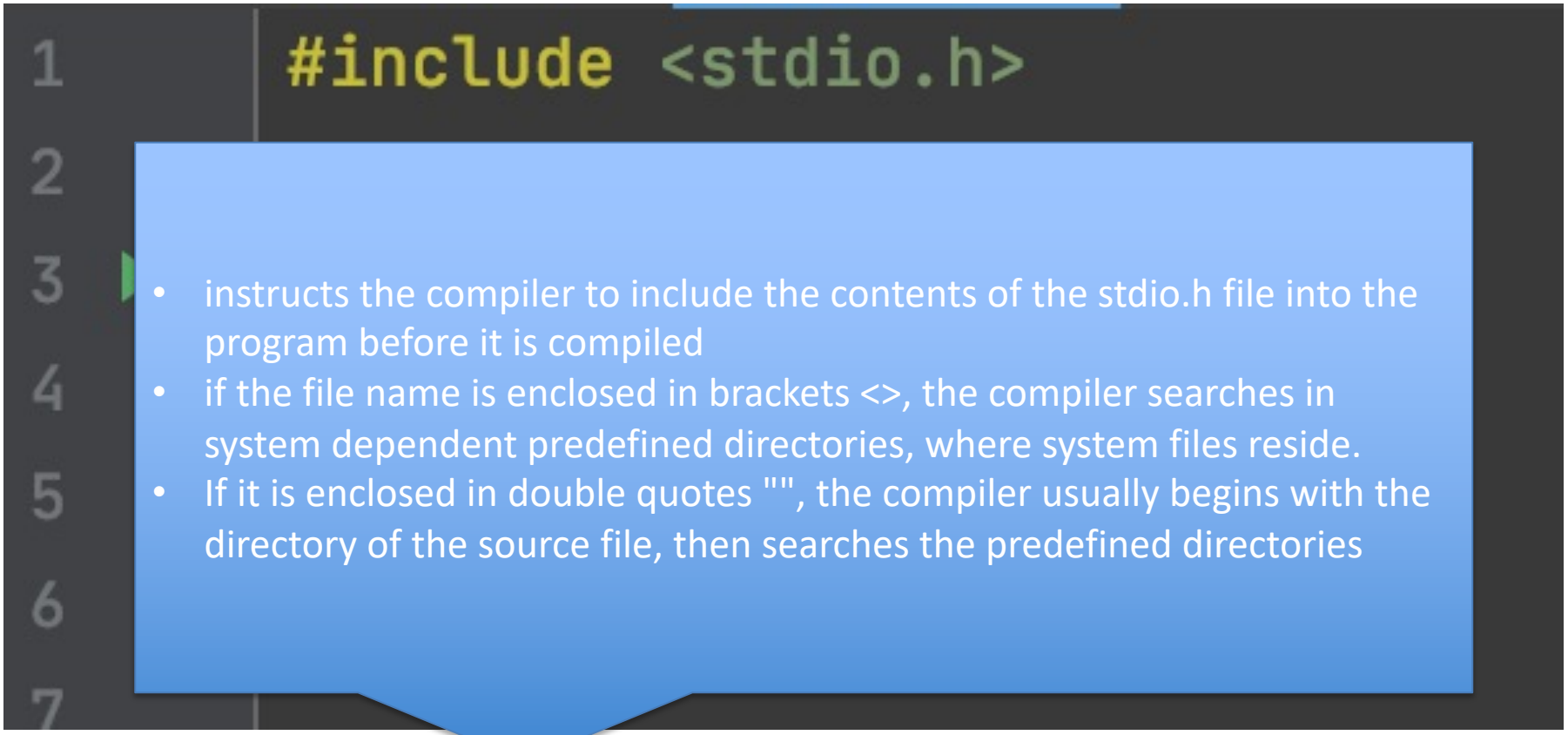
- Please check each individual grade on Canvas, ignore the total grade

IDEA Survey

- Please participate
- Useful Feedback
- Feedback about the TA

Hello World !

#include


1
#include <stdio.h>
2
3
4
5
6
7
• instructs the compiler to include the contents of the stdio.h file into the program before it is compiled
• if the file name is enclosed in brackets <>, the compiler searches in system dependent predefined directories, where system files reside.
• If it is enclosed in double quotes "", the compiler usually begins with the directory of the source file, then searches the predefined directories

The `main()` function

```
3  int main() {
4      printf("Hello World!");
5      return 0;
6  }
7
```

- Every C program must contain a function named main()
- main function will be automatically called when the program runs
- int indicates that the main function will return an integer
  - How about void?

Comments

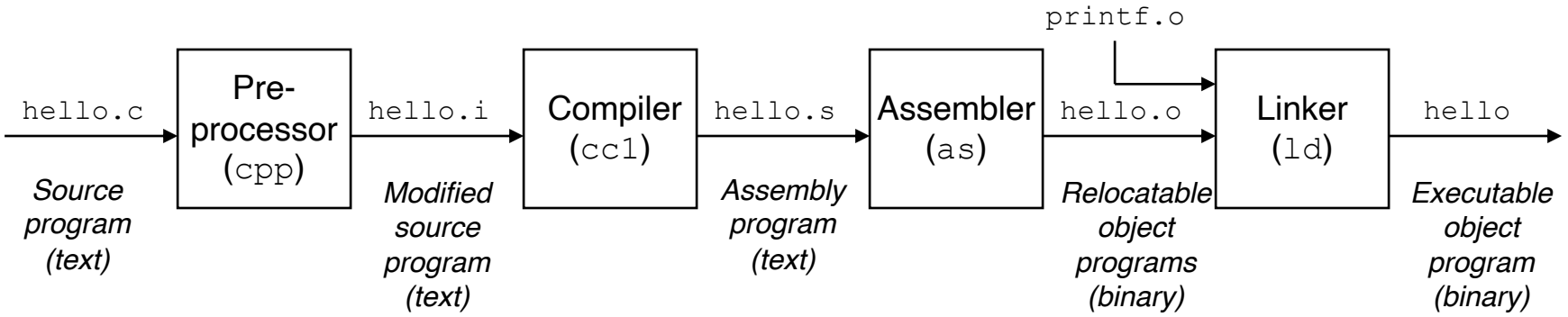
```c
#include <stdio.h>
/* This program calls printf() to display a message on the screen. */
int main(void)
{
        printf("Hey Ho, Let's Go\n");
        return 0;
}
```

Nested comments are not allowed.
For example, the following code is illegal,
and the compiler will raise an error message:

```c
/*
/* Another comment. */
*/
```

Compilation

- `unix> gcc –o hello hello.c`



Compilation System

Compiler Options

- You can compile your C program with various levels of optimization turned on (*e.g., -O, -O3, -Ofast*). Here are some useful/popular compiler and optimization options:
- **The most basic form:** `gcc hello.c` executes the complete compilation process and outputs an executable with name a.out
- **Use option –o:** `gcc hello.c -o hello` produces an output file with name 'hello'.
- **Use option –Wall:** `gcc -Wall hello.c -o hello` enables all the warnings in GCC.
- **Use option –E:** `gcc -E hello.c > hello.i` produces the output of preprocessing stage
- **Use option –S:** `gcc -S hello.c > hello.S` produces only the assembly code
- **Use option –C:** `gcc -C hello.c` produces only the compiled code (without linking)
- **Use option –O:** `gcc -O hello.c` sets the compiler's optimization level.

Running Hello Object File

- Running hello object file on the shell

```
unix> ./hello
hello, world
unix>
```

Variables

- A *variable* in C is a memory location with a given name. The value of a variable is the content of its memory location. A program may use the name of a variable to access its value.
- Here are some basic rules for naming variables. These rules also apply for function names. Be sure to follow them or your code won't compile:
  - The name can contain letters, digits, and *underscore characters* _.
  - The name must begin with either a letter or the underscore character.
  - C is *case sensitive*, meaning that it distinguishes between uppercase and lowercase letters.

Variables / 2

- The following keywords cannot be used as variable names because they have special significance to the C compiler.

auto	do	goto	signed	unsigned
break	double	if	sizeof	void
case	else	int	static	volatile
char	enum	long	struct	while
const	extern	register	switch	
continue	float	return	typedef	
default	for	short	union	

Declaring Variables

- **`data_type`** `name_of_variable;`

C Data Types

Data Type	Usual Size (bytes)	Range of Values(min–max)
`char`	1	−128…127
`short int`	2	−32.768…32.767
`int`	4	−2.147.483.648…2.147.483.647
`long int`	4	−2.147.483.648…2.147.483.647
`float`	4	Lowest positive value: $1.17*10^{-38}$ Highest positive value: $3.4*10^{38}$
`double`	8	Lowest positive value: $2.2*10^{-308}$ Highest positive value: $1.8*10^{308}$
`long double`	8, 10, 12, 16	
`unsigned char`	1	0…255
`unsigned short int`	2	0…65535
`unsigned int`	4	0…4.294.967.295
`unsigned long int`	4	0…4.294.967.295

What is the output ?

```c
#include <stdio.h>

int main() {

    float a = 3.1;
    if (a == 3.1)
        printf("Yes\n");
    else
        printf("No\n");
    printf("%.9f\n", a - 3.1);
    return 0;
}
```

What is the output ?

```c
#include <stdio.h>

int main() {

    float a = 3.1;
    if (a == 3.1)
        printf("Yes\n");
    else
        printf("No\n");
    printf("%.9f\n", a - 3.1);
    return 0;
}
```

```
No
-0.000000095
```

```c
int main () {

   /* variable definition: */
   int a, b;
   int c;
   float f;
   char s[15]="sample string";
   double d;

   /* actual initialization */
   a = 10;
   b = 20;
   d=25.0;

   c = a + b;
   printf("value of c : %d \n", c);

   f = 70.0/3.0;
   printf("value of f : %f \n", f);

   d= 50.0/d;
   printf("value of d : %f \n", d);

   printf("value of s : %s \n", s);

   int functionResult = somefunction();
   printf("value of function call : %d\n",functionResult);
   return 0;
}
```

```
38  int somefunction(){
39      int a=5;
40      int b=7;
41      return (a+b);
42
43  }
```

```
value of c : 30
value of f : 23.333334
value of d : 2.000000
value of s : sample string
value of function call : 12
```

Arithmetic Conversions

```
char c;
short s;
int i;
unsigned int u;
float f;
double d;
long double ld;
i = i+c; /* c is converted to int. */
i = i+s; /* s is converted to int. */
u = u+i; /* i is converted to unsigned int. */
f = f+i; /* i is converted to float. */
f = f+u; /* u is converted to float. */
d = d+f; /* f is converted to double. */
ld = ld+d; /* d is converted to long double. */
```

const

- A variable whose value cannot change during the execution of the program is called *constant*.

```
const int a = 10;
const double PI = 3.14;
int somefunction(const int *data, size_t size);
```

`printf()`

- The `printf()` function is used to display data to stdout (*st*andar*d* *out*put stream).
- The `printf()` function accepts a variable list of parameters.
  - The first argument is a format string, that is, a sequence of characters enclosed in double quotes, which determines the output format.
- The format string may contain escape sequences and conversion specifications.

Escape Sequences

- Escape sequences are used to represent nonprintable characters or characters that have a special meaning to the compiler. An escape sequence consists of a backslash (\) followed by a character.

Escape Sequences

`\a`	Audible alert.
`\b`	Backspace.
`\f`	Form feed.
`\n`	New line.
`\r`	Carriage return.
`\t`	Horizontal tab.
`\v`	Vertical tab.
`\\`	Backslash.
`\'`	Single quote.
`\"`	Double quote.
`\?`	Question mark.

Conversion Specifications

- A conversion specification begins with the % character, and it is followed by one or more characters with special significance. In its simplest form, the % is followed by one of the conversion specifiers below

Conversion Specifier	Meaning
c	Display the character that corresponds to an unsigned integer value.
d ή i	Display a signed integer.
u	Display an unsigned integer.
f	Display a floating-point number. The default precision is six digits.
s	Display a sequence of characters.
e, E	Display a floating-point number in scientific notation. The exponent is preceded by the chosen specifier e or E.
g, G	%e or %E form is selected if the exponent is less than -4 or greater than or equal to the precision. Otherwise, the %f form is used.
P	Display the value of a pointer variable.
x, X	Display an unsigned integer in hex form; %x displays lowercase letters (a-f), while %X displays uppercase letters (A-F).
O	Display an unsigned integer in octal.
n	Nothing is displayed. The matching argument must be a pointer to integer; the number of characters printed so far will be stored in that integer.
%	Display the character %.

```c
#include <stdio.h>
int main(void)
{
        int len;
        printf("%c\n", 'w');
        printf("%d\n", -100);
        printf("%f\n", 1.56);
        printf("%s\n", "some text");
        printf("%e\n", 100.25);
        printf("%g\n", 0.0000123);
        printf("%X\n", 15);
        printf("%o\n", 14);
        printf("test%n\n", &len);
        printf("%d%%\n", 100);
        return 0;
}
```

The program outputs:

`w` (the character constant must be enclosed in single quotes).
`-100`
`1.560000`
`some text` (the string literal must be enclosed in double quotes).
`1.002500e+002` (equivalent to $1.0025 \times 10^2 = 1.0025 \times 100 = 100.25$).
`1.23e-005` (because the exponent is less than `-4`, the number is displayed in scientific form).
`F` (the number `15` is equivalent to `F` in hex).
`16` (the number `14` is equivalent to `16` in octal).
`test` (since four characters have been printed before `%n` is met, the value `4` is stored into `len`.
`100%` (to display the `%` character, we must write it twice).

Precision

```c
#include <stdio.h>
int main(void)
{
        float a = 1.2365;
        printf("%f\n", a);
        printf("%.2f\n", a);
        printf("%.*f\n", 3, a);
        printf("%.f\n", a);
        return 0;
}
```

```
1.236500
1.24
1.237
1
```

scanf()

- The scanf() function is used to read data from stdin (*st*andar*d* *in*put stream) and store that data in program variables.

- The scanf() function accepts a variable list of parameters. The first is a format string similar to that of printf(), followed by the memory addresses of the variables in which the input data will be stored.

