Pointers in C++

* What is a pointer?

- A pointer is defined as a variable that stores the memory address of any other variable.
- It is denoted by an asterisk (*) symbol called indirection operator (or) dereference operator.

* Features of pointers

(i) Execution time with pointer is faster because data is manipulated directly using the address.

(ii) Supports dynamic memory allocation and de-allocation

(iii) Offers high flexibility in management of data.

(iv) Used for creating data structures such as linked lists, trees, graphs, etc.

---

* Pointer Definition

- A pointer is defined like any other variable with appropriate data type.
- But the pointer variable is preceded by asterisk (*) symbol.

Syntax:

```
Datatype * ptrVar, ... ;
```

Datatype → primitive data type (or) user defined (structure & class)

ptrVar → variable name.

(Eg) int *x ;
float *f ;

- Here '*' informs the compiler that,

→ x is an integer pointer and it holds address of integer variable

→ f is a float pointer and holds address of float variable

- The `*` is also called indirection (or) dereference operator (or) value at address

- The indirection operator is used in two ways :
    - For definition *
    - Dereferencing .

* Using Address Operator (&)

- The pointer variable must be bound to memory location.
- It is achieved by assigning address of a variable obtained using address operator (&)

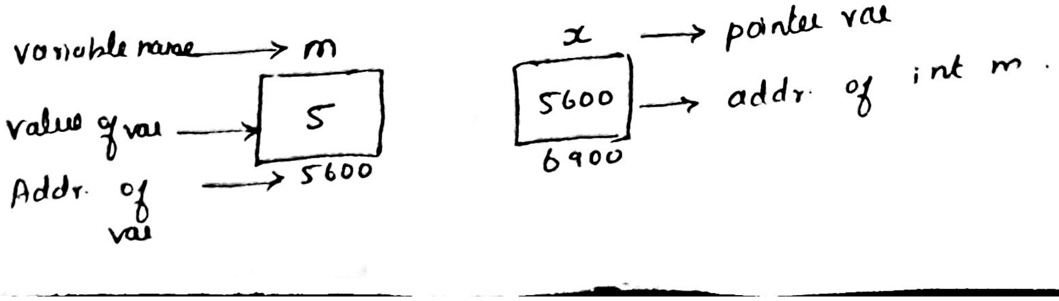
```
     int m = 5 ;
(Eg) int *x; /* Definition of pointer variable */
     x = &m; // Assigning address to pointer
```

* Dereferencing of Pointers

- Dereferencing is the process of accessing and manipulating data stored in memory location pointed to by a pointer .
- The operator * is used to dereference pointers in addition to defining them .
- For the above example , the contents of m is displayed using the stmt

```
cout << *x ;   ⟹ Similar to   cout << m ;
```

- Thus accessing information using pointers is called indirect addressing .



- The contents of memory locations can be modified using the pointer variable

  `*x = 10;`

- Also, the contents of memory location can be read using pointer variable.

  `a = *x`

(Eg)

```
#include <iostream.h>
void main( )
{
    int *p, a, b;
     a = 10 ; b = 20;
     p = &a;
     cout << *p ;    // prints 10
     p = &b;
     cout << *p ;    // prints 20.
     *p = 100;
     cout << *p << b ; // prints 100, 20.
}.
```

Pointers and parameter passing:

- provides two way communication b/w. service requester and service provider.
- address of actual parameters is passed instead of values.

Program for call by reference using pointers

```
# include<iostream.h>

 void   addn ( int *x , *y);

 void main( )

  {
int a = 25 ,b = 10 ;
   cout << " Before fn call "<<"\n";
    cout << " a is "<< a <<endl ;
    cout << " b is " << b ;

    addn (& a , &b);
   Cout << " After fn call " << endl ;
    cout << ' a and b " << a << " " << b ;

  }
 void   addn ( int *x , int *y )
  {
     *x  =  *x+10 ;
     *y  =  *y+10 ;
     cout << " Inside fn" << endl
     cout << " a and b" << *x << " " << *y ;
  }
```

void pointers.

- Also known as generic pointer
- points variable of any data type.

Syntax :

```
void *ptr ;
```

- Uses reserve word 'void' for specifying pointer type.
- Void pointers do not have any type associated with them.
- can hold address of any variable type.

(Eg)

```
void *vptr; int *ptr;
int a;
char c;
vptr = &a   // valid
vptr = &c   // valid.
ptr  = &c   // invalid ⇒ int pointer is assigned to
                         address of char variable.
```

- Since vptr is a void pointer, it can be assigned to address of integer and character variable.

Dereferencing void pointer:

* Prior to dereferencing a pointer to void, it must be typecasted to required data type.

