❖ Pointers Introduction

- A pointer is a derived data type.
- Pointers contain memory addresses as their values.
- Pointers can be used to access and manipulate data stored in the memory.

❖ Benefits of pointers

1) Efficient in handling array.
2) Can be used to return multiple values from a function.
3) Pointers permit references to functions and i.e. it supports passing of functions as arguments to other functions.
4) Helps in saving of data storage space in memory.
5) Supports dynamic memory management.
6) Efficient tool for manipulating data structures like structure, stack and queue.
7) Reduce length and complexity of programs.
8) Increase the execution speed and program execution time.

❖ Understanding pointers

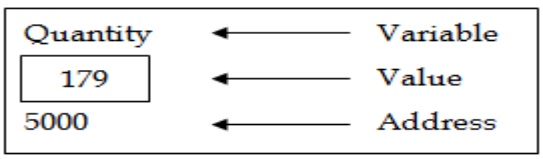
- The computer's memory is a sequential collection of storage cells as shown in the figure.

Memory cell	Address
	0
	1
	2
	3
	.
	65535

- Each cell (byte) has a number called address associated with it.
- The addresses are numbered consecutively, starting from zero.
- The last address depends on the memory size.
- Whenever we declare a variable, the system allocates an appropriate location somewhere in the memory, to hold the value of the variable.
- Since every byte has a unique address number, this location will have its own address number.
- Consider the following statement :

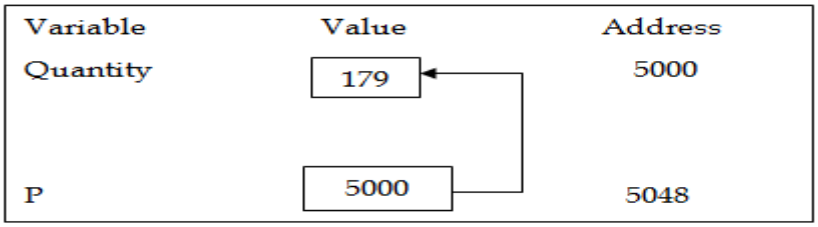
```
int quality  = 179;
```

- This statement instructs the system to find a location for the integer variable quantity and puts the value 179 in that location.
- Suppose the system has chosen the address location 5000 for quantity.



- During the program execution, the system always associates the name quantity with the address 5000.
- We can access the value 179 by using either the name quantity or the address 5000.
- **A pointer variable is a variable that contains an address of another variable.**

- Suppose, we assign the address of quantity to a variable p.
- The link between the variables p and quantity can be visualized as shown below.
- The address of p is 5048.



- Since the value of the variable p is the address of the variable quantity, we may access the value of quantity by using the value of p.
- So we say that the variable p points to the variable quantity.
- Thus p gets the name pointer.
- Pointer Constants: Memory addresses within a computer.
- Pointer value: Address of a variable.
- Pointer variable: The variable that contains a pointer value (address of a variable),

❖ Declaring pointer variables

- Syntax :

```
data_type *pt_name ;
```

1) The asterisk (*) tells that the variable pt_name is pointer variable.
2) pt_name needs a memory location.
3) pt_name points to a variable of type data_type.

- For example,

```
int *p;
```

declares the variable p as a pointer variable that points to an integer data type.

- Similarly the statement,

```
float *x;
```

declares x as a pointer to a floating point variable.

- The declarations cause the compiler to allocate memory locations for the pointer variables p and x.
- Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations.

❖ Pointer Declaration Style

- There are three different style :

```
1) int* p ;        // style1
2) int  *p ;       // style2
3) int * p ;       // style3
```

- Style 2 is preferable because of two reasons:

1) Easy to declare multiple variables in same statement.

```
int *p, x, *q ;
```

2) Easy to access target values.

```
int *p, x, *q ;
x = 10 ;
p = &x ;
y = *p ;
```

```
*p = 20 ;
```

