Pointers: Introduction, Pointers to pointers, Compatibility, L-value, and R-value
Pointer Applications: Arrays and Pointers, Pointer Arithmetic and Arrays, Memory Allocation Function, Array of Pointers, Programming Application.
Processor Commands: Processor Commands.

Define Pointers in C Language and explain with an example.

A pointer is a constant or a special variable that stores the address of another variable with the same data type as its value.

- Pointer variables can be created using any data type such as int, char, float etc.
- A pointer stores only the address of a variable with the same datatype.
- A pointer is declared with the * operator.
- A pointer variable points only to the same data type of its own.
    - Ex: An int pointer stores only the address of an int variable.

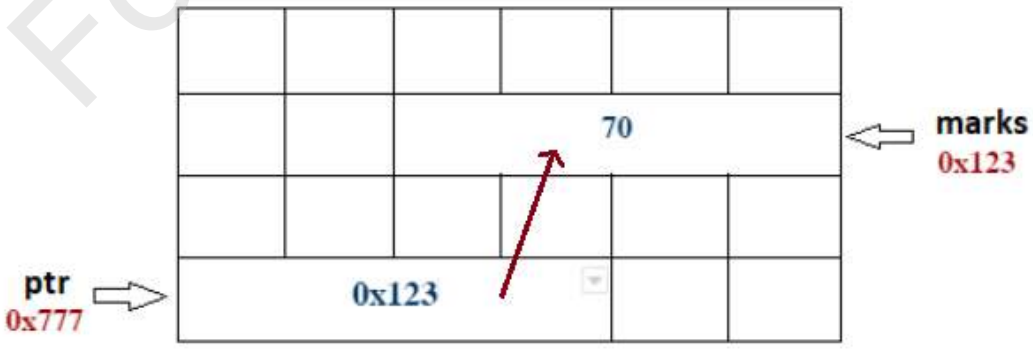
Create a pointer - Syntax Declaration:

```
dataType   *pointerVar;
dataType*    pointerVar;
dataType  *  pointerVar;
```

A variable declaration prefixed with * symbol becomes a pointer variable.
dataType - the data type of the pointer variable
pointerVar - the name of the pointer variable

Assign address as value to a Pointer: Every variable we define in our program is stored at a specific location in the memory.

```
int marks = 70;
int *ptr;
ptr = &marks;
```



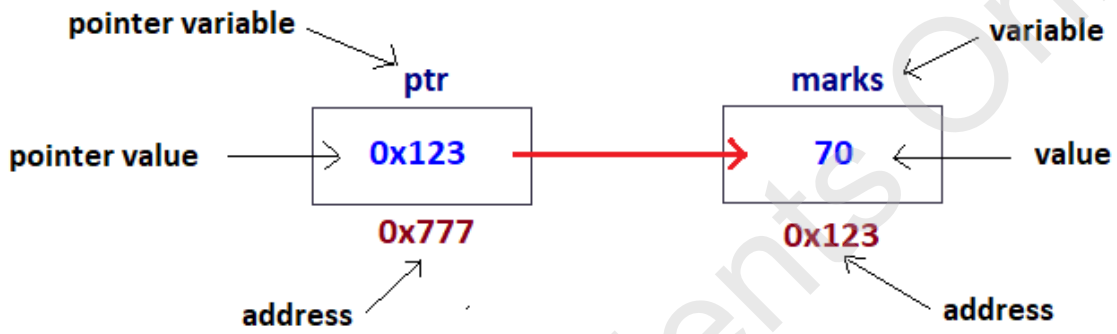
The compiler may allocate 4 bytes for int variable **marks** with value 70 at a memory like 0x123.

- "**marks**" is a normal **integer** variable,
- "**ptr**" is an **integer pointer** variable,
- "**&**" is the **Reference Operator** (also a Unary Operator) to access address of the variable.

We used **&marks** to access the address (**0x123)** of **marks** variable and stored the address in **ptr** pointer. Now, **ptr** variable has **0x123** and pointing to the address of marks variable. The following statement assigns the address of variable "marks" to pointer variable "ptr".

ptr = &marks;

So, the pointer variable **ptr** is pointing to variable **marks**.



Accessing (Variable Value) through Pointer:

- Accessing a variable's value is **Direct access**. However, Accessing another variable's value using a pointer is called **Indirect access**.
- We use * **Indirection Operator** (Dereference Operator) in front of pointer variable to access value of another variable to which the pointer is pointing.
- **Note:** The indirection operator (*) and address operator (**&**) are the inverse of each other. Means, the expression ***(&x)** gives the value of the variable 'x'. (& and * cancel each other).

Syntax to Access variable value from pointer:

***pointerVar**

Example: Program to access value of another variable using pointer variable

```
#include<stdio.h>
int main()
{   int marks = 70;
    int *ptr ;
    ptr = &marks ;  //& is Reference operator to access address
    printf("Address of variable marks = %p\n", ptr);
    printf("Value of variable marks = %d\n", *ptr) ;
    printf("Address of variable ptr = %p\n", &ptr) ;
}
```

Output:

Address of variable marks = 0x123
Value of variable marks = 70
Address of variable ptr = 0x777

Explanation:

marks is a normal variable and
ptr is a pointer variable.