- Typically, the format string contains only conversion specifiers. The conversion characters used in scanf() are the same as those used in printf().

scanf()

```
int i;
float j;
scanf("%d%f", &i, &j);

char str[100];
scanf("%s", str);
```

```c
#include <stdio.h>
int main(void)
{
    char ch;
    int i;
    float f;
    printf("Enter character, int and float: ");
    scanf("%c%d%f", &ch, &i, &f);
    printf("\nC:%c\tI:%d\tF:%f\n", ch, i, f);
    return 0;
}
```

```
Enter character, int and float: s 17 22.6

C:s     I:17    F:22.600000
```

The assignment operator =

```
int a,b,c,d,e;
a=b=c=d=e=25;
```

or even the following is legal

```
a=25;
d=a + ( b = ( (e = a+10) + 40));
```

Arithmetic Operators

- + - / * %
- int/int = cuts off the decimal part

```
int a=7;
int b=5;
a/b will be equal to 1
```

also, be careful with the % operator

```
if ((n%2)==1) is dangerous**
if((n%2)!=0)  is safe
```

** if n is odd and negative

++ and --

- Similar to Java

```
int a=25;
a++; /* is equal to a = a+1; */
a--; /* is equal to a = a-1; */

int c=5,d;
d=c++;
```

or

```
d=++c;
```

Compound Assignment Operators

```
#include <stdio.h>
int main(void)
{
        int a = 4, b = 2;
        a += 6;
        a *= b+3;
        a -= b+8;
        a /= b;
        a %= b+1;
        printf("Num = %d\n", a);
        return 0;
}
```

Comparisons

- > >= < <= != ==
- if (a == 10)

Logical Operators

- ! not operator, && operator, || operator

```
#include <stdio.h>
int main(void)
{
        int a = 4;
        printf("%d\n", !a);
        return 0;
}
```

The Comma Operator

- The comma (,) operator can be used to merge several expressions to form a single expression

```
#include <stdio.h>
int main(void)
{
        int b;
        b = 20, b = b+30, printf("%d\n", b);
        return 0;
}
```

Operator Precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm

```c
#include <stdio.h>

int main() {

    int a=10,b=20,c=30,d=40,e;

    e = (a + b) * c / d;       // (10+20)* 30 / 40
    printf("Value of (a + b) * c / d is : %d\n",  e );

    e = ((a + b) * c) / d;     // ((10+20)* 30 ) / 40
    printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );

    e = (a + b) * (c / d);   // (10+20) * (30/40)
    printf("Value of (a + b) * (c / d) is  : %d\n",  e );

    e = a + (b * c) / d;     //  10 + (20*30)/40
    printf("Value of a + (b * c) / d is  : %d\n" ,  e );

    return 0;
}
```


https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm

```c
#include <stdio.h>

int main() {

    int a=10,b=20,c=30,d=40,e;

    e = (a + b) * c / d;       // (10+20)* 30 / 40
    printf("Value of (a + b) * c / d is : %d\n",  e );

    e = ((a + b) * c) / d;     // ((10+20)* 30 ) / 40
    printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );

    e = (a + b) * (c / d);   // (10+20) * (30/40)
    printf("Value of (a + b) * (c / d) is  : %d\n",  e );

    e = a + (b * c) / d;     //  10 + (20*30)/40
    printf("Value of a + (b * c) / d is  : %d\n" ,  e );

    return 0;
}
```

22

22

0

25

https://www.tutorialspoint.com/cprogramming/c_operators_precedence.htm

if else else if

```
#include <stdio.h>
int main(void)
{
        int a = 10, b = 20, c = 30;
        if(a > 5)
        {
                if(b == 20)
                        printf("1 ");
                if(c == 40)
                        printf("2 ");
                else
                        printf("3 ");
        }
        else
                printf("4\n");
        return 0;
}
```

switch statement

```c
#include <stdio.h>
int main(void)
{
        int a;
        printf("Enter number: ");
        scanf("%d", &a);
        switch(a)
        {
                case 1:
                        printf("One\n");
                break;
                case 2:
                        printf("Two\n");
                break;
                default:
                        printf("Other\n");
                break;
        }
        printf("End\n");
        return 0;
}
```

for loop

```c
#include <stdio.h>
int main(void)
{
    int a;
    for(a = 0; a < 5; a++)
    {
        printf("%d ", a);
    }
    return 0;
}
```

The `break` Statement

```
#include <stdio.h>
int main(void)
{
        int i;
        for(i = 1; i < 10; i++)
        {
                if(i == 5)
                        break;
                printf("%d ", i);
        }
        printf("End = %d\n", i);
        return 0;
}
```

The continue Statement

```
#include <stdio.h>
int main(void)
{
        int i;
        for(i = 1; i < 10; i++)
        {
                if(i < 5)
                        continue;
                printf("%d ", i);
        }
        return 0;
}
```

while loop

```
#include <stdio.h>
int main(void)
{
        int i = 10;
        while(i != 0)
        {
                printf("%d\n", i);
                i--;
        }
        return 0;
}
```

do-while loop

```
#include <stdio.h>
int main(void)
{
        int i = 1;
        do
        {
                printf("%d\n", i);
                i++;
        } while(i <= 10);
        return 0;
}
```

References

- https://www.tutorialspoint.com/cprogramming/c_constants.htm
- C From Theory to Practice - 2nd edition, Nikolaos D. Tselikas and George S. Tselikis

Data Types in C

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges –

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

https://www.tutorialspoint.com/cprogramming/c_data_types.htm

Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision –

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values –

https://www.tutorialspoint.com/cprogramming/c_data_types.htm

for loop

```c
#include <stdio.h>
int main() {
 for (;;){
     printf("This is a strange infinite loop");
 }
}
```

while loop

```c
#include <stdio.h>
int main(void)
{
        int i = 10;
        while(i != 0)
        {
                printf("%d\n", i);
                i--;
        }
        return 0;
}
```

do-while loop

```
#include <stdio.h>
int main(void)
{
        int i = 1;
        do
        {
                printf("%d\n", i);
                i++;
        } while(i <= 10);
        return 0;
}
```

Arrays

- **One-Dimensional Arrays**
  - An array is a data structure that contains a number of values, or else elements, of the same type.
  - Each element can be accessed by its position within the array
  - Always declare the array before your try to use them

    data_type array_name[number_of_elements];

```
int sampleArray[100];
float anotherArray[250];
```

predefined size

```
/* use macros */
#define ARRAY_SIZE 250
float sampleArray[ARRAY_SIZE];
```

```
/* never use const */
const int array_size = 250;
float sampleArray(array_size)
/* this is not legal */
```

sizeof()

```c
#include <stdio.h>
int main() {

    printf("%lu\n", sizeof(char));
    printf("%lu\n", sizeof(int));
    printf("%lu\n", sizeof(float));
    printf("%lu\n", sizeof(double));

    int a = 25;
    double d= 40.55;
    printf("%lu\n",sizeof(a+d));

    int arr[10] = {5,8,9,12};
    printf("\n Size of the array :%lu", sizeof(arr));
    printf("\n Capacity the array :%lu", sizeof(arr)/sizeof(arr[0]));

    int arr2[] = {5,8,9,12};
    printf("\n Size of the array2 :%lu", sizeof(arr2));
    printf("\n Capacity the array2 :%lu", sizeof(arr2)/sizeof(arr2[0]));
    return 0;
}
```

sizeof()

```c
#include <stdio.h>
int main() {

    printf("%lu\n", sizeof(char));    1
    printf("%lu\n", sizeof(int));     4
    printf("%lu\n", sizeof(float));   4
    printf("%lu\n", sizeof(double));  8

    int a = 25;
    double d= 40.55;
    printf("%lu\n",sizeof(a+d));   8

    int arr[10] = {5,8,9,12};
    printf("\n Size of the array :%lu", sizeof(arr));                    40
    printf("\n Capacity the array :%lu", sizeof(arr)/sizeof(arr[0]));    10

    int arr2[] = {5,8,9,12};
    printf("\n Size of the array2 :%lu", sizeof(arr2));                  16
    printf("\n Capacity the array2 :%lu", sizeof(arr2)/sizeof(arr2[0])); 4
    return 0;
}
```

Initialize the Array

```
int arr[3]={10,20,30};
int arr2[10]={10,20};

int arr3[]={10,20,30,40};

/* be careful with the
following*/
const int arr4[] = {10,20,30,40}
```

Assign - Access elements

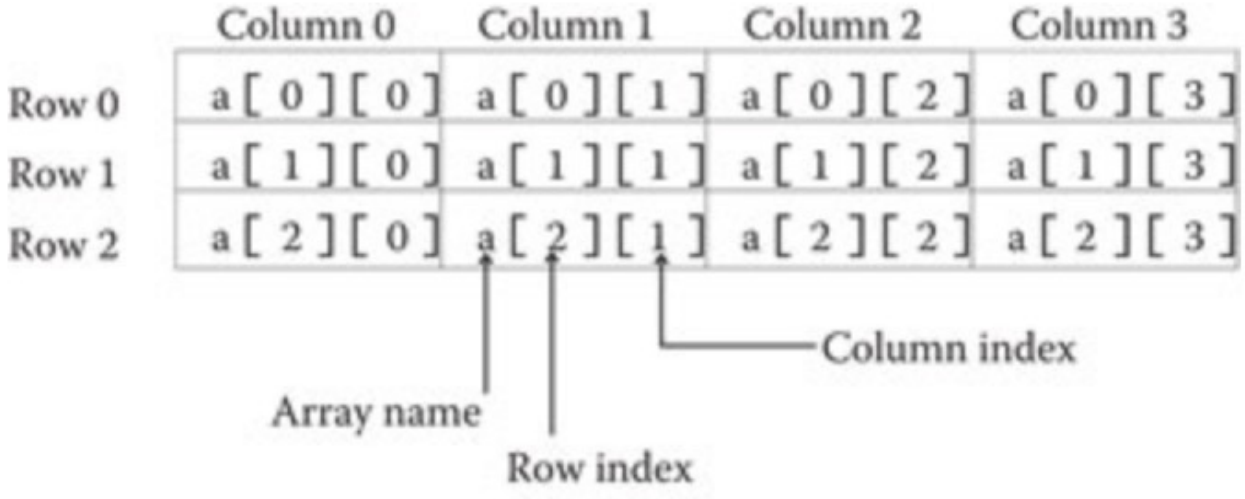
```c
#include <stdio.h>
int main() {
    int i,j=10, arr[10];
    arr[0]=10;
    arr[1]=arr[0]*2;
    for (i=2;i<10;i++){
        arr[i]=j*(i+1);
    }
    for (i=0;i<10;i++)
        printf("\n arr[%d] :%d",i,arr[i]);

    return 0;
}
```

```
arr[0] :10
arr[1] :20
arr[2] :30
arr[3] :40
arr[4] :50
arr[5] :60
arr[6] :70
arr[7] :80
arr[8] :90
arr[9] :100
```

2D Arrays

- **data_type** array_name[number_of_rows][number_of_columns]

```
int a[3][4];
```



initialize 2D array

```
int arr[3][3] = {{10, 20, 30},
{40, 50, 60}, {70, 80, 90}};


int arr[3][4] = {10, 20, 30, 40,
50, 60, 70, 80, 90};
10 20 30 40
50 60 70 80
90 0   0  0
```

Pointers

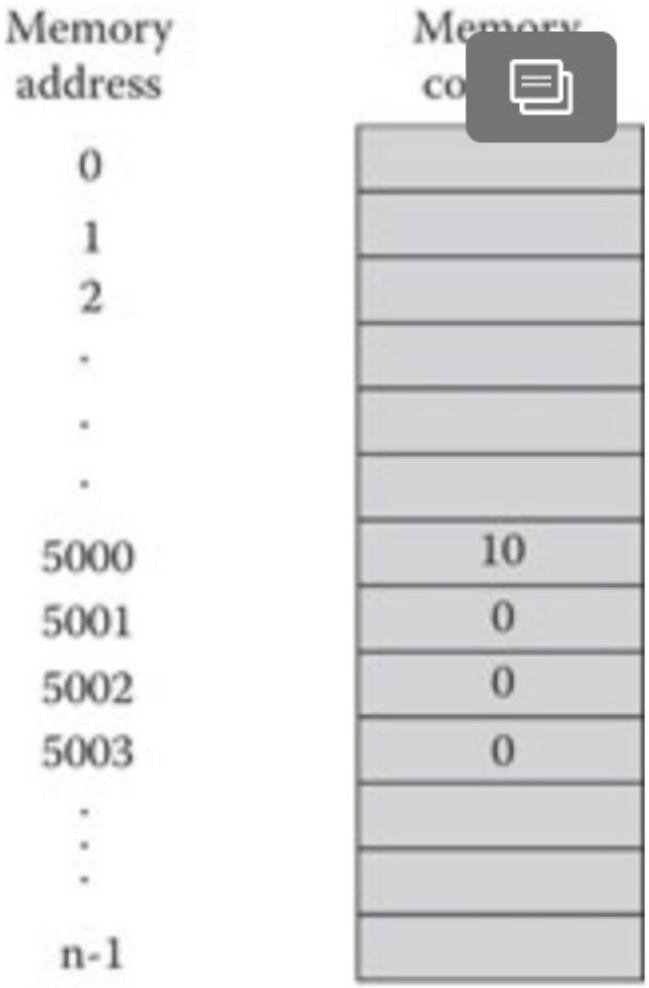
```
int a = 5;
float arr[25];
```

- /* To reach the memory location variables, arrays…etc use ampersand (&) operator */

```
printf("\n %x", &a);
printf("\n %x", &arr);
```

```
e7ad78b8
e7ad78c0
```

Memory Address



Pointers / 2

- How to store this address?
  - we use the pointers
  - Pointers is a variable whose value is the address of another variable
- Declare the pointer before you use it

```
data_type *pointer_name;
int *ptr, a, b, c;
int * ptr, a, b, c;
int* ptr, a, b, c;
```

!!! Caution !!!!

