Unit-5
Arrays and Pointers

Array is a collection of variables of same type that stores the elements in the sequential manner.
It is a data structure.
Array elements consist of **contiguous memory locations** and are accessed randomly using the subscript or index variable.
It stores a fixed-size sequential collection of elements of the same type.
In array, the lowest address corresponds to first element and highest address corresponds to the last element.

Declaring Arrays

To declare an array, first specify the type of the elements and the number of elements required.

Syntax:

```
datatype arrayname [arraysize];
```

The arraysize must be integer constant greater than 0.

Example:

```
int empid[5];    //We created array of 5 employees
```

Array index starts from 0.

Element	Description
empid[0]	It represents variable for 1st employee id.
empid[1]	It represents variable for 2st employee id.
empid[2]	It represents variable for 3st employee id.
empid[3]	It represents variable for 4st employee id.
empid[4]	It represents variable for 5st employee id.

Initializing Arrays

Following are the three ways to initialize an Array:

1. Specify size and initialization will be done using the loop.

Example:

```
int empid[5];
```

```
for(i=0; i<5; i++)
cin>>empid[i];
```

2. Specify size and initialize array in a single statement.

Example:

```
empid[5]={101,102,103,104,105};
```

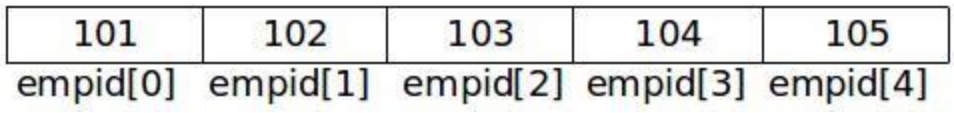
3. Without specifying size

Example:

```
empid[]={101,102,103,104,105};
```

The above example specifies the size of an array equal to 5.

Consider the following representation:



The above representation of an array is the pictorial form of an array, starts with 0.
Accessing Arrays

Example : Program demonstrating the declaring, initializing & accessing the arrays

```
#include<iostream.h>
#include <iomanip>      //used for setw() function
int main ()
{
    int a[ 5 ];     // a is an array of 5 integers
    for ( int i = 0; i < 5; i++ )         // initialize elements of array
    {
       a[i] = i + 101;          // set element at location i to i + 100
    }
    cout<<"Array Element"<<endl;
    for ( int j = 0; j < 5; j++ )          //for displaying array element
    {
       cout<<setw(9)<<j<<setw(13)<<a[j]<<endl;
    }
    return 0;
}
```

Note : setw() is used to format the output.

Output:
Array Element
0 101

1 102
2 103
3 104
4 105

Multi-Dimensional Arrays

Multi-Dimensional Arrays are used to store data which requires multiple references.

Syntax:

data-type name [size1] [size2] . . . [size n];

If we want to create **Two-dimensional array** then we can declare it as:int arr[3][3];

Where,
i. First dimension represents the number of rows.
ii. Second dimension represents the number of columns.

Consider the below representation of Two-dimensional array:

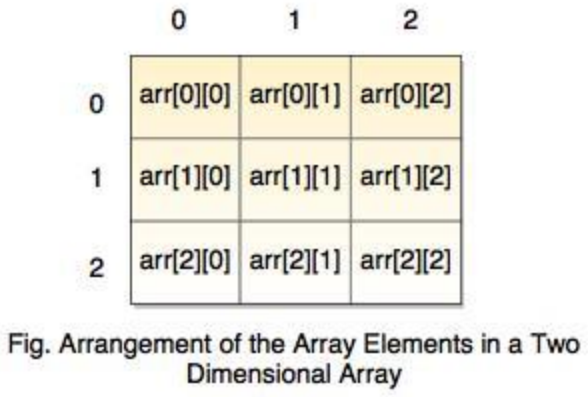


Fig. Arrangement of the Array Elements in a Two Dimensional Array

The size of the array of the above figure is 3 X 3, but the indices will be from 0 to 2 in both rows and columns because array starts from 0.

We can also have an array of more than two dimensions. Three-dimensional array has 3 dimensions, for example int a[2][2][2], but it is rarely used.

Pointer

Pointers are variables used to store the address of another variable.
It is an extremely powerful programming tool.
It helps to improve program's efficiency and allows to handle unlimited amount of data.

Syntax of Pointer declaration:

datatype * ptr_name;

Address of operator (&)

Address of operator is called as the **Referencing operator.**
The **ampersand (&)** is the referencing operator that gives the address of that variable.
It returns the address of the variable associated with the operator.
Address is displayed in hexadecimal form.
Address of variable is the memory location number alloted to the variable.

Example:

```
int *p , a;
a=10;
p=&a;
```

In the above example, **&a** returns the address of the variable **a. p** is a pointer which points to a variable **a.**

p=&a means "copy the address of the variable a in the pointer variable **p**".

