Unit-4

Dynamic Data Structure in C - Pointers

* Pointer *

•> Introduction – Pointer is one of the strongest but also one of the most dangerous features in C. For example a pointer containing an invalid value can cause your program to crash.

Pointers are the most sophisticated feature of C In fact, the power and flexibility that C provides in dealing with pointers serves to set it apart from many other programming languages.

Pointers enable you to effectively represent complex data structures, to change values passed as argument to functions, to work with memory that has been allocated "dynamically". A pointers provide an indirect means of accessing the value of a particular data item.

•> Defination – Pointer is a derived datatype, also a variable that holds or stores the location of a data item in memory or address of variables. Pointers are declared as other datatype. The address of variable is the value of pointer.

Whenever we declare a variable, the system allocates somewhere in the memory, an appropriate location to hold the value of the variable. That location has a number called address.

Memory address are simple numbers, they can be assigned to some variables which can be stored in memory like any other variable. Such variables that hold memory addresses are called pointers.

Prepared By:-
Chandrashekhar Verma
An IT Instructor

•> Why we use Pointer?

Pointers are used frequently in C, as they offer a number of benifits to the programmers. They include-

(i) Pointers are more efficient in handling arrays and data tables.

(ii) Pointers can be used to return multiple values from a function via function arguments.

(iii) Pointers permit (references) to functions and thereby facilitating passing of functions as arguments to other functions.

(iv) The use of pointer arrays to character strings results in saving of data storage space in memory.

•> Rules for Performing Pointer Operations -

The following rules apply when performing Operations on pointer variable -

(i) A pointer variable can be assigned the address of another variable.

(ii) A pointer variable can be assigned the values of another pointer variable.

(iii) A pointer variable can be initialized with NULL or zero values.

(iv) A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.

(v) An integer value may be added or subtracted from a pointer variable.

(vi) When two pointers point to the same array one pointer variable can be subtracted from another.

(vii) when two pointers point to the objects of the same datatype, they can be compared using relational operator

(viii) A pointer variable can't be multiplied by a constant.

(ix) Two pointer variable can't be added.

(x) The value can't be assigned to an arbitary address.

•) Declaration of Pointer – Pointers are also variables and hence must be declared in a program like any other variable. The rules for declaring pointers variables name are the same as ordinary variables. The syntax for declaring pointers –

(114)

```
<datatype> *<pointer_Name> ;
```

•) Address Operator (&) – The symbol '&' is an address operator which is used to access the address of a variable and assign it to a pointer to initialize it.

•) Indirection Operator (*) – The symbol '*' is an indirection operator which is used to access the value of a variable through a pointer. The indirection operator is also known as ~~def~~ dereferencing operator.

Note – An address of variable should be assigned to a pointer before the indirection operator along with the pointer is used in any manipulation.

•) Initialization of Pointer Variable – The process of assigning the address of a variable to a pointer variable is known as initialization. All ~~unit~~ uninitialized pointers will have some unknown values that will be interpreted as memory addresses. The programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable.

Syntax –

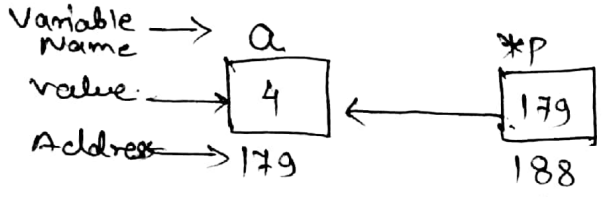
```
<pointer_Variable> = & <variableName> ;
```

Example -

```
int a=4;
int *p ;    // Pointer Declaration.
p = &a ;    // Initialization.
```

(115)

Memory Representation -



Prepared By:-
Chandrashekhar Verma
An IT Instructor

We can also combine the initialization with the declaration -

```
int *p = &a;
```

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable is one step.
Example -

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

// WAP to Explain Pointers in C.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int a=4, *p;
   clrscr();
   p=&a;
   printf("a= %d\n", a);
   printf("&a= %d\n", &a);
   printf("&a= p= %d\n", p);
   printf("&p= %d\n", &p);
   printf("a= *p= %d ", *p);
   getch();
}
```

Output -

```
a= 4
&a= 179
&a= p= 179
&p = 188
a= *p = 4.
```

⇒ Types of Pointers in C. - Types of pointer depends on available data types. Till we have seen about pointer int, char, float, double, void but we also declare the pointer array & structure. Different types of pointer in C are -

(i) NULL Pointer
(ii) Generic Pointer / void pointer
(iii) Dangling Pointer
(iv) Wild Pointer
(v) Complex Pointer
(vi) Near Pointer
(vii) Far Pointer
(viii) Huge Pointer.

1.) Null Pointer - It is always good practice to assign a NULL value to a pointer variable in case you don't have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called Null Pointer.

NULL pointer is a pointer which is pointing to nothing. NULL pointer points the base address of the segment.

// WAP to explain NULL Pointer.

