

CS 332/532 Systems Programming

Lecture 18
-REVIEW-

Professor : Mahmut Unan – UAB CS

Announcement

- Monday (02/28/2022) - Exam 1
 - Multiple Choice
 - True/False
 - Short Answer
- The exam will be closed book/closed note...etc.
- Leave one empty seat
- No restroom break

The ANSI Standard

- The rapid expansion of the C language
 - Many companies developed their own C compilers
- In 1983, the American National Standard Institute (ANSI) began the development of the C standard that was completed and formally approved, in 1989, as ANSI C or Standard C
 - C89
 - C90
 - C95
 - C99
 - C11
 - C18

Hello World !

```
1 #include <stdio.h>
2
3 ► ┌ int main() {
4     printf("Hello World!");
5     return 0;
6 ┌ }
```

#include

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

- 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

```
#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;
}
```

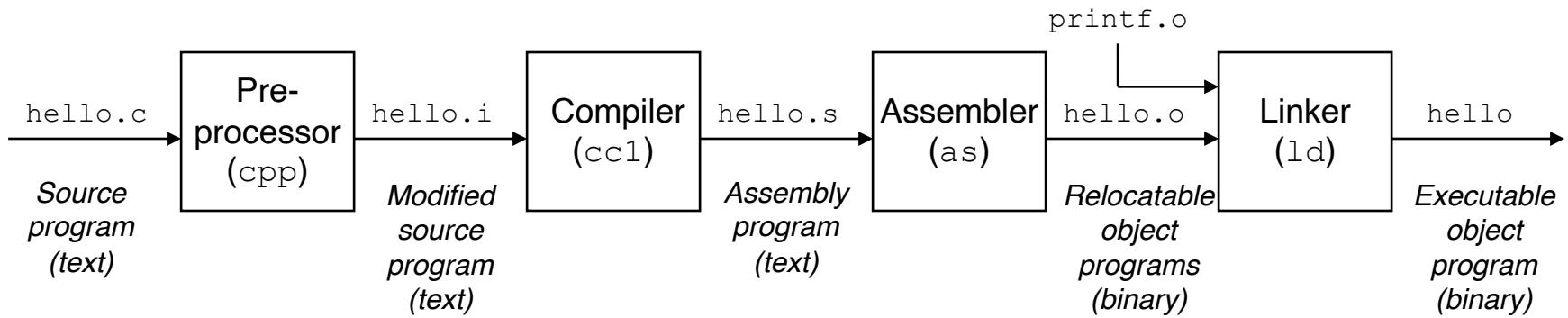
Nested comments are not allowed.

For example, the following code is illegal,
and the compiler will raise an error message:

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

Compilation

- unix> gcc -o hello hello.c



Compilation System

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 ?

```
1 #include <stdio.h>
2
3 ► int main() {
4
5     float a = 3.1;
6     if (a == 3.1)
7         printf("Yes\n");
8     else
9         printf("No\n");
10    printf("%.9f\n", a - 3.1);
11    return 0;
12 }
```

What is the output ?

```
1 #include <stdio.h>
2
3 ► □ int main() {
4
5     float a = 3.1;
6     if (a == 3.1)
7         printf("Yes\n");
8     else
9         printf("No\n");
10    printf("%.9f\n", a - 3.1);
11    return 0;
12 } 13
```

No
-0.000000095

```
10 ► int main () {  
11  
12     /* variable definition: */  
13     int a, b;  
14     int c;  
15     float f;  
16     char s[15] = "sample string";  
17     double d;  
18  
19     /* actual initialization */  
20     a = 10;  
21     b = 20;  
22     d=25.0;  
23  
24     c = a + b;  
25     printf("value of c : %d \n", c);  
26  
27     f = 70.0/3.0;  
28     printf("value of f : %f \n", f);  
29  
30     d= 50.0/d;  
31     printf("value of d : %f \n", d);  
32  
33     printf("value of s : %s \n", s);  
34  
35     int functionResult = somefunction();  
36     printf("value of function call : %d\n", functionResult);  
37     return 0;  
38 }
```

```
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);
```

#define

- The `#define` directive is used to define a macro. In most cases, a macro is a name that represents a numerical value. To define such a macro, we write:

```
#define macro_name value
```

For example

```
#define NUM 100
```

```
#include <stdio.h>
#define NUM 100
int main(void)
{
    int a, b, c;
    a = 20 - NUM;
    b = 20 + NUM;
    c = 3 * NUM;
    return 0;
}
```

printf()

- The `printf()` function is used to display data to `stdout` (*standard output 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.

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, 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 %.

```
#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

```
#include <stdio.h>
int main(void)
{
    float a = 1.2365;
    printf("%f\n", a);
    printf("%.2f\n", a);
    printf("%.3f\n", a);
    printf("%.1f\n", a);
    return 0;
}
```

1.236500	
1.24	
1.237	
1	

scanf ()

- The scanf() function is used to read data from stdin (*standard input 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);
```

```
1 #include <stdio.h>  
2 int main(void)  
3 {  
4     char ch;  
5     int i;  
6     float f;  
7     printf("Enter character, int and float: ");  
8     scanf("%c%d%f", &ch, &i, &f);  
9     printf("\nC:%c\tI:%d\tF:%f\n", ch, i, f);  
10    return 0;  
11 }
```

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

```
1 #include <stdio.h>
2
3 ► int main() {
4
5     int a=10,b=20,c=30,d=40,e;
6
7     e = (a + b) * c / d;          // (10+20)* 30 / 40
8     printf("Value of (a + b) * c / d is : %d\n", e );
9
10    e = ((a + b) * c) / d;       // ((10+20)* 30 ) / 40
11    printf("Value of ((a + b) * c) / d is : %d\n" , e );
12
13    e = (a + b) * (c / d);      // (10+20) * (30/40)
14    printf("Value of (a + b) * (c / d) is : %d\n", e );
15
16    e = a + (b * c) / d;        // 10 + (20*30)/40
17    printf("Value of a + (b * c) / d is : %d\n" , e );
18
19    return 0;
20 }
```

```

1 #include <stdio.h>
2
3 ► int main() {
4
5     int a=10,b=20,c=30,d=40,e;
6
7     e = (a + b) * c / d;          // (10+20)* 30 / 40      22
8     printf("Value of (a + b) * c / d is : %d\n", e );
9
10    e = ((a + b) * c) / d;       // ((10+20)* 30 ) / 40    22
11    printf("Value of ((a + b) * c) / d is : %d\n" , e );
12
13    e = (a + b) * (c / d);      // (10+20) * (30/40)      0
14    printf("Value of (a + b) * (c / d) is : %d\n", e );
15
16    e = a + (b * c) / d;        // 10 + (20*30)/40      25
17    printf("Value of a + (b * c) / d is : %d\n" , e );
18
19    return 0;
20 }
```

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

```
#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

```
1      #include <stdio.h>
2 ►  - int main(void)
3   {
4       int a;
5       for(a = 0; a < 5; a++)
6       {
7           printf("%d ", a);
8       }
9       return 0;
10      }
```

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;
}
```

for loop

```
1      #include <stdio.h>
2      int main() {
3          for (;;) {
4              printf("This is a strange infinite loop");
5          }
6      }
```

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;
}
```

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 you 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()

```
1 #include <stdio.h>
2 ► int main() {
3
4     printf("%lu\n", sizeof(char));
5     printf("%lu\n", sizeof(int));
6     printf("%lu\n", sizeof(float));
7     printf("%lu\n", sizeof(double));
8
9     int a = 25;
10    double d= 40.55;
11    printf("%lu\n", sizeof(a+d));
12
13    int arr[10] = {5,8,9,12};
14    printf("\n Size of the array :%lu", sizeof(arr));
15    printf("\n Capacity the array :%lu", sizeof(arr)/sizeof(arr[0]));
16
17    int arr2[] = {5,8,9,12};
18    printf("\n Size of the array2 :%lu", sizeof(arr2));
19    printf("\n Capacity the array2 :%lu", sizeof(arr2)/sizeof(arr2[0]));
20    return 0;
21 }
```

sizeof()

```
1 #include <stdio.h>
2 ► int main() {
3
4     printf("%lu\n", sizeof(char));    1
5     printf("%lu\n", sizeof(int));    4
6     printf("%lu\n", sizeof(float));   4
7     printf("%lu\n", sizeof(double));  8
8
9     int a = 25;
10    double d= 40.55;
11    printf("%lu\n", sizeof(a+d));    8
12
13    int arr[10] = {5,8,9,12};           40
14    printf("\n Size of the array :%lu", sizeof(arr));
15    printf("\n Capacity the array :%lu", sizeof(arr)/sizeof(arr[0]));  10
16
17    int arr2[] = {5,8,9,12};           16
18    printf("\n Size of the array2 :%lu", sizeof(arr2));
19    printf("\n Capacity the array2 :%lu", sizeof(arr2)/sizeof(arr2[0]));  4
20    return 0;
21 }
```

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

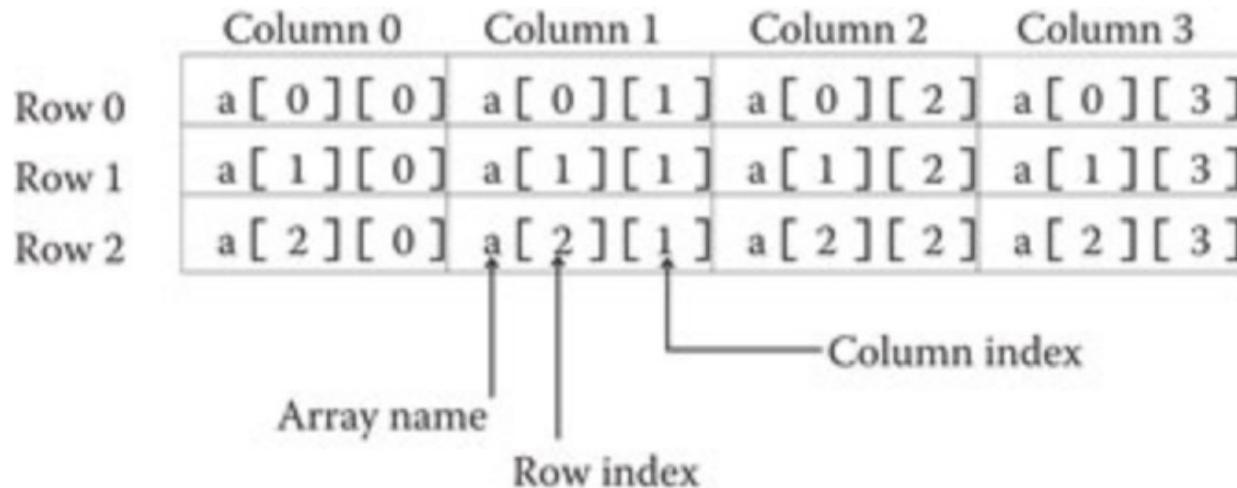
```
1 #include <stdio.h>
2 ► int main() {
3     int i, j=10, arr[10];
4     arr[0]=10;
5     arr[1]=arr[0]*2;
6     for (i=2;i<10;i++){
7         arr[i]=j*(i+1);
8     }
9     for (i=0;i<10;i++)
10        printf("\n arr[%d] :%d",i,arr[i]);
11
12    return 0;
13 }
```

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];
```

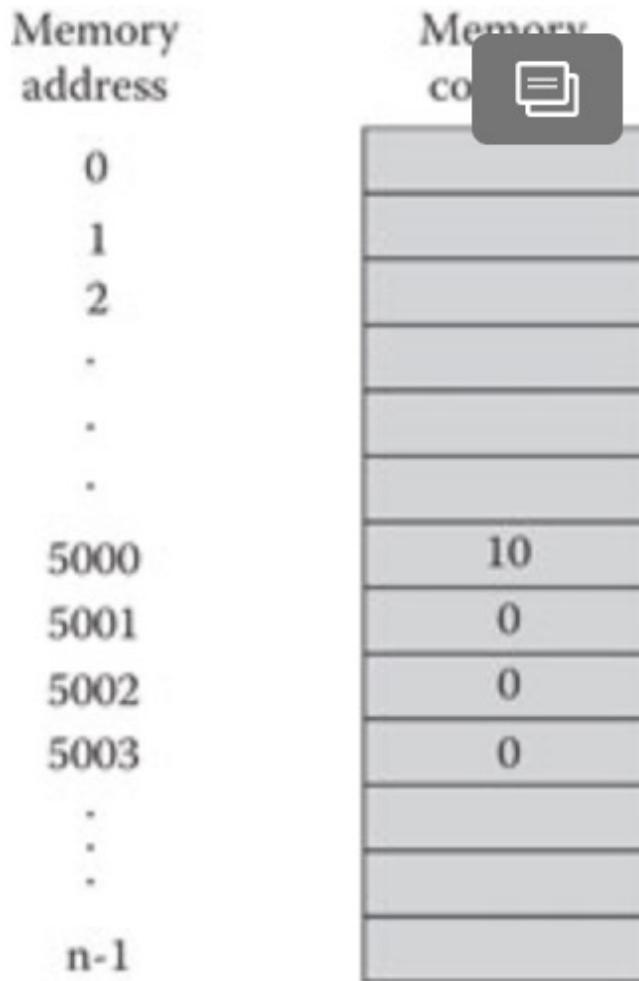
- /* To reach the memory location variables, arrays...etc use ampersand (&) operator */

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

size of a pointer ???

```
1 #include <stdio.h>
2 ► └ int main() {
3     char c;
4     char *ptrC = &c;
5     int a=5;
6     int *ptrI = &a;
7     float f =20.66;
8     float *ptrF = &f;
9     double d = 44.445;
10    double *ptrD = &d;
11
12    printf("size of c: %u\n", sizeof(c));
13    printf("size of ptrC: %u\n", sizeof(ptrC));
14    printf("size of a: %u\n", sizeof(a));
15    printf("size of ptrI: %u\n", sizeof(ptrI));
16    printf("size of f: %u\n", sizeof(f));
17    printf("size of ptrF: %u\n", sizeof(ptrF));
18    printf("size of d: %u\n", sizeof(d));
19    printf("size of ptrD: %u\n", sizeof(ptrD));
20
21 }
```

```
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
```

```
1 #include <stdio.h>
2 ► int main(void)
3 {
4     int *ptr, a;
5     a = 10;
6     ptr = &a;
7     printf("Val = %d\n", *ptr);
8     return 0;
9 }
```