❖ Initialization of pointer variables

- The process of assigning the address of a variable to a pointer variable is known as initialization.
- All uninitialized pointers will have some unknown values that will be interpreted as memory addresses.
- They may not be valid addresses or they may point to some values that are wrong.
- Since the compilers do not detect these errors, the program will produce wrong results.
- Thus it is important to initialize pointer variables before they are used in the program.
- Once a pointer variable has been declared we can use the assignment operator to initialize the variable as follows.

```
int quantity ;
int *p ;          // declaration
p = &quantity ;          // initialization
```

- We can also combine the initialization with declaration.

```
int *p = &quantity ;
```

- We must ensure that the pointer variables always point to the corresponding type of data.
- For example, the following will result in wrong output.

```
float a,b ;
int x, *p ;
p = &a ;          // illegal
b = *p ;
```

- When we declare a pointer to be of int type, the system assumes that any address that the pointer will hold will point to an integer variable.
- It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.

```
int x,*p = &x ;
```

- The following statement is invalid.

```
int *p = &x,  x ;
```

- We can also define a pointer variable with an initial value of NULL or 0.

```
int *p = NULL ;
int *p = 0;
```

- With the exception of NULL and 0, no other constant value can be assigned to a pointer variable.
- The following statement is invalid.

```
int * p = 1234 ;
```

❖ Accessing the Address of a variable

- The actual location of a variable in the memory is system dependent
- So the address of a variable is not known to us immediately.
- To know the address of a variable the operator & is used.
- The operator & returns the address of the variable associated with it.
- For example, the statement would assign the address 5000 to the variable p.

```
p = &quantity ;
```

- The & operator can be used only with a simple variable or an array element.
- The following are illegal use of address operator:
  1) &125 (pointing at constants)
  2) int x[10];
     &x (pointing at array names)
  3) &(x + y) (pointing at expressions)
- If x is an array, then expressions such as

  &x[0] and &x[i + 3]

  are valid and represent the address of 0^{th} and (i + 3)th elements of x.

❖ Accessing a variable through its pointer

- Once a pointer has been assigned the address of a variable, the value of the variable can be accessed using the unary operator *.
- Operator * is called indirection operator or dereferencing operator.
- Consider the following statements:

```
int quantity, *p, n ;
quantity = 179 ;
p = &quantity ;
n = *p ;
```

- When the operator * is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable.
- In this case *p returns the value of the variable quantity, because p is the address of quantity.
- The * means ‘value at address’. Thus the value of n would be 179.
- The two statements

```
p = &quantity ;
n = *p ;
```

  are equivalent to

```
n = * &quantity ;
```

  which in turn is equivalent to n = quantity ;
- The statement

  ptr = &x assigns the address of x to ptr

  y = *ptr assigns the value pointed to by the pointer ptr to y.

  *ptr = 25 assigns the value 25 at the memory location pointed to by ptr.

```
void main()
{
        int x =10 , y ;
        int  *ptr = &x ;
        y = *ptr ;
        printf ("\n Value of x is = %d",x) ;
        printf ("\n %d is stored at address %u", x, &x) ;
        printf ("\n %d is stored at address %u", *&x, &x) ;
        printf ("\n %d is stored at address %u", *ptr, ptr) ;
        printf ("\n %d is stored at address %u", ptr, &ptr) ;
        printf ("\n %d is stored at address %u", y, &y) ;
        *ptr = 25 ;
        printf ("\n Now x = %d", x) ;
}
```

Output :

```
Value of x = 10
10 is stored at address 4104
10 is stored at address 4104
10 is stored at address 4104
4104 is stored at address 4106
10 is stored at address 4108
Now x = 25
```

❖ Pointer Expressions

- Pointers can be used in expressions.
- Suppose p1 and p2 are properly declared and initialized pointers.
- So the following statements are valid.

```
y = *p1 *  *p2 ;          // (*p1) * (*p2)
sum = sum + *p1 ;
z = 5* – *p2 / *p1 ;      // (5 * ( –(*p2) ) ) / (*p1)
```

- C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another.
- p1 + 4, p2 – 2 and p1 – p2 are allowed.
- If p1 and p2 are both pointers to the same array, then p2 – p1 gives the number of elements between p1 and p2.
- We may also use short-hand operators with the pointers.

```
p1++ ;
--p2 ;
sum += p2 ;
```

- In addition to arithmetic operations, pointers can also be compared using the relational operators.
- The expressions such as p1 > p2, p1 == p2 and p1 != p2 are allowed.
- However, any comparisons of pointers that refer to separate and unrelated variables make no sense.

