Q. What is the pointer?

Pointer is basically derived data type and which is used for store the address of another variable and which is used for allocate the dynamic memory at run time. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture.

Q. what is the dynamic memory allocation?

dynamic memory allocation means we can decide how much memory required at program run time or we can allocate memory at run time as per our need and we can grow or shrink memory at run time as per our need at program run time called as dynamic memory allocation.

Q. how to declare the pointer in c language?

If we want to declare the pointer in c language we have to use the * before variable name called as pointer

Syntax: datatype * variablename;

```
        e.g int  *ptr;
```

Q. how to store the address of another variable in pointer?

Syntax:

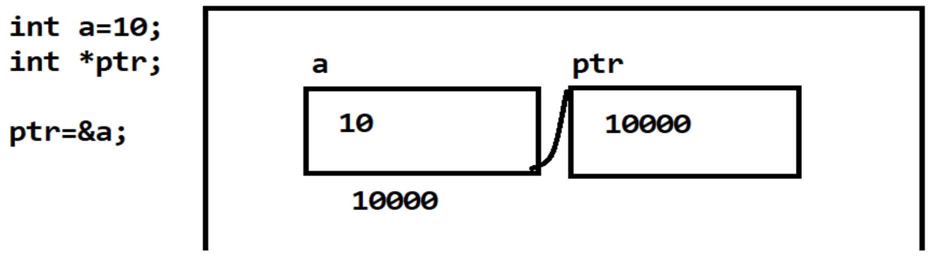
```
datatype *pointername;
pointername=&normalvariable;
```

Example:

```
e.g int a=100,*ptr;
    ptr=&a;
```

if we think about the above statement here we store the address of variable a in pointer variable ptr means ptr points to variable a means ptr can use the value of a or can work on address of variable a.If we store the address of another variable called as referencing.

Following Diagram can show the meaning of above statement



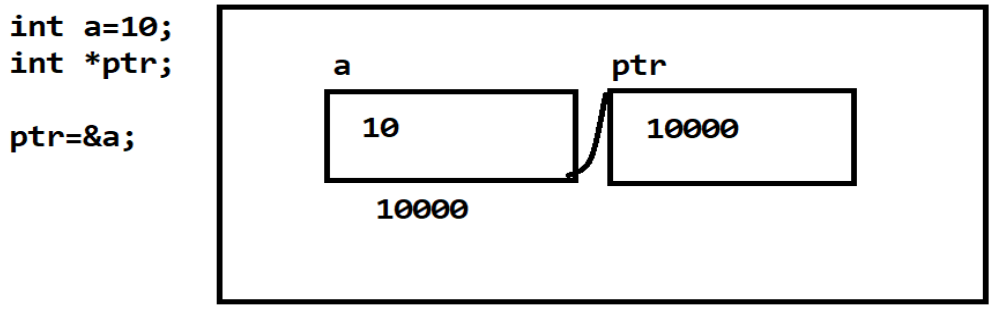
```
int a=10;
int *ptr;

ptr=&a;
```

Q. what is the dereferencing or how to fetch data using pointer?

Dereferencing is a process where we can fetch value from location or address which stored in pointer variable and if we perform dereferencing we have to use the * with a pointer variable