10

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

Example - page 1/2

```
1 #include <stdio.h>
2 ► □ int main()
3 {
4     int *ptr, a;
5     a = 25;
6     /* without using a pointer */
7     printf("Address of a: %p\n", &a);
8     printf("Value of a: %d\n", a);
9
10    /*let's use a pointer */
11    ptr = &a;
12    printf("Address of the pointer : %p\n", ptr);
13    printf("Value of the pointer : %d\n", *ptr);
14
15    /* how about if we change the value of int */
16    a = 125;
17    printf("Address of the pointer : %p\n", ptr);
18    printf("Value of the pointer : %d\n", *ptr);
19
```

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
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

```
1 #include <stdio.h>
2 ► □ int main()
3 {
4     int *ptr, a;
5     a = 25;
6     /* without using a pointer */
7     printf("Address of a: %p\n", &a);
8     printf("Value of a: %d\n", a);
9
10    /*let's use a pointer */
11    ptr = &a;
12    printf("Address of the pointer : %p\n", ptr);
13    printf("Value of the pointer : %d\n", *ptr);
14
15    /* how about if we change the value of int */
16    a = 125;
17    printf("Address of the pointer : %p\n", ptr);
18    printf("Value of the pointer : %d\n", *ptr);
19
```

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
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

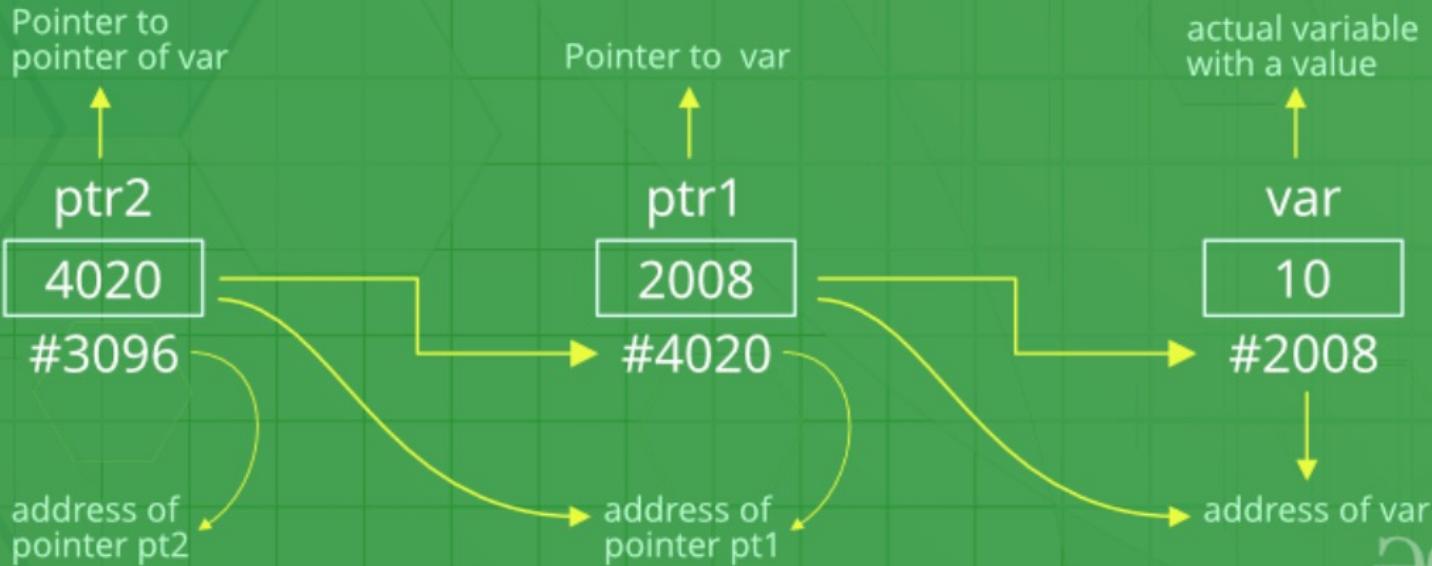
```
#include <stdio.h>

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...

Double Pointer



ĐG

```
#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;
}
```

```
#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

```
1 #include <stdio.h>
2 ► └ int main() {
3     int i, arr[5];
4     double arr2[5];
5
6     for(i = 0; i < 5; i++)
7         printf("address of arr[%d] = %p\n", i, &arr[i]);
8
9     for(i = 0; i < 5; i++)
10        printf("address of arr2[%d] = %p\n", i, &arr2[i]);
11
12 }
```

Arrays & Pointers

```
1      address of arr[0] = 0x7ffeeece0910
2      address of arr[1] = 0x7ffeeece0914
3      address of arr[2] = 0x7ffeeece0918
4      address of arr[3] = 0x7ffeeece091c
5      address of arr[4] = 0x7ffeeece0920
6
7      address of arr2[0] = 0x7ffeeece08e0
8      address of arr2[1] = 0x7ffeeece08e8
9      address of arr2[2] = 0x7ffeeece08f0
10     address of arr2[3] = 0x7ffeeece08f8
11     address of arr2[4] = 0x7ffeeece0900
12 
```

Array Manipulation

```
1 #include <stdio.h>
2 ► int main() {
3     int arr[5] = {10, 20, 30, 40, 50};
4     int *ptr;
5
6     ptr = &arr[3]; // address of the fourth element
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 }
```

Array Manipulation

```
1 Pointer value : 40
2 Next Value : 50
3 Previous Value : 30
4 Address of the Pointer : 0x7ffeeb08e91c
5 Address of the Next Value : 0x7ffeeb08e920
6 Address of the Previous Value : 0x7ffeeb08e918
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 ?

```
1 #include <stdio.h>
2 ▶ int main(void)
3 {
4     int *ptr, totalSum, arr[5] = {10, 20, 30, 40, 50};
5     totalSum = 0;
6     for(ptr = arr; ptr < arr+5; ptr++)
7     {
8         --*ptr;
9         totalSum += *ptr;
10    }
11    printf("Sum = %d\n", totalSum);
12    return 0;
13 }
```

What is the Result ?

```
1 #include <stdio.h>
2 ▶ int main(void)
3 {
4     int *ptr, totalSum, arr[5] = {10, 20, 30, 40, 50};
5     totalSum = 0;
6     for(ptr = arr; ptr < arr+5; ptr++)
7     {
8         --*ptr;
9         totalSum += *ptr;
10    }
11    printf("Sum = %d\n", totalSum);
12    return 0;
13 }
```

145

Passing Pointers to functions

```
1 #include <stdio.h>
2 ← void test(int a);
3 ► ⌂ int main(void)
4 {
5     void (*ptr)(int a);
6     ptr = test;
7     (*ptr)(10);
8     return 0;
9 }
10 ← ⌂ void test(int a)
11 {
12     printf("%d\n", 2*a);
13 }
```

20

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↴ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d\n", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 ↵ { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 ↵ { return a-b; }
```

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↵ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { 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 1 #include <stdio.h>
2 ► int main(void)
3 {
4     char ch;
5     ch = 'a';
6     printf("Char = %c and its ASCII code is %d\n", ch, ch);
7     return 0;
8 }
9
```

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()

```
1 #include <stdio.h>
2 ► □ int main(void)
3 {
4     char str[] = "UAB CS 330 Course";
5     puts(str);
6     puts(str);
7     str[4] = '\0';
8     printf("%s\n", str);
9     return 0;
10 }
```

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

Passing Pointers to functions

```
1 #include <stdio.h>
2 ← void test(int a);
3 ► ⌂ int main(void)
4 {
5     void (*ptr)(int a);
6     ptr = test;
7     (*ptr)(10);
8     return 0;
9 }
10 ← ⌂ void test(int a)
11 {
12     printf("%d\n", 2*a);
13 }
```

20

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↴ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d\n", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { return a-b; }
```

```
1 #include <stdio.h>
2 ↵ int addTwoNumbers(int a, int b);
3 ↵ int subtractTwoNumbers(int a, int b);
4
5 ► ↵ int main(void)
6 {
7     int (*ptr[2])(int a, int b);
8     int i, j, result;
9     ptr[0] = addTwoNumbers;
10    ptr[1] = subtractTwoNumbers;
11
12    printf("Enter two integer numbers: ");
13    scanf("%d %d", &i, &j);
14    if(i > 0 && i < 25)
15        result = ptr[0](i, j);
16    else
17        result = ptr[1](i, j);
18    printf("Result : %d", result);
19    return 0;
20 }
21 ↵ int addTwoNumbers(int a, int b)
22 { return a+b; }
23 ↵ int subtractTwoNumbers(int a, int b)
24 { return a-b; }
```

Enter two integer numbers: 20 30

Result : 50

Enter two integer numbers: 45 20

Result : 25

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()

```
1 #include <stdio.h>
2 ► □ int main(void)
3 {
4     char str[] = "UAB CS 330 Course";
5     puts(str);
6     puts(str);
7     str[4] = '\0';
8     printf("%s\n", str);
9     return 0;
10 }
```

```
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

```
1 #include <stdio.h>
2 ► int main(void)
3 {
4     char str[100];
5     int num;
6     printf("Enter number: ");
7     scanf("%d", &num);
8     printf("Enter text: ");
9     fgets(str, sizeof(str), stdin);
10    printf("%d %s\n", num, str);
11    return 0;
12 }
```

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.

```
1 #include <stdio.h>
2 #include <string.h>
3 ► int main(void)
4 {
5     char str1[100], str2[100];
6     printf("Enter text: ");
7     fgets(str2, sizeof(str2), stdin);
8     strcpy(str1, str2);
9     printf("Copied text: %s\n", str1);
10    return 0;
11 }
```

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 ↵ void test(void);
4     int global;
5 ► int main(void)
6 {
7     int *ptr;
8     int i;
9     static int st;
10
11    /* Allocate memory from the heap. */
12    ptr = (int*) malloc(sizeof(int));
13
14    if(ptr != NULL)
15    {
16        printf("Code seg: %p\n", test);
17        printf("Data seg: %p %p\n", &global, &st);
18        printf("Stack seg: %p\n", &i);
19        printf("Heap: %p\n", ptr);
20        free(ptr);
21    }
22    return 0;
23 }
24 ↵ void test(void)
25 { /* Do something. */
26 }
```

```
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()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 ► int main()
5 {    int *ptr,n,i;
6     /* the number of array elements */
7     printf("How many elements?:\n");
8     scanf("%d",&n);
9
10    ptr = (int*)malloc(n * sizeof(int));
11
12    if (ptr == NULL) {
13        printf("Memory allocation was NOT successful.\n");
14        exit(0);
15    }
16    else {
17        printf("Memory allocation was successful.\n");
18        for (i = 0; i < n; i++)
19            ptr[i] = (i+1) * 10;
20
21        for (i = 0; i < n; ++i)
22            printf("%d, ", ptr[i]);
23    }
24    free(ptr);
25    printf("\nMemory deallocation was successful.\n");
26    return 0;
27 }
```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 ► int main()
5 {    int *ptr,n,i;
6     /* the number of array elements */
7     printf("How many elements?:\n");
8     scanf("%d",&n);
9
10    ptr = (int*)malloc(n * sizeof(int));
11
12    if (ptr == NULL) {
13        printf("Memory allocation was NOT successful.\n");
14
15    }
16
17    }
18
19
20
21
22
23
24
25
26
27 }
```

How many elements?:

8

Memory allocation was successful.

10, 20, 30, 40, 50, 60, 70, 80,

Memory deallocation was successful.

```
return 0;
```

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.

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

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

sizeof()

```
#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;
};
```

```
1 #include <stdio.h>
2 struct student
3 {
4     int code;
5     float grd;
6 };
7 ► int main(void)
8 {
9     struct student s1, s2;
10    s1.code = 1234;
11    s1.grd = 6.7;
12    s2 = s1; /* Copy structure. */
13    printf("C:%d G:%.2f\n", s2.code, s2.grd);
14    return 0;
15 }
```

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.

```
#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.

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

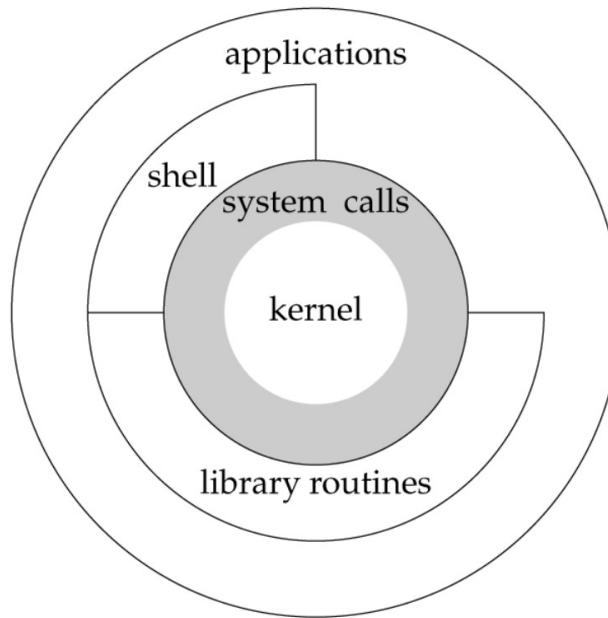


Figure 1.1 Architecture of the UNIX operating system

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

Working in the UNIX Environment

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

Shells

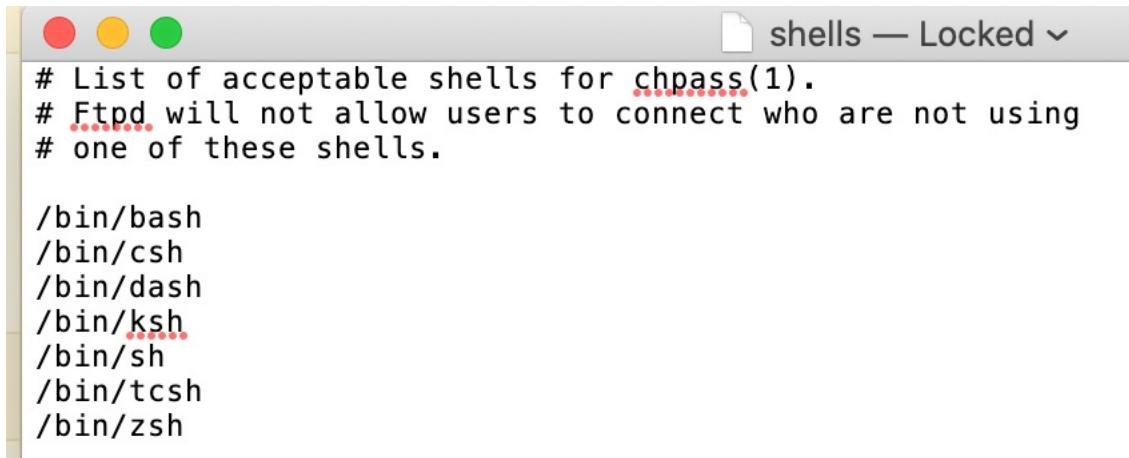
- A shell is the interface between the user and the kernel.
- Users can interact with the shell using shell commands in terminal or from a file (shell script).
- The common shells are;