All three statements are correct and the result of each statement the “ptr” will be declared as the pointer but a,b, and c will be declared as int.
However; it is always better to use the first syntax

```
data_t  pointer_name;
int *ptr, a, b, c;
int * ptr, a, b, c;
int* ptr, a, b, c;
```

size of a pointer ???

```c
#include <stdio.h>
int main() {
    char c;
    char *ptrC = &c;
    int a=5;
    int *ptrI = &a;
    float f =20.66;
    float *ptrF = &f;
    double d = 44.445;
    double *ptrD = &d;

    printf("size of c: %u\n", sizeof(c));
    printf("size of ptrC: %u\n", sizeof(ptrC));
    printf("size of a: %u\n", sizeof(a));
    printf("size of ptrI: %u\n", sizeof(ptrI));
    printf("size of f: %u\n", sizeof(f));
    printf("size of ptrF: %u\n", sizeof(ptrF));
    printf("size of d: %u\n", sizeof(d));
    printf("size of ptrD: %u\n", sizeof(ptrD));
    return 0;
}
```

```
size of c: 1
size of ptrC: 8
size of a: 4
size of ptrI: 8
size of f: 4
size of ptrF: 8
size of d: 8
size of ptrD: 8
```

```c
#include <stdio.h>
int main(void)
{
    int *ptr, a;
    a = 10;
    ptr = &a;
    printf("Val = %d\n", *ptr);
    return 0;
}
```

10

Always initialize the pointer before using it, otherwise you will get segmentation fault error

Example - page 1/2

```c
#include <stdio.h>
int main()
{
    int *ptr, a;
    a = 25;
    /* without using a pointer */
    printf("Address of a: %p\n", &a);
    printf("Value of a: %d\n", a);

    /*let's use a pointer */
    ptr = &a;
    printf("Address of the pointer : %p\n", ptr);
    printf("Value of the pointer : %d\n", *ptr);

    /* how about if we change the value of int */
    a = 125;
    printf("Address of the pointer : %p\n", ptr);
    printf("Value of the pointer : %d\n", *ptr);

```

Example - page 2/2

```
20      /* let's change the value using pointer*/
21      *ptr = 250;
22      printf("Address of a: %p\n", &a);
23      printf("Value of a: %d\n", a);
24
25      /* we can reuse the pointer */
26      int b = 50;
27      ptr = &b;
28      printf("Address of the pointer : %p\n", ptr);
29      printf("Value of the pointer : %d\n", *ptr);
30      return 0;
31  }
```

Example - output

```
Address of a: 0x7ffeee56291c
Value of a: 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 125
Address of a: 0x7ffeee56291c
Value of a: 250
Address of the pointer : 0x7ffeee562918
Value of the pointer : 50
```

Example - page 1/2

```c
#include <stdio.h>
int main()
{
    int *ptr, a;
    a = 25;
    /* without using a pointer */
    printf("Address of a: %p\n", &a);
    printf("Value of a: %d\n", a);

    /*let's use a pointer */
    ptr = &a;
    printf("Address of the pointer : %p\n", ptr);
    printf("Value of the pointer : %d\n", *ptr);

    /* how about if we change the value of int */
    a = 125;
    printf("Address of the pointer : %p\n", ptr);
    printf("Value of the pointer : %d\n", *ptr);

```

Example - page 2/2

```
20      /* let's change the value using pointer*/
21      *ptr = 250;
22      printf("Address of a: %p\n", &a);
23      printf("Value of a: %d\n", a);
24
25      /* we can reuse the pointer */
26      int b = 50;
27      ptr = &b;
28      printf("Address of the pointer : %p\n", ptr);
29      printf("Value of the pointer : %d\n", *ptr);
30      return 0;
31  }
```

Example - output

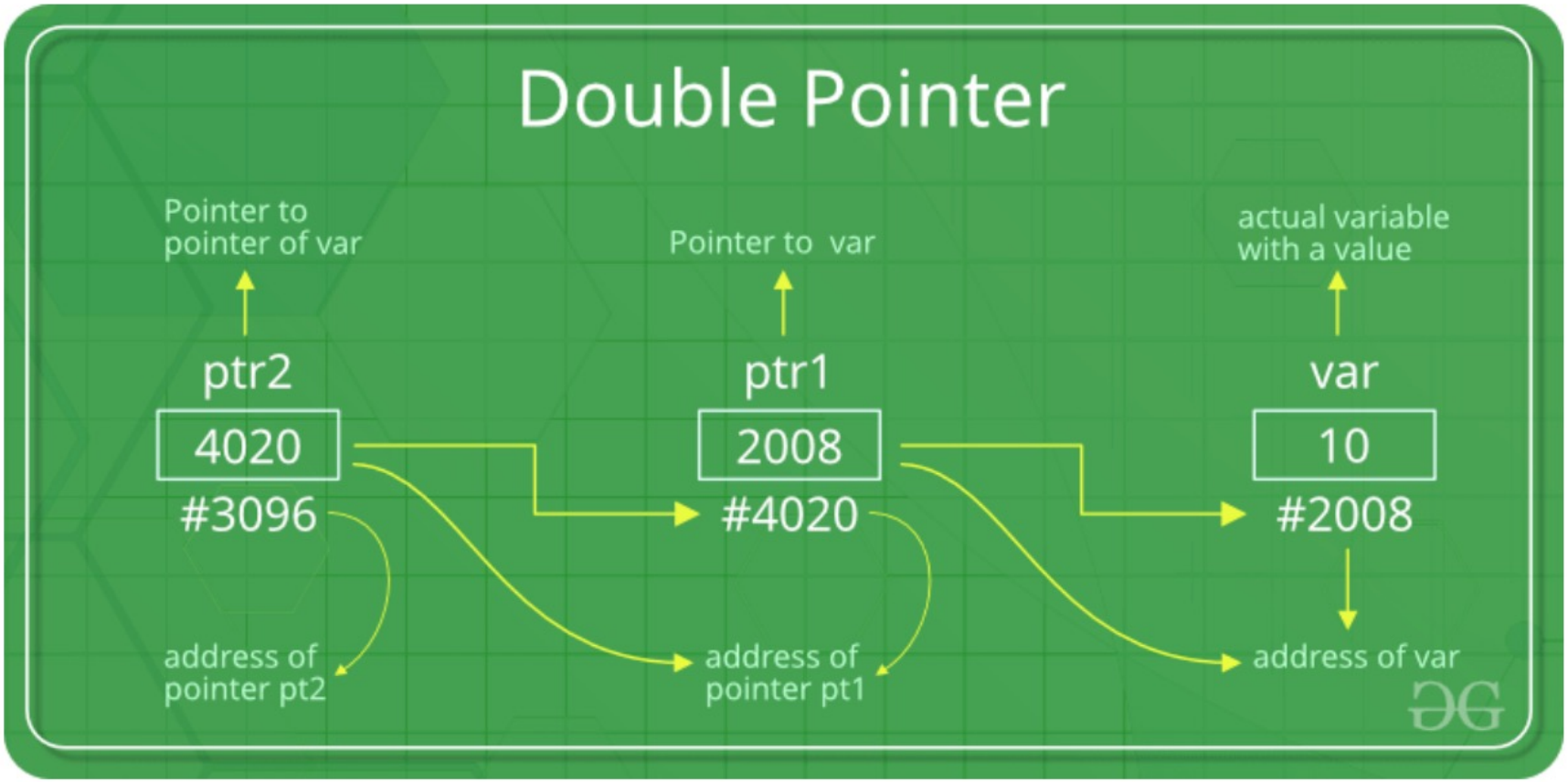
```
Address of a: 0x7ffeee56291c
Value of a: 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 25
Address of the pointer : 0x7ffeee56291c
Value of the pointer : 125
Address of a: 0x7ffeee56291c
Value of a: 250
Address of the pointer : 0x7ffeee562918
Value of the pointer : 50
```

The * and & cancel each other when used together

```c
#include <stdio.h>

int main() {
    int *ptr, i=5;
    ptr = &i;
    printf("%p %p %p %d %p\n", &i, *&ptr, &*ptr, *ptr, &ptr);
    return 0;
}
```

```
0x7ffee636291c 0x7ffee636291c 0x7ffee636291c 5 0x7ffee6362920
```

double pointer? even triple…



https://www.geeksforgeeks.org/double-pointer-pointer-pointer-c/

```c
#include <stdio.h>
int main() {

    int a = 25;
    int *ptr = &a;//the first pointer

    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    *ptr = 45;// change the value
    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    int **ptr2 = &ptr;// second pointer
    printf("Address - First pointer: %p\n", ptr);
    printf("Value -First Pointer: %i\n", *ptr);
    printf("Address - First pointer: %p\n", ptr2);
    printf("Value -First Pointer: %i\n", **ptr2);
    return 0;
}
```

```c
#include <stdio.h>
int main() {

    int a = 25;
    int *ptr = &a;//the first pointer

    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    *ptr = 45;// change the value
    printf("Address : %p\n", ptr);
    printf("Value : %i\n", *ptr);

    int **ptr2 = &ptr;// second pointer
    printf("Address - First pointer: %p\n", ptr);
    printf("Value -First Pointer: %i\n", *ptr);
    printf("Address - First pointer: %p\n", ptr2);
    printf("Value -First Pointer: %i\n", **ptr2);
    return 0;
}
```

```
Address : 0x7ffee76aa928
Value : 25
Address : 0x7ffee76aa928
Value : 45
Address - First pointer: 0x7ffee76aa928
Value -First Pointer: 45
Address - First pointer: 0x7ffee76aa920
Value -First Pointer: 45
```

Arrays & Pointers

```c
#include <stdio.h>
int main() {
    int i,arr[5];
    double arr2[5];

    for(i = 0; i <5; i++)
        printf("address of arr[%d] = %p\n", i, &arr[i]);

    for(i = 0; i <5; i++)
        printf("address of arr2[%d] = %p\n", i, &arr2[i]);
    return 0;
}
```

Arrays & Pointers

```
address of arr[0] = 0x7ffeeece0910
address of arr[1] = 0x7ffeeece0914
address of arr[2] = 0x7ffeeece0918
address of arr[3] = 0x7ffeeece091c
address of arr[4] = 0x7ffeeece0920
address of arr2[0] = 0x7ffeeece08e0
address of arr2[1] = 0x7ffeeece08e8
address of arr2[2] = 0x7ffeeece08f0
address of arr2[3] = 0x7ffeeece08f8
address of arr2[4] = 0x7ffeeece0900
```

Array Manipulation

```c
#include <stdio.h>
int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr;

    ptr = &arr[3]; // address of the fourth element

    printf("\n Pointer value : %d", *ptr);
    printf(" \n Next Value : %d", *(ptr+1));
    printf("\n Previous Value : %d", *(ptr-1));

    printf("\n Address of the Pointer : %p", &(*(ptr)));
    printf("\n Address of the Next Value : %p", &(*(ptr+1)));
    printf("\n Address of the Previous Value : %p", &(*(ptr-1)));
    return 0;
}
```

Array Manipulation

```
Pointer value : 40
Next Value : 50
Previous Value : 30
Address of the Pointer : 0x7ffeeb08e91c
Address of the Next Value : 0x7ffeeb08e920
Address of the Previous Value : 0x7ffeeb08e918
```

```
1
2
3
4
5
6
7
8       printf("\n Pointer value : %d", *ptr);
9       printf(" \n Next Value : %d", *(ptr+1));
10      printf("\n Previous Value : %d", *(ptr-1));
11
12      printf("\n Address of the Pointer : %p", &(*(ptr)));
13      printf("\n Address of the Next Value : %p", &(*(ptr+1)));
14      printf("\n Address of the Previous Value : %p", &(*(ptr-1)));
15      return 0;
16  }
```

What is the Result ?

```c
#include <stdio.h>
int main(void)
{
    int *ptr, totalSum, arr[5] = {10, 20, 30, 40, 50};
    totalSum = 0;
    for(ptr = arr; ptr < arr+5; ptr++)
    {
        --*ptr;
        totalSum += *ptr;
    }
    printf("Sum = %d\n", totalSum);
    return 0;
}
```

What is the Result ?

```c
#include <stdio.h>
int main(void)
{
    int *ptr, totalSum, arr[5] = {10, 20, 30, 40, 50};
    totalSum = 0;
    for(ptr = arr; ptr < arr+5; ptr++)
    {
        --*ptr;
        totalSum += *ptr;
    }
    printf("Sum = %d\n", totalSum);
    return 0;
}
```

145

Passing Pointers to functions

```c
#include <stdio.h>
void test(int a);
int main(void)
{
    void (*ptr)(int a);
    ptr = test;
    (*ptr)(10);
    return 0;
}
void test(int a)
{
    printf("%d\n", 2*a);
}
```

```
20
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr[1](i, j);
    printf("Result  : %d\n", result);
    return 0;
}
int addTwoNumbers(int a, int b)
{   return a+b;  }
int subtractTwoNumbers(int a, int b)
{   return a-b;   }
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr
    printf("Result
    return 0;
}
int addTwoNumbers(int
{   return a+b;  }
int subtractTwoNumber
{   return a-b;   }
```

```
Enter two integer numbers: 20 30
Result   : 50
```

```
Enter two integer numbers: 45 20
Result   : 25
```

The `char` Type

- Since a character in the ASCII set is represented by an integer between 0 and 255, we can use the **char** type to store its value.
- Once a character is stored into a variable, it is the character's ASCII value that is actually stored.

```
char ch;
ch = 'c';
```

  - the value of `ch` becomes equal to the ASCII value of the character 'c'.
  - Therefore,
    - the statements `ch = 'c';` and `ch = 99;` are equivalent.
    - Of course, 'c' is preferable than 99; not only it is easier to read, but also your program won't depend on the character set as well.

```c
#include <stdio.h>
int main(void)
{
    char ch;
    ch = 'a';
    printf("Char = %c and its ASCII code is %d\n", ch, ch);
    return 0;
}

```

```
Char = a and its ASCII code is 97
```

- Since C treats the characters as integers, we can use them in numerical expressions. For example:

```
char ch = 'c’;
int i; ch++; /* ch becomes 'd'. */
ch = 68; /* ch becomes 'D'. */
i = ch-3; /* i becomes 'A',that is 65 */
```

`getchar()` and `putchar()`

- The `getchar()` function is used to read a character from stdin.
- The `putchar()` function writes a character in stdout, for example, `putchar('a')`

Strings

- A string literal is a sequence of characters enclosed in double quotes.
- C treats it as a nameless character array.
- To store a string in a variable, we use an array of characters.
- Because of the C convention that a string ends with the null character, to store a string of **N** characters, the size of the array should be **N+1** at least.

```
char str[8];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```
char str[8] = "message";
```

the compiler copies the characters of the "message" into the str array and adds the null character. In particular, str[0] becomes 'm', str[1] becomes 'e', and the value of the last element str[7] becomes '\0'. In fact, this declaration is equivalent to:

```
char str[8] = {'m', 'e', 's',
's', 'a', 'g', 'e', '\0'};
```

puts()

```c
#include <stdio.h>
int main(void)
{
    char str[] = "UAB CS 330 Course";
    puts(str);
    puts(str);
    str[4] = '\0';
    printf("%s\n", str);
    return 0;
}
```

```
UAB CS 330 Course
UAB CS 330 Course
UAB
```

Passing Pointers to functions

```c
#include <stdio.h>
void test(int a);
int main(void)
{
    void (*ptr)(int a);
    ptr = test;
    (*ptr)(10);
    return 0;
}
void test(int a)
{
    printf("%d\n", 2*a);
}
```

```
20
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr[1](i, j);
    printf("Result  : %d\n", result);
    return 0;
}
int addTwoNumbers(int a, int b)
{   return a+b;  }
int subtractTwoNumbers(int a, int b)
{   return a-b;   }
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr
    printf("Result
    return 0;
}
int addTwoNumbers(int
{   return a+b;  }
int subtractTwoNumber
{   return a-b;   }
```

```
Enter two integer numbers: 20 30
Result   : 50
```

```
Enter two integer numbers: 45 20
Result   : 25
```

The `char` Type

- Since a character in the ASCII set is represented by an integer between 0 and 255, we can use the **char** type to store its value.
- Once a character is stored into a variable, it is the character's ASCII value that is actually stored.

```
char ch;
ch = 'c';
```

  - the value of `ch` becomes equal to the ASCII value of the character 'c'.
  - Therefore,
    - the statements `ch = 'c';` and `ch = 99;` are equivalent.
    - Of course, 'c' is preferable than 99; not only it is easier to read, but also your program won't depend on the character set as well.

```c
#include <stdio.h>
int main(void)
{
    char ch;
    ch = 'a';
    printf("Char = %c and its ASCII code is %d\n", ch, ch);
    return 0;
}

```

```
Char = a and its ASCII code is 97
```

- Since C treats the characters as integers, we can use them in numerical expressions. For example:

```
char ch = 'c’;
int i; ch++; /* ch becomes 'd'. */
ch = 68; /* ch becomes 'D'. */
i = ch-3; /* i becomes 'A',that is 65 */
```

`getchar()` and `putchar()`

- The `getchar()` function is used to read a character from stdin.
- The `putchar()` function writes a character in stdout, for example, `putchar('a')`

Strings

- A string literal is a sequence of characters enclosed in double quotes.
- C treats it as a nameless character array.
- To store a string in a variable, we use an array of characters.
- Because of the C convention that a string ends with the null character, to store a string of **N** characters, the size of the array should be **N+1** at least.

```
char str[8];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```
char str[8] = "message";
```

the compiler copies the characters of the "message" into the str array and adds the null character. In particular, str[0] becomes 'm', str[1] becomes 'e', and the value of the last element str[7] becomes '\0'. In fact, this declaration is equivalent to:

```
char str[8] = {'m', 'e', 's',
's', 'a', 'g', 'e', '\0'};
```

puts()

```c
#include <stdio.h>
int main(void)
{
    char str[] = "UAB CS 330 Course";
    puts(str);
    puts(str);
    str[4] = '\0';
    printf("%s\n", str);
    return 0;
}
```

```
UAB CS 330 Course
UAB CS 330 Course
UAB
```

scanf()

- `scanf()` takes as an argument a pointer to the array that will hold the input string.