```
# include <conio.h>
# include <stdio.h>
void main()
{
    int *ptr = NULL;
    clrscr();
    printf(" The value of ptr is %d", p);
    getch();
}
```

O/p :-

The value of ptr is 0

2.> Generic / Void pointer – It is also known as generalized pointer which can represent any type of variable. General purpose pointer is called as void pointer in C. It does not have any datatype associated with it. It can store address of any type of variable. A void pointer is a C convention for a raw address. The compiler has no idea what type of object a void pointer really points?

When a variable declared as being a pointer of type void, it is known as generic pointer. This is very useful when you want ~~to~~ a pointer to point to data of different types at different times.

Declaration Syntax –

```
void *pointer_name;
```


Example –

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int i=6;
  char c='a';
  void *p1;
  p1 = &i;
  printf(" The value of i = %d\n", *(int *)p1);
  p1 = &c;
  printf(" The value of c = %c", *(char *)p1);
  getch();
}
```

Output :-

```
The value of i = 6
The value of c = a
```

Advantages of Pointer –

(i) Pointers can be used to write compact & efficient programs.

(ii) Data at lower level in memory can be accessed using it address.

(iii) Dynamic memory allocation can be used to handle the values in memory using pointers.

(iv) Block of memory can be allocated to store values in an array by allocating the required number of bytes during runtime of the program.

(v) Pointers are efficiently used to handle a character string by using their addresses.

(vi) Pointer can be used to call function & access data members in a structure efficiently.

* Pointer to Pointer *

A pointer provides the address of the data item pointed to by it. The data item pointed to by a pointer can be an address of another data item. Thus a given pointer can be a pointer to a pointer to an object. Accessing this object from the given pointer then requires two levels of introduction. First, the given pointer is dereferenced to get the pointer to the given object, and then this later pointer is dereferenced to get to the object. It is also known as double pointer.

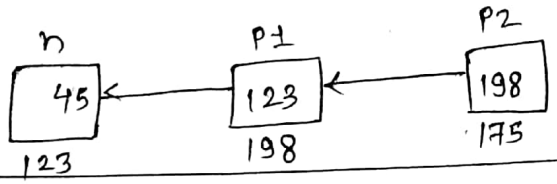
Pointer stores the address of variable, where as double pointer stores the address of the pointer variable. Double dereferencing (**) operator is used to denote double pointer.

Syntax –

<datatype> ** <variableName>;

Example –

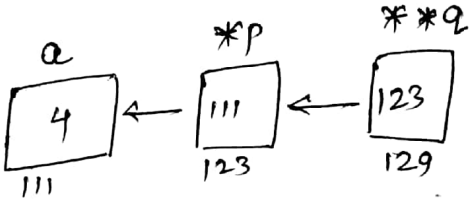
```
int n=45, *p1, **p2;
p1 = &n;
p2 = &p1;
```


```
*p1 = 45
**p2 = 45
p1 = &a
p2 = &p1
```

// WAP to Explain Pointer to Pointer –

```
#include <stdio.h>
#include <conio.h>
void main()
{
  int a=4, *p, **q;
  p=&a;
  q=&p;
  printf("a= *p = **q = %d\n", **q);
  printf(" q = &p = %d\n", q);
  printf(" &q = %d\n", &q);
  getch();
}
```



O/p –

```
a= *p = **q = 4
q = &p = 123
&q = 129
```

* Pointer in Math Expression. *
or
* Pointer Expression *

Pointer Variable can point to simple variable, array, function & also to other pointer, because –

(i) It assigns the address of a variable & other pointers.

(ii) It can also assign NULL (zero) value.

Pointer Expression –

In general, expressions involving pointers confirm to the same rules as other expression. This section examines a few special aspects of pointer expressions :-

(i) Pointer Assignments
(ii) Pointer Arithmetic (Increment, Decrement, Addition Differencing, Comparison, Subtraction)
(iii) Pointer Conversion

1.) Pointer Assignment –

We can use a pointer on the right hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straight forward.