Example:



```
int a=10;
int *ptr;

ptr=&a;
```

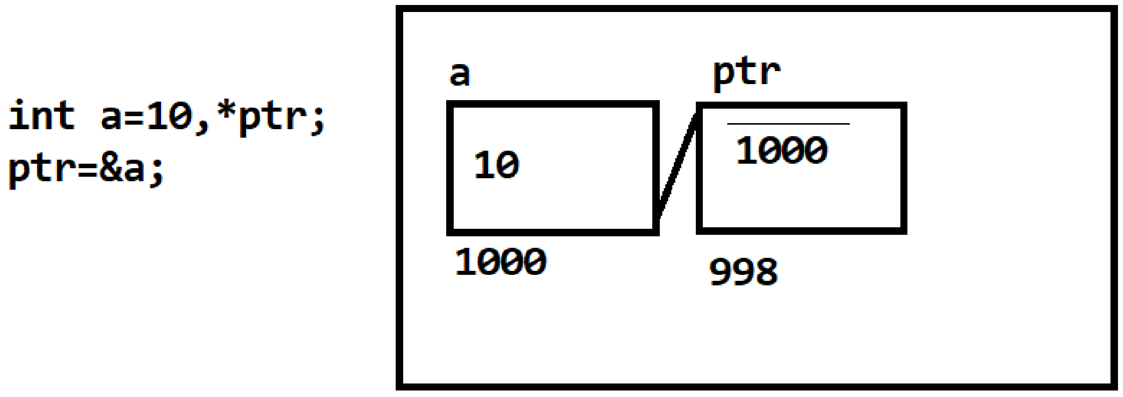
```
printf("%d\n", *ptr); //10
```

If we think about the above diagram in this diagram we store the address of variable a in pointer variable ptr means we store the 10000 in pointer variable ptr and if we use the *ptr means we fetch data from 10000 address so we get the value 10.The above process called as dereferencing.

Q.what happens if we perform the operation on pointer ptr++?

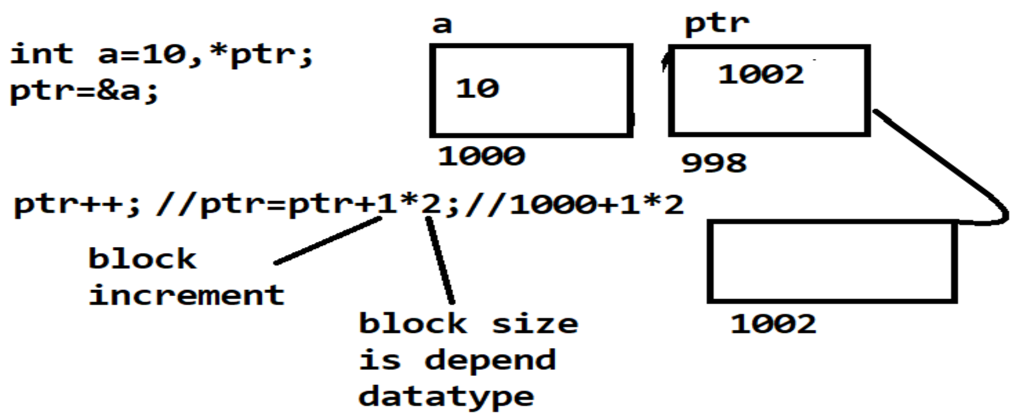
When perform operation ptr++ on pointer then pointer shift on next block or new block and leave the previous address hold by pointer

Example given below



```
int a=10,*ptr;
ptr=&a;
```

If we think about the above statement then variable a store its address in pointer variable ptr means ptr can point or use the address of variable a.



```
int a=10,*ptr;
ptr=&a;
ptr++; //ptr=ptr+1*2;//1000+1*2
```

If we think about the statement ptr++ means pointer move on next block
As per our example ptr contain 1000 address before ptr++ and when ptr++ get executed the pointer move on next block i.e. 1002 as per our example and pointer leave the 1000 address and move on 1002 and hold in pointer means pointer move on next block. Means according to this example we can say pointer can leave the address of memory when he required at program run time.

Q. how we can replace or store value in memory using pointer?

If we want to store the value on specified memory using pointer we can use the *pointername=value;

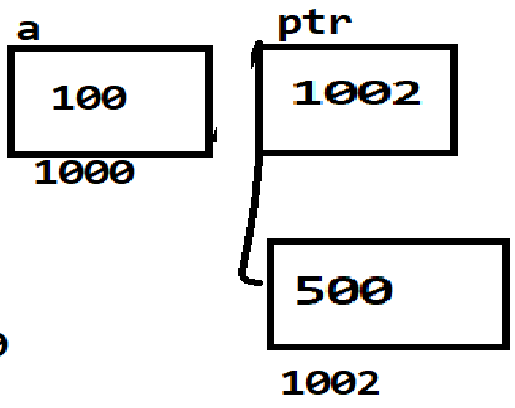
Example:

```
e.g int  a=100,*ptr;
         ptr=&a;

 ptr++;

 *ptr=500;

printf("%d",*ptr);//500
```



If we think about the above code our pointer variable store the address of variable a in ptr variable i.e. 1000 when ptr++ statement get execute then pointer move on new block i.e 1002 block and hold the address 1002 and when *ptr=500 statement executed then 500 value put in 1002 address by pointer variable i.e ptr.