- Address of variable **marks** is stored in pointer variable **ptr**.
- **ptr** is used to access the address of variable **marks** and
- ***ptr** is used to access the value of variable **marks**.

Note: The address of any memory location is an unsigned integer value.
%u prints the address in Unsigned Integer format
%p prints the address in Hexadecimal format

Memory Allocation of Pointer Variables

- Every pointer variable is used to store the address of another variable.
- The address of any memory location is an unsigned integer value.
- In C, unsigned integer requires 4 or 8 bytes of memory depending on C compiler. So, the size of pointer variable is not fixed.
- A 32bit OS allocates 4 bytes and a 64 bit OS allocates 8 bytes for an unsigned integer.

What Are The Types Of Pointers?

The following are different types of pointers in C.

1. **NULL pointer** - Doesn't point anywhere. The initial value is NULL. (%d prints 0).
2. **void pointer** - Pointer with no data type. Accepts address of any data type; **void** is a keyword
3. **Wild pointer** - Uninitialized pointer; contains arbitrary or garbage address
4. **Dangling pointer** - Pointer to a deallocated memory location [in Dynamic Memory allocation]
5. Complex pointer - contains [] , * ,(), data type, identifier

1. **Null Pointer**

A null pointer is created by assigning the value NULL to the pointer. A null pointer can be of any data type. It has a value of 0 in it. The null pointer does not point to any memory location.

Syntax Declaration:

dataType *pointerVar = NULL;

Example:

int *ptr = NULL; //null pointer

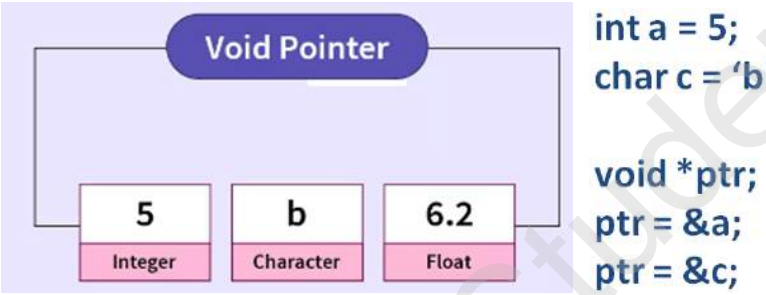
printf("%d",*ptr); //prints 0

2. Void Pointer

A void pointer is a generic pointer that does not have any data type. A void pointer can be typecasted to any data type

Syntax :

void * pointername; // accepts address of any data type variable



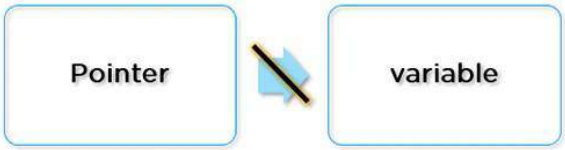
Example:

```
#include<stdio.h>
int main() {
  void *ptr = NULL; //void pointer
  int x = 5;
  ptr = & x;
  //typecasted to int using (int *)
  printf("value of *ptr is %d ", *(int* ) ptr);
  return 0;
}
```

Output: **value of *ptr is 5**

3. Wild Pointer

A wild pointer is an uninitialized pointer. It points to some unknown memory location. Dereferencing a wild pointer has undefined behavior.

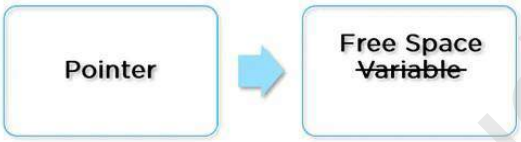


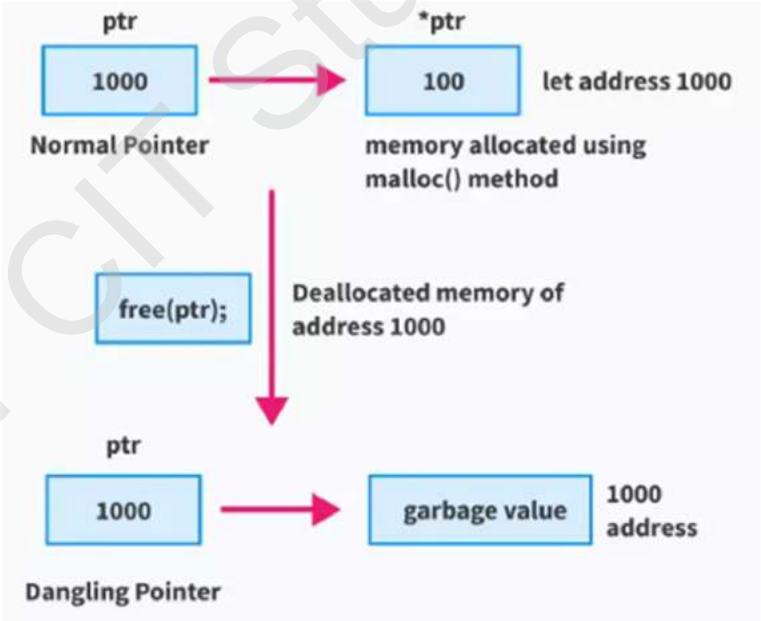
/*Example Program for Wild pointer*/

```
#include <stdio.h>
int main() {
  int *ptr; //Uninitialized wild pointer
  printf("value of *ptr is %d ", *ptr);
  return 0;
}
```

