Arrays

An array is a very popular, linear, homogenous, and useful data structure that is used to store similar types of data elements in contiguous memory locations under one variable name.

One-Dimensional Array Declaration and Initialization:

The declaration of a one-dimensional array is as follows:

Declaration:

```
int a[5];
```

It tells the compiler that `'a'` is an integer type of array, and it should store five integers. In this example, 5 is the subscript enclosed within square brackets. The compiler reserves four bytes of memory for each integer array element; that is, 20 bytes are reserved for storing five integers in the memory.

In the same way, arrays of different data types are declared as follows:

```
char ch[10];
float real[10];
long num[5];
```

The array initialization is done as follows:

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

Here, five elements are stored in an array *'a'*. The array elements are stored sequentially in separate locations. Then, the question arises of how to call each element individually from this bunch of integer elements. The reading of array elements begins from `zero`.

Array elements are accessed with the name of the array, and the number within the square brackets specifies the element number. In other words, array elements are called with array names followed by element numbers.

a[0] refers to 1st element i.e. 1
a[1] refers to 2nd element i.e. 2
a[2] refers to 3rd element i.e. 3
a[3] refers to 4th element i.e. 4
a[4] refers to 5th element i.e. 5

Characteristics of Arrays:

1. The Declaration `int a[5]` is nothing but a creation of five variables of integer types. Instead of declaring five variables for five values, the programmer can define them in an array.
2. All the elements of an array share the same name, and they are distinguished from one another with the help of an element number.
3. The element number in an array plays a major role for calling each element.
4. Any particular element of an array can be modified separately without disturbing the other elements.

   For example, `int a[5] = {1,2,3,4,8};`

   If the programmer needs to replace 8 with 10, he/she is not required to change all the other elements except 8. To carry out this task, the statement `a[4] = 10` can be used. Here, the other four elements are left unchanged.
5. The array elements are stored in contiguous memory locations. The amount of storage required for holding the elements of the array depends on its type and size. The total size in bytes for a single-dimensional array is computed as shown below.

   Total bytes = size of (data type) × size of array

   Example:

   /*Program to read 5 integers through keyboard and display them.*/

```
int main()
{
        int num[5],i;
        for(i = 0;i<5;i++)
        {
            cout<<"\n Enter any no:";
            cin>>num[i];
        }
        cout<<" Numbers are:";
        for(i = 0;i<5;i++)
        {
            cout<<num[i];
        }
        return 0;
}
```

Accessing Array Elements through Pointers: We can quickly and easily access array elements using pointers, as these elements are stored in contiguous memory locations. A pointer variable contains an address, and it is easy to manipulate data with the help of addresses. When a pointer is incremented, the address gets incremented, and we can access the contents of memory locations. This particular method requires less memory; hence, execution is fast.

/***Program to display elements of an array using pointer. Display addresses of elements.***/

```
int main()
{
        int *p,num[5] = {1,2,3,4,5},j;
        p=&num[0];
        cout<<"Number Address"<<endl;
        for (j=0;j<5;j++)
                cout<<" "<<*(p+j) <<" "<<unsigned(p+j)<<endl;
        return 0;
}
```

Output:

Number	Address
1	65452
2	65454
3	65456
4	65458
5	65460

Arrays of Pointers: 'C++' language also supports arrays of pointers. It is nothing but a collection of addresses. Here, we store the addresses of variables for which we have to declare arrays as pointers.

/***Write a program to store addresses of different elements of an array using array of pointers.***/

```
int main()
{
        int *arrp[3];
        int arr[3]={5,10,15}, k;
        for(k=0;k<3;k++)
                arrp[k]=arr+k;
        cout<<"\n\t Address Element"<<endl;
        for (k=0;k<3;k++)
        {
            cout<<"\t" <<unsigned(arrp[k]);
            cout<<"\t"<<*(arrp[k])<<endl;
        }
        return 0;
}
```

Output:

Address	Element
65418	5
65420	10
65422	15

Explanation*:* In the above program, *arrp[3] is declared as an array of pointers. Using first for loop, the addresses of various elements of array 'arr[]' are assigned to '*arrp[]'. The second for loop picks up addresses from '*arrp[]' and displays the values present at those locations. Here, each element of '*arrp[]' points to the respective element of array 'arr[]'.