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



The screenshot shows a terminal window with a title bar labeled "shells — Locked". The window contains the following text:

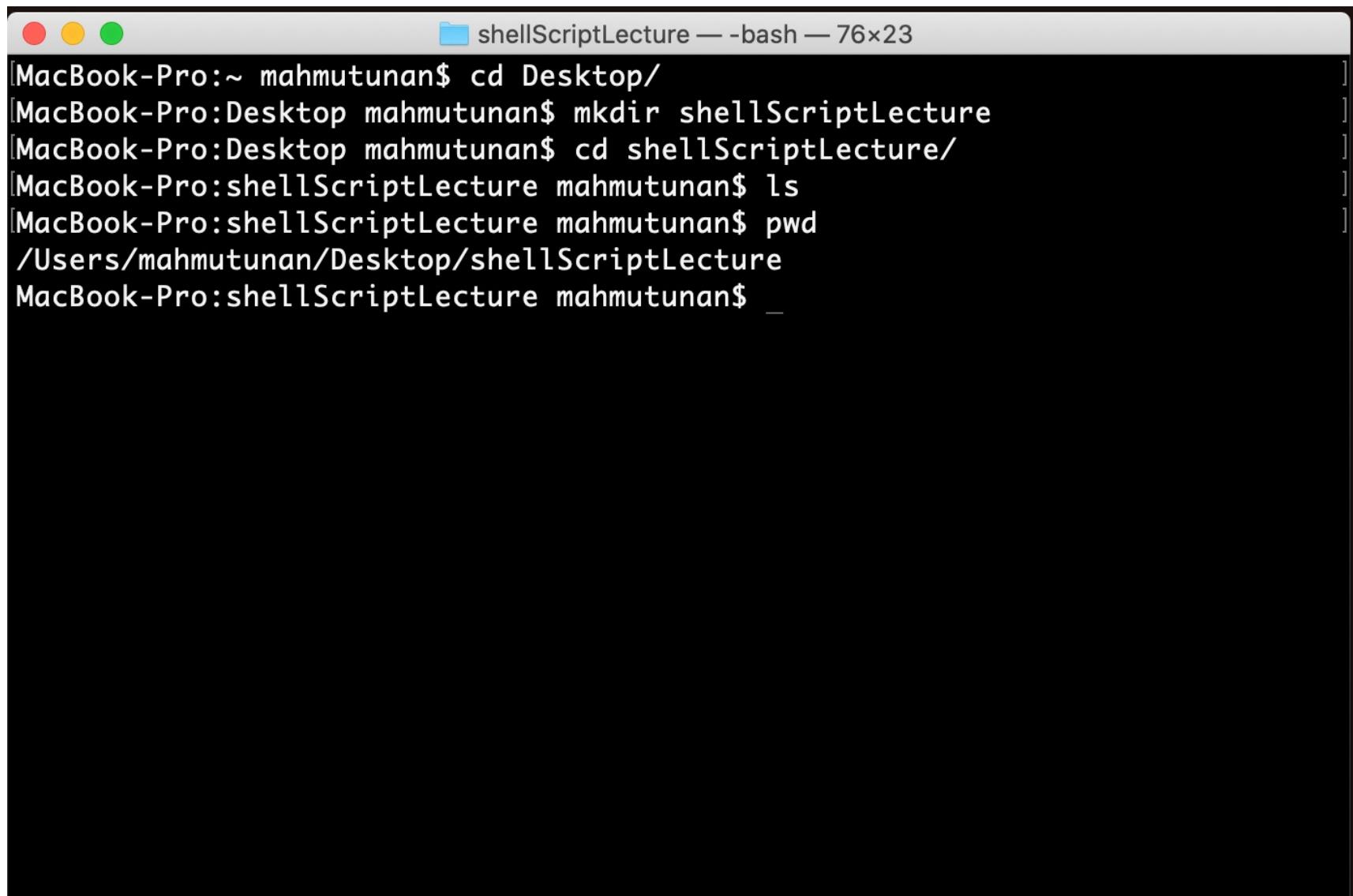
```
# List of acceptable shells for chpass(1).
# Ftpd will not allow users to connect who are not using
# one of these shells.

/bin/bash
/bin/csh
/bin/dash
/bin/ksh
/bin/sh
/bin/tcsh
/bin/zsh
```

Windows Users

- Windows Subsystem 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

A screenshot of a macOS terminal window titled "shellScriptLecture — -bash — 76x23". The window shows a series of shell commands being run by a user named "mahmutunan". The commands include navigating to the Desktop, creating a new directory named "shellScriptLecture", changing into that directory, listing its contents, and printing its full path. The terminal has a dark background with white text and uses standard macOS-style window controls.

```
[MacBook-Pro:~ mahmutunan$ cd Desktop/  
[MacBook-Pro:Desktop mahmutunan$ mkdir shellScriptLecture  
[MacBook-Pro:Desktop mahmutunan$ cd shellScriptLecture/  
[MacBook-Pro:shellScriptLecture mahmutunan$ ls  
[MacBook-Pro:shellScriptLecture mahmutunan$ pwd  
/Users/mahmutunan/Desktop/shellScriptLecture  
MacBook-Pro:shellScriptLecture mahmutunan$ _
```

.sh file

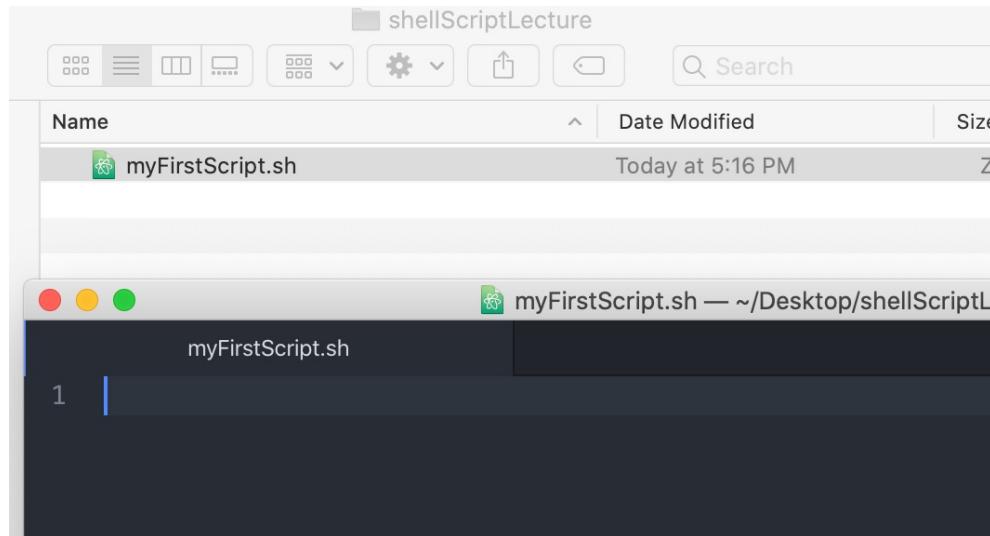
- It is a script programmed for bash
 - It contains instructions written in the Bash language
 - It can be executed by typing text commands within the shell's command-line interface.
- How to run the .sh file?
 - First, give the execute permission
 - chmod 755 somefilename.sh
 - Next, run your script 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
<code>-e</code>	To check if the file exists.
<code>-r</code>	To check if the file is readable.
<code>-w</code>	To check if the file is writable.
<code>-x</code>	To check if the file is executable.
<code>-s</code>	To check if the file size is greater than 0.
<code>-d</code>	To check if the file is a directory.

```
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 "00PPSSS, the file does not exists!"
11 fi
```

```
[MacBook-Pro:shellScriptLecture mahmutunan$ bash fileOperations.sh
00PPSSS, 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"

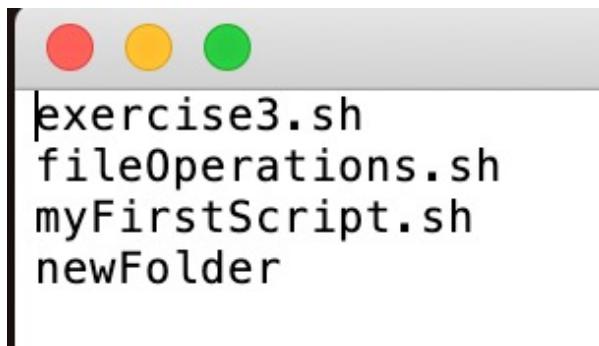
echo -n "How old are you: "
read AGE

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
```

Attachment File →
guru99@VirtualBox:~\$ mail -s "News Today" abc@ymail.com < NewsFlash
E-mail Subject ↑ E-mail Address ↑

Man Page

\$man command

\$man cat

CAT(1)	BSD General Commands Manual	CAT(1)
NAME		
cat -- concatenate and print files		
SYNOPSIS		
cat [-benstuv] [<u>file</u> ...]		
DESCRIPTION		
The cat utility reads files sequentially, writing them to the standard output. The <u>file</u> operands are processed in command-line order. If <u>file</u> is a single dash (`-') or absent, cat reads from the standard input. If <u>file</u> is a UNIX domain socket, cat connects to it and then reads it until EOF. This complements the UNIX domain binding capability available in <i>inetd(8)</i> .		
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

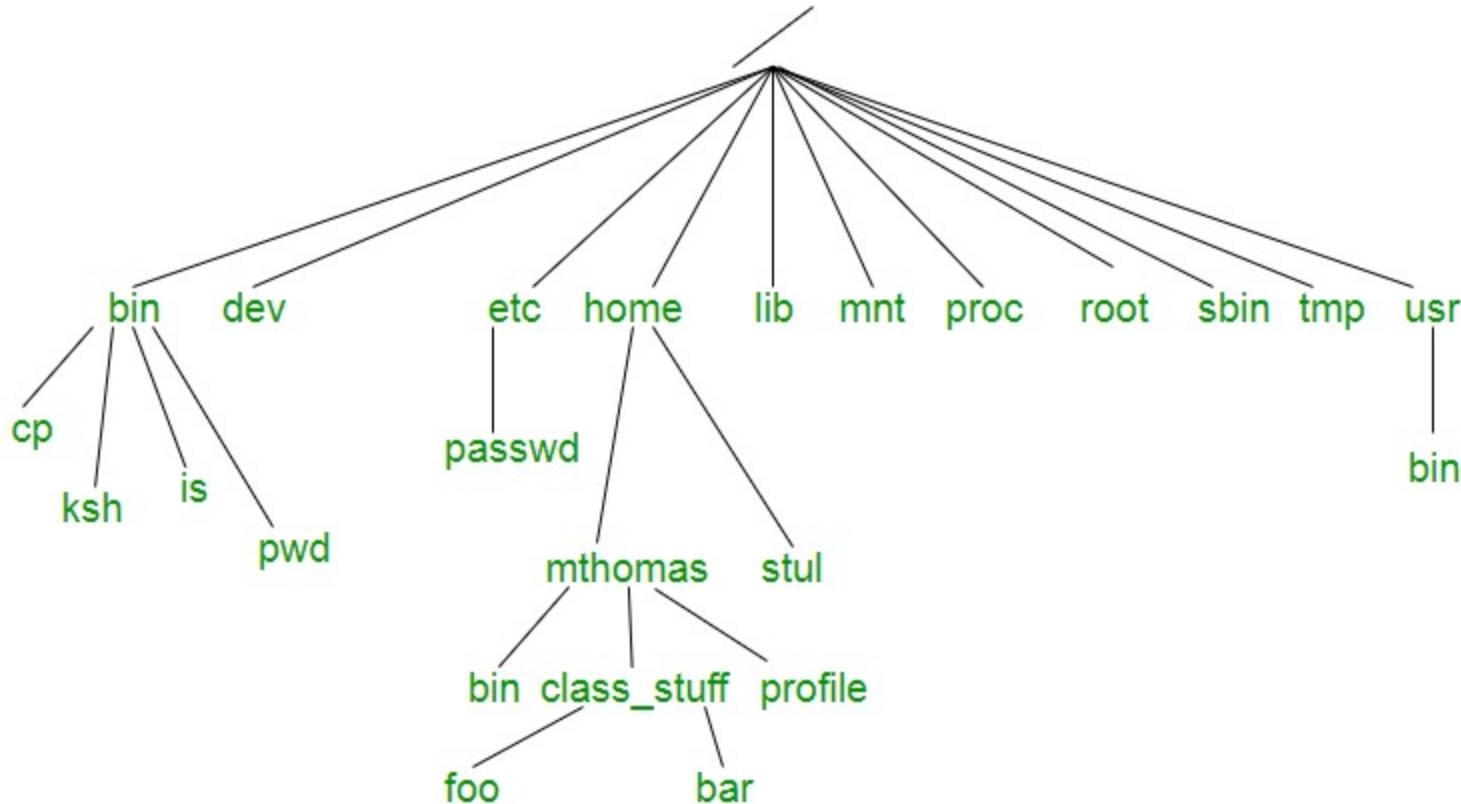
- All data organized into files
- All files organized into directories
- Directories
 - Tree-like structures
 - Multi-level hierarchy (directory tree)
 - Top level - root
 - /
 - All other directories are the children of the root
- File = sequence of bytes

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



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

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](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\(7\)).
```

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.

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.

Exercise

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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5
6  #define BUFFSIZE 4096
7  |
8  ► int main(int argc, char *argv[]) {
9      int readFileDescriptor, writeFileDescriptor;
10     long int n;
11     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