Output: **value of *ptr is 17744 (garbage value)**

4. **Dangling pointer -** Pointer to a deallocated memory location
   [Note: This will be discussed later in Dynamic Memory allocation section]





5. **Complex pointer** - contains [] , * ,(), data type, identifier
   Example:
   char (* ptr)[4]
   Here, ptr is a pointer to a one-dimensional character array of size four.

Explain Pointer to pointer with an example.

A ***Pointer-to-Pointer variable*** will store the address of another pointer variable. Also called a **Double Pointer** variable. . This is a 2-level **indirection**.

Declaration syntax:

dataType **pointerVar;

Pointer2(address of the pointer1) –> Pointer1(address of the variable) –> Variable(value)

Note: pointerVar stores the address of another pointer of the same data type but does not store the normal variable address.

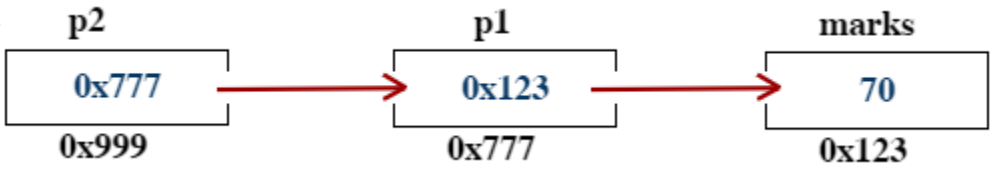
In this situation, a pointer will indirectly point to a variable via another pointer.

Points To Remember

1. Single pointer variable stores the address of a normal variable
2. Double pointer variable stores the address of a single pointer variable
3. Triple pointer variable stores the address of a double pointer variable
4. And so on…

Example:

```
int marks;
int *p1;
int **p2;
marks = 70;
p1 = &marks;
p2 = &p1;
```



Example: Program for Pointer to Pointer

```
/* Program for Pointer to a pointer */
#include <stdio.h>
int main () {
    int  marks;
    int  *p1;
    int  **p2;
    marks = 70;     // assigns value to marks
    p1 = &marks;    // assigns address of marks to p1
    p2 = &p1;       // assigns address of p1 to p2
    printf("marks using marks = %d\n", marks );
    printf("marks using *p1   = %d\n", *p1 );
    printf("marks using **p2  = %d\n", **p2);
    printf("Address of p1  = %p\n", &p1);
    printf("Value at p2    = %p\n", p2);
    return 0;
}
```

Output:

marks using marks = 70
marks using *p1 = 70
marks using **p2 = 70
Address of p1 = 0x777
Value at p2 = 0x777

Explanation:

- p1 pointer variable stores the address of marks variable.
- p2 pointer variable stores the address of p1 pointer variable. It cannot hold the address of marks variable.

- *p2 is 0z777 (the address of p1)
- **p2 is 70 (value of marks variable); this is also written as *(*p2)
- *p1 is 70 (value of marks variable)
- marks is 70 (value of marks variable)

Compatibility:

Pointers are considered compatible using two factors.

1. **Data type**
2. **Level**

Data Type Compatibility - A pointer will save the address of a variable of same data types. This means, a int pointer will save only the address of a int variable. Hence, the Data type must be compatible.

```
int a=10;
char c = 'A';
int *p;
p = &a; //compatible data type
p = &c; //NOT compatible data type
p = (int *)c; //made compatible with Explicit Type Casting operator
```

Level Compatibility - A single pointer at level-1 can save the address of a variable at level-0. A double pointer at level-2 can save the address of a single pointer at level-1. A triple pointer at level-3 can save the address of a double pointer at level-2. And so on. A double pointer at level-2 cannot save the address of a variable at level-0. Hence, the level of address must be compatible.

Level	Address of	Pointer
0	variable (ex: int a= 10;)	int *p1 = &a;
1	pointer 1 (ex: int *p1;)	int **p2 = &p1;
2	pointer 2 (ex: int **p2;)	int ***p3 = &p2;
3	pointer 3 (ex: int ***p3;)	int ****p4 = &p3;

Pointer Expressions and Pointers Arithmetic (or Address Arithmetic) Operations

Pointers store addresses of variables; Pointers do not store normal data type values.
Hence, there are only a few arithmetic operations that are allowed to perform on Pointers.
The pointer arithmetic operations are

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer from a pointer
4. Comparison of pointers of the same type.