<u>**Passing Array Elements to A Function:**</u> We can pass elements to a function by using call-by-value or call-by-address methods. In the call-by-value method, elements (values) of an array are passed to the function; whereas in the call-by-address method, addresses of elements are passed to the function.

The following program demonstrates the call-by-value method:

/***Program to pass elements of an array to a function by using call by value.** */

```
void display(int);

int main()
{
        int num[5]={1,2,3,4,5},i;
        cout<<"\nElements in the reverse order are as follows:";
        for(i=4;i>= 0;i--)
        {
                display(num[i]);
        }
        return 0;
}
void display(int x)
{
        cout<<" "<<x;
}
```

/* **Program to pass elements of an array to a function by using call by address.*/**

```
void display(int *);

 int main()
 {
        int num[5]={1,2,3,4,5},i;

        cout<<"\nElements of the array are as follows:";
        for(i=0;i<5;i++)
        {
            display(&num[i]);
        }
        return 0;
 }
 void display(int *x)
 {
        cout<<" "<<*x;
 }
```

In the above program, addresses of individual array elements are passed to the function display(int *). The x contains the address of an array element, and *x is the value stored at that address.

/* **Program to pass elements of an array to a function by using call by reference.*/**

```
void display(int &);
int main()
{
    int num[5]={1,2,3,4,5},i;

    cout<<"\nElements of the array are as follows:";
    for(i=0;i<5;i++)
    {
            display(num[i]);
    }
    return 0;
}
void display(int &x)
{
    cout<<x<<"\t";
}
```

Passing Complete Array Elements to A Function:

It is also possible to pass the entire elements of an array to a function instead of passing the individual elements of an array. The following program explains the concept:

/* Program to pass entire elements of an array to a function.*/

```
void show(int *, int);
int main()
{
        int num[5]={1,2,3,4,5};
        cout<<"\nElements of the array are as follows:";
        show(&num[0],5);
        return 0;
}
void show(int *x,int y)
{
        for(int i=0;i<y;i++)
        {
                cout<<" "<<*x;
                x++;
        }
}
```

Output: Elements of the array are as follows: 1 2 3 4 5

In the above program, the address of the zeroth element, that is, `&num[0]`, is passed to the function show(). The for loop is used to access the array elements using pointers. The show() function is invoked with two arguments. The first argument is the address of the zeroth element, and the second one is the number that represents the total number of elements in the array.

Initialization of Arrays Using Functions: The programmers always initialize the arrays using statements such as *int d[] = {1,2,3,4,5}*; instead of this, the function can also be directly called to initialize the array. The following program illustrates this point.

/* **Write a program to initialize an array using functions.***/

```
#include <iostream>
using namespace std;
int c();
int main()
{
    int d[3] = {c(),c(),c()};
    cout<<"Array d[] elements are :"<<endl;
    for (int k=0;k<3;k++)
        cout<<d[k]<<endl;
    return 0;
}
int c()
{
    int n;
    cout<<"Enter Number to store the element into an arrray :";
    cin>>n;
    return n;
}
```

Output:

```
Enter Number to store the element into an arrray :1
Enter Number to store the element into an arrray :2
Enter Number to store the element into an arrray :3
Array d[] elements are :
1
2
3
```

Explanation: A function can be called in the declaration of an array. In the above program, `d[]` is an integer array, and `c()` is a user-defined function. When called, the function `c()` reads values through the keyboard. The function `c()` is called from an array; that is, the values returned by the function are assigned to the array. The above program will not work in C.

Two-Dimensional Arrays: Two-dimensional arrays can be considered a rectangular display of elements with rows and columns, and this is also known as a `matrix`.

Consider the following example `int` x[3][3]. The two-dimensional array can be declared as shown in figure.

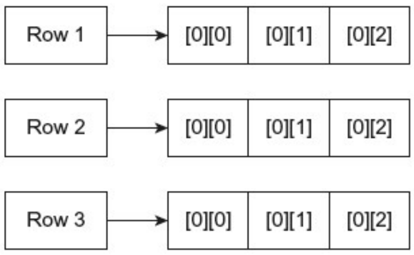


Fig. Two-dimensional array

The arrangement of array elements shown in the figure only for the sake of understanding. Actually, the elements are stored in continuous memory locations. The two-dimensional array is a collection of two one-dimensional arrays. The meaning of the first argument is in x[3][3] and means the number of rows; that is, the number of one-dimensional arrays, and the second argument indicates the number of elements. The x[0][0] means the first element of the first row and column. In one row, the row number remains the same but the column number changes. The number of rows and columns is called the `range` of the array. A two-dimensional array clearly shows the difference between logical assumptions and the physical representation of data. The computer memory is linear and any type of array may one, two- or multi-dimensional array it is stored in continuous memory location as shown in figure.