Q. Gives the names of dynamic memory allocation function in c language?

If we want to work with dynamic memory allocation function in c language we have the following functions
1) malloc ()
2) calloc ()
3) free ()
4) realloc () function in c language.
These all functions present in stdlib.h header file.

Q.Explain the Types of Pointer In C Language?

We have the some types of pointer in c language
1) void pointer or generic pointer
2) dangling of pointer
3) null pointer
4) wild card pointer
5) huge pointer

6) far pointer
7) near pointer

Q. Explain the void pointer or generic pointer in detail?

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be type casted to any type.

```
int a = 10;
char b = 'x';
void *p = &a;  // void pointer holds address of int 'a'
p = &b; // void pointer holds address of char 'b'
```

Advantages of void pointers:

1) malloc() and calloc() return void * type and this allows these functions to be used to allocate memory of any data type (just because of void *)

```
int main(void)
{
   // Note that malloc() returns void * which can be
   // typecasted to any type like int *, char *, ..
   int *x = (int*)malloc(sizeof(int) * n);
}
```

1) void pointers cannot be dereferenced. For example the following program doesn't compile.

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

Output:

```
Compiler Error: 'void*' is not a pointer-to-object type
```

The following program compiles and runs fine.

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

Output:

```
10
```

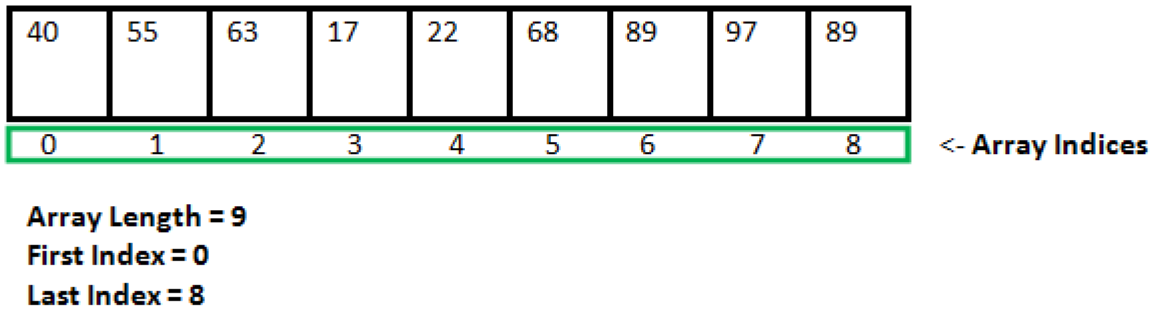
The C standard doesn't allow pointer arithmetic with void pointers. However, in GNU C it is allowed by considering the size of void is 1. For example the following program compiles and runs fine in gcc.

```
include<stdio.h>
int main()
{
   int a[2] = {1, 2};
   void *ptr = &a;
   ptr = ptr + sizeof(int);
   printf("%d", *(int *)ptr);
   return 0;
}
```

output is : 2

Q. Explain the malloc function in Detail?

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



As it can be seen that the length (size) of the array above made is 9. But what if there is a requirement to change this length (size). For Example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

his procedure is referred to as **Dynamic Memory Allocation in C**.
Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

C malloc() function

The "malloc" or "memory allocation" method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

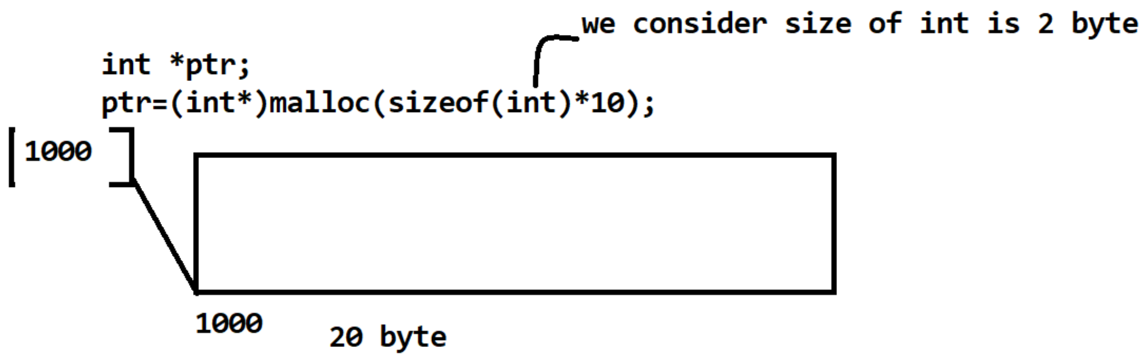
Syntax:

