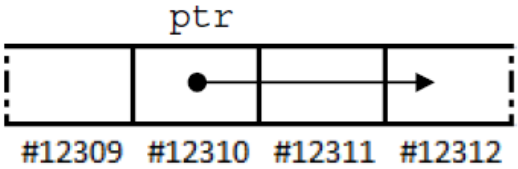
# 1- Pointers

## Variables and memory:

When you refer to the variable by name in your code, the computer must take two steps:

1. Look up the address that the variable name corresponds to
2. Go to that location in memory and retrieve or set the value it contains

• &x evaluates to the address of x in memory.
• *( &x ) takes the address of x and *dereferences* it – it retrieves the value at that location in memory.




The dot-arrow notation indicates that the variable ptr, stored in location 12310, points to the value in memory location 12312.

## Using Pointers:

- More flexible pass-by-reference
- Manipulate complex data structures efficiently
o This is especially useful for manipulating data that, unlike arrays, isn't stored in memory all at once, and so is scattered in various places in memory

To declare a pointer variable named x that points to an integer variable named y:
int *x = &y;
&y returns the address of the integer variable y, and
int *x declares a pointer to an integer value which we set to the address of y.

**The general scheme for declaring pointers is:**
data_type ***pointer_name; // or  data_type *pointer_name = initial_value;

Just like any other data type, we can pass pointers as arguments to functions. The data type of a pointer to an integer is int *, so the same way we'd say void func(int x) {…}, we can say void func(int *x){…}.

Once a pointer is declared, we can *dereference* it with the * operator to access its value:
cout << *x; // Prints the value pointed to by x, which in the above example would be y's value Without the * operator, the identifier x refers to the pointer itself, not the value it points to:
cout << x; //Outputs a hexadecimal representation of the memory address of y

**Pointers and arrays:**
The name of an array is in fact just a pointer to the first element in the array. Saying array[3] tells the compiler to start with the memory address of the first element in array, jump three elements down in memory, and dereference. For instance, an alternate way to express a[3] (where a is some pointer) is *(a+3).

## Passing by reference with pointers:

Example:

```
void squareByRef( int *numPtr )
{
 *numPtr = *numPtr * *numPtr;
}
```

This function, when called on a number, uses pointer syntax to square the value of the variable whose address is passed in.

### char * and char[]:

Arrays of chars and pointers to chars are interchangeable. A string is really just an array of characters. (If it's a hard-coded string that actually appears in your code, it is loaded into program memory and assigned a memory address at program startup.) When you set a char * to a string, you are really setting it to point to the first character in the array that holds the string.

# User Defined Datatypes

## I. **STRUCTURES**

```
struct Employee
 {
   int Empno;
   char *Ename;
   char *Add;
 };
int main()
{
  Employee Emp1;
  cin>>Emp1.Empno;
  Employee Emp2={10,"John", "77 Mass Ave"};
  cout<<Emp2.Empno<<Emp2.Name<<Emp2.Add;//10John77 Mass Ave
  return 0;
}
```

In the above example, “Employee” is s structure which can store 3 values, Empno, Ename and Add of type integer, string and string respectively.
Each structure type variable (Emp1 and Emp2) in our case, has sub-variables Emp1.Empno, Emp1.Ename, Emp1.Add etc, which can be used like any other variables. e.g. we can doEmp1.Empno++; to increment the value of Empno by 1 just as we do in regular integers.

Note: we cannot do operations on an entire structure variable, except in a few cases (for instance initialization can be done as we did for Emp2), but e.g.: cin>>Emp1; or cout<<Emp1; are not valid.

## II. **Classes**

We can put the above defined structure into a class.

```
class Employee
 {  int Empno;
    char *Ename;
    char *Add; };
int main()
{
   Employee Emp1;
   cin>>Emp1.Empno;  //Not allowed
   Employee Emp2={10,"John", "77 Mass Ave"};  //Not Allowed
   cout<<Emp2.Empno<<Emp2.Name<<Emp2.Add;   //Not Allowed
   return 0;
 }
```

Here Empno, Ename and Add (data inside the class) are known as Data Members of the class.
Emp1, and Emp2 are termed as Objects.
However, when we try to work with the data inside the objects, we cannot do so. Even basic input output operations do not work. This is because classes have Visibility Modes for the data inside them.
The three main visibility modes of members of a class are:-
Public: Accessible by objects of a class.
Private: Not accessible by objects of a class
Protected: Not accessible by objects of a class.