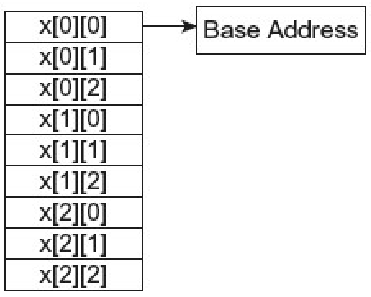


Fig. Storage of two-dimensional array

/*Program to demonstrate two dimensional array.*/

```
int main()
{
        int i,j;
        int a[3][3] = {1,2,3,4,5,6,7,8,9};
        cout<<"\n Array elements and address ";
        cout<<"\n \t Col-0   Col-1   Col-2";
        cout<<"\n \t ======  ======  ====== ";
        for (i=0;i<3;i++)
        {
                for (j=0;j<3;j++)
                        cout<<"\t "<< a[i][j];
        }
        return 0;
}
```

<u>Pointers and Two-Dimensional Arrays:</u> A matrix can represent the two-dimensional elements of an array. In order to display the elements of the two-dimensional array using pointers, it is essential to have ***'&'*** operator as a prefix with the array name followed by element numbers.

/***Write a program to display array elements and their addresses using pointers.*/**

```
int  main()
{
int i,j,*p;
int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}};

cout<<"\n\tElements of an array with their addresses";
p=&a[0][0];
cout<<"\n";
for (i=0;i<3;i++)
{
        for(j=0;j<3;j++)
        {
                cout<<*(p)<<"  "<<unsigned(p)<<"    ";
                p++;
        }
        cout<<endl;
}
return 0;

}
```

Output:

```
Elements of an array with their addresses

        1  65506  2  65508  3  65510
        4  65512  5  65514  6  65516
        7  65518  8  65520  9  65522
```

Explanation: In the above program, the two-dimensional array is declared and initialized. The base address of the array is assigned to integer pointer `'p'`. While assigning the base address of the two-dimensional array, the `'&'` operator is pre-fixed with the array name followed by the element numbers; otherwise, the compiler shows an error. The statement `p = &a[0][0]` is used in this context. The pointer `'p'` is printed and incremented in the `for` loop till it prints the entire array of elements

Three or Multi-Dimensional Arrays:

The ***'C++'*** program helps in creating an array of multi dimensions. The compiler determines the restrictions on it.

The syntax of the declaration of multi-dimensional arrays is as follows:

Data_Type Arrayname [S1][S2][S3]...[Si];

where Si is the size of the i^{th} dimensions.

Three-dimensional arrays can be initialized as follows:

```
int mat[3][3][3] = { 1,2,3, 4,5,6, 7,8,9, 1,4,7, 2,5,8,
                     3,6,9, 1,4,4, 2,4,7, 8,8,5};
```

A three-dimensional array can be considered an array of arrays of arrays. The outer array contains three elements. The inner array is two dimensional with regard to size [3][3].

/* Write a program to explain the working of three-dimensional array. */

```
int main()
{
      int array_3d[3][3][3];
      int a,b,c;

      for (a=0;a<3;a++)
        for (b=0;b<3;b++)
                for (c=0;c<3;c++)
                        array_3d[a][b][c]=a+b+c;

      for (a=0;a<3;a++)
      {
            cout<<"\n";
                  for (b=0;b<3;b++)
                  {
                           for (c=0;c<3;c++)
                           cout<<" "<<array_3d[a][b][c];
                           cout<<"\n";
                  }
      }
      return 0;
 }
```

Pointers

A pointer is a memory variable that stores a memory address. Pointers can have any name, and it is declared in the same fashion as other variables, but it is always denoted by '*' operator.

Features of Pointers

1. Pointers save memory space.
2. Execution time with pointers is faster, because data are manipulated with the address, that is, direct access to memory location.
3. Memory is accessed efficiently with the pointers. The pointer assigns as well as releases the memory space. Memory is dynamically allocated.
4. Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.
5. We can access the elements of any type of array, irrespective of its subscript range.
6. Pointers are used for file handling.
7. Pointers are used to allocate memory in a dynamic manner.
8. In C++, a pointer declared to a base class could access the object of a derived class. However, a pointer to a derived class cannot access the object of a base class.