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int*) malloc(10 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



we consider size of int is 2 byte
int *ptr;
ptr=(int*)malloc(sizeof(int)*10);
1000
1000
20 byte

If space is insufficient, allocation fails and returns a NULL pointer.

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{   int* ptr;
    int n, i;
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);
    ptr = (int*)malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
       printf("Memory successfully allocated using malloc.\n");
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
    return 0;
}
```

Q. Explain the calloc function in c language?

1. **“calloc”** or **“contiguous allocation”** function in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc ().

   Syntax

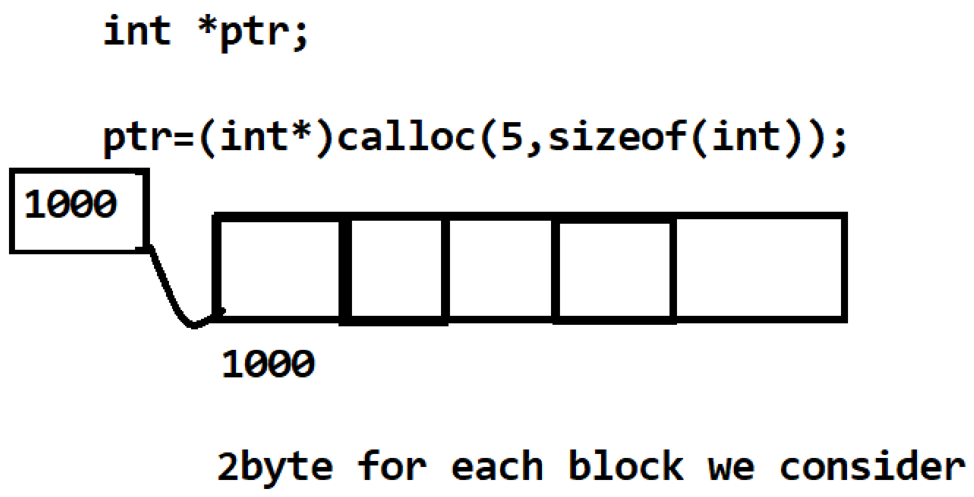
   ptr = (cast-type*)calloc(n, element-size);

   here, n is the no. of elements and element-size is the size of each element.

   For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.


int *ptr;
ptr=(int*)calloc(5,sizeof(int));
1000
1000
2byte for each block we consider

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{   int* ptr;
    int n, i;
    n = 5;
    printf("Enter number of elements: %d\n", n);
   ptr = (int*)calloc(n, sizeof(int));
     if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
 printf("Memory successfully allocated using calloc.\n");
```

```
      for (i = 0; i < n; ++i) {
          ptr[i] = i + 1;
       }
printf("The elements of the array are: ");
       for (i = 0; i < n; ++i) {
          printf("%d, ", ptr[i]);
       }
   }
   return 0;
}
```

Q. Explain the free() function in c language?

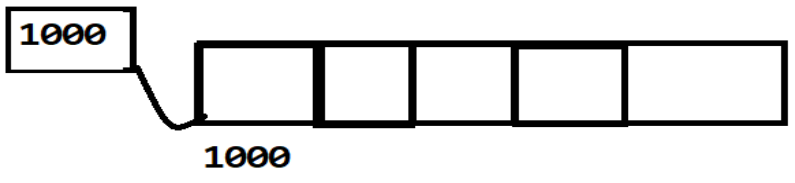
“free” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

free(ptr);

```
int *ptr;

ptr=(int*)calloc(5,sizeof(int));
```



```
2byte for each block we consider

free(ptr);//release the memory of ptr
```

Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
     int *ptr, *ptr1;
    int n, i;
   n = 5;
    printf("Enter number of elements: %d\n", n);
        ptr = (int*)malloc(n * sizeof(int));
    ptr1 = (int*)calloc(n, sizeof(int));
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
       printf("Memory successfully allocated using malloc.\n");
        free(ptr);
        printf("Malloc Memory successfully freed.\n");
        printf("\nMemory successfully allocated using calloc.\n");
          free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }
    return 0;
}
```

Q. Explain the realloc function in c language ?

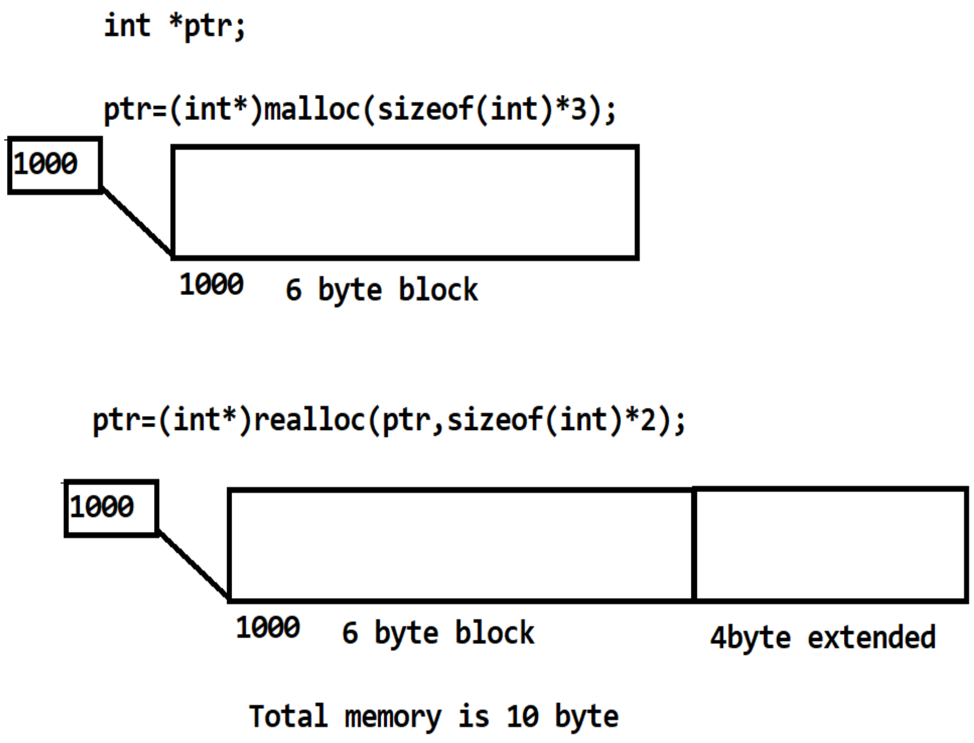
"realloc" or "re-allocation" method in C is used to dynamically change the memory allocation of a previously allocated memory.

In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax:

ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.



```
int *ptr;