Example –

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int x=25;
   int *p1, *p2;
   clrscr();
   p1 = &x;
   p2 = *p1;
   printf("value of x = %d", x);
   printf("\n Address of x = %u", &x);
   printf("\n Value of p1 = %u", p1);
   printf("\n value of p2 = %u", p2);
   getch();
}
```

Output –

```
value of x = 25
Address of x = 25367
Value of p1 = 25367
value of p2 = 25367
```

2.) Pointer Arithmetic – There is 6 types of pointer arithmetic - given as follows -

⇒ (i) Incrementing Pointer :- It generally used in array because we have contiguous memory in array and we know the contents of next memory location. Incrementing Pointer variable depends upon data type of the pointer variable.

(121)

- Syntax – (Formula)

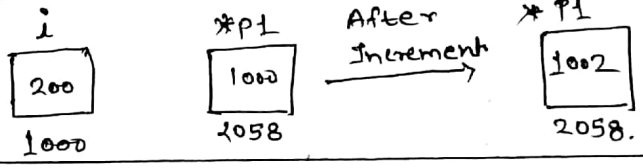
```
new_value = current_Address + i * sizeof(datatype);
```

- Three rules should be used to increment pointer –

```
Address + 1  =  Address
Address ++   =  Address
++ Address   =  Address.
```


- Memory Representation –

```
int i = 20, *p1;
p1 = &i
```



- Example for each datatype –

DataType	Older Address Stored in pointer	Next Address After increment (++)
int	1000	1002
float	1000	1004
char	1000	1001

Incrementing a pointer to an integer data will cause its value to be incremented by 2. This differs from compiler to compiler as memory required to store integer vary compiler to compiler. Increment and Decrement operator on pointer should be used when we have continues memory.

• Example – // WAP to Incrementing / Decrementing Pointer.

(192)

```
#Include <stdio.h>
#include <conio.h>
void main()
{
 int a=13, *p1;
 p1 = &a;
 clrscr();
 Printf("\n Befor Operation");
 Printf(" value of a = %d", a);
 printf("value of p1 = %u", p1);
 a=a+1;
 P1= P1+1  // p1++;
 Printf("\n After Operation (Incrementing)");
 printf(" value of a = %d", a);
 Printf("\n value of p1 = %u", p1);
 P1=P1-1 // P1--;
 a= a-1
 Printf("\n After Operation (Decrementing)");
 Printf(" value of a = %d", a);
 printf(" value of p1 = %u", P1);
 getch();
}
```


Output :-

```
Before Operation
Value of  a = 13
Value of p1 = 5000
After Operation (Incrementing)
Value of a = 14
Value of p1 = 5002
After Operation (Decrementing)
Value of a = 13
Value of p1 = 5000.
```

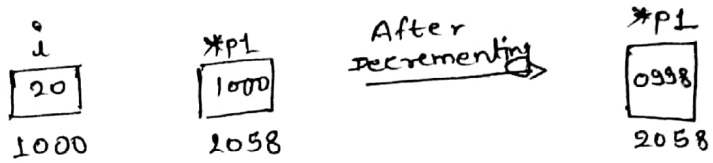
⇒ (ii) Decrementing Pointer :-

• Syntax - (Formula).

```
new_value = Current_Address - i * sizeof (datatype);
```

- Memory Representation -

```
int i=20, *p1;
p1 = &i;
```



- Example for Each DataType -

DataType	Older Address stored in pointer	New Address After Decrement (--)
int	1000	0998
char	1000	0999
float	1000	0996

Decrementing a pointer to an integer data will cause its value to be decremented by 2.

⇒ (iii) Addition Integer Value with Pointer - In C programming we can add integer number to pointer variable. It is perfectly legal in C to add integer to pointer variable.

- Syntax - (formula)

New_value = (address) + (number * size of data type)

⇒ (iv) Subtraction Integer value with Pointer - Suppose we have subtract "n" from pointer of any data type having intial address as "init_address" then after subtraction. we can write -

- Syntax - (formula)

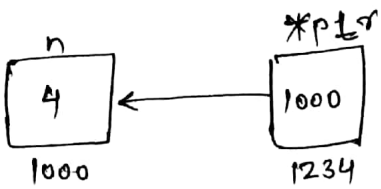
new_value = (address) - (number * size of data type)

- Example -

ptr = init_address - n * (sizeof (datatype));

• Example – // WAP to Addition/Subtraction Integer Value to Pointer

```
#include<stdio.h>
#include <conio.h>
Void main()
{
   int *ptr, n=4;
   ptr= &n;
   ptr= p1 + 3 ;
   clrscr();
   printf (" New value of ptr = %u", ptr);
   ptr= ptr-34;
   printf (" New value of ptr = %u", ptr);
   getch();
}
```



O/p –
New value of ptr = 1006
New value of ptr = 992.

⇒ (v) Differencing / Subtracting Pointers – Differencing means Subtracting two pointers. Subtraction gives the Total number of Objects between them. Subtraction indicates "How apart the two Pointers are ?".

• Example – // WAP to Differencing Pointers.