Value of operator (*)

Value of operator is called as the **De-referencing operator.**
It returns the value stored in the variable pointed by the specified pointer.
The **asterisk (*)** is a dereference operator which means "value pointed to by".

In the above example, the ***p** will return the value of the variable pointed by the pointer **p.**

a=*p means that the value of the variable pointed by the pointer **p** is stored in the variable **a.**

Example : Demonstrating the referencing & de-referencing using pointers

```
#include <iostream.h>
int main()
{
    int *p, a;
```

```
    a = 10;
    cout<<"Address of a (&a): "<<&a<<endl<<endl;
    cout<<"Value of a (a): "<<a<<endl<<endl;
    p = &a;         // Pointer p holds the memory address of variable a
    cout<<"Address that pointer p holds (p): "<<p<<endl<<endl;
    cout<<"Content of the address pointer p holds (*p): "<<*p<<endl;
    a = 20;         //The content inside memory address &a is changed from 10 to 20
    cout <<"Address pointer p holds (p): "<<p<<endl<<endl;
    cout <<"Content of the address pointer p holds (*p): "<< *p << endl;
    *p = 5;
    cout<<"Address of a (&a): "<<&a<<endl<<endl;
    cout<<"Value of a (a): "<<a<<endl<<endl;
    return 0;
}
```

Output:

```
Address of a (&a): 0x7ffc99a8e9a4
Value of a (a): 10
Address that pointer p holds (p): 0x7ffc99a8e9a4
Content of the address pointer p holds (*p): 10
Address pointer p holds (p): 0x7ffc99a8e9a4
Content of the address pointer p holds (*p): 20
Address of a (&a): 0x7ffc99a8e9a4
Value of a (a): 5
```

In the above program, when **a=10;** the value 10 is stored in the address of variable **a.**

When **p=&a;** the pointer **p** holds the address of **a** and the expression ***p** contains the value of that address which is 10;

When **a=20;** the address that pointer p holds is unchanged, but the expression ***p** is changed because the address **&a** (which is same as p) contains 20.

When ***p=5;** the content in the address **p** which is equal to **&a** is changed from 20 to 5. Since the pointer **p** and variable **a** has address, value of **a** is changed to 5.

0x in the beginning represents the address is in hexadecimal form.

Note: You may not get the same address as an output on your system.

Unit-5

Virtual Function and Pure Virtual function

Virtual function is a member function of class declared in base class and overrided in the derived class.

Derived class tells the compiler to perform **late binding** on this function.

Late binding is also called as **Dynamic Binding or Runtime Binding**. In this, function call is resolved at runtime, so compiler determines the type of object at runtime and then it binds the function call.
The **virtual** keyword is used to make a member function of the base class Virtual.

Example: Program demonstrating Virtual Function

```
#include<iostream.h>
class BaseClass
{
   public:
       virtual void display()
       {
           cout << "Base class";
       }
};

class DerivedClass : public BaseClass
{
   public:
       void display()
       {
           cout << "Derived Class";
       }
};
int main()
{
    BaseClass *bc;       // Base class pointer
    DerivedClass dc;   // Derived class object
    bc = &dc;
    bc → display();     // Late Binding Occurs
}
```

Output:
Derived Class

In the above program, using **virtual** keyword with **BaseClass** class's function, late binding takes place and the derived version of function will be called, because base class pointer points to the derived class object.

Important points to be considered

- If a function is declared as virtual in the base class, it will be virtual in all its derived classes.
- The address of virtual function is placed in the virtual table and the compiler uses virtual pointer to point to the virtual function.

- **VTABLE** is a **Virtual Table** contains the address of virtual functions of each class and **VPTR** is a **Virtual Pointer**, which points to the virtual function for that object.
- Virtual function is declared with the keyword virtual in the base class and not in the definition.

Abstract Class and Pure Virtual Function

- **Abstract class** is a class which defines an abstract type and cannot be instantiated.
- It is used to provide an interface for its sub classes.
- Abstract class contains at least one Pure Virtual Function in it.
- **Pure virtual function** is function which has the notation "= 0" in the declaration of that function.
- Pure virtual function don't have implementation, we only declare it.
- If a base class contains at least one virtual function then that class is known as **Abstract class.**

Syntax

```
class class_name
{
    public:
        virtual void display() = 0; //Pure Virtual Function
};
```

- If "= 0" expression is added to the virtual function then function becomes pure virtual function.

Note: Adding **"= 0"** to virtual function does not assign a value. It only indicates that the virtual function is a pure function.

Why are Pure Virtual Functions necessary?