```
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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5
6  #define BUFFSIZE 4096
7  |
8  ► int main(int argc, char *argv[]) {
9      int readFileDescriptor, writeFileDescriptor;
10     long int n;
11     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

```
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 `fseek` 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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <fcntl.h>
5 #include <string.h>
6
7 #define BUFFSIZE 4096
8 #define SEEKSIZE -10
9
10 ► int main(int argc, char *argv[]) {
11     int RWFileDescriptor;
12     long int n;
13     char buf[BUFFSIZE];
14     const char lseekMSG[] = "THIS IS NEW MSG FROM LSEEK!\0";
15
16     if (argc != 2){
17         printf("Usage: %s <filename>\n", argv[0]);
18         exit (-1);
19     }
20
21     RWFileDescriptor = open(argv[1], O_RDONLY);
22
23     if (RWFileDescriptor == -1){
24         printf("Error with file open\n");
25         exit (-1);
26     }
27
```

- 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 }
```

Run the exercise 2

```
(base) mahmutunan@MacBook-Pro Desktop % gcc fileseek.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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #define BUF_SIZE 1024
6
7  int main(int argc, char *argv[]) {
8      if (argc != 2) {
9          printf("Usage: %s <filename>\n", argv[0]);
10         exit(-1);
11     }
12     char *file_Name = argv[1];
13
14     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

```
1 #include <stdio.h>
2
3 int main (int argc, char *argv[]){
4
5     char charArr[20] = "Hello CS330";
6     printf("%s",charArr);
7     return 0;
8
9 }
```

```
1 myExecutable: main.c
2     gcc -o myExecutable main.c
3
```

```
[base] mahmutunan@MacBook-Pro exercise1 % ./myExecutable
(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

```
1 #include "someFunc.h"
2
3 int main (int argc, char *argv[]){
4
5     printHellofunc();
6     return 0;
7
8 }
```

- main.c

```
1 #include <stdio.h>
2 #include "someFunc.h"
3
4 void printHellofunc(){
5
6     printf("Hello CS330");
7
8 }
```

- someFunc.c

```
1 void printHellofunc();
2
```

- 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 %
```

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)
```

Exercise 2

- Let's modify the make file a little bit

```
[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
main.c           someFunc.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

```
1 #include<stdio.h>
2 #include<stdlib.h>
3
4 /* main method */
5 int main(int args, char** argv){
6     int N, i;
7     printf("Please enter number of elements in array: ");
8     scanf("%d", &N);
9
10    float arr[N];
11
12    for (i=0; i<N; i++){
13        printf("Please enter element %d of array: ", (i+1));
14        scanf("%f", &arr[i]);
15    }
16
17    printf("Given array is: ");
18    printf("[");
19    for (i=0; i < N-1; i++){
20        printf("%f, ", arr[i]);
21    }
22    printf("%f]\n", arr[N-1]);
23
24    float temp;
25    int currLoc;
26    for (i=1; i < N; i++){
27        currLoc = i;
28        while (currLoc > 0 && arr[currLoc-1] > arr[currLoc]){
29            temp = arr[currLoc];
30            arr[currLoc] = arr[currLoc-1];
31            arr[currLoc-1] = temp;
32            currLoc--;
33        }
34    }
35
36    printf("Sorted array is: ");
37    printf("[");
38    for (i=0; i < N-1; i++){
39        printf("%f, ", arr[i]);
40    }
41    printf("%f]\n", arr[N-1]);
42
43    return 0;
44 }
```

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

```
1 #include "gettime.h"
2
3 double gettime(void) {
4     struct timeval tval;
5     gettimeofday(&tval, NULL);
6     return((double)tval.tv_sec + (double)tval.tv_usec/1000000.0);
7 }
8
```

- 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
gettme.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, 2.000000]
Sorted array is: [2.000000, 21.000000, 22.000000, 32.000000, 44.000000, 55.000000, 56.000000]
[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

```
1 # Sample Makefile to compile C programs
2 CC = gcc
3 CFLAGS = -Wall -g #replace -g with -O when not debugging
4 DEPS = gettimeofday.h Makefile
5 OBJS = gettimeofday.o insertionsort.o
6 EXECS = insertionsort
7
8 all: $(EXECS)
9
10 %.o: %.c $(DEPS)
11         $(CC) $(CFLAGS) -c -o $@ $<
12
13 insertionsort: $(OBJS)
14         $(CC) $(CFLAGS) -o $@ $^
15
16 clean:
17         /bin/rm -i *.o $(EXECS)
18
19
20
```

- 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);
```

- `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*.

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  
};
```

```
(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
<code>S_ISREG()</code>	regular file
<code>S_ISDIR()</code>	directory file
<code>S_ISCHR()</code>	character special file
<code>S_ISBLK()</code>	block special file
<code>S_ISFIFO()</code>	pipe or FIFO
<code>S_ISLNK()</code>	symbolic link
<code>S_ISSOCK()</code>	socket

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

Exercise 1 - printstat.c

```
1  /* function to print stat data structure */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <unistd.h>
7  #include <time.h>
8
9  void printstat(struct stat sb) {
10  /* copied from the lstat man page example as is */
11  printf("File type:          ");
12
13  switch (sb.st_mode & S_IFMT) {
14  case S_IFBLK:   printf("block device\n");           break;
15  case S_IFCHR:   printf("character device\n");       break;
16  case S_IFDIR:   printf("directory\n");              break;
17  case S_IFIFO:   printf("FIFO/pipe\n");             break;
18  case S_IFLNK:   printf("symlink\n");                break;
19  case S_IFREG:   printf("regular file\n");            break;
20  case S_IFSOCK:  printf("socket\n");                 break;
21  default:        printf("unknown?\n");               break;
22 }
```

Exercise 1 - printstat.c /2

```
23     printf("I-node number:          %ld\n", (long) sb.st_ino);
24
25     printf("Mode:                  %lo (octal)\n",
26            (unsigned long) sb.st_mode);
27
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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/stat.h>
5 #include <unistd.h>
6
7 void printstat(struct stat statbuf);
8
9 int main(int argc, char **argv) {
10     int i;
11     struct stat buf;
12     char *ptr;
13
14     for (i = 1; i < argc; i++) {
15         printf("%s: ", argv[i]);
16         if (lstat(argv[i], &buf) < 0) {
17             printf("lstat error");
18             continue;
19     }
```

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 }
```

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.

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.)

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <dirent.h>
4
5 int main (int argc, char **argv) {
6     struct dirent *dirent;
7     DIR *parentDir;
8
9     if (argc < 2) {
10         printf ("Usage: %s <dirname>\n", argv[0]);
11         exit(-1);
12     }
13     parentDir = opendir (argv[1]);
14     if (parentDir == NULL) {
15         printf ("Error opening directory '%s'\n", argv[1]);
16         exit (-1);
17     }
18     int count = 1;
19     while((dirent = readdir(parentDir)) != NULL){
20         printf ("[%d] %s\n", count, (*dirent).d_name);
21         count++;
22     }
23     closedir (parentDir);
24     return 0;
25 }
```

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

```
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <dirent.h>
8
9 char *filetype(unsigned char type) {
10     char *str;
11     switch(type) {
12         case DT_BLK: str = "block device"; break;
13         case DT_CHR: str = "character device"; break;
14         case DT_DIR: str = "directory"; break;
15         case DT_FIFO: str = "named pipe (FIFO)"; break;
16         case DT_LNK: str = "symbolic link"; break;
17         case DT_REG: str = "regular file"; break;
18         case DT SOCK: str = "UNIX domain socket"; break;
19         case DT_UNKNOWN: str = "unknown file type"; break;
20         default: str = "UNKNOWN";
21     }
22     return str;
23 }
```

Exercise 3 - readdir_v2.c / 2

```
24
25     int main (int argc, char **argv) {
26         struct dirent *dirent;
27         DIR *parentDir;
28
29         if (argc < 2) {
30             printf ("Usage: %s <dirname>\n", argv[0]);
31             exit(-1);
32         }
33         parentDir = opendir (argv[1]);
34         if (parentDir == NULL) {
35             printf ("Error opening directory '%s'\n", argv[1]);
36             exit (-1);
37         }
38         int count = 1;
39         while((dirent = readdir(parentDir)) != NULL){
40             printf ("[%d] %s (%s)\n", count, dirent->d_name, filetype(dirent->d_type));
41             count++;
42         }
43         closedir (parentDir);
44         return 0;
45     }
```

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] writetest.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

```
#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

```
#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 %
```

Standard I/O Library

- This library is specified by ISO C standard because it has been implemented on many OS
- The standard I/O library handles such details as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size

Recall - File Descriptor

- When a file opened
 - nonnegative int assigned
 - this int is used in all operations
- Streams
 - with the standard I/O library, the discussion centers on streams
 - Open or Create a file → associate with a stream

I/O Stream

- *Open I/O Stream:* The standard I/O stream allows you to open a file in read, write, or append modes. This mode can be combined in a single open function call (see Figure 5.2 in Section 5.5 of the textbook for a complete list of options that can be specified). For example:

```
FILE *fptr;  
fptr = fopen ("listings.csv", "r+");
```

<i>type</i>	Description	<i>open(2)</i> Flags
r or rb	open for reading	O_RDONLY
w or wb	truncate to 0 length or create for writing	O_WRONLY O_CREAT O_TRUNC
a or ab	append; open for writing at end of file, or create for writing	O_WRONLY O_CREAT O_APPEND
r+ or r+b or rb+	open for reading and writing	O_RDWR
w+ or w+b or wb+	truncate to 0 length or create for reading and writing	O_RDWR O_CREAT O_TRUNC
a+ or a+b or ab+	open or create for reading and writing at end of file	O_RDWR O_CREAT O_APPEND

Figure 5.2 The *type* argument for opening a standard I/O stream

Input / Output Stream:

- ***Input Stream:*** The standard I/O stream allows us to read from the open file. These functions allow us to read a file character by character – getchar(), line by line – fgets(), or with specific size – fread()
- ***Output Stream:*** The standard I/O stream allow you to write to an open file. These functions allow you to write to a file character by character – putchar(), line by line – fputs(), or with specific size – fwrite()

Exercise 1

- Now let's use these functions and write a program. We will use APIs available in Linux and C to develop different versions of this program

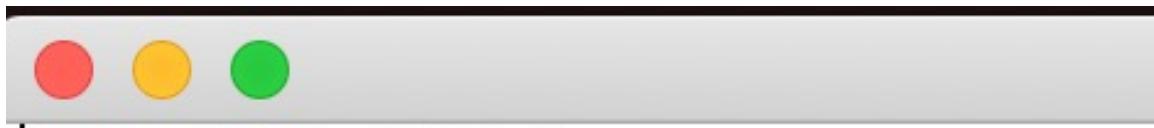
getc()

```
int getc(FILE *stream)
```

- Gets the next character (an unsigned char) from the specified stream
- Advances the position indicator for the stream.

getline1.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int getLine(FILE *fp, char *line);
5
6  int main(int argc, char** argv) {
7      char *str;
8      FILE *fp;
9      int n;
10
11     str = malloc(sizeof(char)*BUFSIZ);
12     fp = fopen(filename: argv[1], mode: "r");
13     if (fp == NULL) {
14         printf("Error opening file %s\n", argv[1]);
15         exit(-1);
16     }
17     while ( (n = getLine(fp, str)) > 0)
18         printf("%d: %s\n", n, str);
19     fclose(fp);
20     return 0;
21 }
22
23 int getLine(FILE *fp, char *line) {
24     int c, i=0;
25     while ((c = getc(fp)) != '\n' && c != EOF)
26         line[i++] = c;
27     line[i] = '\0';
28     return i;
29 }
```



Some text line 1
Line 2
l3

```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline1 getline1.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline1 test.txt
16: Some text line 1
6: Line 2
2: l3
```

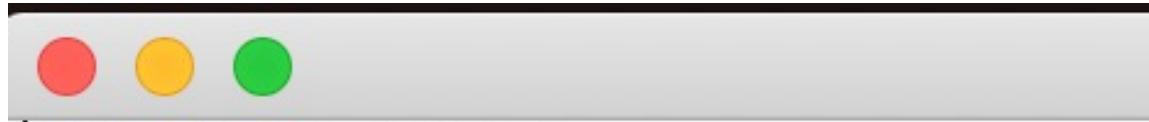
getline()

```
ssize_t getline(char **lineptr, size_t *n, FILE  
*stream);
```

- **getline()** reads an entire line from *stream*, storing the address of the buffer containing the text into **lineptr*. The buffer is null-terminated and includes the newline character, if one was found.
- On success, **getline()** returns the number of characters read, including the delimiter character, but not including the terminating null byte ('\0')

getline2.c

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char** argv) {
6     char *line=NULL;
7     FILE *fp;
8     size_t maxlen=0;
9     ssize_t n;
10
11     printf("BUFSIZ = %d\n", BUFSIZ);
12
13     if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
14         printf("Error opening file %s\n", argv[1]);
15         exit(-1);
16     }
17     while ( (n = getline(&line, &maxlen, fp)) > 0)
18         printf("%ld[%ld]: %s\n", n, maxlen, line);
19
20     fclose(fp);
21     return 0;
22 }
```



```
(base) mahmutunan@MacBook-Pro lecture15 % ./getline2 test.txt
BUFSIZ = 1024
17[32]: Some text line 1

7[32]: Line 2

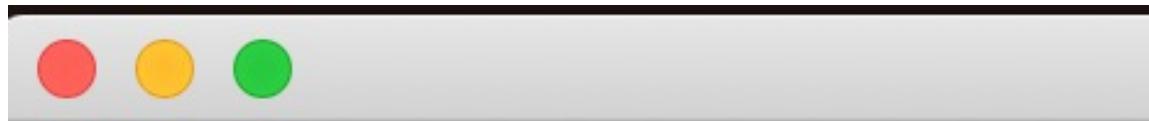
2[32]: l3
```

getdelim()

- `ssize_t getdelim(char **lineptr,
size_t *n, int delim, FILE *stream);`
- `getdelim()` works like `getline()`, except that a line delimiter other than newline can be specified as the *delimiter* argument.
- `getdelim()` returns the number of characters read, including the delimiter character, but not including the terminating null byte

getline3.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char** argv) {
5     char *line=NULL;
6     FILE *fp;
7     size_t maxlen=0;
8     ssize_t n;
9
10    if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
11        printf("Error opening file %s\n", argv[1]);
12        exit(-1);
13    }
14    while ( (n = getdelim(&line, &maxlen, delimiter: ' ', fp)) > 0)
15        printf("%ld: %s\n", n, line);
16
17    fclose(fp);
18    return 0;
19}
20
```



Some text line 1
Line 2
l3