Note: Members of a class are Private by default

So the above code will work if we add the keyword *public* before the members followed by a colon (: )

```
class Employee
{ public:  int Empno;
```

```
        char *Ename;
        char *Add;      };
int  main()
{
    Employee Emp1;
    cin>>Emp1.Empno;
    Employee Emp2={10,"John", "77 Mass Ave"};
    cout<<Emp2.Empno<<Emp2.Name<<Emp2.Add;
    return 0;
}
```

However, we avoid declaring data members in private in order to keep the data "hidden". Thus we should not move the data members to public mode. But, in that mode they are not accessible by objects. How do we then manipulate the data?
Classes have the ability to have functions inside them to manipulate the data. These are known as Member Functions. These can be defined in public mode so that objects can directly access them.
Note: When we say 'members' of a class, we are referring to both data members and member functions.

```
class Employee
{ int Empno;
  char *Ename;
  char *Add;
 public:
  void Input()
    {   cin>>Empno>>Ename>>Add; }
  void Output()
    {   cout<<Empno<<Ename<<Add; }   };
int main()
 {
   Employee Emp1;
   Emp1.Input();    Emp1.Output();
   return 0;
 }
```

Member functions can be defined inside as well as outside the class, but it is usually preferable to define them outside the class, especially if they have control structures such as conditional constructs and loops. This is because some compilers tend to give errors when functions containing these are defined within the classes.
To define functions outside the classes, we have the function prototypes within the classes, and the definition outside.

```
class Employee
{ int Empno;
  char *Ename;
  char *Add;
 public:
  void Input();
  void Output();  };
int main()
 {
   Employee Emp1;
   Emp1.Input();
   Emp1.Output();
   return 0;
 }
void Employee::Input()
  {
   cin>>Empno>>Ename>>Add;
  }
void Employee::Output()
  {
    cout<<Empno<<Ename<<Add;
  }
```

:: is called ‘scope resolution operator’, and is used to indicate that the functions belong to a particular class. For instance, Employee::Input() means function Input() belongs to class Employee. Had we only written void Input() and defined it, it would have meant that this function is not related to the class.
Member functions in a class are identical to regular functions-they are defined normally, have a return type, name and arguments. They can also be overloaded like regular functions. Also regardless of the order in which they are defined, every member function in the class can call the other.

Consider the following program which utilizes the same class Employee with a few extensions:

```
class Employee
 { int Empno;
  char *Ename;
  char *Add;
  float Salary, Comm, TotSal;
 public:
  void Input();
  void Output();
  void ChangeEmpno(int);//function with argument of type int
  void ChangeEmpno2(int =2);//function with default argument
  float Tot_sal();//function with return type float
 };
int main()
 {
  Employee Emp1;
  Emp1.Input();          Emp1.Output();
  cout<<Emp1.Tot_Sal();
  Emp1.ChangeEmpno();
  Emp1.ChangeEmpno(458);
  Emp1.ChangeEmpno2(100);
  return 0;
  }
void Employee::Input()
  {   cin>>Empno>>Ename>>Add>>Salary>>Comm;
      TotSal=Tot_Sal();    //Input() is calling Tot_Sal()
  }
void Employee::Output()
  {   cout<<Empno<<Ename<<Add<<Salary<<Comm<<TotSal;
  }
void Employee::ChangeEmpno()
  {
     int NewEmpno;
     cin>>NewEmpno;
     Empno=NewEmpno;
  }
```

```
void Employee::ChangeEmpno(int NewEmpno)
  {    Empno=NewEmpno;  }
void Employee::ChangeEmpno2(int IncEmpno)
   {    Empno+=IncEmpno;  }
float Employee::Tot_sal()
   {   return Salary+Comm;   }
```

## Passing Class Objects as Arguments to Functions

Similar to other variables, class objects can also be passed as arguments to functions. These can be passed by value or by reference.

Passing by Value:

```
int SampleFunction1 (Employee E)
  {   E.Input();
      E.Output();   }
```

Passing by Reference:

```
int SampleFunction2(Employee &E)
  {   E.Input();
      E.Output();   }
```

## Having a class as return-type of a function:

This indicates that we will be returning an object of the class Employee to the calling function when this function is called.

```
Employee SampleFunction3()
{
    Employee E1;
    E1.Input();
    return E1;
}
```