Syntax :

```
*((int *) vptr)
```

void pointer.

pointer typecasting

Example

```
# include <iostream.h>
void main( )
{
  int  a = 100 ;
  void  *ptr ;
  ptr = &a ;
  cout << " The value of a is" << *((int*) ptr) << endl;
}.
```

O/P:

```
The value of a is 100.
```

Arithmetic operations on pointer variables

- Arithmetic operators used with pointers are.
  - Binary operators : + (addition) and - (subtraction)
  - Unary operators : ++ (increment) and -- (decrement)

(Eg)

```
void main( )
{
  int  x, *x1 ;
  char y , *y1 ;
  float z ,  *z1 ;
  x1 = &x
  y1 = &y
  z1 = &z.
  cout << x1 << " " << x1++ ;  // prints  7500 and 7502
  cout << y1 << " " << y1++ ;  //   "    6750 and 6751
  cout << z1 << " " << z1++ ;  //   "    6800 and 6804
   cout << x1+10 ; //prints  7512
}
```

- Thus if a pointer to an integer is incremented using ++, then the address contained in the pointer is incremented by two.

Note: Arithmetic operations cannot be performed on void pointers without type casting.

Pointers and arrays:

* Array values can be accessed efficiently using pointers.

* The address of first element of the array (base address) is assigned to the pointer. Then the pointer can be moved to other array elements using pointer arithmetic operations.

(Eg) int x [4] = {10, 20, 30, 40}

int *p;

p = & x [0] (or) p = x ; => assigns first element address to pointer.

	[0]	[1]	[2]	[3]
x	10	20	30	40
	6400	6404	6408	6412

- Here & x [0] is 6400. Hence P = 6400.

- Pointer can be moved to next element as

```
p++ ;
cout << p ; // prints 6404.
```

Example:

```
#include <iostream.h>
void main ()
{
  int *ptr , var [10] ;
  ptr = var; //(or) ptr = & var [0] ;
```

```
for(int i=0; i<4; i++)
  {
    cin >> *ptr;
    cout << "The entered array element is" << *ptr << endl;

    ptr++;
  }
```

(or)

```
for(int i =0; i<4; i++
  {
   cin >> *(var+i);
   cout <<" The entered array element is " << *(var+i);
  }
```

Memory management Operators:

* Dynamic memory allocation - allocating memory during runtime on demand.

* Two operators for runtime (or) dynamic memory management.
  - new → for dynamic memory allocation
  - delete → " " " deallocation.

(1) new operator:
  - used to create objects of any type.

Syntax: $\boxed{\text{pointer-variable = new datatype;}}$

(Eg) int *P;

P = new int; ⇒ 'new' operator allocates sufficient memory to hold data of type int and returns the address of object.

equivalent to ⇓

int *P = new int; ⇒ declaration of pointers and assignments can be combined.

*p = 25 ; => assigns value 25 .

* Memory can also be initialized using new operator :

int *p = new int(25) ;

Syntax : pointer-variable = new data-type (value) ;

* new can also be used to create memory space for arrays, structures and classes .

Syntax for 1D array :

pointer-variable = new data-type [size] ;

(Eg) int *p = new int [10] ; => creates memory for an array of 10 integers.

p[0] -> first element of array

- for creating multi-dimensional arrays with new ,

int *p = new int [3][2][4] ; // legal

int *p = new int [m] [5] [4] ; // legal => 1st dimension can be a variable .

" = " int [3] [5] [ ] ; // illegal

int *p = new int [ ] [5] [2] ; // illegal .

(ii) delete Operator :

- used to destroy created data object to release memory space for reuse .

Syntax : delete pointer-variable ;

(Eg) delete p ; => p is the pointer that points to the data object created with new

- to free a dynamically allocated array,

<u>Syntax</u>:

```
delete [size] pointer-variable;
```

(Eg) delete [ ] p; ⇒ deletes entire array pointed to by p

<u>Note</u>:

* If sufficient memory is not available for allocation, new returns a null pointer.

(Eg)

```
int *p= new int;
if (!p)
  {
   cout << "Allocation failed\n";
  }
```

} checks for null pointer.

(Eg)

```
#include <iostream.h>
void main()
{
 int i,n, *p;
 cout << "Enter no. of elements\n";
 cin >> n;
 p = new int[n];
 if (!p)
    cout << "Allocation failed\n";
 else
  {
   for (i=0 ; i<n; i++)
    {
     cout << "Enter number\n";
     cin >> p[i];
    }
   cout << "The entered numbers :\n";
   for (i=0; i<n ; i++)
      cout << p[i] << " ";
   delete [ ] p;
  }
}
```

<u>Note</u>: The data object created by 'new' will exist until it is explicitly destroyed by delete