```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline3 getline3.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline3 test.txt
5: Some
5: text
5: line
7: 1
Line
4: 2
l3
(base) mahmutunan@MacBook-Pro lecture15 %
```

gets()

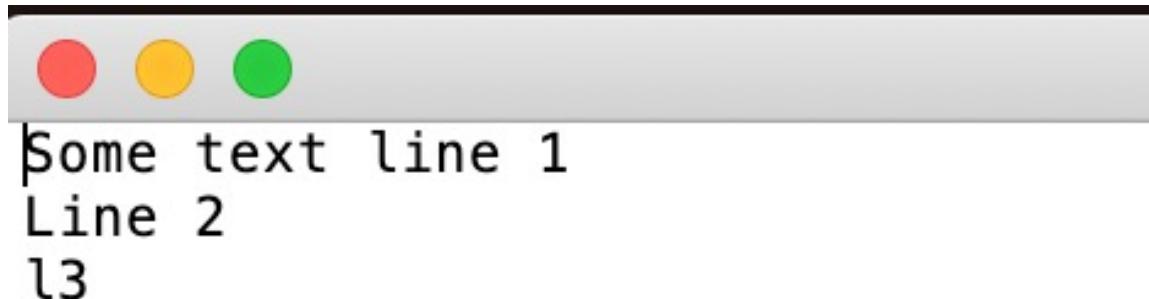
- `char *gets(char *str)`
- The C library function `char *gets(char *str)` reads a line from `stdin` and stores it into the string pointed to by `str`.
- It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.
- It reads string from standard input and prints the entered string, but it suffers from Buffer Overflow as `gets()` doesn't do any array bound testing.
- `gets()` keeps on reading until it sees a newline character.
- To avoid Buffer Overflow, `fgets()` should be used instead of `gets()` as `fgets()` makes sure that not more than `MAX_LIMIT` characters are read.

fgets()

- `char *fgets(char *str, int n, FILE *stream)`
- **str** – This is the pointer to an array of chars where the string read is stored.
- **n** – This is the maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as str is used.
- **stream** – This is the pointer to a FILE object that identifies the stream where characters are read from.
- On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned

getline4.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char** argv) {
6     char *line;
7     FILE *fp;
8
9     line = malloc(sizeof(char)*BUFSIZ);
10
11    if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
12        fprintf(stderr,"Error opening file %s\n", argv[1]);
13        exit(-1);
14    }
15    while ( fgets(line, BUFSIZ, fp) != NULL )
16        fprintf(stdout,"%ld: %s\n", strlen(line), line);
17
18    fclose(fp);
19    return 0;
20}
```



Some text line 1
Line 2
l3

```
[base] mahmutunan@MacBook-Pro lecture15 % gcc -o getline4 getline4.c
[base] mahmutunan@MacBook-Pro lecture15 % ./getline4 test.txt
17: Some text line 1
7: Line 2
2: l3
[base] mahmutunan@MacBook-Pro lecture15 %
```

fscanf()

- `int fscanf(FILE *stream, const char *format, ...)`

Conversion type	Description
d, i	signed decimal
o	unsigned octal
u	unsigned decimal
x, X	unsigned hexadecimal
f, F	double floating-point number
e, E	double floating-point number in exponential format
g, G	interpreted as f, F, e, or E, depending on value converted
a, A	double floating-point number in hexadecimal exponential format
c	character (with l length modifier, wide character)
s	string (with l length modifier, wide character string)
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters written so far
%	a % character
C	wide character (XSI option, equivalent to lc)
S	wide character string (XSI option, equivalent to ls)

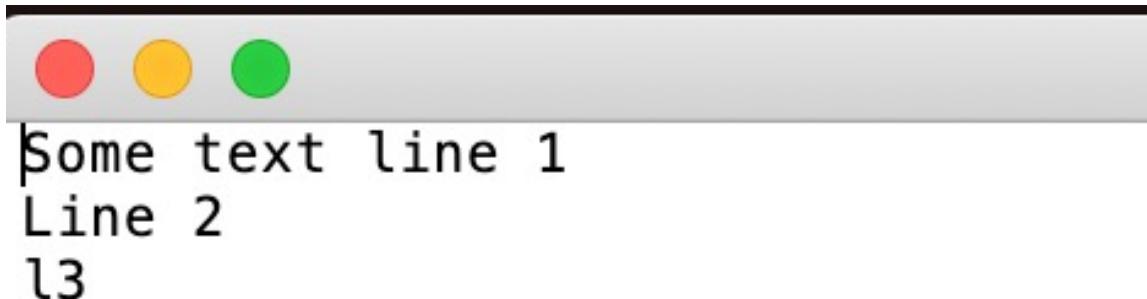
Figure 5.9 The conversion type component of a conversion specification

Conversion type	Description
d	signed decimal, base 10
i	signed decimal, base determined by format of input
o	unsigned octal (input optionally signed)
u	unsigned decimal, base 10 (input optionally signed)
x,X	unsigned hexadecimal (input optionally signed)
a,A,e,E,f,F,g,G	floating-point number
c	character (with 1 length modifier, wide character)
s	string (with 1 length modifier, wide character string)
[matches a sequence of listed characters, ending with]
[^	matches all characters except the ones listed, ending with]
p	pointer to a void
n	pointer to a signed integer into which is written the number of characters read so far
%	a % character
C	wide character (XSI option, equivalent to 1c)
S	wide character string (XSI option, equivalent to 1s)

Figure 5.10 The conversion type component of a conversion specification

getline5.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char** argv) {
6      char *line;
7      FILE *fp;
8
9      line = malloc(sizeof(char)*BUFSIZ);
10
11     if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
12         printf("Error opening file %s\n", argv[1]);
13         exit(-1);
14     }
15     while ( fscanf(fp, "%s", line) != EOF )
16         printf("%ld: %s\n", strlen(line), line);
17
18     fclose(fp);
19     return 0;
20 }
```



```
[base] mahmutunan@MacBook-Pro lecture15 % gcc -o getline5 getline5.c  
[base] mahmutunan@MacBook-Pro lecture15 % ./getline5 test.txt  
4: Some  
4: text  
4: line  
1: 1  
4: Line  
1: 2  
2: l3
```

fprintf()

```
int fprintf(FILE *stream, const char *format, ...)
```

stream

The stream where the output will be written.

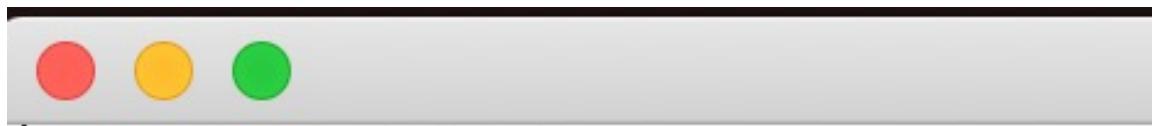
format

Describes the output as well as provides a placeholder to insert the formatted string. Here are a few examples:

Format	Explanation	Example
%d	Display an integer	10
%f	Displays a floating-point number in fixed decimal format	10.500000
.1f	Displays a floating-point number with 1 digit after the decimal	10.5
%e	Display a floating-point number in exponential (scientific notation)	1.050000e+01
%g	Display a floating-point number in either fixed decimal or exponential format depending on the size of the number (will not display trailing zeros)	10.5

getline6.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char** argv) {
6     char *line;
7     FILE *fp, *fpout;
8
9     line = malloc(sizeof(char)*BUFSIZ);
10
11    if ((fp = fopen(filename: argv[1], mode: "r")) == NULL) {
12        fprintf(stderr,"Error opening file %s\n", argv[1]);
13        exit(-1);
14    }
15    if ((fpout = fopen(filename: argv[2], mode: "w")) == NULL) {
16        fprintf(stderr,"Error opening file %s\n", argv[2]);
17        exit(-1);
18    }
19    while ( fscanf(fp, "%s", line) != EOF )
20        fprintf(fpout,"%ld: %s\n", strlen(line), line);
21
22    fclose(fp);
23    fclose(fpout);
24    return 0;
25 }
```



```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline6 getline6.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline6 test.txt output.txt
(base) mahmutunan@MacBook-Pro lecture15 % _
```



Example 2

- We will now write a program to read a comma separated file (“listing.csv”) and use the C structures to store and display the data on the console.
- The sample input file used is :

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	id	host_id	host_name	neighbourhood	neighbourhood_group	latitude	longitude	room_type	price	minimum_nights	number_of_reviews	calculated_host_listings_count	availability_365
2	2015	2217	Ian	Mitte	Brunnenstr. 9	52.5345373	13.4025569	Entire home/apt	60	4	118	4	141
3	2695	2986	Michael	Pankow	Prenzlauer Berg	52.5485128	13.4045528	Private room	17	2	6	1	0
4	3176	3718	Britta	Pankow	Prenzlauer Berg	52.5349962	13.4175787	Entire home/apt	90	62	143	1	220
5	3309	4108	Jana	Tempelhof - Schöneberg	Schöneberg	52.4988549	13.3490645	Private room	26	5	25	1	297
6	7071	17391	Bright	Pankow	Helmholtzplatz	52.5431573	13.4150911	Private room	42	2	197	1	26
7	9991	33852	Philipp	Pankow	Prenzlauer Berg	52.5330308	13.4160468	Entire home/apt	180	6	6	1	137
8	14325	55531	Chris + Olive	Pankow	Prenzlauer Berg	52.5478464	13.4055622	Entire home/apt	70	90	23	3	129
9	16401	59666	Melanie	Friedrichshain-Kreuzberg	Frankfurter Allee	52.510514	13.4578502	Private room	120	30	0	1	365
10	16644	64696	Rene	Friedrichshain-Kreuzberg	Neukölln	52.5047923	13.4351019	Entire home/apt	90	60	48	2	159
11	17409	67590	Wolfram	Pankow	Prenzlauer Berg	52.5290709	13.4128434	Private room	45	3	279	1	42
12	17904	68997	Matthias	Neukölln	Reuterstraße	52.4954763	13.4218213	Entire home/apt	49	5	223	1	232
13	20858	71331	Marc	Pankow	Prenzlauer Berg	52.5369524	13.407615	Entire home/apt	129	3	56	1	166
14	21869	64696	Rene	Friedrichshain-Kreuzberg	Neukölln	52.5027333	13.4346199	Entire home/apt	70	60	60	2	129
15	22415	86068	Kiki	Friedrichshain-Kreuzberg	Neukölln	52.4948506	13.4285006	Entire home/apt	98	3	61	2	257
16	22677	87357	Ramfis	Mitte	Brunnenstraße	52.5343484	13.4055765	Entire home/apt	160	3	223	1	228
17	23834	94918	Tanja	Friedrichshain-Kreuzberg	Tempelhofer Vorstadt	52.4897144	13.3797476	Entire home/apt	65	60	96	1	275
18	24569	99662	Dominik	Pankow	Prenzlauer Berg	52.5307909	13.4180844	Entire home/apt	90	3	18	2	3
19	25653	99662	Dominik	Pankow	Prenzlauer Berg	52.5302587	13.419467	Entire home/apt	90	4	5	2	15
20	26543	112675	Terri	Pankow	Helmholtzplatz	52.5440624	13.4213765	Entire home/apt	197	3	163	1	336
21	28156	55531	Chris + Olive	Pankow	Prenzlauer Berg	52.5467194	13.405117	Entire home/apt	70	90	28	3	191
22	28268	121580	Elena	Friedrichshain-Kreuzberg	Neukölln	52.5133852	13.4699475	Entire home/apt	90	5	30	1	55
23	28711	84157	Emanuela	Neukölln	Reuterstraße	52.4861061	13.434817	Entire home/apt	60	2	1	10	341
24	29279	54283	Marine	Pankow	Helmholtzplatz	52.5417876	13.4238832	Entire home/apt	130	60	69	3	221

listing.c

- The given file has 13 different attributes, and these attributes can be divided into three different datatypes: integer, character array, and float.
- And collectively we can create a C structure to represent these attributes and then create an array of such structures to store multiple entities.
- 1. Define a structure called listing with all attributes as individual members of the struct listing.

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

#define LINESIZE 1024

struct listing {
    int id, host_id, minimum_nights, number_of_reviews, calculated_host_listings_count, availability_365;
    char *host_name, *neighbourhood_group, *neighbourhood, *room_type;
    float latitude, longitude, price;
};
```

strtok

- `char *strtok(char *str, const char *delim);`
- `char *strtok_r(char *str, const char *delim, char **saveptr);`
- strtok, strtok_r - extract tokens from strings
- The `strtok()` function breaks a string into a sequence of zero or more nonempty tokens. On the first call to `strtok()`, the string to be parsed should be specified in `str`. In each subsequent call that should parse the same string, `str` must be `NULL`.

atoi

- atoi, atol, atoll - convert a string to an integer
- int atoi(const char **nptr*) ;
- long atol(const char **nptr*) ;
- long long atoll(const char **nptr*) ;
- Return value: The converted value or 0 on error.

<https://man7.org/linux/man-pages/man3/atoi.3.html>

atof

- **atof** - convert a string to a double

```
double atof(const char *nptr);
```

RETURN VALUE : The converted value

<https://man7.org/linux/man-pages/man3/atof.3.html>

strdup

- strdup- duplicate a string

```
char *strdup(const char *s);
```

On success, the strdup() function returns a pointer to the duplicated string. It returns NULL if insufficient memory was available, with errno set to indicate the cause of the error.

<https://www.man7.org/linux/man-pages/man3/strndup.3.html#:~:text=DESCRIPTION%20top,copies%20at%20most%20n%20bytes.>

listing.c

- 2. Define a function which can help to parse each line in the file and return the above defined structure.
- For this task you need to learn the string tokenizer function (*strtok*) which is available in *<string.h>* header file.
- You can find out more about the *strtok* function by typing *man strtok*.
- You will notice that when you invoke the *strtok* function for the first time you provide the pointer to the character array and on subsequent invocations of *strtok* we use *NULL* as the argument.

listing.c

```
15  struct listing getfields(char* line){
16      struct listing item;
17
18      item.id = atoi(strtok(line, ","));
19      item.host_id = atoi(strtok(NULL, ","));
20      item.host_name = strdup(strtok(NULL, ","));
21      item.neighbourhood_group = strdup(strtok(NULL, ","));
22      item.neighbourhood = strdup(strtok(NULL, ","));
23      item.latitude = atof(strtok(NULL, ","));
24      item.longitude = atof(strtok(NULL, ","));
25      item.room_type = strdup(strtok(NULL, ","));
26      item.price = atof(strtok(NULL, ","));
27      item.minimum_nights = atoi(strtok(NULL, ","));
28      item.number_of_reviews = atoi(strtok(NULL, ","));
29      item.calculated_host_listings_count = atoi(strtok(NULL, ","));
30      item.availability_365 = atoi(strtok(NULL, ","));
31
32      return item;
33 }
```

listing.c

Now, create a function to display the items in the struct

```
36 void displayStruct(struct listing item) {
37     printf("ID : %d\n", item.id);
38     printf("Host ID : %d\n", item.host_id);
39     printf("Host Name : %s\n", item.host_name);
40     printf("Neighbourhood Group : %s\n", item.neighbourhood_group);
41     printf("Neighbourhood : %s\n", item.neighbourhood);
42     printf("Latitude : %f\n", item.latitude);
43     printf("Longitude : %f\n", item.longitude);
44     printf("Room Type : %s\n", item.room_type);
45     printf("Price : %f\n", item.price);
46     printf("Minimum Nights : %d\n", item.minimum_nights);
47     printf("Number of Reviews : %d\n", item.number_of_reviews);
48     printf("Calculated Host Listings Count : %d\n", item.calculated_host_listings_count);
49     printf("Availability_365 : %d\n\n", item.availability_365);
50 }
```

listing.c

- 3. Now use *fopen* function to open file in read only mode. Notice that the *fopen* function returns a pointer of *FILE* type (file pointer) unlike the *open* function that returns an *integer* value as the file descriptor.