```
#include<stdio.h>
# include <conio.h>
 void main()
{  int num, *p1, *p2;
   p1 = &num;
   p2 = p1 + 2;
   printf ("%d", p2-p1);
   getch();
}
```

Output –
2

- Explanation – Suppose the Address of Num = 1000.

Statement	Value of p1	Value of p2.
int num, *p1, *p2;	Garbage	Garbage
p1 = #	1000	Garbage
p2 = p1 + 2;	1000	1004
p2 - p1 ;	1000	1004

- Syntax – (Formula)

Result = (p2-p1) / size of Data Type.

Step : 1 Compute Mathematical Difference (Numerical Difference).

p2 - p1 = value of p2 - value of p1
= 1004 - 1000
= 4

Step : 2 Finding Actual Difference (Technical Difference)

Result = 4 / size of Integer
= 4/2
= 2

Numerically subtraction (p2-p1) differs by 4. As both are Integers they are numerically Differed by 4 and Technically by 2 objects. Suppose Both pointers of float then they will be differed numerically by 8 and Technically by 2 objects. Consider the below statement and refer the following ~~data~~ table -

```
int num = p2 - p1;
```

If Pointers DataType are	Numerical Difference	Technical Difference
Integer	4	2
Float	8	2
character	1	1

⇒ (vi) Comparing Two Pointer Variable – Pointer Comparison is valid only if the two pointers is are pointing to same array. All relational operators can be used for comparing pointers of same type. All Equality & Inequality operators can be used to with all pointer types. Pointer can't be divided or multiplied.

- Example –


```
#Include<stdio.h>
#include<conio.h>
void main( )
 {
   int *p1, *p2;
   float *p3;
   p1 = (int *) 1000;
   p2 = (int *) 2000;
   p3 = (float *) 3000;
   if ( p2 > p1)
        printf(" p2 is far from p1");
  if (p3 > p2)
        printf(" p3 is far from p2");
  getch();
}
```

In C language, it is legal to compare two pointer of different data type can be compared. Following operations on pointers :

>	⇒	Grater Than
<	⇒	Less Than
>=	⇒	Grater Than & Equal To
<=	⇒	Less Than & Equal To
==	⇒	Equals
!=	⇒	Not Equal.

One can perform different arithmetic operations on pointer such as increment, decrement but still we have some more arithmetic operations that can't be performed on pointer -

(i) Addition of two addresses.
(ii) Multiplying two addresses.
(iii) Division of two addresses.
(iv) Modulo operation on pointer.
(v) Can't perform bitwise AND, OR, XOR operations on pointer.
(vi) Can't perform NOT operation or negation operation.

* Pointers Vs Arrays. *

• Pointer to Array — The elements of an array can be accessed by using pointer. The following facts regarding the relation between pointers & arrays should be noted -

- Array elements always stored in contiguous memory location irrespective of the size of the array.
- The size of the data type which the pointer variable refers to, is dependent on the data type pointed to by the pointer.

A pointer to Array is a pointer which points the memory address of an array.

Example -

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int ary[5] = {1,2,3,4,5}, i, *ptr;
  ptr = ary;
  for (i=0; i<5 ; i++)
  {
    printf("%u \t %d \n", ptr, *ptr);
    ptr ++;
  }
  getch();
}
```

Output →

```
65516   1
65518   2
65520   3
65522   4
65524   5
```

Array of Pointers

An array of pointers is similar to an array of any predefined data type. As a pointer variable always contains an address, an array of pointers is a collection of addresses. These can be addresses of ordinary isolated variables or of array elements.

The elements of an array of pointers are stored in the memory just like the elements of any other kind of array. All rules that apply to other arrays also apply to the array of pointers.