Pointer Declaration

Pointer variables can be declared as follows:

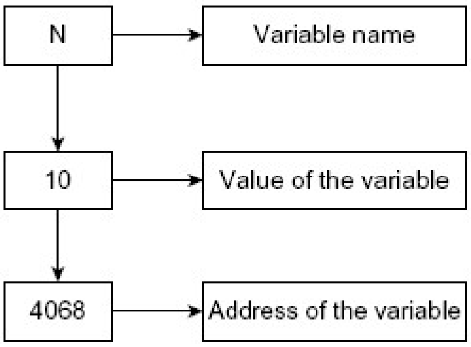
Example:

```
int *x;
float *f;
char *y;
```

In the first statement, 'x' is an integer pointer, and it informs the compiler that it holds the address of any integer variable. In the same way, 'f' is a float pointer that stores the address of any float variable, and 'y' is a character pointer which stores the address of any character variable.

The indirection operator (*) is also called the *dereference operator*. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

/* Write a program to display the address of the variable.*/



```
#include <iostream>
using namespace std;

int main()
{
        int n;

        cout<<"Enter a Number = ";
        cin>>n;
        cout<<"Value of n = "<<n;
        cout<<"Address of n= " <<(unsigned)&n;
        return 0;
}
```

Output:

```
        Enter a Number = 10
        Value of n = 10
        Address of n=4068
```

/* Write a program to declare a pointer. Display the value and address of the variable-using pointer. */

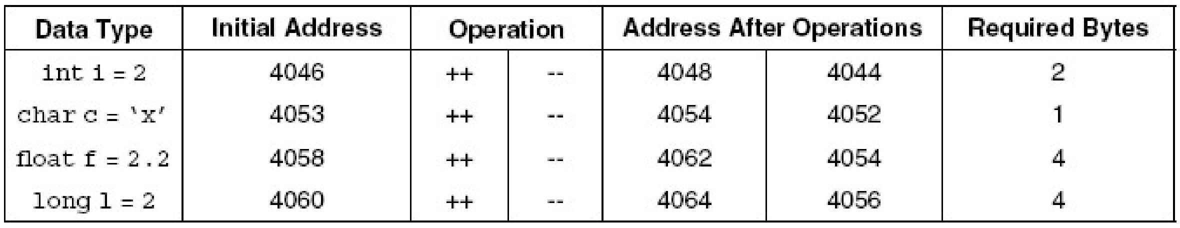
```
#include <iostream>
using namespace std;

int main()
{
   int *p;
   int x=10;

   p=&x;

   cout<<"\n x="<<x <<"  &x="<<&x;
   cout<<"\n x="<<*p<<"  &x="<<p;

   return 0;
}
```

Output:

```
x=10  &x=0x22ff18
x=10  &x=0x22ff18
```

<u>Arithmetic Operations with Pointers</u>

We can perform different arithmetic operations by using pointers. Increment, decrement, prefix, and postfix operations can be performed with pointers. The effects of these operations are shown in Table.



Data Type	Initial Address	Operation		Address After Operations		Required Bytes
int i = 2	4046	++	--	4048	4044	2
char c = 'x'	4053	++	--	4054	4052	1
float f = 2.2	4058	++	--	4062	4054	4
long l = 2	4060	++	--	4064	4056	4

From the above table, while referring to the first entry, we can observe that on increment of the pointer variable for integers, the address is incremented by two; that is, 4046 is the original address and on increment, its value will be 4048, because integers require two bytes. Similarly, when the pointer variable for integer is decreased, its address 4048 becomes 4046.