- Since we're using the name of the array as a pointer, we don't add the address operator **&** before its name.

- Because `scanf()` stops reading once it encounters the space character, only the word this is stored into str. Therefore, the program outputs this.

- To force `scanf()` to read multiple words, we can use a more complex form such as `scanf("%[^\n]", str);`

gets() fgets()

```
char *gets(char *str);
```

`gets()` is not safe, don't use it

```c
#include <stdio.h>
int main(void)
{
    char str[100];
    int num;
    printf("Enter number: ");
    scanf("%d", &num);
    printf("Enter text: ");
    fgets(str, sizeof(str), stdin);
    printf("%d %s\n", num, str);
    return 0;
}
```

```
Enter number: 1223132312312312312323123123231231231231232131231231231231231231231231231231231231
Enter text: -1
```

The `strlen()` Function

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

The size_t type is defined in the C library as an unsigned integer type (usually as **unsigned int**).

`strlen()` returns the number of characters in the string pointed to by str, not counting the null character.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str1[100], str2[100];
    printf("Enter text: ");
    fgets(str2, sizeof(str2), stdin);
    strcpy(str1, str2);
    printf("Copied text: %s\n", str1);
    return 0;
}
```

```
Enter text: Hello CS330
Copied text: Hello CS330
```

Search the following functions

strcat()

strcmp()

Functions →Array as Arguments

- When a parameter of a function is a one-dimensional array, we write the name of the array followed by a pair of brackets.

- The length of the array can be omitted; in fact, this is the common practice.

- For example:

```
void test(int arr[]);
```

- When passing an array to a function, we write only its name, without brackets. For example:

```
test(arr);
```

Memory Blocks

- Code
- Data
- Stack
- Heap

```c
#include <stdio.h>
#include <stdlib.h>
void test(void);
int global;
int main(void)
{
    int *ptr;
    int i;
    static int st;

    /* Allocate memory from the heap. */
    ptr = (int*) malloc(sizeof(int));

    if(ptr != NULL)
    {
        printf("Code seg: %p\n", test);
        printf("Data seg: %p %p\n", &global, &st);
        printf("Stack seg: %p\n", &i);
        printf("Heap: %p\n", ptr);
        free(ptr);
    }
    return 0;
}
void test(void)
{ /* Do something. */
}
```

```
Code seg: 0x106e1ff30
Data seg: 0x106e21024 0x106e21020
Stack seg: 0x7ffee8de091c
Heap: 0x7f8a55405840
```

Static Memory Allocation

- In static allocation, the memory is allocated from the stack.
- The size of the allocated memory is fixed; we must specify its size when writing the program and it cannot change during program execution.
- For example, with the statement:

```
float grades[1000];
```

Dynamic Memory Allocation

- In dynamic allocation, the memory is allocated from the heap during program execution. Unlike static allocation, its size can be dynamically specified.
- Furthermore, this size may dynamically shrink or grow according to the program's needs.
- Typically, the default stack size is not very large, the size of the heap is usually much larger than the stack size.

malloc()

```
void *malloc(size_t size);
```

The `size_t` type is usually a synonym of the **unsigned int** type.

The size parameter declares the number of bytes to be allocated.

If the memory is allocated successfully;

`malloc()` returns a pointer to that memory, NULL otherwise.

Check the following functions

`realloc()`

`calloc()`

`free()`

`memcpy()`

`memmove()`

`memcmp()`

Static Memory Allocation

- In static allocation, the memory is allocated from the stack.
- The size of the allocated memory is fixed; we must specify its size when writing the program and it cannot change during program execution.
- For example, with the statement:

```
float grades[1000];
```

Dynamic Memory Allocation

- In dynamic allocation, the memory is allocated from the heap during program execution. Unlike static allocation, its size can be dynamically specified.
- Furthermore, this size may dynamically shrink or grow according to the program's needs.
- Typically, the default stack size is not very large, the size of the heap is usually much larger than the stack size.

malloc()

```
void *malloc(size_t size);
```

The `size_t` type is usually a synonym of the **unsigned int** type.

The size parameter declares the number of bytes to be allocated.

If the memory is allocated successfully;

`malloc()` returns a pointer to that memory, NULL otherwise.

Check the following functions

`realloc()`

`calloc()`

`free()`

`memcpy()`

`memmove()`

`memcmp()`

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

int main()
{   int *ptr,n,i;
    /* the number of array elements */
    printf("How many elements?:\n");
    scanf("%d",&n);

    ptr = (int*)malloc(n * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation was NOT successful.\n");
        exit(0);
    }
    else {
        printf("Memory  allocation was successful.\n");
        for (i = 0; i < n; i++)
            ptr[i] = (i+1) * 10;

        for (i = 0; i < n; ++i)
            printf("%d, ", ptr[i]);
    }
    free(ptr);
    printf("\nMemory deallocation was successful.\n");
    return 0;
}
```

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

int main()
{   int *ptr,n,i;
    /* the number of array elements */
    printf("How many elements?:\n");
    scanf("%d",&n);

    ptr = (int*)malloc(n * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation was NOT successful.\n");
    return 0;
}
```

```
How many elements?:
8
Memory  allocation was successful.
10, 20, 30, 40, 50, 60, 70, 80,
Memory deallocation was successful.
```

Structures & Unions

```
struct structure_tag {
 member_list;
} structure_variable_list;
```

A **struct** declaration defines a type. Although the structure_tag is optional, we prefer to name the structures we declare and use that name later to declare variables.

```c
struct company
{
    char name[50];
    int start_year;
    int field;
    int tax_num;

    int num_empl;
    char addr[50];
    float balance;
};
```

sizeof()

```c
#include <stdio.h>
struct date
{
        int day;
        int month;
        int year;
};
int main(void)
{
        struct date d;
        printf("%u\n", sizeof(d));
        return 0;
}
```

sizeof()

```
struct test1
{
        char c;
        double d;
        short s;
};
struct test2
{
        double d;
        short s;
        char c;
};
```

```c
#include <stdio.h>
struct student
{
    int code;
    float grd;
};
int main(void)
{
    struct student s1, s2;
    s1.code = 1234;
    s1.grd = 6.7;
    s2 = s1; /* Copy structure. */
    printf("C:%d G:%.2f\n", s2.code, s2.grd);
    return 0;
}
```

Unions

- Like a structure, a union contains one or more members, which may be of different types. The properties of unions are almost identical to the properties of structures; the same operations are allowed as on structures.
- Their difference is that the members of a structure are stored at *different* addresses, while the members of a union are stored at the *same* address.

```c
#include <stdio.h>
union sample
{
    char ch;
    int i;
    double d;
};
int main(void)
{
    union sample s;
    printf("Size: %u\n", sizeof(s));
    return 0;
}
```

Operating Systems

- What is an operating system?
  - What stands between the user and the bare machine
  - The most basic and the important software to operate the computer
  - Similar role to that conductor of an orchestra
- It manages the computer's memory and processes, as well as all of its software and hardware.
- It also allows you to communicate with the computer without knowing how to speak the computer's language (hide the complexity from user)
- Without an operating system, a computer is useless.

https://edu.gcfglobal.org/en/computerbasics/understanding-operating-systems/1/

The Role of OS

- OS exploits the hardware resources of one or more processors to provide a set of services to system users
- OS manages secondary memory and I/O devices on behalf of its users
- In short,
  - OS manages the computer's resources, such as the central processing unit, memory, disk drives, and printers
  - establishes a user interface
  - executes and provides services for applications software.

OS

- A general –purpose, modern OS can exceed 50 million lines of code
- New OS are being written all the time
  - E-book reader
  - Tablet
  - Smartphone
  - Mainframe
  - Server
  - PC
  - …..

Why to learn OS?

- To be able to write concurrent code
- Resource management
- Analyze the performance
- To fully understand how your code works
- ….
- In short,
  - this class isn't to teach you how to CREATE an OS from scratch, but to teach you how an OS works

Unsolved problem

Operating systems are an unsolved problem in computer science. Because;

- *Most of them do not work well.*
  - *Crashes, not fast enough, not easy to use, etc.*
- *Usually they do not do everything they were designed to do.*
  - *Needs are increasing every day*
- *They do not adapt to changes so easily.*
  - *New devices, processors, applications.*
- ……

Operating System Services

- execute a new program
- open a file
- read a file
- allocate a region of memory
- get the current time of day
- so on

UNIX Architecture

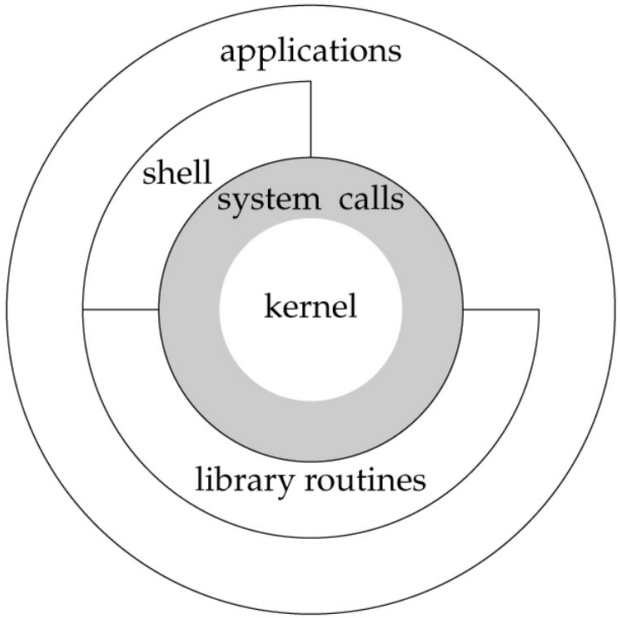


Figure 1.1 Architecture of the UNIX operating system

Linux vs UNIX

- Linux refers to the kernel of the GNU/Linux operating system. More generally, it refers to the family of derived distributions.
- Unix refers to the original operating system developed by AT&T. More generally, it refers to family of derived operating systems.
- GNU/Linux and derivates like Debian and Fedora. System-V Unix and derivatives like IBM-AIX and HP-UX; Berkeley Unix and derivatives like FreeBSD and macOS
- Linux is broadly available as configurable software download and installer. UNIX is typically shipped along with hardware e.g. MacBook

https://www.softwaretestinghelp.com/unix-vs-linux/#:~:text=Linux%20refers%20to%20the%20kernel,family%20of%20derived%20operating%20systems.

Working in the UNIX Environment

- UNIX like OS
  - Solaris
  - FreeBSD
  - macOS
  - NetBSD
  - ….
- Logging In
  - login name - password
  - password file
    - /etc/passwd

Shells

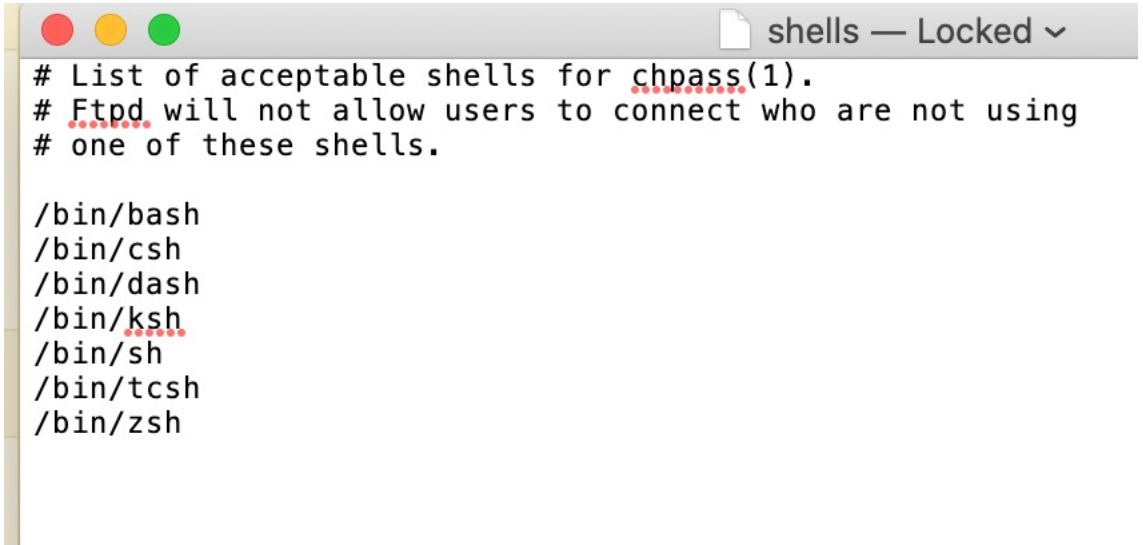
- A shell is is the interface between the user and the kernel.
- Users can interact with the shell using shell commands in terminal or from a file (shell script).
- The common shells are;

Name	Path	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Bourne shell	`/bin/sh`	•	•	copy of `bash`	•
Bourne-again shell	`/bin/bash`	optional	•	•	•
C shell	`/bin/csh`	link to `tcsh`	optional	link to `tcsh`	•
Korn shell	`/bin/ksh`	optional	optional	•	•
TENEX C shell	`/bin/tcsh`	•	optional	•	•

Figure 1.2 Common shells used on UNIX systems

MacOS users

- Start the Terminal app on your Mac
- Terminal > Preferences, then click General.
- Under “Shells open with,” select “Command (complete path),” then enter the path to the shell you want to use.
- If you want to check the available shells in your mac;
  - go to `/etc` folder and check the `shells` file

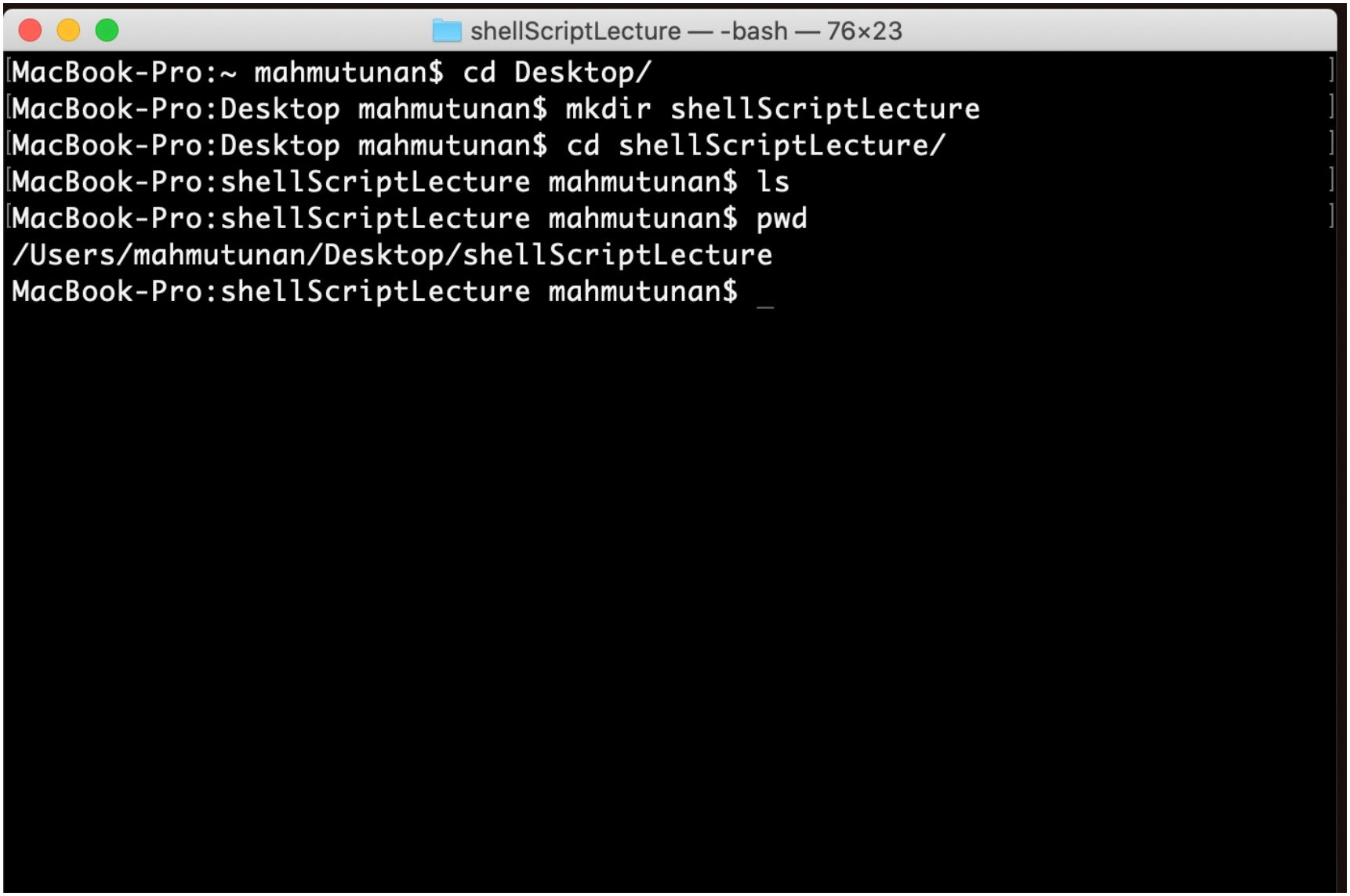


Windows Users

- Windows Subsytem for Linux
  - Bash Shell
- Git Bash

- https://www.geeksforgeeks.org/use-bash-shell-natively-windows-10/
- https://www.howtogeek.com/249966/how-to-install-and-use-the-linux-bash-shell-on-windows-10/

Exercise 1 - first script file



.sh file

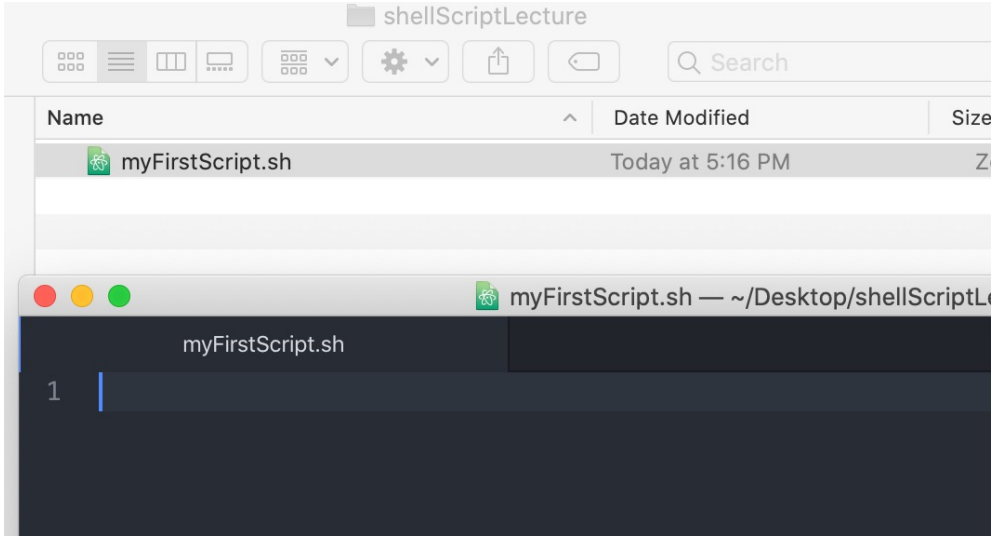
- It is a script programmed for bash
  - It contains instructions written in the Bash language
  - It can be executed by typing text commands within the shell's command-line interface.
- How to run the .sh file?
  - First, give the execute permission
    - chmod 755 somefilename.sh
  - Next, run your script file
    - sh somefilename.sh
    - bash somefilename.sh
    - ./somefilename.sh
    - if you want to run it as a root user
      - sudo bash somefilename.sh

myFirstScript.sh

- You can use your terminal to create the file and use nano to edit the file