- We may not use pointers in division or multiplication.
- For example, expressions such as p1 /p2 or p1 * p2 or p1/ 3 are not allowed.
- Similarly, two pointers cannot be added. That is p1 + p2 is illegal.

```
void main()
{
        int a, b, *p1, *p2, x, y, z ;
        a = 12 ;
        b = 4 ;
        p1 = &a ;
        p2 = &b ;
        x = *p1 *  *p2 – 6 ;
        y = 4* – *p2 / *p1 + 10 ;
        printf("\n Address of a = %u", p1) ;
        printf("\n Address of b = %u", p2) ;
        printf("\n a = %d", a) ;
        printf("\n b = %d", b) ;
        printf("\n x = %d", x) ;
        printf("\n y = %d", y) ;
        *p2 = *p2 + 3 ;
        *p1 = *p2 – 5 ;
        z = *p1 * *p2 – 6 ;
        printf("\n a = %d", a) ;
        printf("\n b = %d", b) ;
        printf("\n z = %d", z) ;
}
```

Output:

```
Address of a = 4020
Address of b = 4016
a = 12
b = 4
x = 42
y = 9
a = 2
b = 7
z = 8
```

❖ Pointer Increments and Scale factor

- Pointers can be incremented like

```
p1 = p1 + 2 ;
p2 = p1 + 1 ;
```

- However, an expression like

```
p1++ ;
```

  will cause the pointer p1 to point to the next value of its type.
  For example, if p1 is an integer pointer with an initial value say 2800, then after the operation p1 = p1 + 1, the value of p1 will be 2802 and not 2801.
- That is, when we increment a pointer, its value is increased by the length of the data type that it points to.
- **This length called the scale factor.**
- The numbers of bytes used to store various data types depends on the system and can be found by making use of the sizeof operator.
- For example, if x is a variable then sizeof(x) returns the number of bytes needed for the variable.
- The length of various data types are as follows:

➢

Data Type	Scale Factor
characters	1 byte
integers	2 bytes
floats	4 bytes
long integers	4 bytes
doubles	8 bytes

❖ Pointers and Arrays

- When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.
- The base address is the location of the first element (index 0) of the array.
- The compiler also defines the name array name as a constant pointer to the first element.
- Suppose we declare an array x as follows:

  int x[5] = {1, 2, 3, 4, 5} ;
- Suppose the base address of x is 1000, the five elements will be stored as follows:

Elements →	X[0]	X[1]	X[2]	X[3]	X[4]
Value →	1	2	3	4	5
Address →	1000	1002	1004	1006	1008

- The name x is defined as a constant pointer pointing to the first element, x[0] and therefore the value of x is 1000, the location where x[0] is stored. That is,

  x = &x[0] = 1000
- If we declare p as an integer pointer , then we can make the pointer p to point to the array x by the following assignment:

  p = x ;
- This is equivalent to

  p = &x[0] ;
- Now, we can access every value of x using p++ to move from one element to another.
- The relationship between p and x is shown as:

  p = &x[0] // (=1000)
  p + 1 = &x[1] // (=1002)
  p + 2 = &x[2] // (=1004)
  p + 3 = &x[3] // (=1006)
  p + 4 = &x[4] // (=1008)
- The address of an element is calculated using its index and the scale factor of the data type. For example,

  address of x[3] = base address + (3 * scale factor of int)
  = 1000 + (3 * 2) = 1006
- When handling arrays, instead of using array indexing, we can use pointers to access array elements.
- *(p + 3) means the value of x[3].