1. Increment & Decrement Operation on Pointer

A pointer is incremented by the size of its data type.
Formula for pointer increment:

```
pointerVar++;   // PointerAddress + Size of Pointer Datatype
```

A pointer is decremented by the size of its data type.
Formula for pointer decrement:

```
pointerVar - -;   // PointerAddress - Size of Pointer Datatype
```

Ex:
```
int arr[5] = {10,20,30,40,50};
int *ptr;
ptr = arr;        //ptr points to value 10
ptr++;            //ptr points to value 20
ptr - -;          //ptr points to values 10
```

Scale factor is possible on pointers. Pointers allow us to increase or decrease its value by the length (or size) of data type that it points to. This length is called scale factor.

2. Addition of integer to a pointer

When a pointer is added with a value, the value is first multiplied by the size of data type and then added to the pointer.

Formula: AddressAtPointer + (Number * SizeInBytesOfDatatype)

Ex: float sal[3]={11.10,20.20,30.30}; float *fptr = sal; //assume &sal[0] is address 7000
fptr = fptr + 2 ; // fptr = 7000 + (2 * 4 bytes) = 7008 bytes

3. Subtraction of integer from a pointer

When a pointer is subtracted with a value, the value is first multiplied by the size of the data type and then subtracted from the pointer.

Formula: **AddressAtPointer - (Number * SizeInBytesOfDatatype)**

Ex: float sal[3]={10.75, 20.75, 30.75}; float *fptr = &sal[2]; //assume &sal[2] is address 7008
fptr = fptr - 1 ; // fptr = 7008 - (1 * 4 bytes) = 7004 bytes

Addition Operation on Pointer

The formula for an Addition operation on pointer variables,

AddressAtPointer + (NumberToBeAdded * SizeInBytesOfDatatype)

4. Comparison of Pointers of same data type

The comparison operation is performed between the pointers of the same datatype only. We can use all comparison operators (relational operators) with pointers. We can't perform multiplication and division operations on pointers.

Example: Program to show all pointer arithmetic operations

```
#include<stdio.h>
int main()
{
    float a=10.75, *fptr, *fptr2 ;
    fptr = &a ;  // Asume address of a is 7000
    fptr2 = &a;
    printf("fptr value before increment: %u\n", fptr) ; //7000
    fptr++;  // fptr=7000 + 4 bytes=7004
    printf("fptr value after increment: %u\n", fptr) ; //7004
    fptr--;  // fptr=7000 - 4 bytes=7000
    printf("fptr value after decrement: %u\n", fptr) ; //7000

    fptr = fptr + 2;  // fptr=7000 + (2*4 bytes)=7008
    printf("fptr value after addition: %u\n", fptr) ; //7008

    fptr = fptr - 1;  // fptr=7008 - (1*4 bytes)=7004
    printf("fptr value after subtraction: %u\n", fptr) ; //7004

    float *p1, *p2;
    p1 = &a;
    p2 = &a;
    printf("p1=%u, p2=%u \n", p1, p2);
```

```
    if (p1>p2)
      printf("p1 is greater \n");
    else if (p1<p2)
      printf("p1 is lesser \n");
    else
      printf("Both p1 and p2 have same address \n");

   return 0;
}
```

Output:

```
fptr value before increment: 7000
fptr value after increment: 7004
fptr value after decrement: 7000
fptr value after addition: 7008
fptr value after subtraction: 7004
p1=7000, p2=7000
Both p1 and p2 have same address
```

Pointers to Arrays in C

- When we declare an array the compiler allocates memory for all array values.
- The address of the first element of an array is called the base address of that array.
- The array name stores the base address and is a constant pointer to the first element of that array.
- Since the array name is a constant pointer we can't modify its value.
- We can use the array name to access the address and value of all the elements of that array.

Ex: **int marks[6] ;**

Here, the compiler allocates 24 bytes of memory (6 x 4 bytes for int). The address of first memory location (i.e., marks[0]) is stored in a constant pointer called **marks**. That means in the above example, **marks** is a pointer to **marks[0]**.

Example: Program to show a pointer to an array

```
#include <stdio.h>
int main()
{
    int marks[4] = {55, 75, 95, 15} ;
    int *ptr ;
    ptr = marks ;
    printf("Base Address of 'marks' array = %u\n", ptr) ;
```

```
    ptr = marks + 2 ;
    //"ptr" is assigned with address of "marks[2]" element
    printf("Current Address of ptr = %u\n", ptr) ;
    printf("Value of 'marks[0]' = %d \n", *marks) ;
    printf("Value of 'marks[3]' = %d \n", *(marks+3)) ;

    return 0;
}
```

Output:

Base Address of 'marks' array = 6422284
Current Address of ptr = 6422292
Value of 'marks[0]' = 55
Value of 'marks[3]' = 15

In the above example program, the array name **marks** can be used as follows.
marks is same as **&marks[0]**
marks + 1 is same as &marks[1]
marks + 2 is same as &marks[2]
marks + 3 is same as &marks[3]
marks + 4 is same as &marks[4]
marks + 5 is same as &marks[5]
*marks is same as marks[0]
*(marks + 1) is same as marks[1]
*(marks + 2) is same as marks[2]
*(marks + 3) is same as marks[3]
*(marks + 4) is same as marks[4]
*(marks + 5) is same as marks[5]

Pointers to Multi-Dimensional Array

In case of multi-dimensional array also the array name acts as a constant pointer to the base address of that array.