```
MacBook-Pro:shellScriptLecture mahmutunan$ touch myFirstScript.sh
MacBook-Pro:shellScriptLecture mahmutunan$ nano myFirstScript.sh
```

- OR, you can use any editor to create and edit the file



myFirstScript.sh

```
1  #!/bin/bash
2
3  # some comment
4  echo Hello CS332!!!
5
6  LECTURE="Lecture 8"
7  echo "This is $LECTURE"
8
9  echo -n "How old are you: "
10 read AGE
11 if [[ $AGE -ge 18 ]]
12 then
13   echo "You can vote"
14 else
15   echo "You are not eligible to vote"
16 fi
17
```

myFirstScript.sh

```
1  #!/bin/bash
2
3  # some comment
4  echo Hello CS332!!!
5
6  LECTURE="Lecture 8"
7  echo "This is $LECTURE"
8
```

```
MacBook-Pro:shellScriptLecture mahmutunan$ bash myFirstScript.sh
Hello CS332!!!
This is Lecture 8
How old are you: 21
You can vote
MacBook-Pro:shellScriptLecture mahmutunan$ bash myFirstScript.sh
Hello CS332!!!
This is Lecture 8
How old are you: 11
You are not eligible to vote
MacBook-Pro:shellScriptLecture mahmutunan$
```

FILE Conditions

File operators

Operator	Note
`-e`	To check if the file exists.
`-r`	To check if the file is readable.
`-w`	To check if the file is writable.
`-x`	To check if the file is executable.
`-s`	To check if the file size is greater than 0.
`-d`	To check if the file is a directory.

https://dyclassroom.com/unix/shell-programming-file-operators

```
1  #!/bin/sh
2
3  FILE_NAME="someFileThatDoesntExist.txt"
4
5  # check
6  if [ -e $FILE_NAME ]
7  then
8    echo "Heyyooo, the file exists!"
9  else
10   echo "OOPPSSS, the file does not exists!"
11 fi
```

```
MacBook-Pro:shellScriptLecture mahmutunan$ bash fileOperations.sh
OOPPSSS, the file does not exists!
```

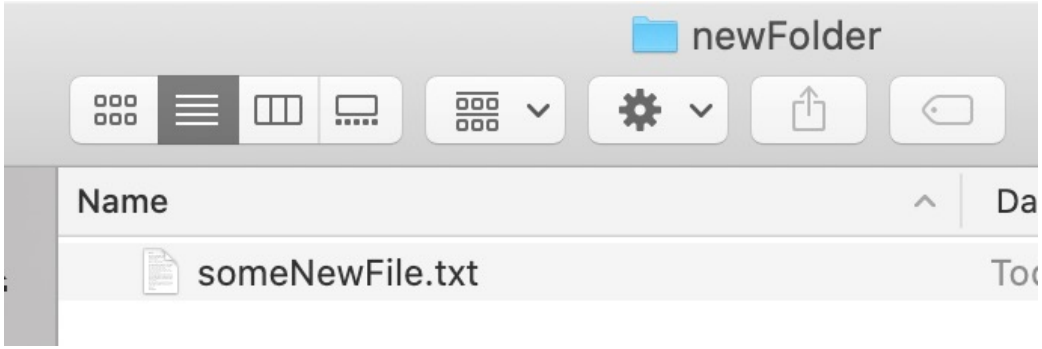
Loops & Arrays

```
MY_COURSES="CS203 CS330 CS332"
for COURSE in $MY_COURSES
do
   echo $COURSE
done
```

```
CS203
CS330
CS332
```

```
mkdir newFolder
touch "newFolder/someNewFile.txt"
echo "This message goes to the file" >> "newFolder/someNewFile.txt"
echo "This message appears on the terminal"
```

```
This message appears on the terminal
MacBook-Pro:shellScriptLecture mahmutunan$
```

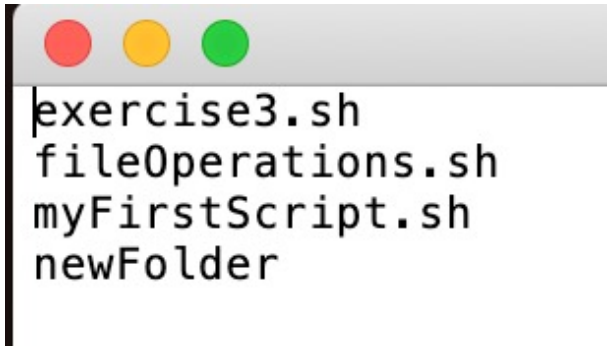


```
1  #!/bin/sh
2  clear
3  echo "Current Directory :"
4  pwd
5  echo "What is in this directory? :"
6  ls
7
8  head "myFirstScript.sh"
9
10 echo "Disk Usage :"
11 df -h
12
13 exit
14
```

```
Current Directory :
/Users/mahmutunan/Desktop/shellScriptLecture
What is in this directory? :
exercise3.sh            fileOperations.sh       myFirstScript.sh        newFolder
#!/bin/bash

# some comment
echo Hello CS332!!!

LECTURE="Lecture 8"
echo "This is $LECTURE"

echo -n "How old are you: "
read AGE
Disk Usage :
Filesystem                                Size   Used  Avail Capacity iused      ifree %iused
/dev/disk1s6                             466Gi   10Gi  221Gi     5%  488316 4881964564    0%
devfs                                    231Ki  231Ki    0Bi   100%     800          0  100%
/dev/disk1s1                             466Gi  208Gi  221Gi    49% 1063805 4881389075    0%
/dev/disk1s4                             466Gi   15Gi  221Gi     7%      15 4882452865    0%
/dev/disk1s5                             466Gi   10Gi  221Gi     5%  487048 4881965832    0%
map auto_home                              0Bi    0Bi    0Bi   100%       0          0  100%
ome
Box                                      466Gi  208Gi  221Gi    49% 1063805  586421779    0%
/Users/mahmutunan/Documents/Atom.app     466Gi  199Gi  243Gi    45%  971867 4881481013    0%
k/l02njqgs56v17md44tfknd7c0000gn/T/AppTranslocation/71F900AC-7A89-4E16-BC6F-66BBF16E283E
/dev/disk1s3                             466Gi  1.0Gi  221Gi     1%      94 4882452786    0%
/dev/disk3s1                             309Mi  229Mi   80Mi    75%    1433 4294965846    0%
MacBook-Pro:shellScriptLecture mahmutunan$ _
```

I/O Redirection

- Regular UNIX system commands;
  - take input from terminal (stdin)
  - writes output to terminal (stdout)
- Output redirection
  - Output to a file
  - `> filename` notation will be used
  - `ls >> "newFolder/someNewFile.txt”`



- Input Redirection
  - < filename

```
Mail -s "Subject" to-address < Filename
```

https://www.guru99.com/linux-redirection.html

Man Page

```
$man command
$man cat
```

```
CAT(1)                    BSD General Commands Manual                    CAT(1)

NAME
     cat -- concatenate and print files

SYNOPSIS
     cat [-benstuv] [file ...]

DESCRIPTION
     The cat utility reads files sequentially, writing them to the standard output.  The file operands
     are processed in command-line order.  If file is a single dash (`-') or absent, cat reads from
     the standard input.  If file is a UNIX domain socket, cat connects to it and then reads it until
     EOF.  This complements the UNIX domain binding capability available in inetd(8).

     The options are as follows:

     -b      Number the non-blank output lines, starting at 1.

     -e      Display non-printing characters (see the -v option), and display a dollar sign (`$') at
             the end of each line.

     -n      Number the output lines, starting at 1.

     -s      Squeeze multiple adjacent empty lines, causing the output to be single spaced.

     -t      Display non-printing characters (see the -v option), and display tab characters as `^I'.

     -u      Disable output buffering.

