

## Part: 51

### Pointers to Objects and Arrow Operator in CPP

In this tutorial, we will discuss pointers to objects and arrow operator in C++

#### Pointer to objects in C++

As discussed before pointers are used to store addresses of variables which have data types like int, float, double etc. But pointer can also store the address of an object. An example program is shown below to demonstrate the concept of pointer to objects.

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

class Complex{
    int real, imaginary;
public:
    void getData(){
        cout<<"The real part is "<< real<<endl;
        cout<<"The imaginary part is "<< imaginary<<endl;
    }

    void setData(int a, int b){
        real = a;
        imaginary = b;
    }
};

int main(){
    Complex *ptr = new Complex;
    (*ptr).setData(1, 54); is exactly same as
    (*ptr).getData(); is as good as

    return 0;
}
```

#### Code Snippet 1: Pointer to objects Example Program 1

As shown in a code snippet 1,

1. We created a class **“Complex”**, which contains two private data members **“real”** and **“imaginary”**.
2. The class **“complex”** contains two members functions **“getdata”** and **“setdata”**
3. The Function **“setdata”** will take two parameters and assign the values of parameters to the private data members **“real”** and **“imaginary”**
4. The Function **“getdata”** will print the values of private data members **“real”** and **“imaginary”**
5. In the main program object is created dynamically by using the **“new”** keyword and its address is assigned to the pointer **“ptr”**
6. The member function **“setdata”** is called using the pointer **“ptr”** and the values **“1, 54”** are passed.
7. The member function **“getdata”** is called using the pointer **“ptr”** and it will print the values of data members.

The main thing to note here is that we called the member function with pointers instead of object but still it will give same result because pointer is pointing to the address of that object.

The output of the following program is shown below,

```
The real part is 1
The imaginary part is 54
```

**Figure 1:** Pointer to Objects Program 1 Output

### Arrow Operator in C++

Another example program for the pointer to Objects and the use of the “**Arrow**” Operator is shown below.

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

class Complex{
    int real, imaginary;
public:
    void getData(){
        cout<<"The real part is "<< real<<endl;
        cout<<"The imaginary part is "<< imaginary<<endl;
    }

    void setData(int a, int b){
        real = a;
        imaginary = b;
    }
};

int main(){
    Complex *ptr = new Complex;
    ptr->setData(1, 54);
    ptr->getData();

    // Array of Objects
    Complex *ptr1 = new Complex[4];
    ptr1->setData(1, 4);
    ptr1->getData();
    return 0;
}
```

**Code Snippet 2:** Pointer to Objects with Arrow Operator Example Program 2

As shown in code snippet 2,

1. We created a class “**Complex**”, which contains two private data members “**real**” and “**imaginary**”.
2. The class “**complex**” contains two members functions “**getdata**” and “**setdata**”
3. The Function “**setdata**” will take two parameters and assign the values of parameters to the private data members “**real**” and “**imaginary**”
4. The Function “**getdata**” will print the values of private data members “**real**” and “**imaginary**”
5. In the main program object is created dynamically by using the “**new**” keyword and its address is assigned to the pointer “**ptr**”
6. The member function “**setdata**” is called using the pointer “**ptr**” with the arrow operator “**->**” and the values “**1, 54**” are passed.

7. The member function **“getdata”** is called using the pointer **“ptr”** with the arrow operator **“->”** and it will print the values of data members.
8. Array of objects is created dynamically by using the **“new”** keyword and its address is assigned to the pointer **“ptr1”**
9. The member function **“setdata”** is called using the pointer **“ptr1”** with the arrow operator **“->”** and the values **“1, 4”** are passed.
10. The member function **“getdata”** is called using the pointer **“ptr1”** with the arrow operator **“->”** and it will print the values of data members.

The main thing to note here is that we called the member function with pointers by using arrow operator **“->”** instead of the dot operator **“.”** but still it will give the same results.