Ex: **int marks[3][3] ;**

Here, the array name marks acts as a constant pointer to the base address (address of marks[0][0]) of that array.

In the above example of a two-dimensional array, the element marks[1][2] is accessed as ***(*(marks + 1) + 2)**.

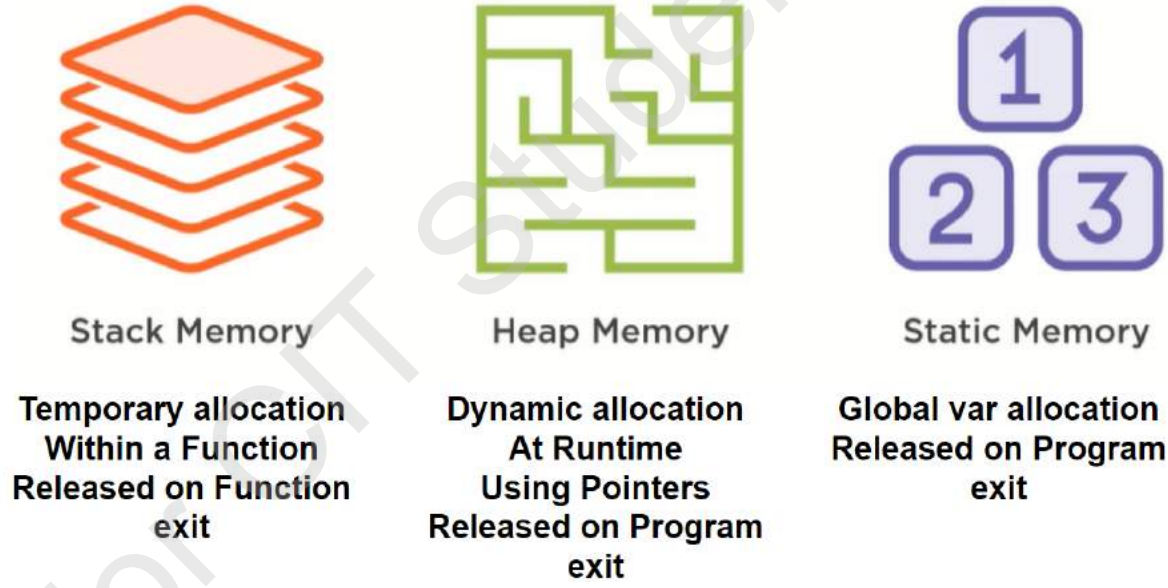
Dynamic Memory Allocation in C

When we declare variables, memory is allocated in the stack. The stack memory is fixed until the end of the program execution. When we create an array, we must specify the size at the time of the declaration itself and it can not be changed during the program execution. This is a major problem if we do not know the number of values to be stored in an array beforehand.

To solve this we use the concept of Dynamic Memory Allocation.

The allocation of memory during the program execution is called dynamic memory allocation.
or
Dynamic memory allocation is the process of allocating the memory manually at the time of program execution.

Dynamic memory allocation allocates memory from heap storage.



Following are the four standard library functions to allocate memory dynamically. They are defined in the header file "**stdlib.h**".

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

malloc()

malloc() is the standard library function used to allocate a memory block of specified number of bytes and returns void pointer. The void pointer can be casted to any datatype. If malloc() function unable to allocate memory due to any reason it returns NULL pointer.

Syntax

void* malloc(size_in_bytes)

calloc()

- calloc() is the standard library function used to allocate multiple memory blocks of the specified number of bytes contiguously and initializes them to ZERO.
- calloc() function returns void pointer. If calloc() function unable to allocate memory due to any reason it returns a NULL pointer.
- Generally, calloc() is used to allocate memory for array and structure. calloc() function takes two arguments and they are
    1. The number of blocks to be allocated,
    2. Size of each block in bytes

Syntax

void* calloc(number_of_blocks, size_of_each_block_in_bytes)

Example: Program for calloc().