Example –

```
#include <stdio.h>
#include <conio.h>
void main()
{
   int a[5] = { 1,2,3,4,5 }, i, *p[5];
   clrscr();
   for (i=0 ; i<=4 ; i++)
   {
      p[i] = &a[i];
   }
   for (i=0 ; i<5 ; i++)
   {
      printf("%u \t %d\n", p[i], *p[i]);
   }
   getch();
}
```

Output –

```
65516    1
65518    2
65520    3
65522    4
65524    5
```

"Pointer Vs Functions"

* Pointer to Function *

(29) Another confusing but powerful feature of 'C' is the function pointer. Like Integers, Characters & floats have addresses in memory which can be referred with the help of pointer function also have a physical address In the memory. This address is the entry point of the function which can be assigned to a pointer & the pointer can be used to invoke the function.

We can obtain the address of a function with the help of name of function without any paranthes This is similar to the way an array's address is obtained when only the array name without indexes is used.

```
// A program to demonstrate how to obtain address
   of a function using only function name.
#include <stdio.h>
#include <conio.h>
void disp();
void main()
{
  clrscr();
  printf("Address of disp() function : %d \t %u ", disp, disp);
  disp();
  getch();
}
void disp()
{
  printf("\nHello");
}
```


output -

```
Address of disp() function : 689    689
Hello
```

// A program to demonstrate pointer to function or invoking of a function using pointer.

(30)

```
#include <stdio.h>
#include <conio.h>
int sum (int , int );
void main()
{
  int a,b, c;
  int (*p) (int ,int)  // Declaration of pointer to a function
  printf (" enter two numbers :");
  scanf (" %d%d", &a, &b);
  p= sum ;          // Assigning address of function.
  c= (*p)(a,b);     // Invoking the function.
  printf (" sum = %d", c);
  getch();
}
int sum (int a, int b)
{
   return (a+b);
}
```


Note - Here *p is a function pointer.

Output :-

enter two numbers : 4 6 ↵
Sum =10.

* Function Returning Pointer *

A function can return a single value by its name or return multiple value through pointer parameters. Since pointers are a datatype in C, we can also force a function to return a pointer to the calling function.

// A program to demonstrate function returning pointer.

```
# include <stdio.h>
# include <conio.h>
int *large (int,int);
void main()
{
   int a,b;
   int *p;
   printf(" enter two number : ");
   scanf(" %d %d", &a, &b);
   p= large (a,b);
   printf(" large value = %d ", *p);
   getch();
}
int *large (int a, int b):
{
   if (a>b)
      return &a;
   else
      return &b;
}
```


Output :-

```
enter two number : 4  7 ↵
large value = 7
```

The function "large" receive the address of variable a and b, devides which one is large using the pointer. Then return the address of its locations. The return value is then assign to the pointer variable p in the calling function.

* Passing Pointer to Function * (Pointer as Function Argument)

A pointer can be used as an argument in function declaration. When a function with a pointer argument is called the calling program will pass the address (not a value) of a variable to the argument.

In C language, a function can be called by the calling program in two ways -

1. Call by value
2. Call by reference.

1.) Call by Value - When a function is called by the calling program, the values to the arguments in the function are supplied by the calling program. The values supplied can be used inside the function. Any alteration to the value inside the function is not accepted in the calling program but the change is locally available in the function. This method is refered as calling a function by value.

```
// Program that calls a function by value.
#include<stdio.h>
#include<conio.h>
void main()
{
   int a=40, b=10;
   void add(int x, int y);
   clrscr();
   printf("\n Before function call \n a=%d \t b=%d",a,b);
   add(a,b);
   printf("\n After function call \n a=%d \t b=%d",a,b);
   getch();
}
void add(int x,int y)
{
    x= x+10;
    y= y+10;
    printf("\n Inside Function \n a=%d \t b=%d", x, y);
}
```

Output -

```
Before function call
  a= 40      b=10
Inside function
  a= 50      b=20
After function call
  a= 40      b=10
```

2.) Call by Reference - A function can be declared with pointers as its arguments. Such functions are called by the calling program with the address of a variable as argument from it. The addressess of the variables are substituted to the pointers and any alteration to its value inside the function is automatically carried out in the location. The change is directly ~~mand~~ made and is accepted by the calling program. This method is referred as calling a function by reference.

```
// Program that calls a function by references.
#include<stdio.h>
#include<conio.h>
void main()
{
   int a=40, b=10;
   void add (int *x, int *y);
   clrscr();
   printf("\n Before function call \n a=%d \t b=%d", a, b);
   add (&a,&b);
   Printf("\n After function call \n a=%d \t b=%d", a, b);
   getch();
}
void add (int *x, int *y)
{
   *x = *x+10;
   *y = *y +10;
   printf("\n Inside function \n a=%d \t b=%d", *x, *y);
}
```

Output -
```
Before function call
  a=40    b=10
Inside function
  a=50    b=20
After function call
  a=50    b=20
```


* Pointer to Structure *

We Known that the name of an array stands for the address of its zeroth element. The same thing is true of names of arrays of structure variable. Suppose product is an array variable to struct type. The name product represents the address of its zeroth element.