ptr=(int*)malloc(sizeof(int)*3);
```

```
ptr=(int*)realloc(ptr,sizeof(int)*2);
```

Total memory is 10 byte

Q. Explain Difference Between malloc() and calloc() with Examples?

The functions malloc() and calloc() are library functions that allocate memory dynamically. Dynamic means the memory is allocated during runtime (execution of the program) from the heap segment.

Initialization

malloc() allocates a memory block of given size (in bytes) and returns a pointer to the beginning of the block. malloc() doesn’t initialize the allocated memory. If you try to read from the allocated memory without first initializing it, then you will invoke undefined behavior, which will usually mean the values you read will be garbage.

calloc() allocates the memory and also initializes every byte in the allocated memory to 0. If you try to read the value of the allocated memory without initializing it, you’ll get 0 as it has already been initialized to 0 by calloc().

Parameters

malloc() takes a single argument, which is the number of bytes to allocate.

Unlike malloc(), calloc() takes two arguments:
1) Number of blocks to be allocated.
2) Size of each block in bytes.

Return Value

After successful allocation in malloc() and calloc(), a pointer to the block of memory is returned otherwise NULL is returned which indicates failure.

Q. Difference between Static and Dynamic Memory Allocation in C ?

Memory Allocation: Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. The memory allocation is done either before or at the time of program execution. There are two types of memory allocations:

Static Memory Allocation: Static Memory is allocated for declared variables by the compiler. The address can be found using the *address of* operator and can be assigned to a pointer. The memory is allocated during compile time.

Dynamic Memory Allocation: Memory allocation done at the time of execution(run time) is known as **dynamic memory allocation**. Functions calloc() and malloc() support allocating dynamic memory. In the Dynamic allocation of memory space is allocated by using these functions when the value is returned by functions and assigned to pointer variables.

.No	**Static Memory Allocation**	**Dynamic Memory Allocation**
1	In the static memory allocation, variables get allocated permanently, till the program executes or function call finishes.	In the Dynamic memory allocation, variables get allocated only if your program unit gets active.
2	Static Memory Allocation is done before program execution.	Dynamic Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory
4	It is less efficient	It is more efficient
5	In Static Memory Allocation, there is no memory re-usability	In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required
6	In static memory allocation, once	In dynamic memory allocation, when

	the memory is allocated, the memory size can not change.	memory is allocated the memory size can be changed.
7	In this memory allocation scheme, we cannot reuse the unused memory.	This allows reusing the memory. The user can allocate more memory when required. Also, the user can release the memory when the user needs it.
8	In this memory allocation scheme, execution is faster than dynamic memory allocation.	In this memory allocation scheme, execution is slower than static memory allocation.
9	In this memory is allocated at compile time.	In this memory is allocated at run time.
10	In this allocated memory remains from start to end of the program.	In this allocated memory can be released at any time during the program.
11	**Example:** This static memory allocation is generally used for array.	**Example:** This dynamic memory allocation is generally used for linked list.

Q. what happens when you don't free memory after using malloc () ?

The "malloc" or "memory allocation" method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with a default garbage value.

Syntax:

ptr = (cast-type*) malloc(byte-size)

For Example:

ptr = (int) malloc(10 * sizeof(int));*
*Since the size of int is 4 bytes, this statement will allocate 2**0 bytes** of memory. And, the pointer **ptr** holds the address of the first byte in the allocated memory. But the memory allocation using malloc() is not de-allocated on its own. So, “free()” method is used to de-allocate the memory. But the **free()** method is not compulsory to use. If **free()** is not used in a program the memory allocated using **malloc()** will be de-allocated after completion of the execution of the program (included program execution time is relatively small and the program ends normally). Still, there are some important reasons to **free()** after using **malloc()**:*

- Use of free after malloc is a good practice of programming.
- There are some programs like a digital clock, a listener that runs in the background for a long time and there are also such programs which allocate memory periodically. In these cases, even small chunks of storage add up and create a problem. Thus our usable space decreases. This is also called “memory leak”. It may also happen that our system goes out of memory if the de-allocation of memory does not take place at the right time.

So, the use of the **free()** functions depends on the type of the program but it is recommended to avoid the unwanted memory issues like a memory leak.
Below is the program to illustrate the use of **free()** and **malloc()**:

Q. How to deallocate memory without using free() in C?

Solution: Standard library function realloc() can be used to deallocate previously allocated memory. Below is function declaration of “realloc()” from “stdlib.h”

```
void *realloc(void *ptr, size_t size);
```

If “size” is zero, then call to realloc is equivalent to “free(ptr)”. And if “ptr” is NULL and size is non-zero then call to realloc is equivalent to “malloc(size)”.

Let us check with simple example.

```
#include <stdio.h>

#include <stdlib.h>

int main(void)

{  int *ptr = (int*)malloc(10);

 realloc(ptr, 0);

   return 0;