/* Program on pointer incrementation and decrementation.*/

```
int main()
{
        int x=10;
        int *p;
        p=&x;
        cout<<"\n Address of p:"<<unsigned(p);
        p=p+4;
        cout<<"\n Address of p:"<<unsigned(p);
        p=p-2;
        cout<<"\n Address of p:"<<unsigned(p);
        return 0;
}
```

Output:

```
Address of p:65524
Address of p:65532
Address of p:65528
```

/* Program on changing the values of variables using pointer.*/

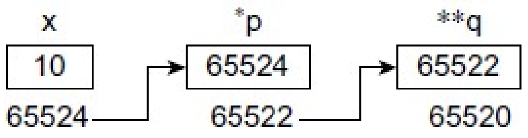
```
#include <iostream>
using namespace std;

int main() {
        int x=10;
        int *p;
        p=&x;
        cout<<"\n Value of x:"<<*p;
        *p=*p+10;
        cout<<"\n Value of x:"<<*p;
        *p=*p-2;
        cout<<"\n Value of x:"<<*p;
        return 0;
}
```

Output:

```
        Value of x: 10
        Value of x: 20
        Value of x: 18
```

<u>Pointer to Pointer</u>

Pointer to pointer is a pointer that stores the address of another pointer. There can be a chain of pointers depending on applications/requirements. In the Figure,, x is a simple variable, p is a pointer to the variable x, and q is a pointer to p. The values of variables 'x,* p, and **q' are shown in the boxes, and their addresses are shown outside the boxes.



/*Program to demonstrate the concept of pointer to pointer.*/

```
#include <iostream>
using namespace std;

int main() {
        int x=10;
        int *p;
        int **q;
        int ***z;
        p=&x;
        q=&p;
        z=&q;
        cout<<"\n Value of x = "<<x;
        cout<<"\n Value of x = "<<*p;
        cout<<"\n Value of x = "<<**q;
        cout<<"\n Value of x = "<<***z;
        cout<<"\n Adderss of x "<<unsigned(&x);
        cout<<"\n Adderss of p "<<unsigned(&p);
        cout<<"\n Adderss of q "<<unsigned(&q);

        return 0;

}
```

OUTPUT

```
Value of x = 10
 Value of x = 10
 Value of x = 10
 Value of x = 10

 Adderss of x 2293528
 Adderss of p 2293524
 Adderss of q 2293520
```

<u>void Pointers</u>

When a variable is declared as being a pointer to type **void** it is known as a *generic pointer*. Since you cannot have a variable of type **void**, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term *Generic pointer*. This is very useful when you want a pointer to point to data of different types at different times.

/* Program to illustrate the use of void pointer */

```
int main()
{
        int i;
        char c;
        void *the_data;
```

```
            i = 6;
            c = 'a';

            the_data = &i;
            cout<<"The data points to the integer value :"<< *(int*) the_data;

            the_data = &c;
            cout<<"\nThe data now points to the character:"<< *(char*) the_data;

            return 0;
    }
```

Output:

```
The data points to the integer value : 6
The data now points to the character: a
```

wild Pointers or Dangling Pointers

Uninitialized pointers are known as wild pointers because they point to some arbitrary memory location and may cause a program to crash or behave badly.

```
#include <iostream>
using namespace std;

int main()
{
            int *p;  /* wild pointer */
            *p = 12; /* Some unknown memory location is being corrupted. This
                          should never be done. */
            cout<<unsigned(p);

}
```

Please note that if a pointer p points to a known variable then it's not a wild pointer. In the below program, p is a wild pointer till this points to a.

```
int main()
{
      int  *p; /* wild pointer */
      int a = 10;
      p = &a;  /* p is not a wild pointer now*/
      *p = 12; /* This is fine. Value of a is changed */
}
```

The pointer becomes a `wild` pointer due to the following reasons:

1. Pointer declared but not initialized
2. Pointer alteration
3. Accessing destroyed data

this Pointer

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (i.e., the object on which the function is invoked). This pointer is known as *this pointer*. It is internally created at the time of function call.

/* Program to print the address of the object using this pointer*/

```
#include <iostream>
using namespace std;
class integer
{
        int x;
public:
        void show_addr();
};
void integer::show_addr()
{
        cout<<"My Object's Address="<<this<<"\n";
}
int main() {
        integer a,b,c;
        cout<<"A addr:   "<<&a<<endl;
        a.show_addr();
        return 0;

}
```

Output:

```
        A addr:0x22ff1c

        My Object's Address=0x22ff1c
```

/* Program to add two object contents using this pointer*/

```
#include <iostream>
using namespace std;
class Add
{
        int val;
public:
        void setdata(int val)
        {
                this->val=val;
        }
        void display()
        {
                cout<<val<<endl;
        }
        Add sum(Add v2)
        {
                val=val+v2.val;
                return *this;
        }
};
int main() {
        Add v1,v2;
        v1.setdata(3);
        v2.setdata(4);

        Add s;
        s=v1.sum(v2);
        cout<<"Sum is=";
        s.display();
        return 0;

}
```

Output:

```
        Sum is=7
```

Pointer to Derived Classes and Base Class

It is possible to declare a pointer that points to the base class as well as the derived class. One pointer can point to different classes. For example, X is a base class and Y is a derived class. The pointer pointing to X can also point to Y.