The address of a given structure variable can be obtainted by using the '(&) operator. Structure pointer only store the structure variable address.

⊙ Accessing Structure member Using pointer –

Structure member may be accessed using pointer to structure this is pointer through the use of a special access operation (->) this is called arrow operator. The arrow operator (->), A hyfen followed by a grater than (>) symbol also belongs to the highest precident group like that (.) dot operator.


```
// WAP to explain pointer to structure.
#include<stdio.h>
#include<conio.h>
struct stud
{
    char name[20];
    int age;
};
void main()
{
  struct stud a, *p;
  p=&a;
  clrscr();
  printf(" enter student name & age : ");
  scanf(" %s %d", &a.name, &a.age); //scanf("%s%d", p->name, p->age);
  printf("\n Name = %s ", p->name);
  printf("\n Age = %d ", p->age);
  getch();
}
```

Output →

```
Enter student name & age : Ajay
23
Name = Ajay
Age = 23
```


* Pointer Vs Constant *

⊙ **Constant Pointers** – A constant pointer is a pointer that can't change the address its holding. In other words we can say that once a constant pointer points to variable than it can't point to any other variable.

These type of pointers are the one which can't change address they are pointing to. This means that suppose there is a pointer which points to a variable (or stores the address of that variable). Now if we try to point some other variable (or try to make the pointer store address of some other variable), then constant pointers are incapable of this.

Constant Pointers can't be modified. Modification in Integer to which it points to is allowed. Modification made in pointer is not allowed.

Syntax to declare Constant Pointer –

<datatype> * const <pointerName>

Example –

```
# include <stdio.h>
# include <conio.h>
void main()
{
  int var1=70, var2=0;
  int * const ptr = &var1;
  clrscr();
  //ptr = &var2; //cause Error- can't modify a const object
  printf("%d \t %u", *ptr, ptr);
  getch();
}
```

Output –

70 65524

·> Pointer to Constant – As evident from the name, a pointer through which one can't change the value of variable it points is known as pointer to constant. These type of pointers can change the address they point to but can't change the value kept at those address.

These type of pointers are the one which can't change the value they are pointing to. This means they can't change the value of the variable whose address they are holding.

Syntax to declare pointer to Constant -

Const <datatype> * <pointerName>;

Example -

```
# include<stdio.h>
# include <conio.h>
void main()
{
   int var1=4;
   Const int *ptr = &var1;
   //*ptr = 1;   //Cause Error - Can't modify a const object.
   printf("%d\t%u\n", *ptr, ptr);
   getch();
}
```

Output -

4 65530.

- Constant Pointer to a Constant - It is a mixture of Constant Pointer & Pointer to constant. A Constant Pointer to a constant which is a pointer that can neither change the address it is pointing and nor it can change the value kept at that address.

Syntax -

Const <datatype> * const <pointerName>;

Example -

Const int * const ptr;

//WAP to explain constant pointer to a constant

```
#include <stdio.h>
# include <conio.h>
void main()
{
   int v1=4, v2=6;
   Const int *const ptr = &v1;
   // *ptr = 1;
   // ptr = &v2;
   printf("%d\t%u", *ptr, ptr);
   getch();
}
```

-> Both cause Error - Can't modify a const object.

Output -

4 65530.

137 ⇒ Difference between Constant Pointer & Pointer to Constant

S.No.	Pointer to Constant	Constant Pointer.
1.	*ptr = 20, statement is invalid in pointer to constant i.e. Assigning value is illegal.	*ptr = 20 is valid in constant pointer i.e. assigning value is perfectly legal.
2.	ptr ++ statement is valid in pointer to constant	p++ statement is invalid in constant pointer.
3.	Pointer can be incremented or decremented.	Pointer can't be incremented or decremented.
4.	Pointer is pointing to Constant data. object	Constant pointer is pointing to data. object
5.	Declaration - const int *ptr;	Declaration - int * const ptr;

* Pointer & String / Pointer & Character Array *

Pointer to string / pointer to array of string is a pointer which pointing to an array which content is string, this is known as pointer to strings.