     -v      Display non-printing characters so they are visible.  Control characters print as `^X'
             for control-X; the delete character (octal 0177) prints as `^?'.  Non-ASCII characters
```

UNIX Files System

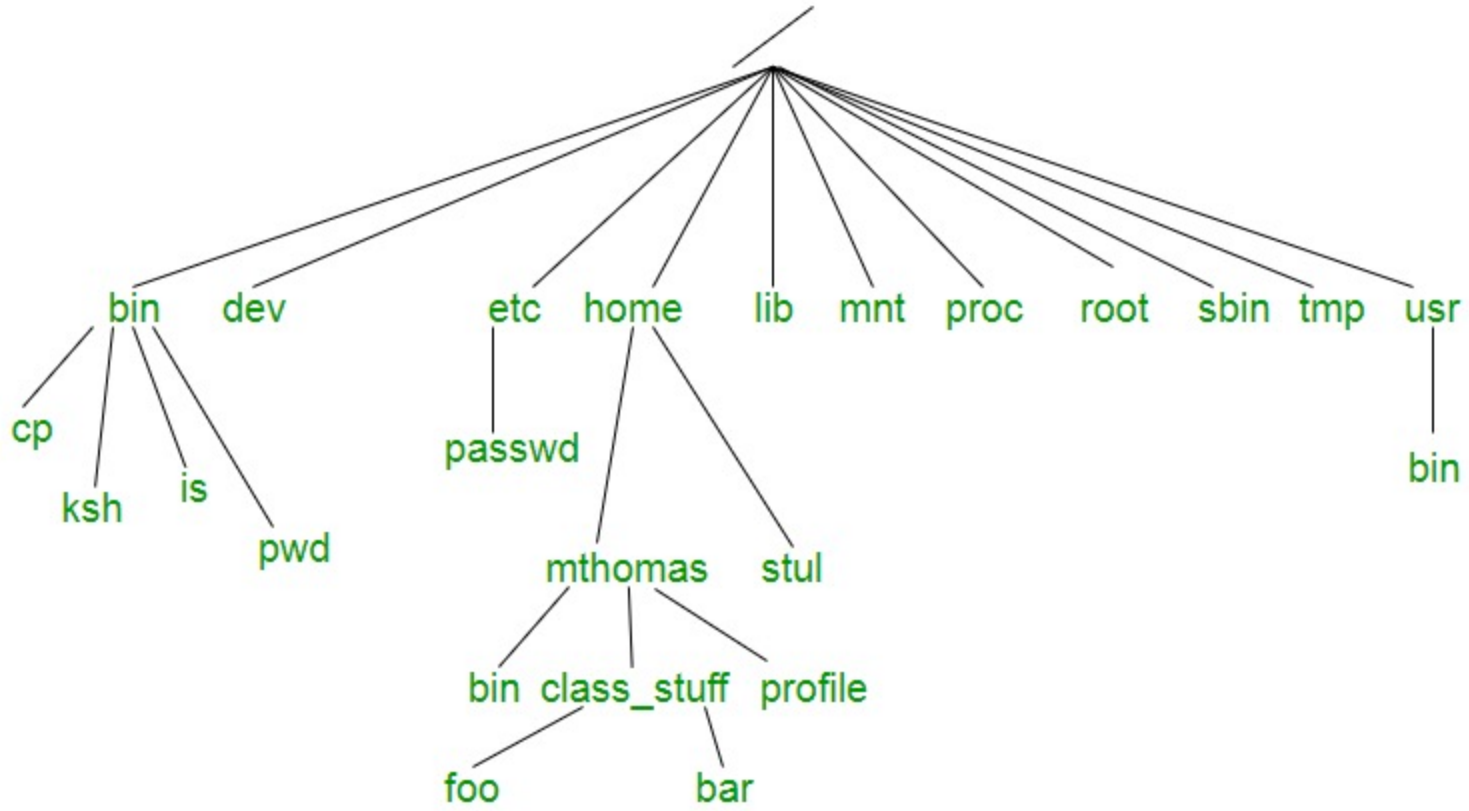
- All data organized into files
- All files organized into directories
- Directories
  - Tree-like structures
  - Multi-level hierarchy (directory tree)
  - Top level - root
    - /
    - All other directories are the children of the root
- File = sequence of bytes

Windows vs Unix

- Windows files are stored in folders on different data drives
  - C: D: E: ….
- Unix files are ordered in a tree structure
  - root and the children
- Peripherals such as hard drives, CD-Rom, printers, scanners
  - Windows consider them as devices
  - Unix consider them as files
- Naming convention
  - Windows → UAB and uab are the same, can't be under the same folder
  - Unix → they are different; UAB, uab, Uab, uAb…

Unix Directory Structure

- A file system consists of files, relationships to other files, as well as the attributes of each file
- root contains other files and directories
- each file or directory
  - uniquely identified by;
    - name
    - the directory that contains the file/directory
    - unique identifier (inode)
  - includes;
    - file type, size, owner, protection/privacy. time stamp
- each file is self contained



picture: https://www.geeksforgeeks.org/unix-file-system/

Listing the names of all files

```
#include "apue.h"
#include <dirent.h>

int
main(int argc, char *argv[])
{
    DIR             *dp;
    struct dirent   *dirp;

    if (argc != 2)
        err_quit("usage: ls directory_name");

    if ((dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ((dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Figure 1.3 List all the files in a directory

File Types

- Directory Files
- Ordinary Files
  - text
  - data
  - program instruction….
  - …
- Special Files
  - devices
  - shortcuts
  - …

Unix File Descriptors

- It is a non-negative integer number that uniquely identifies an open file in Unix.
- It describes a data resource, and how that resource may be accessed.
- The first three file descriptors;
  - STDIN (standard input). 0
  - STDOUT (standard output). 1
  - STDERR (standard error). 2

File Permissions

- use `ls -l` command to display the permissions
  - r w x - → read write execute no permission
- Owners permission -> First three characters
- Group permission -> next three characters
- World (other) Permission -> last three characters
- For example;

  -rwxrw-r--

  drwxr-xr--

Change Permission

- Use the `chmod` command to set permissions (read, write, execute) on a file/directory for the owner, group and the world

Number	Permission Type	Symbol
0	No Permission	---
1	Execute	--x
2	Write	-w-
3	Execute + Write	-wx
4	Read	r--
5	Read + Execute	r-x
6	Read +Write	rw-
7	Read + Write +Execute	rwx

https://www.guru99.com/file-permissions.html

UNIX file I/O Functionalities

- Processes needs system calls to handle file operation
- Opening a file
  - open or openat functions

```
fd = open(path,flag,mode)
```

https://man7.org/linux/man-pages/man2/open.2.html

flag

O_RDONLY

O_WRONLY

O_RDWR

O_EXEC

O_APPEND

O_CREAT

……

……

mode

- The third argument → *mode* specifies the permissions to use in case a new file is created.

The following symbolic constants are provided for *mode*:

S_IRWXU 00700 user (file owner) has read, write, and execute permission

S_IRUSR 00400 user has read permission

S_IWUSR 00200 user has write permission

S_IXUSR 00100 user has execute permission

S_IRWXG 00070 group has read, write, and execute permission

S_IRGRP 00040 group has read permission

S_IWGRP 00020 group has write permission

S_IXGRP 00010 group has execute permission

S_IRWXO 00007 others have read, write, and execute permission

S_IROTH 00004 others have read permission

S_IWOTH 00002 others have write permission

S_IXOTH 00001 others have execute permission

According to POSIX, the effect when other bits are set in *mode* is unspecified. On Linux, the following bits are also honored in *mode*:

S_ISUID 0004000 set-user-ID bit

S_ISGID 0002000 set-group-ID bit (see inode(7)).

S_ISVTX 0001000 sticky bit (see inode(7)).

https://man7.org/linux/man-pages/man2/open.2.htm

close

- system call
- the file descriptor is returned to the pool of available descriptors

```
int close(int fd);
```

- `close()` returns zero on success. On error, -1 is returned, and errno is set appropriately.

read()

- **ssize_t read(int** *fd*, **void** **buf*, **size_t** *count*);

- **read**() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

- On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

https://man7.org/linux/man-pages/man2/read.2.html

write()

- **ssize_t write(int** *fd*, **const void** **buf*, **size_t** *count*);

- **write**() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*.

- On success, the number of bytes written is returned. On error, -1 is returned, and *errno* is set to indicate the cause of the error.

https://www.man7.org/linux/man-pages/man2/write.2.html

Exercise

- C code to copy one file and copy the contents of that file to a new file

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

#define BUFFSIZE 4096

int main(int argc, char *argv[]) {
    int  readFileDescriptor, writeFileDescriptor;
    long int n;
    char buf[BUFFSIZE];
```

- Check if the correct numbers of the argument are given. There should be three arguments: name of the program, input file name, and output file name. If the number of arguments is not three then the program should print an error message and terminate. Also, input and output file names should not be the same.

```
12
13      if (argc != 3){
14          printf("Usage: %s <source_filename> <destination_filename>\n", argv[0]);
15          exit (-1);
16      }
17
```

- Use the *open* function in *read only mode* to read the input file.
- The open function takes the name of the file as the first argument and the open flag as the second argument.
- The open flag specifies if the file should be opened in read only mode (O_RDONLY), write only mode (O_WRONLY), or read-write mode (O_RDWR).
- There is an optional third argument that specifies the file permissions called *mode*, we will not use the optional third argument here. The third argument specifies the file permissions of the new file created for writing. Note that the UNIX file uses {*read, write, execute*} (*rwx)* permissions for the user, group, and everyone

```
18      readFileDescriptor = open(argv[1], O_RDONLY);
```

```
18      readFileDescriptor = open(argv[1], O_RDONLY);
```

- The function returns a file descriptor which is typically a non-negative integer.
- If there is an error opening the file, the function returns -1.
- Note that most programs have access to three standard file descriptors: standard input (stdin) - 0, standard output (stdout) - 1, and standard error (stderr) - 2.
- Instead of using the values 0, 1, and 2 we can also use the POSIX name STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO, respectively.

- Use the open function to create a new write for writing the output such that if the file does not exist it will create a new file and if a file with the given name exists it will overwrite the existing file.
- This is accomplished by ORing the different open flags: O_CREAT, O_WRONLY, and O_TRUNC. O_CREAT specifies to open a new empty file if the file does not exist and requires the third file permission argument to be provided.
- O_WRONLY specifies that the file should be open for writing only and the O_TRUNC flag specifies that if the file already exists, then it must be truncated to zero length (i.e., destroy any previous data).

```
19      writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
```

- Check the file descriptor to see if there is a problem or not

```
18      readFileDescriptor = open(argv[1], O_RDONLY);
19      writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
20
21      if (readFileDescriptor == -1 || writeFileDescriptor == -1){
22          printf("Error with file open\n");
23          exit (-1);
24      }
```

- Now read the file by reading fixed chunks of data using the *read* function by providing the file descriptor, input buffer address, and the maximum size of the buffer provided .
- A successful read returns the number of bytes read, 0 if the end-of-file is reached, or -1 if there is an error.
- After you read the data in to the buffer write the buffer to the new file using the *write* function.
- The *write* function takes the file descriptor, buffer to write, and the number of bytes to write. If the write is successful, it will return the number of bytes actually written.

```
26      while ((n = read(readFileDescriptor, buf, BUFFSIZE)) > 0){
27          if (write(writeFileDescriptor, buf, n) != n){
28              printf("Error writing to output file\n");
29              exit (-1);
30          }
31      }
```

- check the error condition

```
32      if (n < 0){
33          printf("Error reading input file\n");
34          exit (-1);
35      }
36
```

- After completing the copy process use the *close* function to close both file descriptors. The close function takes the file descriptor as the argument and returns 0 on success and -1 in case of an error.

```
38      close(readFileDescriptor);
39      close(writeFileDescriptor);
40      return 0;
41  }
```

Exercise

- C code to copy one file and copy the contents of that file to a new file

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

#define BUFFSIZE 4096

int main(int argc, char *argv[]) {
    int  readFileDescriptor, writeFileDescriptor;
    long int n;
    char buf[BUFFSIZE];
```

- Check if the correct numbers of the argument are given. There should be three arguments: name of the program, input file name, and output file name. If the number of arguments is not three then the program should print an error message and terminate. Also, input and output file names should not be the same.

```
12
13      if (argc != 3){
14          printf("Usage: %s <source_filename> <destination_filename>\n", argv[0]);
15          exit (-1);
16      }
17
```

- Use the *open* function in *read only mode* to read the input file.
- The open function takes the name of the file as the first argument and the open flag as the second argument.
- The open flag specifies if the file should be opened in read only mode (O_RDONLY), write only mode (O_WRONLY), or read-write mode (O_RDWR).
- There is an optional third argument that specifies the file permissions called *mode*, we will not use the optional third argument here. The third argument specifies the file permissions of the new file created for writing. Note that the UNIX file uses {*read, write, execute*} (*rwx)* permissions for the user, group, and everyone

```
18      readFileDescriptor = open(argv[1], O_RDONLY);
```

```
18      readFileDescriptor = open(argv[1], O_RDONLY);
```

- The function returns a file descriptor which is typically a non-negative integer.
- If there is an error opening the file, the function returns -1.
- Note that most programs have access to three standard file descriptors: standard input (stdin) - 0, standard output (stdout) - 1, and standard error (stderr) - 2.
- Instead of using the values 0, 1, and 2 we can also use the POSIX name STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO, respectively.

- Use the open function to create a new write for writing the output such that if the file does not exist it will create a new file and if a file with the given name exists it will overwrite the existing file.
- This is accomplished by ORing the different open flags: O_CREAT, O_WRONLY, and O_TRUNC. O_CREAT specifies to open a new empty file if the file does not exist and requires the third file permission argument to be provided.
- O_WRONLY specifies that the file should be open for writing only and the O_TRUNC flag specifies that if the file already exists, then it must be truncated to zero length (i.e., destroy any previous data).

```
19      writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
```

- Check the file descriptor to see if there is a problem or not

```
18      readFileDescriptor = open(argv[1], O_RDONLY);
19      writeFileDescriptor = open(argv[2], O_CREAT|O_WRONLY|O_TRUNC, 0700);
20
21      if (readFileDescriptor == -1 || writeFileDescriptor == -1){
22          printf("Error with file open\n");
23          exit (-1);
24      }
```

- Now read the file by reading fixed chunks of data using the *read* function by providing the file descriptor, input buffer address, and the maximum size of the buffer provided .
- A successful read returns the number of bytes read, 0 if the end-of-file is reached, or -1 if there is an error.
- After you read the data in to the buffer write the buffer to the new file using the *write* function.
- The *write* function takes the file descriptor, buffer to write, and the number of bytes to write. If the write is successful, it will return the number of bytes actually written.

```
26      while ((n = read(readFileDescriptor, buf, BUFFSIZE)) > 0){
27          if (write(writeFileDescriptor, buf, n) != n){
28              printf("Error writing to output file\n");
29              exit (-1);
30          }
31      }
```

- check the error condition

```
32      if (n < 0){
33          printf("Error reading input file\n");
34          exit (-1);
35      }
36
```

- After completing the copy process use the *close* function to close both file descriptors. The close function takes the file descriptor as the argument and returns 0 on success and -1 in case of an error.

```
38      close(readFileDescriptor);
39      close(writeFileDescriptor);
40      return 0;
41  }
```

Run the example

gcc filecopy.c -o exercise1

./exercise1 smallTale.txt outputFile.txt

```
MacBook-Pro:Desktop mahmutunan$ gcc filecopy.c -o exercise1
MacBook-Pro:Desktop mahmutunan$ ./exercise1 smallTale.txt outputFile.txt
MacBook-Pro:Desktop mahmutunan$
```

lseek()

off_t lseek(int fd, off_t offset, int whence);

- repositions the file offset of the open file description associated with the file descriptor *fd* to the argument ***offset*** according to the directive ***whence*** as follows:

https://man7.org/linux/man-pages/man2/lseek.2.html

- **SEEK_SET** The file offset is set to *offset* bytes.
- **SEEK_CUR** The file offset is set to its current location plus *offset* bytes.
- **SEEK_END** The file offset is set to the size of the file plus *offset* bytes.

Example 2

- Now, we will use *lseek* function to move to particular location in the file and modify it by performing following steps;

1. You can use the output file of Example #1.

Or, you can use any txt file.

I will create a new txt file and put some text

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

#define BUFFSIZE 4096
#define SEEKSIZE -10

int main(int argc, char *argv[]) {
    int  RWFileDescriptor;
    long int n;
    char buf[BUFFSIZE];
    const char lseekMSG[] = "THIS IS NEW MSG FROM LSEEK!\0";

    if (argc != 2){
        printf("Usage: %s <filename>\n", argv[0]);
        exit (-1);
    }

    RWFileDescriptor = open(argv[1], O_RDONLY);

    if (RWFileDescriptor == -1){
        printf("Error with file open\n");
        exit (-1);
    }

```

- 2. Use lseek to read last 10 bytes of file and print it on console.
  - The *lseek* function takes three arguments: file descriptor, the file offset, and the base address from which the offset is to be implemented (often referred to as *whence*).
  - In this example, we are trying to read the last 10 bytes in the file, so we set the whence to the end of the file using SEEK_END and specify the offset as 10.
  - You can look at the man page for lseek to find out other predefined whence values: SEEK_SET, SEEK_CUR, etc.
  - The lseek function returns the new file offset on a successful and returns -1 when there is an error.