- The pointer accessing method is much faster than array indexing.

```
void main ( )
{
        int *p, sum=0, i ;
        int x[ 5 ] = {5, 9, 6 ,3, 7} ;
        p = x ;
        printf("\n Element\tValue\tAddress") ;
        for(i=0;i<5;i++)
        {
                printf("\n x[%d] \t %d \t %u", i, *p, p) ;
                sum = sum + *p ;
                p++ ;
        }
        printf("\n Sum  = %d", sum) ;
        printf("\n &x[0]  = %u", &x[ 0 ]) ;
        printf("\n p  = %u", p) ;
}
```

Output:

Element	Value	Address
x[0]	5	166
x[1]	9	168
x[2]	6	170
x[3]	3	172
x[4]	7	174

Sum = 30
&x[0] = 166
p = 176

❖ Pointers as Function Arguments

- When an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements.
- If x is an array, when we call sort(x), the address of x[0] is passed to the function sort.
- The function uses this address for manipulating the array elements.

- Similarly, we can pass the address of a variable as an argument to a function.
- When we pass addresses to a function, the parameters receiving the addresses should be pointers.
- The process of calling a function using pointers to pass the addresses of variables is known as call by reference.
- The function which is called by reference can change the value of the variable used in the call.

```
void main ( )
{
        int x = 20;
        change (&x) ;          //call by reference or  address
        printf (" \n%d", x) ;
}
change ( int *p )
{
        *p = *p + 10 ;
}
```

- When the function change() is called , the address of the variable x, not its value, is passed into the function change().
- Inside change(), the variable p is declared as a pointer and therefore p is the address of the variable x. The statement,

  *p = *p + 10 ; (add 10 to the value stored at the address p.)

- Since p represents the address of x, the value of x is changed from 20 to 30.
- Therefore, the output of the program will be 30 not 20.
- Thus call by reference provides a mechanism by which the function can change the stored values in the calling function.
- This mechanism is also known as call by address or pass by pointers.

```
void exchange (int *, int *) ;
void main ( )
{
        int x, y ;
        x = 100 ;
        y = 200 ;
        printf (“\n Before Exchange ”) ;
        printf (“\n x = %d”, x) ;
        printf (“\n y = %d”, y) ;
        exchange ( &x, &y ) ;
        printf (“\n After Exchange ”) ;
        printf (“\n x = %d”, x) ;
        printf (“\n y = %d”, y) ;
}
exchange(int *a, int *b)
{
         int t ;
        t =*a ;
        *a = *b ;
        *b = t ;
}
```

Output :

Before Exchange
x = 100
y = 200
After Exchange
x = 200
y = 100

❖ Functions that return multiple values

- Normally a function can return only one value using a return statement, because a return statement can return only one value.
- Functions can have arguments not only to receive information (input parameters) but also to send back information to the calling function (output parameters).
- This mechanism of sending back information can be achieved using address of operator (&) and indirection operator (*).

```
void mathop (int x, int y, int *add, int *sub)
{
        *add = x + y;
        *sub = x - y ;
}
void main()
{
        int x = 50, y = 10, a, s ;
        mathop (x, y, &a, &s) ;
        printf ("\n Addition = %d", a) ;
        printf ("\n Subtraction = %d", s) ;
}
```

Output :

Addition = 30
Subtraction = 10

- The actual arguments x and y are input arguments, s and d are output arguments.
- In the function call, while we pass the actual values x and y to the function, we pass the addresses of locations where the values of s and d are stored in the memory.
- When the function is called the following assignment occurs:

  value of x to a
  value of y to b
  address of s to add
  address of d to sub
- The variables add and sub point to the memory locations of s and d respectively.
- In the body of the function, we have two statements :

  *add = a + b ;
  *sub = a – b ;
- The first one adds the values a and b and the result is stored in the memory location pointed to by add.
- Similarly, the value of a – b is stored in the location pointed to by sub, which is the location of d.
- After the function call is implemented, the value of s is a + b and the value of d is a – b.
- The variables *add and *sub are known as pointers and add and sub as pointer variables.

❖ Pointer and Structure

- The names of arrays of structures to store the address of its zeroth element.

Syntax:

```
struct tag_name
{
   datatype member1;
   datatype member2;
   ……..
}struct_variable,*ptr_name;
```

Example:

```
struct student
{
    int sno;
    char sname[20];
}s[2],*ptr;
```

- In above example declare **s** as array of two elements. i.e structure variable

And ptr as a pointer variable to points a data of structure student.