- Lets one take example of Class Shape. The class Shape has a function called draw(). The other classes like Circle, Square and Triangle are derive from Shape class.
- In the class Shape, it does not make any sense to provide a definition for the draw() function, because of one simple reason class Shape has no specific pattern or design. It is simply meant to act as a base class.
- In the Circle, Square and Triangle classes we can define a draw() function with proper function definition because they should just draw either Circle or Square (respectively) on the page. But in the Shape class, it makes no sense to provide a function definition for the draw() function. And therefore a draw() function in the Shape class should be a Pure Virtual Function.

Characteristics of Abstract Class

- Abstract class is mainly used for Upcasting, so that its derived classes can use its interface.
- It cannot be instantiated, but pointers and references of Abstract class type can be created.
- It can have normal functions and variables along with a Pure virtual function.

Example: Program demonstrating Abstract Class and Pure Virtual Function

```
#include <iostream.h>
class Shape                  // Abstract class
{
   protected:
     float length;
   public:
     void get_input()         //Note: This function is not virtual.
     {
        length=5;
     }
     virtual float area() = 0; // Pure virtual function
};

class Square : public Shape
{
   public:
     float area()
     {
         return length*length;
     }
};
class Circle : public Shape
{
   public:
     float area()
     {
          return 3.14*length*length;
     }
};

int main()
{
   Square s;
   Circle c;
   s.get_input();
   cout<<"Area of Square: "<<s.area()<<endl;
   c.get_input();
   cout<<"Area of Circle: "<<c.area()<<endl;
   return 0;
}
```

Output:
Area of Square: 25
Area of Circle: 78.5

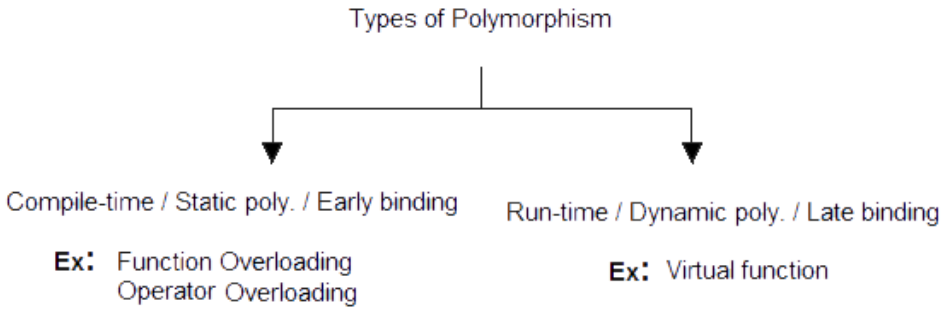
In the above program, pure virtual function **virtual float area = 0** is defined inside **class Shape**, so this class is an abstract class and you cannot create object of **class Shape**.

If we create object for abstract class, it will display error, because when we create a pure virtual function in Abstract class, we reserve a slot for a function in the Virtual Table (VTABLE) and it does not put any address in that slot. So the Virtual Table will be incomplete and because of that the compiler will not let the creation of object for such class and will display an error message.

Polymorphism

Polymorphism is a feature of OOPs that allows the object to behave differently in different conditions.
In C++ we have two types of polymorphism

- Compile time Polymorphism (Static or Early Binding)
- Runtime Polymorphism (Dynamic or Late Binding)



- **Compile time Polymorphism:**

If matching of function call with the correct function definition happens at compile time, this type of polymorphism is called Compile-Time Polymorphism. It is also called **Static Binding** or **Early Binding.** Static Binding is implemented in a program at the time of coding.
Even though there are two or more functions with same name, compiler uniquely identifies each function depending on the parameters passed to those functions.

Function overloading is an example of compile time polymorphism. More than one

function with same name, with different signature in a class or in a same scope is called function overloading
Another example of compile time polymorphism is **Operator overloading**. Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.

- **Runtime Polymorphism:**

In Run time polymorphism, function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. This is called D**ynamic Binding** or **Late Binding.**
Virtual function is an example of dynamic polymorphism.
Virtual function is used in situation, when we need to invoke derived class function using base class pointer. Giving new implementation of derived class method into base class and the calling of this new implemented function with base class's object is done by making base class function as virtual function. This is how we can achieve "Runtime Polymorphism".

Difference Between Static and Dynamic Binding

BASIS FOR COMPARISON	STATIC BINDING	DYNAMIC BINDING
Event Occurrence	Events occur at compile time are "Static Binding".	Events occur at run time are "Dynamic Binding".
Information	All information needed to call a function is known at compile time.	All information need to call a function come to know at run time.
Advantage	Efficiency.	Flexibility.
Time	Fast execution.	Slow execution.
Alternate name	Early Binding.	Late Binding.
Example	Overloaded function call, Overloaded operators.	Virtual function in C++.