```
52     int main(int argc, char* args[]) {
53         struct listing list_items[22555];
54         char line[LINESIZE];
55         int i, count;
56
57         FILE *fptr = fopen("listings.csv", "r");
58         if(fptr == NULL){
59             printf("Error reading input file listings.csv\n");
60             exit (-1);
61     }
```

listing.c

- 4. Then loop through till the end of file (use fgets function) and store all data in the array of structures
- We can close the file using fclose function

```
62
63     count = 0;
64     while (fgets(line, LINESIZE, fptr) != NULL){
65         list_items[count++] = getfields(line);
66     }
67     fclose(fptr);
```

listing.c

- 5. Now invoke the function to display the structure in a loop.

```
69     for (i=0; i<count; i++)
70         displayStruct(list_items[i]);
71
72     return 0;
73 }
74
```

Process Management

Process

- Fundamental to the structure of operating systems

A *process* can be defined as:

A program in execution

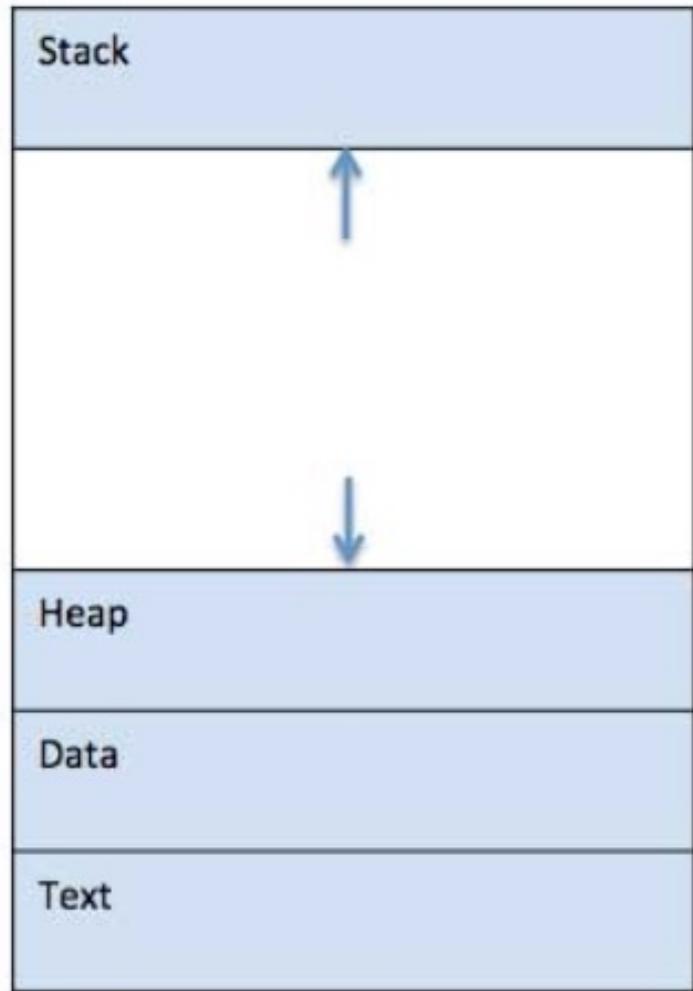
An instance of a running program

The entity that can be assigned to, and executed on, a processor

A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

Process

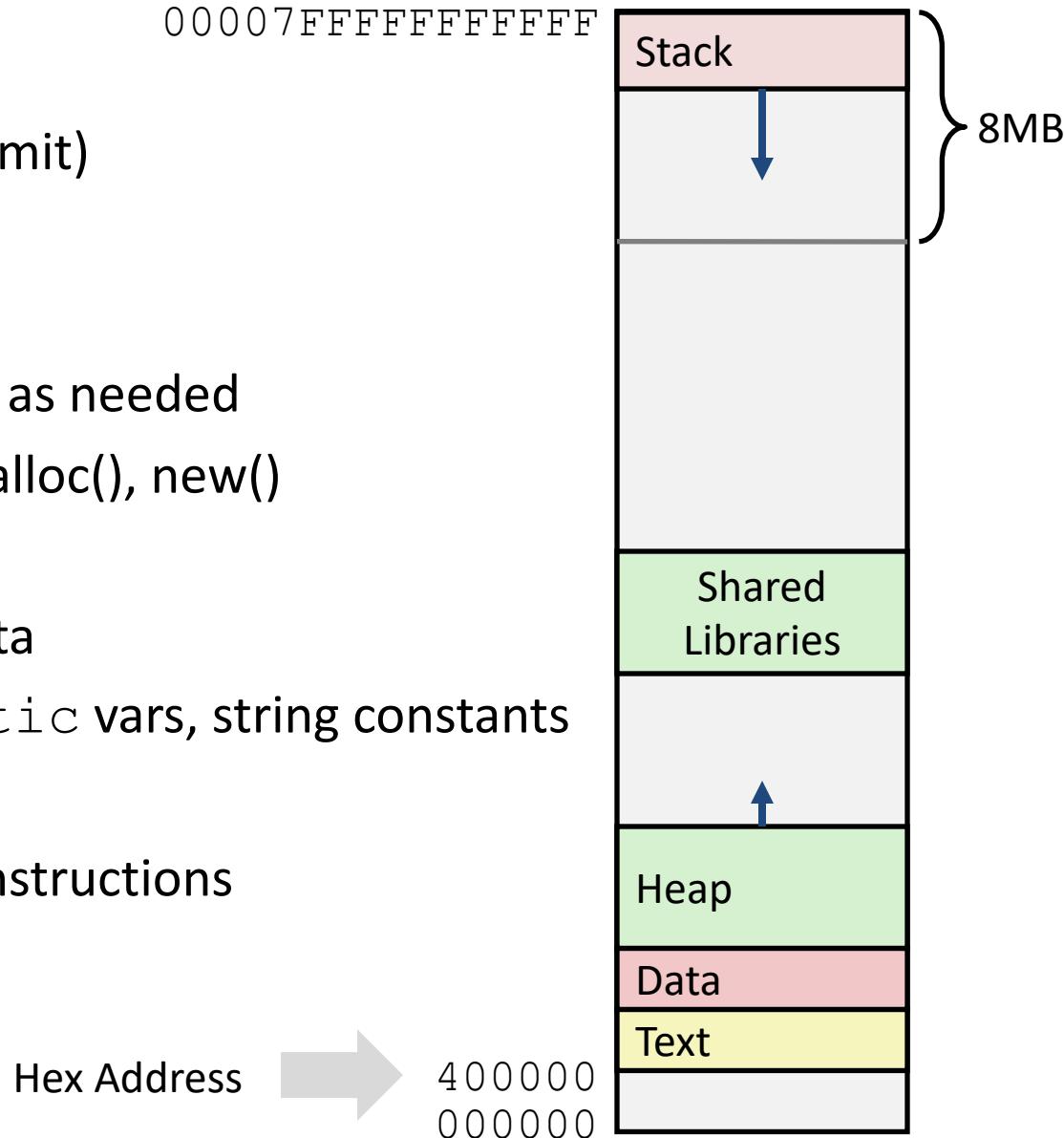
- When a program is loaded into the memory and it becomes a process, it can be divided into four sections
 - stack
 - heap
 - text
 - data.



x86-64 Linux Memory Layout

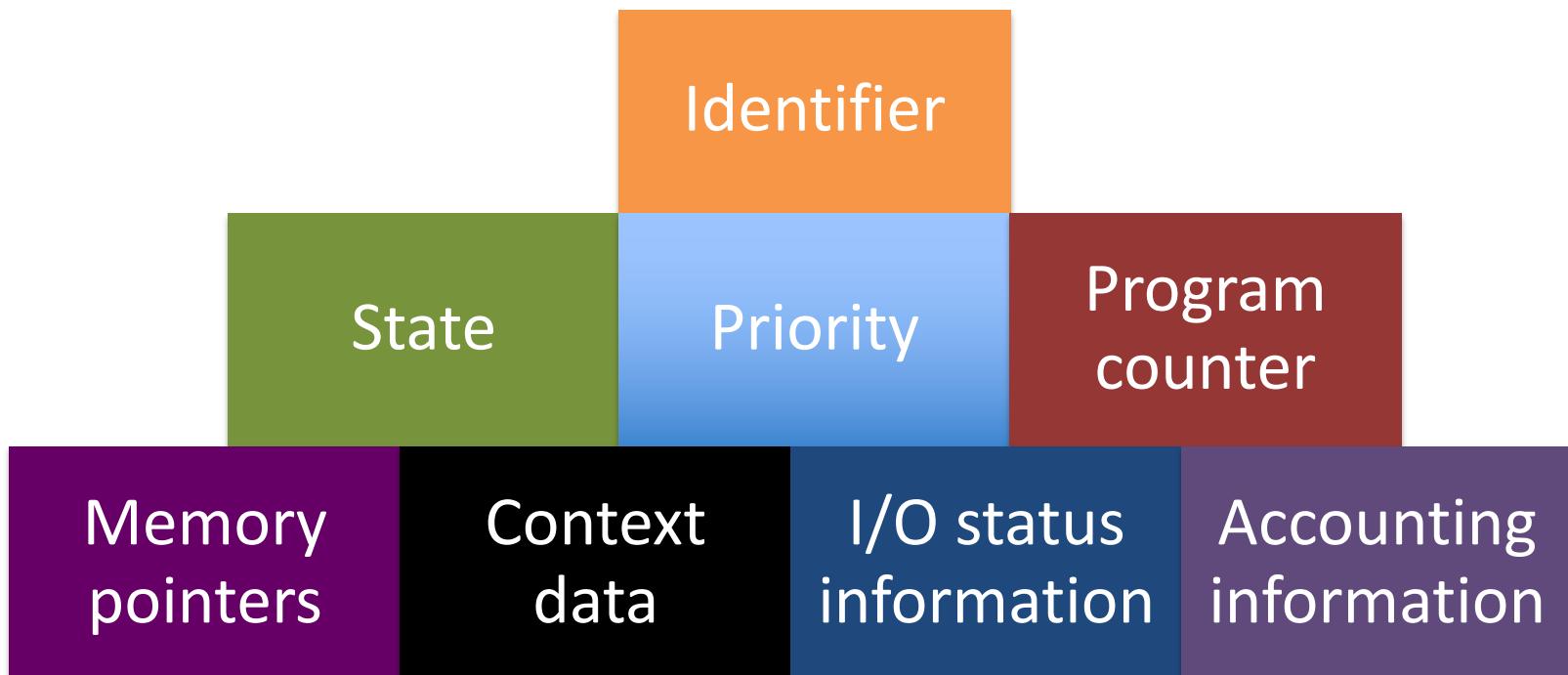
not drawn to scale

- Stack
 - Runtime stack (8MB limit)
 - E. g., local variables
- Heap
 - Dynamically allocated as needed
 - When call `malloc()`, `calloc()`, `new()`
- Data
 - Statically allocated data
 - E.g., global vars, static vars, string constants
- Text / Shared Libraries
 - Executable machine instructions
 - Read-only



Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements, including:



Process Execution

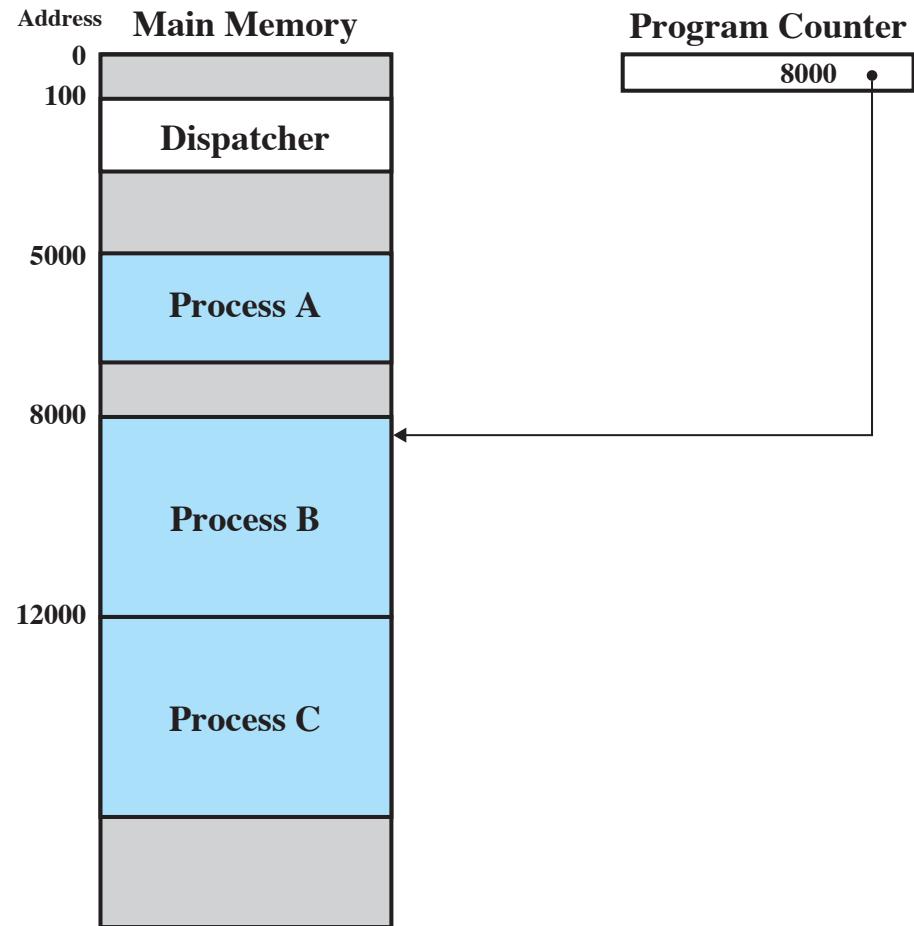


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A

(b) Trace of Process B

(c) Trace of Process C

5000 = Starting address of program of Process A

8000 = Starting address of program of Process B

12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

1	5000		
2	5001		
3	5002		
4	5003		
5	5004		
6	5005		
		----- Timeout	
7	100		
8	101		
9	102		
10	103		
11	104		
12	105		
13	8000		
14	8001		
15	8002		
16	8003		
		----- I/O Request	
17	100		
18	101		
19	102		
20	103		
21	104		
22	105		
23	12000		
24	12001		
25	12002		
26	12003		
		----- Timeout	
27	12004		
28	12005		
		----- Timeout	
29	100		
30	101		
31	102		
32	103		
33	104		
34	105		
35	5006		
36	5007		
37	5008		
38	5009		
39	5010		
40	5011		
		----- Timeout	
41	100		
42	101		
43	102		
44	103		
45	104		
46	105		
47	12006		
48	12007		
49	12008		
50	12009		
51	12010		
52	12011		
		----- Timeout	

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Process Creation

*Process
spawning*

- When the OS creates a process at the explicit request of another process

*Parent
process*

- Is the original, creating, process

Child process

- Is the new process

Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

Five-State Process Model

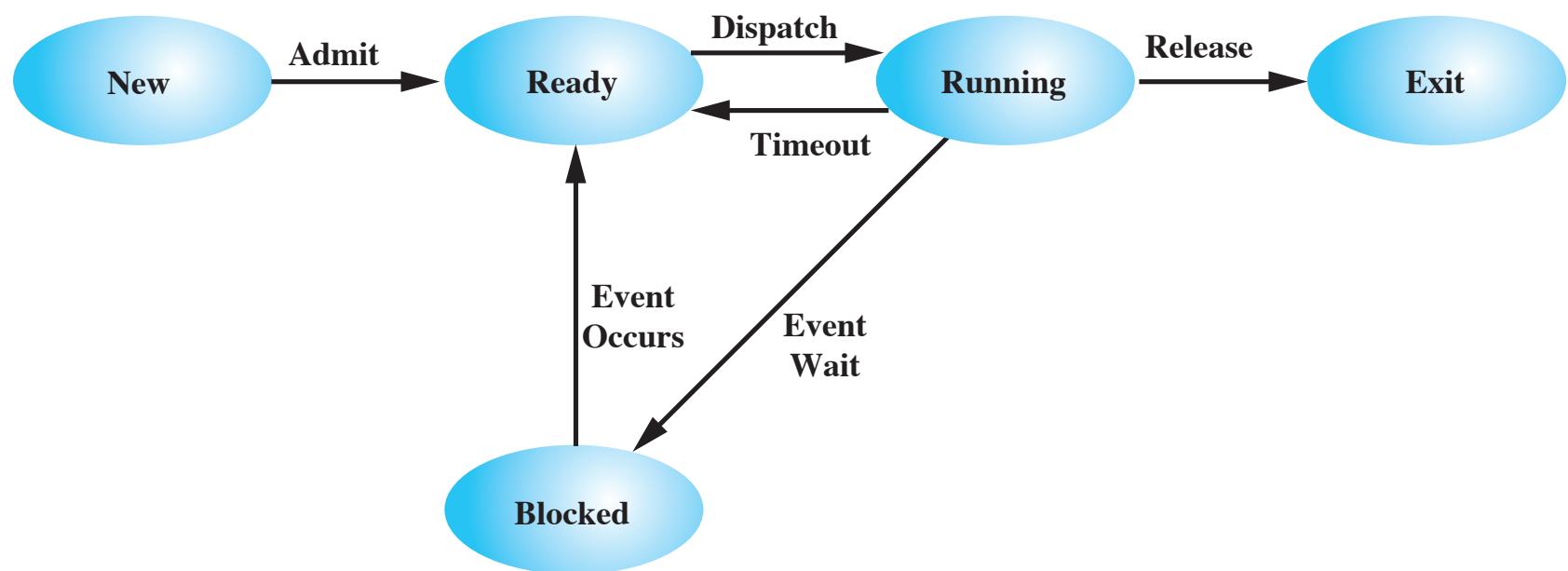


Figure 3.6 Five-State Process Model

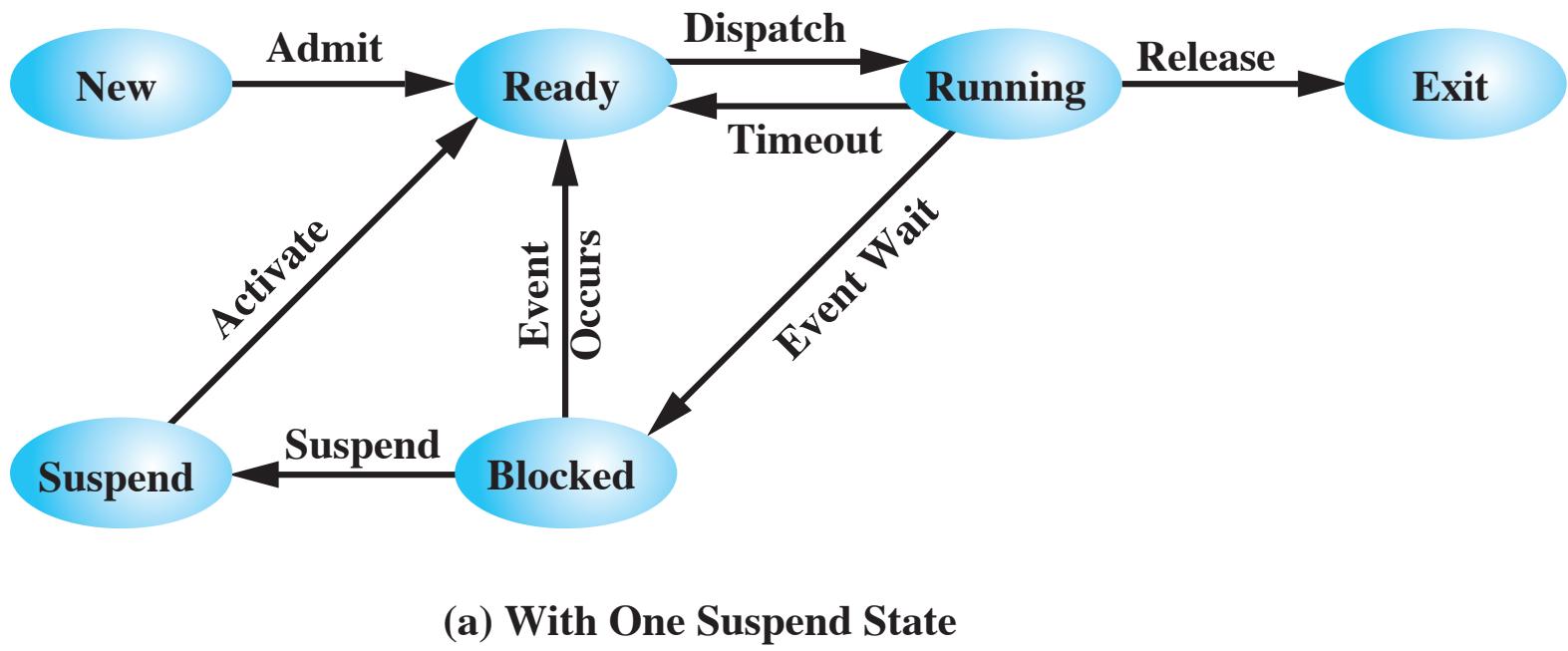
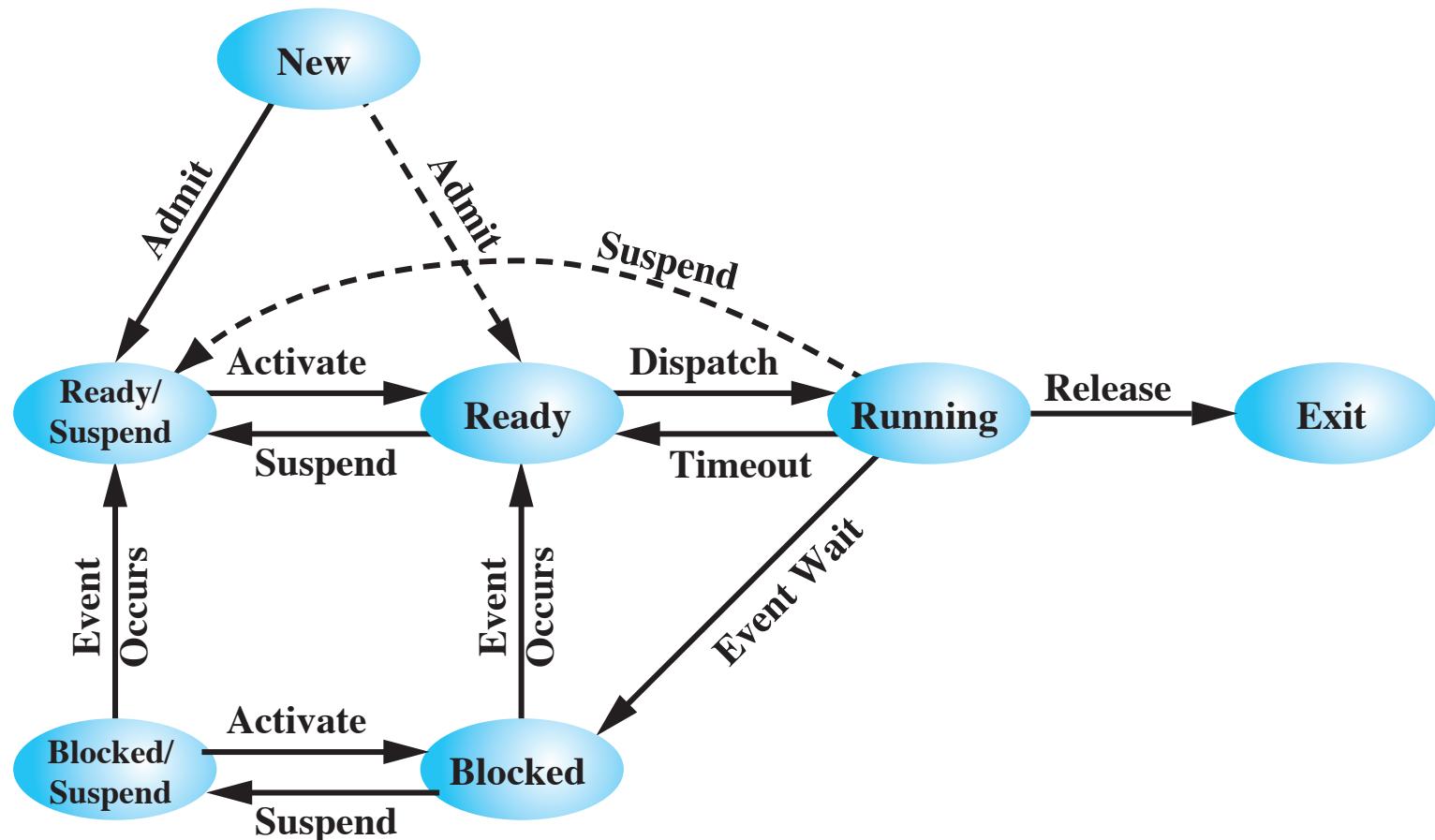


Figure 3.9 Process State Transition Diagram with Suspend States