/* Program to declare a pointer to the base class and access the member variable of base class.*/

```
class B
{
      public :
      int b;
      void display()
      {
      cout<<"b = " <<b <<endl;
      }
};
class D : public B
{
         public :
                 int d;
                 void display()
                 {
                         cout<<"b= " <<b <<"\n" <<" d="<<d <<endl;
                 }
 };
int  main()
{
          B *cp;
          B base;
          cp=&base;
          cp->b=100;
          // cp->d=200; Not Accessible
          cout<<"\n cp points to the base object \n";
          cp->display();
          D d;
          cout<<"\n cp points to the derived class \n";
          cp=&d;
          cp->b=150;
           //cp->d=300; Not accessible
          cp->display();
          return 0;

}
```

Output:

```
 cp points to the base object
b = 100
cp points to the derived class
b = 150
```

/* Program to declare a pointer to the derived class and access the member variable of base and derived class.*/

```
#include <iostream>
using namespace std;

class B
{
    public :
    int b;
    void display()
    {
          cout<<"b = "<<b <<endl;
    }
};
class D : public B
{
        public:
                int d;
                void display()
                {
                                cout<<"b= "<<b <<endl;
                                cout<<"d= "<<d <<endl;
                }
};
int main()
{
        D *cp;
        D d;
        cp=&d;
        cp->b=100;
        cp->d=350;
        cout<<"\n cp points to the derived object \n";
        cp->display();
        return 0;
}
```

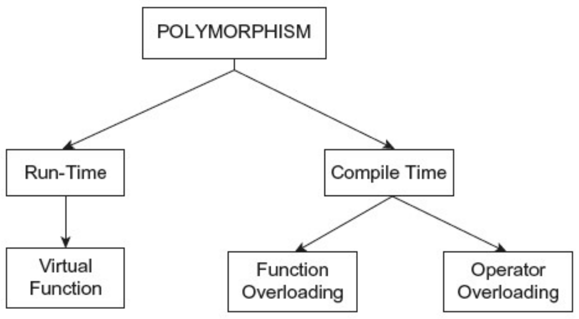
Output:

```
cp points to the derived object
b= 100
d= 350
```

Polymorphism

The word *poly* means many, and *morphism* means several forms. Both the words are derived from Greek language. Thus, by combining these two words, a new whole word called polymorphism is created, which means various forms.

In C++, the function can be bound at either compile time or run time. Deciding a function call at compile time is called compile time or early or static binding. Deciding a function call at run time is called run time or late or dynamic binding. Dynamic binding permits to suspend the decision of choosing a suitable member function until run time. Two types of polymorphism are shown in Figure.



A polymorphism is a technique in which various forms of a single function can be defined and shared by various objects to perform an operation.

Binding in C++

Binding refers to the process that is to be used for converting functions and variables into machine language addresses. The C++ supports two types of binding: static or early binding and dynamic or late binding.

Static (Early) Binding: By default, matching of function call with the correct function definition happens at compile time. This is called static binding or early binding or compile-time binding. Static binding is achieved using function overloading and operator overloading. Even though there are two or more functions with same name, compiler uniquely identifies each function depending on the parameters passed to those functions. Consider the following example.

```
#include <iostream>
using namespace std;

class Base
{
        public:
        void display()
        {
                        cout<<"Base"<<endl;
        }
};

class Derived:public Base
{
        public:
        void display()
        {
                        cout<<"Derived"<<endl;
        }
};
int main()
{
        Derived d;
        d.Base::display(); // Invokes base class function
        d.display(); // Invokes derived class function
        return 0;
}
```

Output:

```
Base
Derived
```

Explanation: In the above program both the classes contain display() member function. Both the classes contain a similar function name. In function `main()` Hence, in order to invoke the `display()` function of the base class, the scope access operator is used. When base and derived classes have similar function names, in such a situation, it is very essential to provide information to the compiler at compile time about the member functions.