Example -

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char *p, str[20];
    p = str;
    clrscr();
    printf("enter string - ");
    gets(str);
    while (*p != '\0')
    {   printf("Address = %u \t value = %c \n", p, *p);
        p++;
    }
    getch();
}
```

Output -

```
enter string = Hello
Address = 65506   value = H
Address = 65507   value = e
Address = 65508   value = l
Address = 65509   value = l
Address = 65510   value = o
```

* Dynamic Memory Allocation *

memory is allocated in two ways :-

(i) static memory allocation.
(ii) dynamic memory allocation.

1.> Static Memory Allocation - Static means anything that happen during compilation. In static allocation memory is allocated during compile time & memory can't be increased while executing program.

Certain problems related with static allocation are -

(a) We should know the free space memory needed for allocation.

(b) Without making correction to the program we can't increase or decrease the size.

•> Memory allocation Process - Global variables, static variables & program instructions get their memory in permanent storage area whereas local variable are stored in memory area called Stack. The memory space between these two region is known as Heap area. The region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.

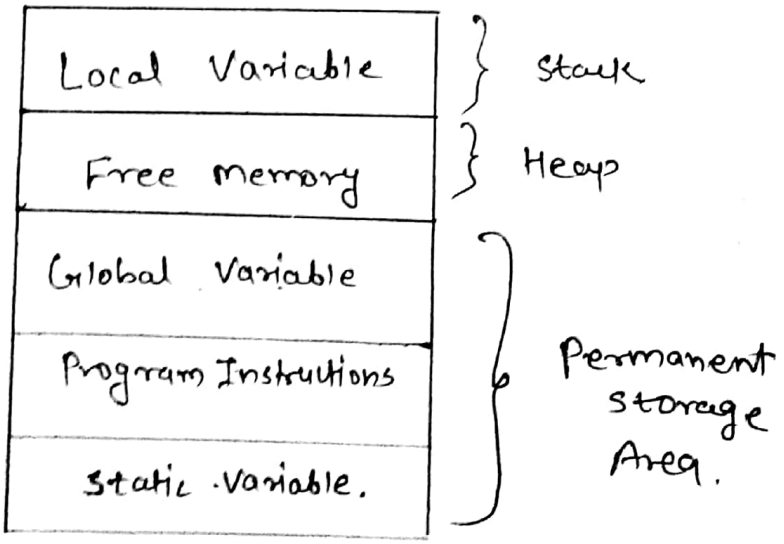


fig. - Storage of a C Program.

2.) Dynamic Memory Allocation – The process of allocating & de-allocating the memory at run time it is called as dynamic memory allocation. Dynamic memory allocation is the solution of the problem of static.

When we are working with array or string static memory allocation will take place that is compile time memory management. When we are allocating the memory at compile we can't extend the memory at run time, if it is not sufficient. By using compile time memory management we can't utilize the memory property. In implementation when we need to utilize the memory more effieiently then go for dynamic memory allocation.

By using dynamic memory allocation whenever we want which type we want or how much we type that time & size and that we much create dynamically.

Dynamic memory allocation refers to the method of allocating a block of memory and releasing it when the memory is not required at the time of running the program. A block of memory can be used to store values of simple or subscripted variables & a block of memory can be allessed using a pointer.

Following are the functions used in dynamic memory allocation & these functions are available in the header file alloc.h and stdlib.h,

(i) malloc() – allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space. It allocates single block of requested memory.

(ii) calloc() – allocates space for an array of elements, initialize them to zero & then returns a void pointer to the memory. allocates multiple block of requested memory.

(iii) realloc() - reallocates or modify the size of previously allocated space occupied by malloc() or calloc() function.

(iv) free() - release previously allocated memory.

•> malloc() :- It is used to allocate a single block of memory & to store values of specific datatype. It also assign the address of the 1st byte of the allocated space to a pointer. It assign Garbage Value initially to all blocks in memory.

Syntax —

```
ptr = (datatype *) malloc (No_of_element * sizeof(datatype));
```

Here —

(a) (datatype *) :- type conversion or typecasting refers to changing an entity of ~~th~~ one ~~dtu~~ datatype into another. Similarly address of block of memory is casted to address of pointer.

(b) No_of_elements :- It is nothing but the "No of element created at run time".

(c) sizeof (datatype) :- It calculates the memory required for variable, datatype etc.

Example —

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

In the above example The total allocation will be of 20 bytes. If the above line will execute successfully then 20 byte will be allocated otherwise it return NULL.

The allocated space can be used to 10 int type values and this block of memory can be accessed by the pointer ptr of the same type.

// A Program to demonstrate malloc() function in C.