(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

Modes of Execution

User Mode

- Less-privileged mode
- User programs typically execute in this mode

System Mode

- More-privileged mode
- Also referred to as control mode or kernel mode
- Kernel of the operating system

Unix Processes

- The OS tracks processes through a five-digit ID number
 - **pid** or **process ID.**
- Each process in the system has a unique **pid**.
- If you want to list the running processes, use the **ps** (process status) command
 - to display the full option

`ps -f`

UID, PID, PPID, C, STIME, TTY, CMD, TIME.....

Unix Processes

- to stop a process
 - kill
 - kill PID
 - kill -9 PID
- Init Process
 - PID 1 and PPID 0
- Foreground Process
- Background Process

fork()

- fork - create a child process

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

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

fork()

- We can use the fork() system call to create a new process (referred to as the child process) which is an identical image of the calling process (referred to as the parent process).
- Here is the C interface for the fork() system call:

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

fork()

- If the fork() call is successful, then it returns the process ID of the child process to the parent process and returns 0 in the child process.
- fork() returns a negative value in the parent process and sets the corresponding errno variable (external variable defined in *errno.h*) if there is any error in process creation and the child process is not created.
- We can use perror() function (defined in *stdio.h*) to print the corresponding system error message. Look at the man page for perror to find out more about the perror() function.

fork()

- Once the parent process creates the child process, the parent process continues with its normal execution.
- If the parent process exits before the child process completes its execution and terminates, the child process will become a zombie process (*i.e.*, a process without a parent process).
- Alternatively, the parent process could wait for the child process to terminate using the `wait()` function.
- The `wait()` system call will suspend the execution of the calling process until one of the child process terminates and if there are no child processes available the `wait()` function returns immediately.

Process Control

- Process creation is by means of the kernel system call, *fork ()*
- When a process issues a fork request, the OS performs the following functions:
 - 1 • Allocates a slot in the process table for the new process
 - 2 • Assigns a unique process ID to the child process
 - 3 • Makes a copy of the process image of the parent, with the exception of any shared memory
 - 4 • Increments counters for any files owned by the parent, to reflect that an additional process now also owns those files
 - 5 • Assigns the child process to the Ready to Run state
 - 6 • Returns the ID number of the child to the parent process, and a 0 value to the child process

Hello World!

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

Hello, World!

Hello World!

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

Hello, World!

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

Hello, World!
Hello, World!

Hello World!