- The assignment **ptr=s;**

would assign the address of zeroth element of s to ptr. i.e **ptr=&s[0].**

- Access the structure member through Pointer:

The symbol **->** is called the arrow operator and is made up of minus (-) sign and >(greater than) sign.

So access the structure member of student using the ptr variable as below.

```
ptr->sno;        ptr->sname;
```

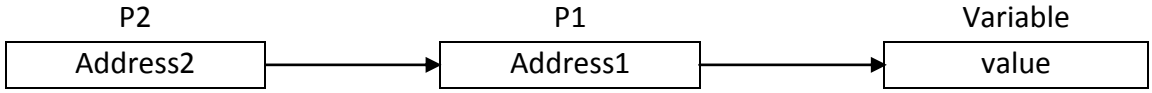
- we can also use the notation (*ptr).sno to access the sno. The parentheses around *ptr are necessary because the member operator "." Has a higher precedence than the operator *.

e.g

```
struct student
{
    int sno;
    char sname[20];
}s[2],*ptr;
void main()
{
 for (ptr=s;ptr<s+2;ptr++)
 {
   scanf("%d",ptr->sno);
   scanf("%s",ptr->sname);
   printf("%d",ptr->sno);   //    printf("%d", (*ptr).sno);
   printf("%s",ptr->sname);    printf("%s", (*ptr).sname);
 }
}
```

❖ Chain of pointers (pointer to a pointer)

- It is possible to make a pointer to point to another pointer, thus creating a chain of pointer as shown below.



- Here, the pointer variable p2 contains the address of the pointer variable p1, which points to the location that contains the desired value. This is known as multiple indirections.
- A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name.
- For example,

```
int **p2 ;
```

- This declaration tells the compiler that p2 is a pointer to a pointer of int type.
- The pointer p2 is not pointer to an integer, but rather a pointer to an integer pointer.
- We can access the target value indirectly pointed to by pointer to a pointer by applying the indirection operator twice.

void main()
{
int x, *p1, **p2 ;
x = 100 ;
p1 = &x ;
p2 = &p1 ;
printf("\n x = %d", **p2);
}
	Output :
	x = 100

- Here p1 integer pointer and p2 is a pointer to an integer pointer.

❖ Dynamic Memory Allocation

- C language requires the number of elements in an array to be specified at compile time.

- Our initial judgment of size, if it is wrong, may cause failure of the program or wastage of memory space.
- Many languages permit a programmer to specify an array's size at run time.
- Such languages have the ability to calculate and assign, the memory space required by the variables in a program during execution.
- **The process of allocating memory at run time is known as dynamic memory allocation.**
- There are four library routines (functions) known as "**memory management functions**" that can be used for allocating and freeing memory during program execution.

Function	Task
malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	Frees previously allocated space.
realloc	Modifies the size of previously allocated space.

❖ Allocating a block of memory : malloc ( )

Use:	➢ A block of memory allocated using the function malloc. ➢ It reserves a block of memory of specified size and return a pointer of type void. This means that we can assign it to any type of pointer.
Syntax:	ptr = (cast-type *)malloc(byte-size);
	➢ Where ptr is a pointer of type cast-type. ➢ The malloc returns a pointer (of cast-type) to an area of memory with size byte-size.
Example	x = (int *) malloc (100 *sizeof (int)) ; ➢ When this statement is executed, a memory space equivalent to 200 bytes is reserved and the address of the first byte of memory allocated is assigned to the integer pointer x. Similarly, the statement cptr = (char *) malloc (10) ; allocates 10 bytes of space for the character pointer cptr.

❖ Allocating multiple blocks of memory : calloc( )

Use:	➢ calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. ➢ While malloc allocate a **single block of storage** space, calloc allocates **multiple block of storage**, each of the same size and then sets all bytes to zero.
Syntax:	ptr = (cast-type *) calloc (n, elem-size) ;
	The above statement allocates contiguous space for n blocks, each of size elem-size bytes.
Example	x = (int *) calloc (5,10) ; allocates 10 bytes of space of 5 block for the integer pointer x.