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

int main () {
   int i, n;
   int *ptr;

   printf("Enter number of elements:");
   scanf("%d",&n);

   ptr = (int*)calloc(n, sizeof(int));
   printf("Enter %d numbers:\n",n);
   for( i=0 ; i < n ; i++ ) {
      printf("Enter value for Memory %u at %d : ", &ptr[i],i);
      scanf("%d",&ptr[i]);
```

```
    }

    return(0);
}
```

Output:

Enter number of elements:5
Enter 5 numbers:
Enter value for Memory 13178576 at 0 : 10
Enter value for Memory 13178580 at 1 : 20
Enter value for Memory 13178584 at 2 : 30
Enter value for Memory 13178588 at 3 : 40
Enter value for Memory 13178592 at 4 : 50

realloc()

- realloc() is the standard library function used to modify or expand the size of memory blocks that were previously allocated using malloc() or calloc().
- realloc() function returns void pointer.
- If realloc() function unable to allocate memory due to any reason it returns NULL pointer.

Syntax

void* realloc(*pointer, new_size_in_bytes)

free()

- free() is the standard library function used to deallocate memory block that was previously allocated using malloc() or calloc().
- free() function returns void pointer.
- free() function deallocates all blocks.

Syntax

void free(*pointer)

Example: Program for malloc(), realloc() and free()

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main () {
   char *branch;
```

```
    branch = (char *) malloc(15);
    strcpy(branch, "Electronics");
    printf("Initial allocation \n");
    printf("String = %s,  Address = %u, size = %d bytes\n", branch,
branch, strlen(branch));

    (char *) realloc(branch,30);
    strcpy(branch, "Electronics Engineering");
    printf("After Re-allocation \n");
    printf("String = %s,  Address = %u, size = %d bytes\n", branch,
branch, strlen(branch));
    free(branch);

    return(0);
}
```

Output:

Initial allocation
String = Electronics, Address = 14882472, size = 11 bytes
After Re-allocation
String = Electronics Engineering, Address = 14882472, size = 23 bytes

---

L-Value and R-Value

- L-value or locator value is an expression that designates an object. The object represents Location or Address in the memory.
- R-value is the result of evaluating an expression that will be assigned to L-value.

Example:

int i = 10; //Valid, because i has an address in memory and is a lvalue

int i;
10 = i; //Not Valid, 10 doesn't have a memory location and hence is an rvalue. So assigning the value of i to 10 doesn't make any sense.

Arrays of Pointers

Similar to an array of values, we can also use array of pointers in C.

Syntax:
dataType *arrayName[MaxSize];

Ex: **int *arr[5];**
Here,
arr[0] will hold address of one integer variable,
arr[1] will hold the address of another integer variable and so on.

Example1: Program for Array of Pointers

```
#include<stdio.h>
#define MAX 5
int main()
{
int *arr[MAX];
int a=10, b=20, c=30, d=40, e=50, i;
arr[0] = &a;
arr[1] = &b;
arr[2] = &c;
arr[3] = &d;
arr[4] = &e;

printf("Address of a = %u\n",arr[0]);
printf("Address of b = %u\n",arr[1]);
printf("Address of c = %u\n",arr[2]);
printf("Address of d = %u\n",arr[3]);
printf("Address of e = %u\n",arr[4]);

for(i = 0; i < MAX; i++)
{
  printf("At address = %u\t, the Value = %d\n", arr[i], *arr[i]);
}
return 0;
}
```

Output:

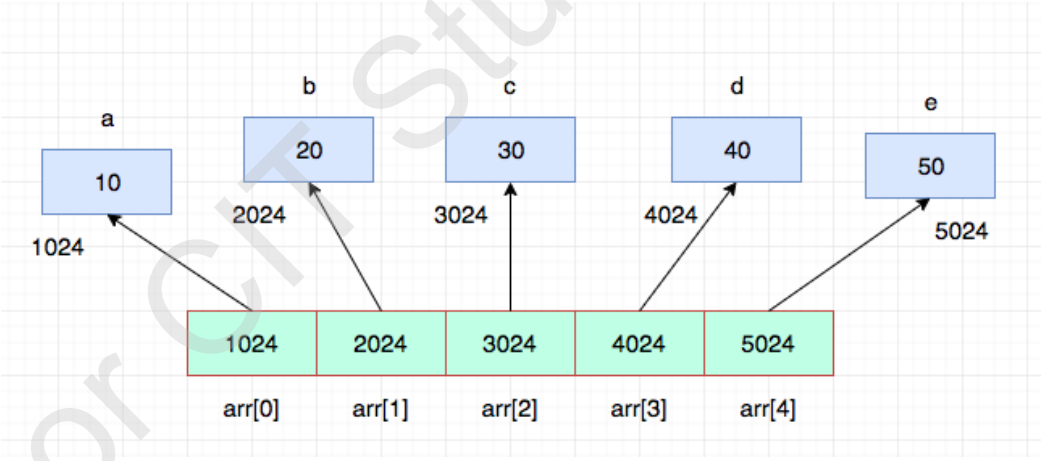
Address of a = 1024
Address of b = 2024
Address of c = 3024
Address of d = 4024
Address of e = 5024
At address = 1024, the Value = 10
At address = 2024, the Value = 20
At address = 3024, the Value = 30
At address = 4024, the Value = 40
At address = 5024, the Value = 50

Dereferencing array of pointer

Dereference is accessing the value stored at the pointer using * operator.

Since each array index pointing to a variable's address, we need to use ***arr[index]** to access the value stored at the particular index's address.

arr[index] will have address
***arr[index]** will get the value.