Dynamic (Late) Binding: C++ provides facility to specify that the compiler should match function calls with the correct definition at the run time; this is called dynamic binding or late binding or run-time binding. Dynamic binding is achieved using virtual functions, base class pointer points to derived class object and Inheritance.

Pointer to Base and Derived Class Objects:

In inheritance, the properties of existing classes are extended to the new classes. The new classes that can be created from the existing base class are called as derived classes. Pointers can be declared to the base or derived class. Pointers to objects of the base class are type compatible with pointers to objects of the derived class. i.e, base class pointer can point to objects of both the base and derived class. In other words, a pointer to the object of the base class can point to the object of the derived class; whereas a pointer to the object of the derived class cannot point to the object of the base class.

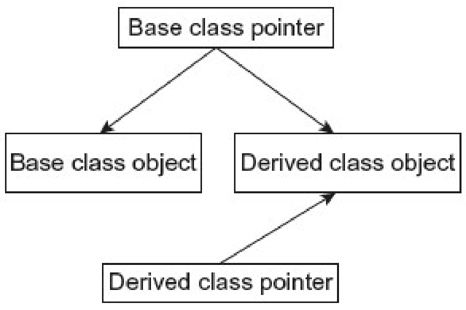


Fig. Type compatibility of base and derived class pointers

Virtual Functions:

A virtual function is a member function that is declared with in a base class and is redefined by derived class.

To create a virtual function, precede the functions declaration in the base class with the keyword '**virtual**'. It signals the compiler that we don't want static linkage for this function. So, when a class containing virtual function is inherited, the derived class redefines the virtual function to fit its own needs. i.e., the definition creates a specific method.

Rules for Virtual Functions:

1. The virtual function should not be static and must be a member of a class.
2. The virtual function may be declared as a friend for another class. An object pointer can access the virtual functions.
3. A constructor cannot be declared as virtual, but a destructor can be declared as virtual.
4. The virtual function should be defined in the public section of the class. It is also possible to define the virtual function outside the class. In such a case, the declaration is done inside the class, and the definition is outside the class. The virtual keyword is used in the declaration and not in the function definition
5. It is also possible to return a value from virtual functions similar to other functions.
6. The prototype of the virtual function in the base class and derived class should be exactly the same. In case of a mismatch, the compiler neglects the virtual function mechanism and treats these functions as overloaded functions.
7. Arithmetic operations cannot be used with base class pointers.
8. If a base class contains a virtual function and if the same function is not redefined in the derived classes, in such a case, the base class function is invoked.

Example:

```
class Base
{
        public:
        virtual void display()
        {
                        cout<<"Base"<<endl;
        }
};

class Derived:public Base
{
        public:
        void display()
        {
                        cout<<"Derived"<<endl;
        }
};
```

```
        int main()
        {
                Derived b;
                Base *a=&b;
                a->display();
                return 0;

        }

        Output:
                Derived
```

/* C++ program to use pointer for both base and derived class and call the member function. Use virtual keyword. */

```
    #include <iostream>
    using namespace std;
    class super
    {
            public:
            virtual void display()
            {
                    cout<<"\n In function display() - class super";
            }
            virtual void show()
            {
                    cout<<"\nIn function show() - class super";
            }
    }
    ;
    class sub: public super
    {
            public:
            void display()
            {
                    cout<<"\nIn function display() class sub";
            }
            void show()
            {
                    cout<<"\nIn function show() class sub";
            }
    };
    int main()
```

```
{
        super sup;
        sub s;
        super *sp;
        cout<<"\n Super Pointer points to class super\n";
        sp=&sup;
        sp->display();
        sp->show();

        cout<<"\n\n Super Pointer points to derived class sub\n";
        sp=&s;
        sp->display();
        sp->show();
        return 0;

}
```

Output:

```
Super Pointer points to class super
        In function display() - class super
        In function show() - class super

Super Pointer points to derived class sub
        In function display() class sub
        In function show() class sub
```