}
```

Q. what are the applications of pointer in c language?

1) To pass arguments by reference.

2) To modify variable of function in other.

3) For efficiency purpose. Example passing large structure without reference would create a copy of the structure (hence wastage of space).

4) For accessing array elements.

5) To return multiple values

6) Dynamic memory allocation

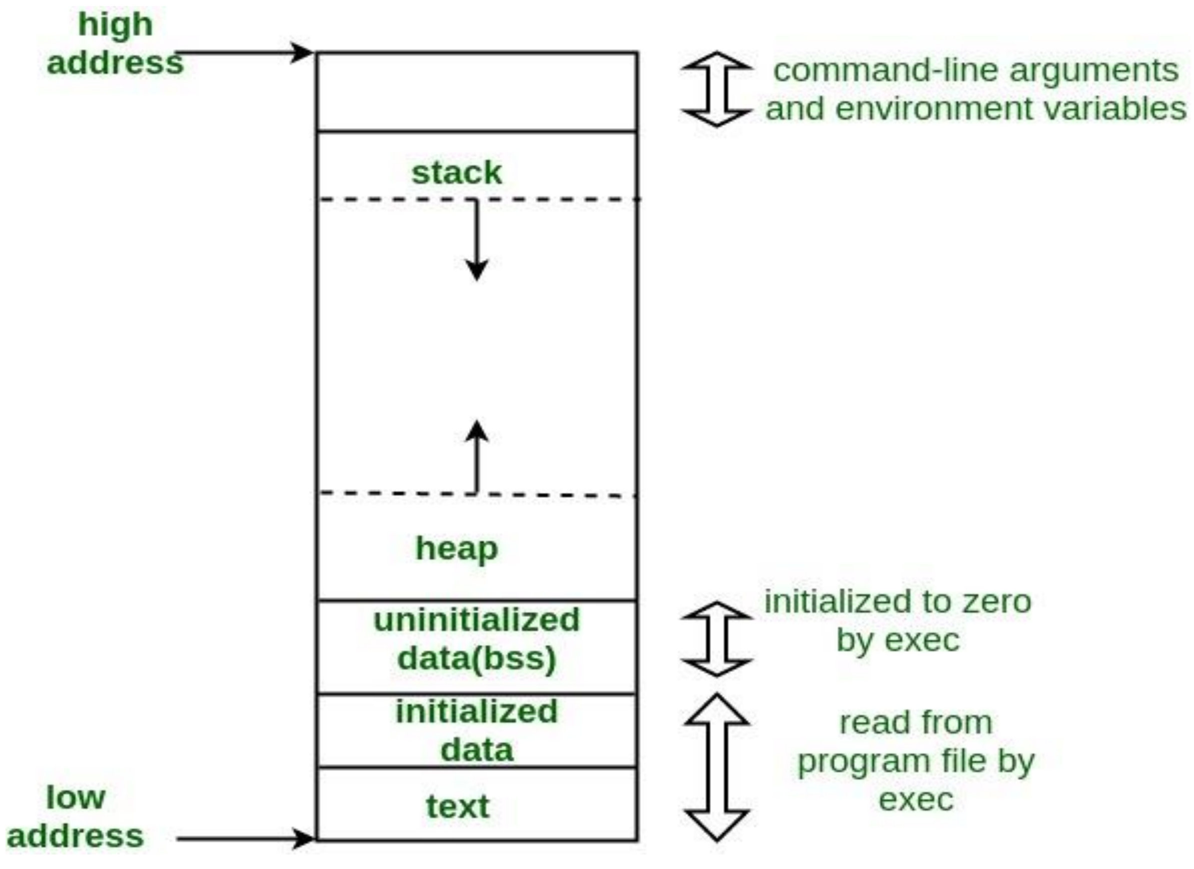
7) To implement data structures

8) To do system level programming where memory addresses are useful

Q. Explain the memory layout of c program?

A typical memory representation of a C program consists of the following sections.

1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack


high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

A typical memory layout of a running process

1. Text Segment:

A text segment, also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

2. Initialized Data Segment:

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of the virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer.

Note that, the data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into the initialized read-only area and the initialized read-write area.

For instance, the global string defined by char s[] = "hello world" in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in the initialized read-write area. And a global C statement like const char* string = "hello world" makes the string literal "hello world" to be stored in the initialized read-only area and the character pointer variable string in the initialized read-write area.

Ex: static int i = 10 will be stored in the data segment and global int i = 10 will also be stored in data segment

3. Uninitialized Data Segment:
Uninitialized data segment often called the “**bss**” segment, named after an ancient assembler operator that stood for “**block started by symbol**.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing
uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.
For instance, a variable declared static int i; would be contained in the BSS segment.
For instance, a global variable declared int j; would be contained in the BSS segment.

4. Stack:
The stack area traditionally adjoined the heap area and grew in the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow in opposite directions.)
The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture, it grows toward address zero; on some other architectures, it grows in the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.
Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how

recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

5. Heap:

Heap is the segment where dynamic memory allocation usually takes place.
The heap area begins at the end of the BSS segment and grows to larger addresses from there. The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

MCQ on Pointers Chapter

Q. what will be the output of given code?