```
28      if (lseek(RWFileDescriptor, SEEKSIZE, SEEK_END) >= 0){
29          if((n = read(RWFileDescriptor, buf, BUFFSIZE)) > 0){
30              if (write(STDOUT_FILENO, buf, n) != n) {
31                  printf("Error writing to file\n");
32                  exit (-1);
33              }
34          } else {
35              printf("Error reading file\n");
36              exit (-1);
37          }
38      } else {
39          printf("lseek error (Part 1)\n");
40          exit (-1);
41      }
42      close(RWFileDescriptor);
```

- 3. Use lseek to write a string character “THIS IS NEW MSG FROM LSEEK!” at the beginning of the file.

```
43
44      RWFileDescriptor = open(argv[1], O_WRONLY);
45      if (lseek(RWFileDescriptor, 0, SEEK_SET) >= 0){
46          if (write(RWFileDescriptor, lseekMSG, strlen(lseekMSG)) != strlen(lseekMSG)) {
47              printf("Error writing to file\n");
48          }
49      } else {
50          printf("lseek error (Part 2)\n");
51      }
52
53      close(RWFileDescriptor);
54
55      return 0;
56  }
57
```

Run the exercise 2

```
(base) mahmutunan@MacBook-Pro Desktop % gcc filelseek.c -o exercise2
(base) mahmutunan@MacBook-Pro Desktop % cat testfile.txt
THIS IS NEW MSG FROM LSEEK!s a test message. Do you see it?%
```

```
(base) mahmutunan@MacBook-Pro Desktop % ./exercise2 testfile.txt
```

```
(base) mahmutunan@MacBook-Pro Desktop % cat testfile.txt
THIS IS NEW MSG FROM LSEEK!s a test message. Do you see it?
```

Lab 4 - exercise

- Check out what values are printed if you change the *offset* and *whence* to the following values when you are using the *lseek* function with the *readFileDescriptor*:

offset	**whence**
0	SEEK_SET
0	SEEK_END
-1	SEEK_END
-10	SEEK_CUR

Exercise 3

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define BUF_SIZE 1024

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <filename>\n", argv[0]);
        exit(-1);
    }
    char *file_Name = argv[1];

    int writeFd;
```

```
15
16      fprintf(stdout, "Opening file :%s\n", file_Name);
17      writeFd = open(file_Name, O_RDWR, O_NONBLOCK, O_APPEND);
18
19      if (writeFd < 0) {
20          printf("Error with file open\n");
21          exit(-1);
22      }
23
24      fprintf(stdout, "Seeking the beginning of the file \n");
```

```
25
26      if (lseek(writeFd, 0, SEEK_SET) >= 0) {
27          fprintf(stdout, "Writing the message into %s\n", file_Name);
28          char buffer[BUF_SIZE] = "Message from Exercise 3\n";
29          write(writeFd, buffer, BUF_SIZE);
30          close(writeFd);
31
32          return 0;
33      }
34  }
```

output

```
(base) mahmutunan@MacBook-Pro Desktop % gcc exercise3.c -o exercise3
(base) mahmutunan@MacBook-Pro Desktop % ./exercise3 output.txt
Opening file :output.txt
Seeking the beginning of the file
Writing the message into output.txt
(base) mahmutunan@MacBook-Pro Desktop %
```

Make Utility

- *make* is a utility that is used to automatically detect which program need to be recompiled while working on a large number of source programs and will recompile only those programs that have been modified.
- The *make* utility uses a *Makefile* to describe the rules for determining the dependencies between the various programs and the compiler and compiler options to use for compiling the programs.
- In case of C programs, an executable is created from object files (*.o files) and object files are created from source files.
- Source files are often divided into header files (*.h files) and actual source files (*.c files).

- The simplest way to organize the code compilation
- Make figures out automatically which files it needs to update, based on which source files have changed.

Exercise 1

```c
#include <stdio.h>

int main (int argc, char *argv[]){

char charArr[20]="Hello CS330";
printf("%s",charArr);
return 0;

}
```

```
myExecutable: main.c
  gcc -o myExecutable main.c

```

```
(base) mahmutunan@MacBook-Pro exercise1 % ls
Makefile        main.c
(base) mahmutunan@MacBook-Pro exercise1 % make
gcc -o myExecutable main.c
(base) mahmutunan@MacBook-Pro exercise1 % ./myExecutable
Hello CS330%
(base) mahmutunan@MacBook-Pro exercise1 %
```

Exercise 2

```c
#include "someFunc.h"

int main (int argc, char *argv[]){

  printHellofunct();
  return 0;

}
```

- main.c

```c
#include <stdio.h>
#include "someFunc.h"

void printHellofunct(){

  printf("Hello CS330");

}
```

- someFunc.c

```c
void printHellofunct();

```

- someFunc.h

Exercise 2

- we can compile multiple file using command line

```
(base) mahmutunan@MacBook-Pro exercise2 % ls
main.c          someFunc.c      someFunc.h
(base) mahmutunan@MacBook-Pro exercise2 % gcc -o exercise2 main.c someFunc.c -I
(base) mahmutunan@MacBook-Pro exercise2 % ./exercise2
Hello CS330%
(base) mahmutunan@MacBook-Pro exercise2 %
```

- The -I. is included so that gcc will look in the current directory (.) for the include file someFunc.h

Exercise 2

- We can use the make file to automate the compilation process

Makefile

```
1  exercise2withMake: main.c someFunc.c
2    gcc -o exercise2withMake main.c someFunc.c -I
3
```

```
(base) mahmutunan@MacBook-Pro exercise2 % ls
Makefile        exercise2       main.c          someFunc.c      someFunc.h
(base) mahmutunan@MacBook-Pro exercise2 % make
gcc -o exercise2withMake main.c someFunc.c -I
(base) mahmutunan@MacBook-Pro exercise2 % ./exercise2withMake
Hello CS330%
(base) mahmutunan@MacBook-Pro exercise2 %
```

Exercise 2

- Let's modify the make file a little bit

Makefile

```
1  CC=gcc
2  CFLAGS=-I
3  DEPS=someFunc.h
4  OBJ=main.o someFunc.o
5
6  %.o: %.c $(DEPS)
7    $(CC) -c -o $@ $< $(CFLAGS)
8
9  exercise2withMake: $(OBJ)
10   $(CC) -o $@ $^ $(CFLAGS)
```

```
(base) mahmutunan@MacBook-Pro exercise2 % ls
Makefile        main.c          someFunc.c      someFunc.h
(base) mahmutunan@MacBook-Pro exercise2 % make
gcc -c -o main.o main.c -I
gcc -c -o someFunc.o someFunc.c -I
gcc -o exercise2withMake main.o someFunc.o -I
(base) mahmutunan@MacBook-Pro exercise2 % ls
Makefile                main.o                  someFunc.o
exercise2withMake       someFunc.c
main.c                  someFunc.h
(base) mahmutunan@MacBook-Pro exercise2 % ./exercise2withMake
Hello CS330%
(base) mahmutunan@MacBook-Pro exercise2 %
```

Exercise - Lab04

- To illustrate the use of make, let us consider adding a new function to measure the time taken by the insertion sort program that we wrote in Lab 2.
- Instead of adding this method to the same file as the insertion sort, let us create a new file and create a header file that has the method prototype.

insertionsort.c

gettime.c | insertionsort.c

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

/* main method */
int main(int args, char** argv){
  int N, i;
  printf("Please enter number of elements in array: ");
  scanf("%d", &N);

  float arr[N];

  for (i=0; i<N; i++){
    printf("Please enter element %d of array: ", (i+1));
    scanf("%f", &arr[i]);
  }

  printf("Given array is: ");
  printf("[");
  for (i=0; i < N-1; i++){
    printf("%f, ", arr[i]);
  }
  printf("%f]\n", arr[N-1]);

  float temp;
  int currLoc;
  for (i=1; i < N; i++){
    currLoc = i;
    while (currLoc > 0 && arr[currLoc-1] > arr[currLoc]){
      temp = arr[currLoc];
      arr[currLoc] = arr[currLoc-1];
      arr[currLoc-1] = temp;
      currLoc--;
    }
  }

  printf("Sorted array is: ");
  printf("[");
  for (i=0; i < N-1; i++){
    printf("%f, ", arr[i]);
  }
  printf("%f]\n", arr[N-1]);

  return 0;
}
```

gettime.h

gettime.h

```
1  #ifndef _GETTIME_H_
2  #include <stdio.h>
3  #include <sys/time.h>
4  double gettime(void);
5  #endif
```

gettime.c

gettime.c

```c
#include "gettime.h"

double gettime(void) {
    struct timeval tval;
    gettimeofday(&tval, NULL);
    return((double)tval.tv_sec + (double)tval.tv_usec/1000000.0);
}

```

- Note that we can compile the file `gettime.c` separately and link the object file with any other program that uses the ***gettime*** function.
- To use the gettime function in the insertion sort program, we have to include the file `gettime.h` and invoke the ***gettime*** function before and after the call to ***insertionsort*** function

- Here are the steps involved in incrementally compiling and linking these two different files:

```
(base) mahmutunan@MacBook-Pro Desktop % cd lecture11
(base) mahmutunan@MacBook-Pro lecture11 % ls
gettime.c       gettime.h       insertionsort.c
(base) mahmutunan@MacBook-Pro lecture11 % gcc -c gettime.c
(base) mahmutunan@MacBook-Pro lecture11 % gcc -c insertionsort.c
(base) mahmutunan@MacBook-Pro lecture11 % gcc -o insertionsort insertionsort.o gettime.o
(base) mahmutunan@MacBook-Pro lecture11 % ./insertionsort
Please enter number of elements in array: 7
Please enter element 1 of array: 21
Please enter element 2 of array: 22
Please enter element 3 of array: 44
Please enter element 4 of array: 55
Please enter element 5 of array: 32
Please enter element 6 of array: 56
Please enter element 7 of array: 2
Given array is: [21.000000, 22.000000, 44.000000, 55.000000, 32.000000, 56.000000, 2.0000
00]
Sorted array is: [2.000000, 21.000000, 22.000000, 32.000000, 44.000000, 55.000000, 56.000
000]
(base) mahmutunan@MacBook-Pro lecture11 % _
```

`gcc -c` compiles source files without linking

```
$ gcc -c myfile.c
```

This compilation generates *myfile.o* object file.

- Also note that we don't have to recompile `gettime.c` if we are only making changes to the file `insertionsort.c`.
- These dependencies is what we can describe in a make file and let the make utility determine which files what been updated and recompile those files.

makefile

Makefile

```
# Sample Makefile to compile C programs
CC = gcc
CFLAGS = -Wall -g #replace -g with -O when not debugging
DEPS     = gettime.h Makefile
OBJS     = gettime.o insertionsort.o
EXECS    = insertionsort

all:     $(EXECS)

%.o:     %.c $(DEPS)
         $(CC) $(CFLAGS) -c -o $@ $<

insertionsort: $(OBJS)
         $(CC) $(CFLAGS) -o $@ $^

clean:
         /bin/rm -i *.o $(EXECS)


```

- If the Makefile is saved as Makefile or makefile, you can invoke make utility by typing make.
- If you use a different file name other than Makefile or makefile then you have to specify the makefile using the -f option to make. If you type make, you should see the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % make
gcc -Wall -g  -c -o gettime.o gettime.c
gcc -Wall -g  -c -o insertionsort.o insertionsort.c
gcc -Wall -g  -o insertionsort gettime.o insertionsort.o
```

- If you change gettime.h then you should see all files recompiled and the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % touch gettime.h
(base) mahmutunan@MacBook-Pro lecture11 % make
gcc -Wall -g  -c -o gettime.o gettime.c
gcc -Wall -g  -c -o insertionsort.o insertionsort.c
gcc -Wall -g  -o insertionsort gettime.o insertionsort.o
```

If you change gettime.c then you should see the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % touch gettime.c
(base) mahmutunan@MacBook-Pro lecture11 % make
gcc -Wall -g  -c -o gettime.o gettime.c
gcc -Wall -g  -o insertionsort gettime.o insertionsort.o
```

- However, if you only change insertionsort.c you will see the following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % make
gcc -Wall -g  -c -o insertionsort.o insertionsort.c
gcc -Wall -g  -o insertionsort gettime.o insertionsort.o
```

- If you have not modified any files, if you execute make, you will see that following output:

```
(base) mahmutunan@MacBook-Pro lecture11 % make
make: Nothing to be done for `all'.
```

stat lstat

- We'll start with the *stat* and *lstat* functions.
- Both function return a structure called *stat*, and members of stat structure provide information about the file or directory which was provided as the argument to these functions.

stat lstat

- stat, fstat, lstat, fstatat - get file status

```
int stat(const char *pathname, struct stat
*statbuf);

int fstat(int fd, struct stat *statbuf);

int lstat(const char *pathname, struct stat
*statbuf);
```

https://man7.org/linux/man-pages/man2/stat.2.html

- `stat()` and `fstatat()` retrieve information about the file pointed to by *pathname*

- `lstat()` is identical to `stat()`, except that if *pathname* is a symbolic link, then it returns information about the link itself, not the file that the link refers to.

- `fstat()` is identical to `stat()`, except that the file about which information is to be retrieved is specified by the file descriptor *fd*.

https://man7.org/linux/man-pages/man2/stat.2.html

The `stat` structure

- All of these system calls return a *stat structure*, which contains the following fields:

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

https://man7.org/linux/man-pages/man2/stat.2.html

```
(base) mahmutunan@MacBook-Pro lecture12 % touch someNewFile.txt
(base) mahmutunan@MacBook-Pro lecture12 % echo "some text into file" > someNewFile.txt
(base) mahmutunan@MacBook-Pro lecture12 % cat someNewFile.txt
some text into file
(base) mahmutunan@MacBook-Pro lecture12 % stat -x someNewFile.txt
  File: "someNewFile.txt"
  Size: 20           FileType: Regular File
  Mode: (0644/-rw-r--r--)         Uid: (  501/mahmutunan)  Gid: (   20/   staff)
Device: 1,4   Inode: 10604554    Links: 1
Access: Mon Sep 21 11:44:20 2020
Modify: Mon Sep 21 11:44:18 2020
Change: Mon Sep 21 11:44:18 2020
(base) mahmutunan@MacBook-Pro lecture12 % _
```

File Types

- Regular File
- Directory File
- Block Special File
- Character Special File
- FIFO
- Socket
- Symbolic links

The type of the file

Macro	Type of file
`S_ISREG()`	regular file
`S_ISDIR()`	directory file
`S_ISCHR()`	character special file
`S_ISBLK()`	block special file
`S_ISFIFO()`	pipe or FIFO
`S_ISLNK()`	symbolic link
`S_ISSOCK()`	socket

Figure 4.1 File type macros in `<sys/stat.h>`

Exercise 1 - printstat.c