```
int main() {  
    int processId = fork();  
    printf("Hello, World! from= %d\n", processId);  
    return 0;  
}
```

```
Hello, World! from= 96037  
Hello, World! from= 0
```

```
int main() {  
    printf("This is before the fork statement\n");  
    fork();  
    printf("After the FIRST fork\n");  
    fork();  
    printf("After the SECOND fork \n");  
    fork();  
    printf("After the THIRD fork \n");  
    return 0;  
}
```

```
After the FIRST fork  
After the SECOND fork  
After the FIRST fork  
After the THIRD fork  
After the SECOND fork  
After the SECOND fork  
After the SECOND fork  
After the THIRD fork
```

```
int main() {  
  
    fork();  
    fork();  
    fork();  
    fork();  
    printf("4 forks will work 16 times\n");  
    return 0;  
}
```

```
4 forks will work 16 times  
4 forks will work 16 times
```

getpid()

```
int main() {  
  
    printf("before calling the fork %d\n",getpid());  
    fork();  
    printf("after calling the FIRST fork %d\n",getpid());  
    fork();  
    printf("after calling the SECOND fork %d\n",getpid());  
}
```

```
before calling the fork 96717  
after calling the FIRST fork 96717  
after calling the SECOND fork 96717  
after calling the FIRST fork 96718|  
after calling the SECOND fork 96719  
after calling the SECOND fork 96718  
after calling the SECOND fork 96720
```

wait()

wait, waitpid, waitid - wait for process to change state

```
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int
options);
int waitid(idtype_t idtype, id_t id, siginfo_t
*infop, int options);
```

<https://www.man7.org/linux/man-pages/man2/waitid.2.html>

wait()

- The wait() call returns the PID of the child process that terminated when successful, otherwise, it returns -1.
- The wait() call also sets an integer value that is passed as an argument to the function which can be inspected with various macros provided in <sys/wait.h> to determine how the child process completed (e.g., terminated normally, terminated by a signal).

`wait()` `waitpid()`

- If the calling process created more than one child process, we can use the `waitpid()` system call to wait on a specific child process to change state.
- A state change could be any one of the following events: the child was terminated; the child was stopped by a signal; or the child was resumed by a signal. Similar to `wait()`, `waitpid()` returns the PID of the child process that changed state when successful, otherwise, it returns -1.

wait() waitpid()

- Here are the C APIs for the `wait()` and `waitpid()` system calls:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int
options);
```

```
int main() {
    int processId = fork();
    int count;
    fflush(stdout);

    if (processId==0){
        count=1;
    }else{
        count=6;
    }
    if (processId!=0){
        wait();
    }
    int i;
    for (i=count;i<count+5;i++){
        printf("%d",i);
        fflush( stdout );
    }
}
```

12345678910

Process finished with exit code 0

Example 1

- We will create a sample program to illustrate how to use fork() to create a child process, wait for the child process to terminate, and display the parent and child process ID in both processes.

fork.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     int status;
10
11     pid = fork();
12     if (pid == 0) { /* this is child process */
13         printf("This is the child process, my PID is %ld and my parent PID is %ld\n",
14                (long)getpid(), (long)getppid());
15     } else if (pid > 0) { /* this is the parent process */
16         printf("This is the parent process, my PID is %ld and the child PID is %ld\n",
17                (long)getpid(), (long)pid);
18
19         printf("Wait for the child process to terminate\n");
20 }
```

fork.c

```
18     printf("Wait for the child process to terminate\n");
19     wait(&status); /* wait for the child process to terminate */
20     if (WIFEXITED(status)) { /* child process terminated normally */
21         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
22     } else { /* child process did not terminate normally */
23         printf("ERROR: Child process did not terminate normally!\n");
24         /* look at the man page for wait (man 2 wait) to
25            determine how the child process was terminated */
26     }
27 }
28 } else { /* we have an error in process creation */
29     perror("fork");
30     exit(EXIT_FAILURE);
31 }
32
33 printf("[%ld]: Exiting program .....%n", (long)getpid());
34
35 return 0;
36 }
37 }
```

fork.c

```
[base] mahmutunan@MacBook-Pro lecture17 % ./exercise1
This is the parent process, my PID is 90695 and the child PID is 90696
Wait for the child process to terminate
This is the child process, my PID is 90696 and my parent PID is 90695
[90696]: Exiting program .....
Child process exited with status = 0
[90695]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture17 % █
```

exec()

- execl, execlp, execle, execv, execvp, execvpe - execute a file
- The **exec()** family of functions replaces the current process image with a new process image.

exec()

- Note that the child process is a copy of the parent process and control is split at the invocation of the fork() call between the parent and the child process.
- If we like the child process to execute a different program other than making a copy of the parent process, we can use the exec family of system calls to replace the current process image with a new one.

- Here is the C APIs for the exec family of system calls:

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

```
int execl(const char *pathname, const char
          *arg, ...);
int execlp(const char *filename, const char
           *arg, ...);
int execle(const char *pathname, const char
           *arg, ..., char * const envp[]);
int execv(const char *pathname, char *const
          argv[]);
int execvp(const char *filename, char *const
          argv[]);
int execvpe(const char *filename, char *const
            argv[], char *const envp[]);
```

- We will use the `execl()` to replace the child process created by `fork()`.
- The `execl()` function takes as arguments the full pathname of the executable along with a pointer to an array of characters for each argument.
- Since we can have a variable number of arguments, the last argument is a null pointer.

p1.c

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv) {
    printf("Hello from p1, process id= () %d\n", getpid());
    char *args[]={ "Hello", "CS", "332", NULL};
    execv( path: "./p2", args);
    printf("we are not supposed to see this text");
    return 0;
}
```

p2

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

int main(int argc, char **argv) {
    printf("Hello from p2, process id= () %d\n", getpid());
    printf("The arguments are %s %s %s\n", argv[0], argv[1], argv[2]);
    printf("Now, the child process will terminate\n");
    return 0;
}
```

compile & run

```
[base] mahmutunan@MacBook-Pro lecture18 % gcc -Wall p1.c -o p1
[base] mahmutunan@MacBook-Pro lecture18 % gcc -Wall p2.c -o p2
[base] mahmutunan@MacBook-Pro lecture18 % ./p1
Hello from p1, process id= () 30983
Hello from p2, process id= () 30983
The arguments are Hello CS 332
Now, the child process will terminate
(base) mahmutunan@MacBook-Pro lecture18 %
```

p1.c

- Let's modify it a little bit and fork the process

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char **argv) {
    printf("Hello from p1, process id= () %d\n", getpid());
    int pid=fork();
    int status;
    if (pid == 0) {
        char *args[] = {"Hello", "CS", "332", NULL};
        execv( path: "./p2", args);
        printf("we are not supposed to see this text");
    }
    else if(pid>0){
        wait(&status);
    }
    return 0;
}
```

p2.c

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

int main(int argc, char **argv) {
    printf("Hello from p2, process id= () %d\n", getpid());
    printf("The arguments are %s %s %s\n", argv[0], argv[1], argv[2]);
    printf("Now, the child process will terminate\n");
    return 0;
}
```

compile&run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p1.c -o p1
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p2.c -o p2
(base) mahmutunan@MacBook-Pro lecture18 % ./p1
Hello from p1, process id= () 30922
Hello from p2, process id= () 30923
The arguments are Hello CS 332
Now, the child process will terminate
(base) mahmutunan@MacBook-Pro lecture18 %
```

forkexec.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char **argv) {
8      pid_t pid;
9      int status;
10
11      pid = fork();
```

```

12     if (pid == 0) { /* this is child process */
13         execl( path: "/usr/bin/uname", arg0: "uname", "-a", (char *)NULL);
14         printf("If you see this statement then execl failed ;-(\n");
15         perror("execl");
16         exit(-1);
17     } else if (pid > 0) { /* this is the parent process */
18         printf("Wait for the child process to terminate\n");
19         wait(&status); /* wait for the child process to terminate */
20         if (WIFEXITED(status)) { /* child process terminated normally */
21             printf("Child process exited with status = %d\n", WEXITSTATUS(status));
22         } else { /* child process did not terminate normally */
23             printf("Child process did not terminate normally!\n");
24             /* look at the man page for wait (man 2 wait) to determine
25              how the child process was terminated */
26         }
27     } else { /* we have an error */
28         perror("fork"); /* use perror to print the system error message */
29         exit(EXIT_FAILURE);
30     }
31
32     printf("[%ld]: Exiting program .....\\n", (long)getpid());
33
34     return 0;
35 }
```

compile & run

```
[base] mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexec1.c -o exercise1
[base] mahmutunan@MacBook-Pro lecture18 % ./exercise1
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[1290]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

- Let us look at the different versions of the exec functions. There are two classes of exec functions based on whether the argument is a list of separate values (l versions) or the argument is a vector (v versions):
- functions that take a variable number of command-lines arguments each as an array of characters terminated with a null character and the last argument is a null pointer –
*(char *)NULL (execl, execvp, and execle)*
- functions that take the command-line arguments as a pointer to an array of pointers to the arguments, similar to argv parameter used by the main method (execv, execvp, and execvpe)

- Functions that have *p* in the name use *filename* as the first argument while functions without *p* use the *pathname* as the first argument. If the filename contains a slash character (/), it is considered as a pathname, otherwise, all directories specified by the PATH environment variable are searched for the executable.
- Functions that end in *e* have an additional argument – a pointer to an array of pointers to the environment strings.

forkexecv.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6
7 int main(int argc, char **argv) {
8     pid_t pid;
9     int status;
10    char *args[] = {"uname", "-a", (char *)NULL};
11
12    pid = fork();
13    if (pid == 0) { /* this is child process */
14        execv( path: "/usr/bin/uname", args);
15        printf("If you see this statement then exec failed ;-(\n");
16        perror("execv");
17        exit(-1);
18    } else if (pid > 0) { /* this is the parent process */
```

forkexecv.c

```
18     } else if (pid > 0) { /* this is the parent process */
19         printf("Wait for the child process to terminate\n");
20         wait(&status); /* wait for the child process to terminate */
21         if (WIFEXITED(status)) { /* child process terminated normally */
22             printf("Child process exited with status = %d\n", WEXITSTATUS(status));
23         } else { /* child process did not terminate normally */
24             printf("Child process did not terminate normally!\n");
25             /* look at the man page for wait (man 2 wait) to determine
26              how the child process was terminated */
27         }
28     } else { /* we have an error */
29         perror("fork"); /* use perror to print the system error message */
30         exit(EXIT_FAILURE);
31     }
32
33     printf("[%ld]: Exiting program .....\\n", (long)getpid());
34
35     return 0;
36 }
37 }
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecv.c -o exercise2
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise2
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[30193]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

In order to see the difference between execl and execv, here is a line of code executing a

```
ls -l -R -a
```

with execl :

```
execl("/bin/ls", "ls", "-l", "-R", "-a", NULL);
```

with execv :

```
char* arr[] = {"ls", "-l", "-R", "-a", NULL};  
execv("/bin/ls", arr);
```

The array of char* sent to execv will be passed to /bin/ls as argv
(in `int main(int argc, char **argv)`)

forkexecvp.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  ▶ int main(int argc, char **argv) {
8      pid_t pid;
9      int status;
10
11     if (argc < 2) {
12         printf("Usage: %s <command> [args]\n", argv[0]);
13         exit(-1);
14     }
15
16     pid = fork();
17     if (pid == 0) { /* this is child process */
18         execvp(argv[1], &argv[1]);
19         printf("If you see this statement then exec failed ;-(\n");
20         perror("execvp");
21         exit(-1);
```

forkexecvp.c

```
22 } else if (pid > 0) { /* this is the parent process */
23     printf("Wait for the child process to terminate\n");
24     wait(&status); /* wait for the child process to terminate */
25     if (WIFEXITED(status)) { /* child process terminated normally */
26         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
27     } else { /* child process did not terminate normally */
28         printf("Child process did not terminate normally!\n");
29         /* look at the man page for wait (man 2 wait) to determine
30            how the child process was terminated */
31     }
32 } else { /* we have an error */
33     perror("fork"); /* use perror to print the system error message */
34     exit(EXIT_FAILURE);
35 }
36
37 printf("[%ld]: Exiting program .....%n", (long)getpid());
38
39 return 0;
40 }
41 }
```

hello.c

```
#include <stdio.h>
int main(int argc, char **argv) {
    printf("Hello from the execvp()\n");
    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecvp.c -o exercise3
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall hello.c -o hello
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise3 ./hello
Wait for the child process to terminate
Hello from the execvp()
Child process exited with status = 0
[30387]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

forkexecvp2.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char **argv) {
8      pid_t pid;
9      int status;
10
11     if (argc < 2) {
12         printf("Usage: %s <command> [args]\n", argv[0]);
13         exit(-1);
14     }
15
16     pid = fork();
17     if (pid == 0) { /* this is child process */
18         execvp(argv[1], &argv[1]);
19         printf("If you see this statement then exec failed ;-(\n");
20         perror("execvp");
21         exit(-1);
```

```
22 } else if (pid > 0) { /* this is the parent process */
23     char name[BUFSIZ];
24
25     printf("[%d]: Please enter your name: ", getpid());
26     scanf("%s", name);
27     printf("[%d-stdout]: Hello %s!\n", getpid(), name);
28     fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);
29
30     wait(&status); /* wait for the child process to terminate */
31     if (WIFEXITED(status)) { /* child process terminated normally */
32         printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33     } else { /* child process did not terminate normally */
34         printf("Child process did not terminate normally!\n");
35         /* look at the man page for wait (man 2 wait) to determine
36            how the child process was terminated */
37     }
38 } else { /* we have an error */
39     perror("fork"); /* use perror to print the system error message */
40     exit(EXIT_FAILURE);
41 }
42
43 printf("[%ld]: Exiting program .....%n", (long)getpid());
44
45 return 0;
46 }
47 }
```