```
#include<stdio.h>

void main()

{    int i = 10;

     void *p = &i;

     printf("%d\n", (int)*p);

}
```

Q. what will be the output of given code?

The declaration
int (*p) [5];
means

A. p is one dimensional array of size 5, of pointers to integers.

B. p is a pointer to a 5 elements integer array.

C. The same as int *p[]

D. None of these.

Q. what will be the output of given code?

```
void main() {
 int a[] = {1,2,3,4,5}, *p;
 p = a;
 ++*p;
 printf("%d ", *p);
   p += 2;
  printf("%d ", *p);
}
```

Q. What is the output of the following C code?

```
 char *ptr;
  char mystring[] = "abcdefg";
  ptr = myString;
  ptr += 5;
```

Q. Is the NULL pointer same as an uninitialised pointer?

a) True
b) False

Q. Which of the following statements correct about k used in the below statement?

```
char ****k;
```

a) k is a pointer to a pointer to a pointer to a char
b) k is a pointer to a pointer to a pointer to a pointer to a char
c) k is a pointer to a char pointer
d) k is a pointer to a pointer to a char

Q. What will be the output of the program?

```
char *p = 0;
char *t = NULL;
```

Q. What will be the output of the program?

```
void main() {
  printf("%d %d", sizeof(int *), sizeof(int **));
}
```

Q. What will be the output of the program if the array begins at 1000 and each integer occupies 2 bytes?

```
void main() {
  int i = 10;
  void *p = &i;
  printf("%f\n", *(float *)p);
}
```

Q. What would be the equivalent pointer expression for referring the array element a[i][j][k][l]?

a) ((((a+i)+j)+k)+l)

b) *(*(*(*(a+i)+j)+k)+l)

c) (((a+i)+j)+k+l)

d) ((a+i)+j+k+l)

Q. Can you combine the following two statements into one?

```
char *p;

p = (char*) malloc(100);
```

A. char p = *malloc(100);

B. char *p = (char) malloc(100);

C. char *p = (char*)malloc(100);

D. char *p = (char *)(malloc*)(100);

Q. what will be the output of given code?

```
void main()

{   int k = 5;

    int *p = &k;

    int **m = &p;

    **m = 6;

    printf("%d\n", k);

}
```

Q. what will be the output of given code?

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

```
 int main()
 {   int i = 0, j = 1;
     int *a[] = {&i, &j};
     printf("%d", (*a)[0]);
     return 0;
 }
```

Q. what will be the output of given code?

```
void main()
{  int a[3] = {1, 2, 3};
int *p = a;
int *r = &p;
printf("%d", (**r));
}
```

Q. what will be the output of given code?

```
int main()
{  int a = 1, b = 2, c = 3;
int *ptr1 = &a, *ptr2 = &b, *ptr3 = &c;
int **sptr = &ptr1; //-Ref
*sptr = ptr2;
}
```

Q. what will be the output of given code?

a) int *a[] = {{1, 2, 3}, {2, 3, 4, 5}};

b) int a[][] = {{1, 2, 3}, {2, 3, 4, 5}};

c) Both A and B

d) None of the above

Q. what will be the output of given code?

```
int main()
{ char *p = NULL;
  char *q = 0;
  if (p)
    printf(" p ");
 else
    printf("nullp");
if (q)
   printf("q\n");
else
   printf(" nullq\n");
}
```

Q. what will be the output of given code?

```
#include<stdio.h>

int main()

{      const int a = 5;

          const int *ptr;

          ptr = &a;
```

```
        *ptr = 10;

        printf("%d\n", a);

        return 0;

}
```

Q. what will be the output of given code?

```
#include<stdio.h>

int main()

{   printf("%d", sizeof(void *));

        return 0;

}
```