Explanation:

arr[0] is holding the address of the variable **a.** i.e. **arr[0] = 1024**

*arr[0] -> *1024 -> value stored at the memory address 1024 is 10. So, ***arr[0] = 10.**
Similarly, *arr[1] = 20, *arr[2] = 30 and so on.

Application of Array of Pointers

Assume that we are developing a software application that uses different sensors to track the temperature. In this case, we can use an array of pointers to hold the memory address of each sensor so that it will be very easy to manipulate the sensor status.

Example

sensor[0] will hold the address of the 1st sensor.
sensor[1] will hold the address of the second sensor and so on.
Since it is an array, we can directly interact with the particular sensor using array index.
We can get the temperature status of 1st sensor using sensor[0], 2nd sensor status using sensor[1] and so on.

Example2: Program for Array of Pointers using for loop

```
#include <stdio.h>
const int MAX = 3;
int main () {
   int  scores[] = {11, 22, 33};
   int i, *ptr[MAX];

   for ( i = 0; i < MAX; i++) {
      ptr[i] = &scores[i]; /* assigns the address of integer. */
   }
   for ( i = 0; i < MAX; i++) {
      printf("Value of scores[%d] at %p = %d\n", i, ptr[i], *ptr[i] );
   }
   return 0;
}
```

Output:

Value of scores[0] at 0061FEF8 = 11
Value of scores[1] at 0061FEFC = 22
Value of scores[2] at 0061FF00 = 33

Example3: Program for Array of Pointers using strings

```
#include <stdio.h>
const int MAX = 6;
int main () {
char *veg[] = {"Tomato", "Carrot", "Eggplant", "Spinach", "Pepper",
"Onion" };
int a = 0;
for ( a = 0; a < MAX; a++) {
  printf("Vegetable[%d] at %p = %s\n", a, &veg[a], veg[a] );
```

```
}
return 0;
}
```

Output:

Vegetable[0] at 0061FF04 = Tomato
Vegetable[1] at 0061FF08 = Carrot
Vegetable[2] at 0061FF0C = Eggplant
Vegetable[3] at 0061FF10 = Spinach
Vegetable[4] at 0061FF14 = Pepper
Vegetable[5] at 0061FF18 = Onion

Applications of Pointers

1. Pointers help **pass arguments by reference** to other functions.
2. Pointers are used for **accessing array elements.**
3. Pointers are the only way we can **return multiple values** from a function.
4. Pointers allow **Dynamic Memory Allocation** where users can take control of memory management.
5. Pointers play a crucial role in **implementing data structures.**
6. Pointers enable to do **system-level programming** where memory addresses are useful.
7. Pointers are used for **handling files** in OS.
8. Pointers are useful to **avoid copies of large data** and save memory space by using only the address.
9. Pointers **avoid memory wastage** when amount of data varies during program execution.

Advantages of Pointers:

- Pointers **save memory** space.
- **Faster Execution** time with pointers because data are manipulated with direct access to address.
- **Efficient access** to memory as the pointer assigns and releases the memory dynamically.
- Pointers are useful for representing **two-dimensional and multi-dimensional arrays**.
- Pointer arithmetic enable an array of any type **without array's index or subscript** range. (Access through address)

Disadvantages of Pointers:

- A pointer may read a wrong value if it points to an incorrect address.
- Segmentation fault can occur due to uninitialized pointer.
- Pointers might lead to a memory leak if we have not deallocated a memory after its use. So, we must use free(ptr) after the completion of your logic execution.

Example: Pointer Application on Arrays without using Index number

```
#include <stdio.h>
int main() {
  int x[5] = {10, 20, 30, 40, 50};
  int* ptr;
  // ptr is assigned the address of the third element
  ptr = &x[2];

  printf("*ptr = %d \n", *ptr);   // 30
  ptr=ptr+2;
  printf("*(ptr+2) = %d \n", *(ptr)); // 50
  printf("*(ptr-1) = %d", *(ptr-1));  // 20

  return 0;
}
```

Output:

*ptr = 30
*(ptr+2) = 50
*(ptr-1) = 40

Example: Pointer Application on Dynamic Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>
int main() {
int *a, i, n;
printf("Enter number of elements: ");
scanf("%d",&n);
a = malloc(n * sizeof(int));
for (i = 0; i < n; i++)
     a[i] = i;
printf("Size of array: %d bytes \n", n*sizeof(a));
free(a);
return 0;
}
```

Output:

Enter number of elements: 100
Size of array: 400 bytes

Storage Classes in Pointers

Every variable and function has a type and a storage class. The storage classes work on lifetime, scope/visibility and memory of variable where they are stored.

Four storage classes:

1. **auto**
2. **register**
3. **extern**
4. **static**

Storage Class	Keyword	Lifetime	Visibility	Initial Value
Automatic	auto	Function Block	Local	Garbage
External	extern	Whole Program	Global	Zero
Static	static	Whole Program	Local	Zero
Register	register	Function Block	Local	Garbage

auto

- Allocates memory in RAM
- All variables in a block or function are by default "auto" storage class.
- Use of "auto" keyword is optional.
- Memory allocated upon entering block, released at exit

Ex:

```
auto int a, *p;
```

register

- Allocates memory in CPU cache
- "register" keyword to allocates memory in CPU
- Recommended for small data variable
- Purpose is to speed program execution by keeping very frequently accessed variables (loop counters) immediately available

Ex:

```
register int i;
for (i = 0; i < MAX; i++)
…
```

extern

- Extern tells compiler to look for a variable in external file
- Extern can be used for data sharing between multiple C files.
- Must use "extern" keyword to tell the variable is declared in external file

Ex:

extern int *p;

```
file1.c
//Storage class extern parent; a header file
#include<stdio.h>
int pass_marks = 65;
int *p=&pass_marks;

file2.c
//Storage class extern child
#include<stdio.h>
#include "p_extern1.c"
extern void verify(int m);
int main(){
extern int *p;
int marks = 70;

if (marks>=*p)
  printf("Passed the exam");
else
  printf("Failed the exam");

return 0;
}
```

static:

- static variables hold their value even after they are out of their scope.
  - **Global scope -** variables are accessi for whole program
  - **Local scope -** variables are accessible only in a block or a function in which they were defined.
- Once declared and initialized they are not re-declared i.e., once the memory is allocated, then they exist till the termination of that program.
- Useful in multiple function calls. Static variable gets initialized only once and retains the changed values for all function calls or iterations;

```c
#include<stdio.h>
void display_count()
{   //auto int k = 0;      //prints 1 1 1
    static int k = 0;    //prints 1 2 3
    k = k + 1;
    printf("\n %d",k);
}
int main() {
int j;
for(int j=1;j<=3;j++)
  display_count();
return 0;
}
```

Special Notes

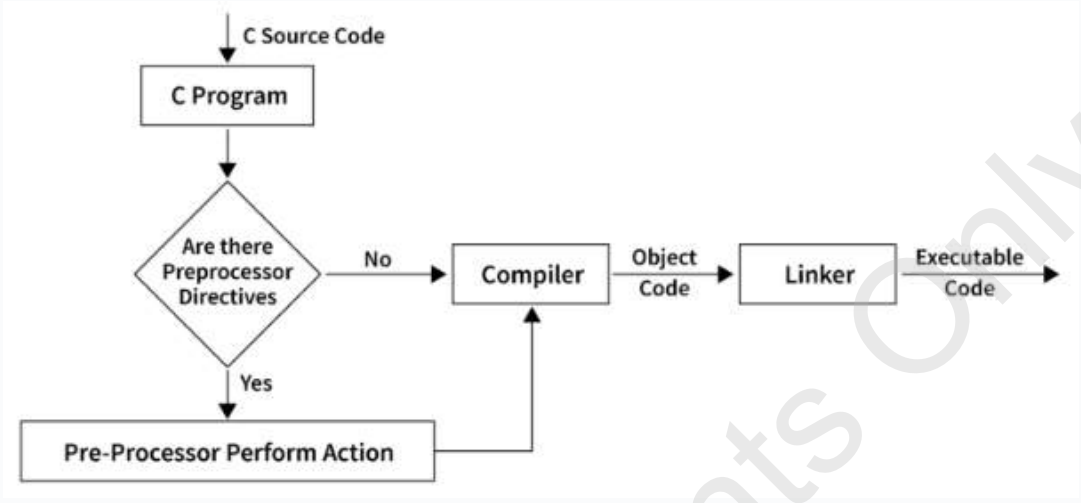
- **const int a;** /* a is a const int */
- **const int *b;** /* b is a pointer to a const int */
- **int * const c;** /* c is a const pointer to int */
- **const int * const d;** /* d is a const pointer to a const int */

Examples of Pointers to void

Valid	Invalid
int *p; float *q; void *v;	
p = 0; p = NULL;	
p = v = q;	
p = (int *) q;	p = q;
p = (int *)11;	P = 11
p = malloc(4 * sizeof(int));	

Preprocessor Directives in C

Preprocessor commands are used before compiling the C program. All the preprocessor directive names start with a hash # which means all the statements starting with # will first go to the preprocessor before the source code is compiled.



There are four types of derivatives in C.

1. Macros
2. File Inclusion
3. Conditional Compilation, and
4. Line Control derivatives.

List of Preprocessor Directives and their Purpose

Directive	Function
#include	Includes a header file in the source program
#define	Defines Macro substitution
#undef	Undefines Macro
#ifdef	Tests for a Macro definition
#ifndef	Checks whether a Macro is defined or not
#if	Checks a compile time condition
#elif	Checks another compile time condition
#else	Specifies alternative when #if condition fails
#endif	Specifies end of #if

Example: Program using selective proprocessor directives

```
#include <stdio.h>
#define Age 18   //Age is called a Macro
int main()
{
   #ifdef Age    // conditional compilation directives
   printf("This person is over %d years old.\n", Age);
   #endif  // also a conditional compilation directive

   printf("So, he is eligible.\n");
   return 0;
}
```

Output:

```
This person is over 18 years old.
So, he is eligible.
```