```
#include<string.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main()
{
  char *m;
  clrscr();
  m = (char *) malloc (20 * sizeof(char));
  if (m==NULL)
      printf("Couldn't be able to allocate");
  else
      strcpy(m,"Hello World");
  printf("DMA Content : %s", m);
  free(m);
  getch();
}
```

Output :-

DMA Content : Hello World

(141)

•> Calloc() :- Calloc() function is used to allocate memory in multiple blocks of same size during the program execution. The space allocated is used to store values of an array of structure. This function assigns the address of the 1st byte of the allotted space to a pointer. It also asign zero as the initial value for all bytes in the block of memory. Calloc() function takes two argument - no_of_element, & sizeof each element.

Syntax -

```
ptr = (datatype *) calloc (no_of_elements, sizeof(datatype));
```

Example -

```
ptr = (int *) calloc (10, sizeof(int));
```

In the above example it creates 10 blocks and each block size of 2 byte (size of int). The total space allocated is 20 bytes. The allocated space is used to store 10 int type values & are accessed by the pointer ptr of the same type. Note that a NULL pointer is returned when there is insufficient space in memory.

// WAP to demonstrate calloc() function.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
   int *p, size, i;
   clrscr();
   printf("enter size :");
   scanf("%d", &size);
   p = (int *) calloc (size, sizeof(int));
   if (p==NULL)
   {
      printf(" no allocation");
      exit (0);
   }
   for (i=0; i<size ;i++)
   {
      scanf (" %d", p);
      p++;
   }
   p--;
  for (i=0; i<size; i++)
  {
     printf (" %u = %d\n", p, *p);
     p--;
  }
  free (p);
  getch();
}
```

Output –

```
enter size = 5
    23
    45
    34
    22
    11

1876 = 11
1874 = 22
1872 = 34
1870 = 45
1868 = 23
```


•> realloc() :- realloc() function is used to modify or reallocate the memory space which is previously ~~allocated~~ allotted. This facilitate to increase or reduce the allocated space at a later stage in a program. It is just a reallocation of memory & the first byte address of the newly allotted space is assigned to the pointer.

If enough space doesn't exist in memory of current block to extend, new block is allocated for the full size of reallocation, then copies the existing data to new block & then frees the old block.

Syntax –

```
ptr = (datatype *) realloc (pointer, newsize);
```

Here –

The 'pointer' is a name of pointer to previous block. and the newsize is a size ~~of~~ for reallocated memory space.

Example –

```
ptr = (int *) malloc (100);
.....
ptr = (int *) realloc (p, 400);
```


In the above example the previously allotted space of 100 bytes to the pointer ptr is modified as 400 bytes at a later stage of the program.

•> free() :- free() function is used to release the memory space which is ~~allocate~~ allotted using malloc() or calloc() or realloc() function. Only one argument is required in this function i.e. pointer. The free function returns the memory pointed to by pointer to the heap. This makes the memory available for future allocation.

Syntax –

```
free (ptr);
```

Here ptr is a pointer.

The free() function is the opposite of malloc() in that it returns previously allocated memory to the system. Once the memory has been freed, it may be reused by a subsequent call to malloc().

// WAP to demonstrate realloc() & free().

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{
   char *m;
   m = (char *) malloc (20 * sizeof(char));
   if (m == NULL)
      printf("No allocation");
   else
      strcpy (m, "Hello");
   printf("DMA Content : %s\n", m);
   m = (char *) realloc (m, 100 * sizeof(char));
   if (m == NULL)
      printf("No allocation");
   else
      strcpy (m," space is extended upto 100 characters");
   printf("DMA Resized memory content : %s", m);
   free(m);
   getch();
}
```


Output –
DMA Content : Hello
DMA Resized Memory content : space is extended upto 100 characters.

⇒ Difference between calloc() and malloc().

SNO.	Calloc()	malloc()
1.	It is used to allocate storage.	It is used to allocate single memory at run time.
2.	The storage area is automatically set to zero.	The storage area automatically set to garbage value.
3.	It is used to allocate same memory size to each item.	It is used to provide different memory to each item.
4.	Used to store value of structure & array.	Used to store value of datatype.
5.	Syntax – ptr = (datatype *) calloc (no_of_item, size);	Syntax – ptr = (datatype*) malloc (size);
6.	It takes two argument.	It takes one argument.