/*C++ program to create array of pointers. Invoke functions using array objects.*/

```
#include <iostream>
using namespace std;
class A
{
        public:
        virtual void show() { cout<<"A\n";   }
};
class B : public A
{
        public:
                void show() {cout<<"B\n";}
};
class C : public A
{
        public:
        void show()    {        cout<<"C\n";  }
};
class D : public A
{
        public:
        void show()    {        cout<<"D\n";  }
};
class E : public A
{
        public:
        void show()    {        cout<<"E";    }
};
int main()
{
        A a;
        B b;
        C c;
        D d;
        E e;
        A *pa[]={&a,&b,&c,&d,&e};
        for ( int j=0;j<5;j++)
                pa[j]->show();
        return 0;

}
```

Output:

```
        A
        B
        C
        D
        E
```

Pure Virtual Functions

Pure virtual Functions are virtual functions with no definition. Declaration of pure virtual function

virtual void display() =0; // pure function

In the above declaration of the function, the `display()` is a pure virtual function. The assignment operator is not used to assign zero to this function. It is used just to instruct the compiler that the function is a pure virtual function and that it will not have a definition.

A pure virtual function declared in the base class cannot be used for any operation. The class containing the pure virtual function cannot be used to declare objects. Such classes are known as **abstract classes** or **pure abstract classes.**

/* C++ program to declare pure virtual functions.*/

```
#include <iostream>
using namespace std;
class Base
{
        public:
        virtual void display()=0;
};

class Derived:public Base
{
        public:
        void display()
        {
                        cout<<"Derived"<<endl;
        }
};
int main()
{
        Derived b;
        Base *a=&b;
        a->display();
        return 0;
}
```

Output:

```
        Derived
```

Explanation: In the above program, the **display()** function of the base class is declared a pure function. The pointer object *a holds the address of the object of the derived class and invokes the function display() of the derived class. Here, the function display() of the base class does nothing. If we try to invoke the pure function using the statement b->Base::display(), the program is terminated with the error "abnormal program termination."

Abstract Classes

Abstract Class is a class which contains at least one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

While defining such an abstract class, the following points should be kept in mind:

1. Do not declare an object of abstract class type.
2. An abstract class can be used as a base class.
3. The derived class should not have pure virtual functions. Objects of the derived class can be declared.

Example:

```
class Base          //Abstract base class
{
        public:
         virtual void show() = 0;          //Pure Virtual Function
};

class Derived:public Base
{
         public:
         void show() {
                cout << "Implementation of Virtual Function in Derived class";
         }
};

int main()
{
 //Base obj;      //Compile Time Error
 Base *b;
 Derived d;
```

```
        b = &d;
        b->show();
        }
```

<u>Virtual functions in derived classes:</u>

In C++, once a member function is declared as a virtual function in a base class, it becomes virtual in every class derived from that base class. In other words, it is not necessary to use the keyword virtual in the derived class while declaring redefined versions of the virtual base class function.

```
#include <iostream>
using namespace std;

class Base          //Abstract base class
{
        public:
                virtual void show()
                {
                        cout << "Implementation of Virtual Function in Derived class";
                }
};

class Derived:public Base
{
};

int main()
{
        Base *b;
        Derived d;
        b = &d;
        b->show();
}
```

Output:

```
                Implementation of Virtual Function in Derived class
```

Run-Time Polymorphism Example:

```
#include <iostream>
using namespace std;

class CPolygon
{

        int width, height;
    public:
        void get (int first, int second)
        {
            width= first;
            height= second;
        }
        virtual int area()=0;

};
class CRectangle: public CPolygon
{
    public:
        int area()
        {
            return (width * height);
        }
};

class CTriangle: public CPolygon
{
    public:
        int area()
        {
            return ((width * height )/ 2);
        }
};
int main ()
{
    CRectangle rectangle;
    CTriangle triangle;
    CPolygon * ptr_polygon;

    ptr_polygon = &rectangle;
    ptr_polygon->get(2,2);
    cout << ptr_polygon->area () << endl;
```

```
        ptr_polygon = &triangle;
        ptr_polygon->get(2,2);
        cout << ptr_polygon->area () << endl;

        return 0;
}
```

Output:

```
        4
        2
```

Object Slicing: Virtual functions can be invoked using a pointer or reference. If we do so, object slicing takes place. The following program takes you to the real thing:

```
class Base
{
        public:
        virtual void display()=0;
};

class Derived:public Base
{
        public:
        void display()
        {
                        cout<<"Derived"<<endl;
        }
};
int main()
{
        Derived b;
        b. display();//object reference
        Base *a=&b;
        a->display(); //pointer
        return 0;
}
```

Output:

```
        Derived
        Derived
```