❖ Releasing the used space : free( )

Use:	➢ With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. ➢ The release of storage space becomes important when the storage is limited. ➢ **When we no longer need the data we stored in a block of memory, and we do not intend to use that block for storing any other information, we may release that block of memory for future use, using the free function.**
Syntax:	free (ptr) ;
	ptr is a pointer to a memory block, which has already been created by malloc or calloc.
Example	x = (int *) malloc (100) ; free(x);

❖ Altering the size of a block : realloc ( )

Use:	➢ It may be possible that the previously allocated memory is not sufficient and we need additional space for more elements. ➢ It is also possible that the memory allocated is much larger than necessary and we want to reduce it. ➢ In both the cases, we can change the memory size already allocated with the help of the function realloc. ➢ This process is called reallocation of memory.
Syntax:	ptr = realloc (ptr, newsize) ;
	➢ This function allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the new memory block. ➢ The newsize may be larger or smaller than the size.
Example	x = (int *) malloc (100) ; x=realloc(x,200);

	UNIT - 3
	MCQ
1	_____operator is used with a pointer to access the value of the variable whose address is contained in the pointer. (a) address of (b) sizeof (c) **indirection** (d) member selection
2	int a, *p = &a; Which of the following statement will not add 1 to a variable? (a) a++; (b) *p=*p+1; (c) (*p)++; (d) ***p++;**
3	Given the following declarations: int x; double d; int *p; double *q; Which of the following expression is allowed? (a) **p=&x;** (b) q=&x; (c) p=&d; (d) p=x;
4	Which of the following defines a pointer variable to an integer? (a) int &ptr; (b) int **ptr; (c) int &&ptr (d) **int *ptr;**

5	Which of the following defines and initializes a pointer to the address of x? (a) int *ptr = *x; (b) **int *ptr = &x;** (c) int &ptr = *x; (d) int *ptr = ^x;
6	For the given the declarations , which statement is not valid? int i; float f; int *pd; float *pf; (a) **pd=pf;** (b) i=5; (c) pd=&i; (d) pf=&f;
7	Which of the following statements about pointers and arrays is true? (a) The only way to reference data in array is with index operator. (b) The name of the array is a pointer variable. (c) **The following expressions are identical when ary is an array: ary and &ary[0]** (d) The following expressions are identical when ary is an array: *ary and &ary[0]
8	Which of the following is not a C memory allocation function? (a) malloc() (b) calloc() (c) realloc() (d) **alloc()**
9	If ary is name of an integer array with 10 elements then which of the following statement is false? (a) The two expressions *(ary + 5) and ary[5] are same (b) Name of array ary is a pointer constant to the first element of array. (c) The two expressions ary and &ary[0] are same. (d) **If p is an integer pointer variable then p=ary; is invalid statement.**
10	Given a pointer ptr to a structure stu containing a field called name which of the following statements correctly refer name? (a) **ptr->name** (b) ptr->stu.name (c) ptr.name (d) ptr->stu->name
11	How will you free the allocated memory? (a) remove(ptr) (b) **free(ptr)** (c) dealloc(ptr)(d) destroy(ptr)
12	*&ptr gives _____ (**a) value stored at the location pointed to by ptr** (b) address stored at the location pointed to by ptr (c) address pointed to by ptr (d) address of ptr

	Unit-3
	Short Questions
1	Define pointer variable.
2	Differentiate pointer and pointer variable.
3	List benefits of pointer.
4	Give different forms to declare a pointer variable. Which one is preferable?
5	List out memory management functions.
6	Define dynamic memory allocation.
7	What does malloc() do? Write the syntax for using malloc().
8	What does calloc() do? Write the syntax for using calloc().
9	What does realloc() do? Write the syntax for using realloc().
10	What does free() do? Write the syntax for using free().
11	List out any 4 scale factor along with the data types.

12	Explain pointer increment and scale factor in brief.
	Unit-3
	Long Questions
1	Explain declaration and initialization of pointer variable with syntax and example.
2	Explain pointer to pointer with example.
3	How can we access a variable through its pointer?
4	Explain pointer arithmetic expressions in detail.
5	Explain pointer to an array with example.
6	Explain pointers as function arguments with example.
7	Explain function returning multiple values with example.
8	Explain malloc() with example.
9	Explain calloc() with example.
10	Explain realloc() with example.
11	Explain free() with example.