```c
/* function to print stat data structure */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>

void printstat(struct stat sb) {
  /* copied from the lstat man page example as is */
  printf("File type:                ");

  switch (sb.st_mode & S_IFMT) {
  case S_IFBLK:  printf("block device\n");            break;
  case S_IFCHR:  printf("character device\n");        break;
  case S_IFDIR:  printf("directory\n");               break;
  case S_IFIFO:  printf("FIFO/pipe\n");               break;
  case S_IFLNK:  printf("symlink\n");                 break;
  case S_IFREG:  printf("regular file\n");            break;
  case S_IFSOCK: printf("socket\n");                  break;
  default:       printf("unknown?\n");                break;
  }
```

Exercise 1 - printstat.c /2

```
23
24      printf("I-node number:            %ld\n", (long) sb.st_ino);
25
26      printf("Mode:                     %lo (octal)\n",
27         (unsigned long) sb.st_mode);
28
29      printf("Link count:               %ld\n", (long) sb.st_nlink);
30      printf("Ownership:                UID=%ld   GID=%ld\n",
31         (long) sb.st_uid, (long) sb.st_gid);
32
33      printf("Preferred I/O block size: %ld bytes\n",
34         (long) sb.st_blksize);
35      printf("File size:                %lld bytes\n",
36         (long long) sb.st_size);
37      printf("Blocks allocated:         %lld\n",
38         (long long) sb.st_blocks);
39
40      printf("Last status change:       %s", ctime(&sb.st_ctime));
41      printf("Last file access:         %s", ctime(&sb.st_atime));
42      printf("Last file modification:   %s", ctime(&sb.st_mtime));
43  }
44
45
```

Exercise 1 - lstat.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

void printstat(struct stat statbuf);

int main(int argc, char **argv) {
    int i;
    struct stat buf;
    char *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
            printf("lstat error");
            continue;
        }
```

Exercise 1 - lstat.c /2

```
20          if (S_ISREG(buf.st_mode))
21              ptr = "regular";
22          else if (S_ISDIR(buf.st_mode))
23              ptr = "directory";
24          else if (S_ISCHR(buf.st_mode))
25              ptr = "character special";
26          else if (S_ISBLK(buf.st_mode))
27              ptr = "block special";
28          else if (S_ISFIFO(buf.st_mode))
29              ptr = "fifo";
30          else if (S_ISLNK(buf.st_mode))
31              ptr = "symbolic link";
32          else if (S_ISSOCK(buf.st_mode))
33              ptr = "socket";
34          else
35              ptr = "** unknown mode **";
36          printf("%s\n", ptr);
37
38      printstat(buf);
39      }
40      exit(0);
41  }
42
```

Example 1 -compile&run

```
(base) mahmutunan@MacBook-Pro lecture12 % gcc -o exercise1 printstat.c lstat.c
(base) mahmutunan@MacBook-Pro lecture12 % ln -s /Users/mahmutunan/Desktop/ABEt_links.txt aSymbolicLink
(base) mahmutunan@MacBook-Pro lecture12 % mkdir newFolder
(base) mahmutunan@MacBook-Pro lecture12 % ls
aSymbolicLink   exercise1       lstat.c         newFolder       printstat.c     someNewFile.txt
(base) mahmutunan@MacBook-Pro lecture12 % ./exercise1 aSymbolicLink someNewFile.txt newFolder
```

```
aSymbolicLink: symbolic link
File type:                symlink
I-node number:            10610993
Mode:                     120755 (octal)
Link count:               1
Ownership:                UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:                40 bytes
Blocks allocated:         0
Last status change:       Mon Sep 21 12:38:35 2020
Last file access:         Mon Sep 21 12:38:35 2020
Last file modification:   Mon Sep 21 12:38:35 2020
```

Example 1 -compile&run /2

```
someNewFile.txt: regular
File type:                regular file
I-node number:            10604554
Mode:                     100644 (octal)
Link count:               1
Ownership:                UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:                20 bytes
Blocks allocated:         8
Last status change:       Mon Sep 21 12:29:58 2020
Last file access:         Mon Sep 21 12:29:58 2020
Last file modification:   Mon Sep 21 11:44:18 2020
```

```
newFolder: directory
File type:                directory
I-node number:            10611008
Mode:                     40755 (octal)
Link count:               2
Ownership:                UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:                64 bytes
Blocks allocated:         0
Last status change:       Mon Sep 21 12:38:39 2020
Last file access:         Mon Sep 21 12:38:39 2020
Last file modification:   Mon Sep 21 12:38:39 2020
(base) mahmutunan@MacBook-Pro lecture12 % _
```

Open & Read the directories

- Till now we talked about how filesystem store files and directories information and access details. Now it’s time to learn how to open and read the directories and traverse the file system. To achieve this task, you need to learn about three functions:
- *opendir* - this function will allow us to open a directory with the given path
- *readdir* - this function will read what’s inside the directory
- *closedir* - this will close the open directory

opendir

- opendir, fdopendir - open a directory

```
DIR *opendir(const char *name);
DIR *fdopendir(int fd);
```

The `opendir()` function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

The `fdopendir()` function is like opendir(), but returns a directory stream for the directory referred to by the open file descriptor *fd*. After a successful call to fdopendir(), *fd* is used internally by the implementation, and should not otherwise be used by the application.

https://man7.org/linux/man-pages/man3/opendir.3.html

readdir

- readdir - read a directory

```
struct dirent *readdir(DIR *dirp);
```

The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dirp.

On success, readdir() returns a pointer to a dirent structure. (This structure may be statically allocated; do not attempt to free(3) it.)

https://www.man7.org/linux/man-pages/man3/readdir.3.html

closedir

- closedir — close a directory stream

```
int closedir(DIR *dirp);
```

- The closedir() function shall close the directory stream referred to by the argument dirp. Upon return, the value of dirp may no longer point to an accessible object of the type DIR. If a file descriptor is used to implement type DIR, that file descriptor shall be closed.

Exercise 2 - readdir.c

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

int main (int argc, char **argv) {
  struct dirent *dirent;
  DIR *parentDir;

  if (argc < 2) {
    printf ("Usage: %s <dirname>\n", argv[0]);
    exit(-1);
  }
  parentDir = opendir (argv[1]);
  if (parentDir == NULL) {
    printf ("Error opening directory '%s'\n", argv[1]);
    exit (-1);
  }
  int count = 1;
  while((dirent = readdir(parentDir)) != NULL){
    printf ("[%d] %s\n", count, (*dirent).d_name);
    count++;
  }
  closedir (parentDir);
  return 0;
}
```

Exercise 2 - compile & run

```
(base) mahmutunan@MacBook-Pro lecture12 % gcc -o exercise2 readdir.c
(base) mahmutunan@MacBook-Pro lecture12 % ./exercise2
Usage: ./exercise2 <dirname>
(base) mahmutunan@MacBook-Pro lecture12 % ./exercise2 ./
 [1] .
 [2] ..
 [3] someNewFile.txt
 [4] .DS_Store
 [5] exercise2
 [6] readdir.c
 [7] exercise1
 [8] aSymbolicLink
 [9] printstat.c
 [10] newFolder
 [11] lstat.c
(base) mahmutunan@MacBook-Pro lecture12 % ./exercise2 ./newFolder
 [1] .
 [2] ..
(base) mahmutunan@MacBook-Pro lecture12 %
```

Exercise 3 - readdir_v2.c

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

char *filetype(unsigned char type) {
  char *str;
  switch(type) {
  case DT_BLK: str = "block device"; break;
  case DT_CHR: str = "character device"; break;
  case DT_DIR: str = "directory"; break;
  case DT_FIFO: str = "named pipe (FIFO)"; break;
  case DT_LNK: str = "symbolic link"; break;
  case DT_REG: str = "regular file"; break;
  case DT_SOCK: str = "UNIX domain socket"; break;
  case DT_UNKNOWN: str = "unknown file type"; break;
  default: str = "UNKNOWN";
  }
  return str;
}
```

Exercise 3 - readdir_v2.c / 2

```c

int main (int argc, char **argv) {
  struct dirent *dirent;
  DIR *parentDir;

  if (argc < 2) {
    printf ("Usage: %s <dirname>\n", argv[0]);
    exit(-1);
  }
  parentDir = opendir (argv[1]);
  if (parentDir == NULL) {
    printf ("Error opening directory '%s'\n", argv[1]);
    exit (-1);
  }
  int count = 1;
  while((dirent = readdir(parentDir)) != NULL){
    printf ("[%d] %s (%s)\n", count, dirent->d_name, filetype(dirent->d_type));
    count++;
  }
  closedir (parentDir);
  return 0;
}
```

Exercise 3 - readdir_v2.c / compile&run

```
(base) mahmutunan@MacBook-Pro lecture13 % gcc -o exercise3 readdir_v2.c
(base) mahmutunan@MacBook-Pro lecture13 % ./exercise3 ./
 [1] . (directory)
 [2] .. (directory)
 [3] someNewFile.txt (regular file)
 [4] writestat.c (regular file)
 [5] .DS_Store (regular file)
 [6] exercise2 (regular file)
 [7] readstat.c (regular file)
 [8] exercise3 (regular file)
 [9] readdir.c (regular file)
 [10] exercise1 (regular file)
 [11] aSymbolicLink (symbolic link)
 [12] printstat.c (regular file)
 [13] newFolder (directory)
 [14] funcptr.c (regular file)
 [15] lstat.c (regular file)
 [16] readdir_v2.c (regular file)
(base) mahmutunan@MacBook-Pro lecture13 % _
```

Exercise 4

- We can use the read/write system calls from Lab-04 to write the stat structure and read the stat structure.

- Note that data is written as a binary file.

writestats.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

void printstat(struct stat statbuf);

int main(int argc, char** argv) {
    int fd;
    struct stat statbuf;

    if (lstat(argv[1], &statbuf) < 0) {
        printf("Error reading file/directory %s\n", argv[1]);
        perror("lstat");
        exit(-1);
    }
    printstat(statbuf);

    if ((fd = open(argv[2], O_CREAT | O_WRONLY, 0755)) == -1) {
        printf("Error opening file %s\n", argv[2]);
        perror("open");
        exit(-1);
    }
    write(fd, &statbuf, sizeof(struct stat));
    close(fd);

    return 0;
}
```

readstats.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
#include <fcntl.h>

void printstat(struct stat sb);

int main(int argc, char** argv) {
    int fd;
    struct stat statbuf;

    if ((fd = open(argv[1], O_RDONLY)) == -1) {
        printf("Error opening file %s\n", argv[1]);
        perror("open");
        exit(-1);
    }
    read(fd, &statbuf, sizeof(struct stat));
    close(fd);

    printstat(statbuf);

    return 0;
}
```

Exercise 4 - compile&run

```
(base) mahmutunan@MacBook-Pro lecture13 % gcc -Wall -o exercise4 writestat.c printstat.c
(base) mahmutunan@MacBook-Pro lecture13 % ./exercise4 lstat.c stat.out
File type:                regular file
I-node number:            10642103
Mode:                     100644 (octal)
Link count:               1
Ownership:                UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:                1041 bytes
Blocks allocated:         8
Last status change:       Mon Sep 21 19:40:17 2020
Last file access:         Mon Sep 21 19:40:19 2020
Last file modification:   Mon Sep 21 12:23:14 2020
(base) mahmutunan@MacBook-Pro lecture13 % gcc -Wall -o exercise4_read readstat.c printstat.c
(base) mahmutunan@MacBook-Pro lecture13 % ./exercise4_read stat.out
File type:                regular file
I-node number:            10642103
Mode:                     100644 (octal)
Link count:               1
Ownership:                UID=501   GID=20
Preferred I/O block size: 4096 bytes
File size:                1041 bytes
Blocks allocated:         8
Last status change:       Mon Sep 21 19:40:17 2020
Last file access:         Mon Sep 21 19:40:19 2020
Last file modification:   Mon Sep 21 12:23:14 2020
(base) mahmutunan@MacBook-Pro lecture13 % _
```

Hints for the HW2

- You can find a more elaborate example in Figure 4.22 in the textbook.
- This program takes as input a directory name, traverses a file hierarchy, counts the different types of files in the given file hierarchy, and the prints the summary (as shown in Figure 4.4).
- This program uses function pointers i.e., you can pass a function as an argument to a function similar to passing variables of different type.
- This enables us to perform different operations on a file as we traverse the file hierarchy.

Function Pointers - recall

- In the C environment, like normal data pointers, we can have pointers to functions

```c
void someFunction(int x)
{
    printf("Square of x is %d\n", (x*x));
}

int main()
{
    void (*functionPointer)(int) = someFunction;
    functionPointer(10);
    return 0;
}
```

Passing Pointers to functions

```c
#include <stdio.h>
void test(int a);
int main(void)
{
    void (*ptr)(int a);
    ptr = test;
    (*ptr)(10);
    return 0;
}
void test(int a)
{
    printf("%d\n", 2*a);
}
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr[1](i, j);
    printf("Result  : %d\n", result);
    return 0;
}
int addTwoNumbers(int a, int b)
{   return a+b;  }
int subtractTwoNumbers(int a, int b)
{   return a-b;   }
```

```c
#include <stdio.h>
int addTwoNumbers(int a, int b);
int subtractTwoNumbers(int a, int b);

int main(void)
{
    int (*ptr[2])(int a, int b);
    int i, j, result;
    ptr[0] = addTwoNumbers;
    ptr[1] = subtractTwoNumbers;

    printf("Enter two integer numbers: ");
    scanf("%d %d", &i, &j);
    if(i > 0 && i < 25)
        result = ptr[0](i, j);
    else
        result = ptr
    printf("Result
    return 0;
}
int addTwoNumbers(int
{   return a+b;  }
int subtractTwoNumber
{   return a-b;   }
```


```
Enter two integer numbers: 20 30
Result   : 50
```

```
Enter two integer numbers: 45 20
Result   : 25
```

Exercise 5

```c
/* Sample program to illustrate how to use function pointers */
#include <stdio.h>
typedef int MYFUNC(int a, int b);

int add(int a, int b) {
    printf("This is the add function\n");
    return a + b;
}

int sub(int a, int b) {
    printf("This is the subtraction function\n");
    return a - b;
}

int opfunc(int a, int b, MYFUNC *f) {
    return f(a, b);
}

int main(int argc, char *argv[]) {
    int a = 10, b = 5;
    printf("Passing add function....\n");
    printf("Result = %d\n", opfunc(a, b, add));
    printf("Passing sub function....\n");
    printf("Result = %d\n", opfunc(a, b, sub));
    return 0;
}
```

- In this example we define a function *opfunc* that takes as input a pointer to a function that takes two integer arguments and returns an integer value.
- We use *typedef* to define the function signature so that we can use this as a type in the function definition.
- Then we can have different functions with the given type signature and these functions can perform different operations. In this example, we define two functions to perform addition and subtraction on the two arguments passed to the function.
- Now we can invoke the *opfunc* by providing two integer values and the corresponding function to perform the required operation.

- Example 4.22 uses this mechanism through which we can define different functions to perform different operations on a given file as we traverse the file hierarchy.
- In this example, we just count the different files types, however, we could perform other operations such as check the file size or file permission or any other user-defined operation.