The output of the following program is shown below,

```
The real part is 1
The imaginary part is 54
The real part is 1
The imaginary part is 4
```

**Figure 2:** Pointer to Objects Program 2 Output

## Part: 52

### Array of Objects Using Pointers in C++

In this tutorial, we will discuss an array of objects using pointers in C++

### Array of Objects Using Pointers in C++

Array of objects can be defined as an array that's each element is an object of the class. In this tutorial, we will use the pointer to store the address of an array of objects. An example program is shown below to demonstrate the concept of an array of objects using pointers.

```
#include<iostream>
using namespace std;
class ShopItem
{
    int id;
    float price;
public:
    void setData(int a, float b){
        id = a;
        price = b;
    }
    void getData(void){
        cout<<"Code of this item is "<< id<<endl;
        cout<<"Price of this item is "<<price<<endl;
    }
};
```

#### Code Snippet 1: Array of Objects Using Pointers Example Program

As shown in a code snippet 1,

1. We created a class **“ShopItem”**, which contains two private data members **“id”** and **“price”**.
2. The class **“ShopItem”** contains two members functions **“setdata”** and **“getdata”**
3. The Function **“setdata”** will take two parameters and assign the values of parameters to the private data members **“id”** and **“price”**
4. The Function **“getdata”** will print the values of private data members **“id”** and **“price”**

```
int main(){
    int size = 3;
    ShopItem *ptr = new ShopItem [size];
    ShopItem *ptrTemp = ptr;
    int p, i;
    float q;
    for (i = 0; i < size; i++){
        cout<<"Enter Id and price of item "<< i+1<<endl;
        cin>>p>>q;
        // (*ptr).setData(p, q);
        ptr->setData(p, q);
        ptr++;
    }
    for (i = 0; i < size; i++)
    {
        cout<<"Item number: "<<i+1<<endl;
        ptrTemp->getData();
        ptrTemp++;
    }
    return 0;
}
```

## Code Snippet 2: Main Program

As shown in code snippet 2,

1. We created an integer variable **“size”** and assigned the value **“3”** to it.
2. Array of objects of size **“3”** is created dynamically by using the **“new”** keyword and its address is assigned to the pointer **“ptr”**
3. The address of pointer **“ptr”** is assigned to another pointer **“ptrTemp”**
4. Two integer variables **“p”** and **“i”** are declared and one float variable **“q”** is declared
5. We created a **“for”** loop which will run till the size of array and will take input for **“id”** and **“price”** from user at run time. In this **“for”** loop **“setdata”** function is called using pointer **“ptr”**; the function will set the values of **“id”** and **“price”** which user will enter. The value of the pointer **“ptr”** is incremented by 1 in every iteration of loop.
6. We created another **“for”** loop which will run till the size of array and will print the number of the item. In this **“for”** loop **“getdata”** function is called using pointer **“ptr”**; the function will print the values of **“id”** and **“price”**. The value of the pointer **“ptrTemp”** is incremented by 1 in every iteration of loop.

The main thing to note here is that in the first **“for”** loop we are incrementing the value of the pointer **“ptr”** because it is pointing to the address of array of objects and when loop will run every time the function **“setdata”** will be called by the different object. If we don't increment the value of the pointer **“ptr”** each time function **“setdata”** will be called by the same object. Likewise, in the second loop we are incrementing the pointer **“ptrTemp”** so that the function **“getdata”** could be called by each object in the array.

The input and output of the following program is shown below,

```
Enter Id and price of item 1
1001
1.1
Enter Id and price of item 2
1002
1.2
Enter Id and price of item 3
1003
3.3
```

Figure 1: Array of Objects Using Pointer Program Input

```
Item number: 1
Code of this item is 1001
Price of this item is 1.1
Item number: 2
Code of this item is 1002
Price of this item is 1.2
Item number: 3
Code of this item is 1003
Price of this item is 3.3
```

Figure 2: Array of Objects Using Pointer Program Output

## Part: 53

### This Pointer in C++

In this tutorial, we will discuss 'this' pointer in C++

#### 'this' Pointer in C++

**"this"** is a keyword that is an implicit pointer. **"this"** pointer points to the object which calls the member function. An example program is shown below to demonstrate the concept of **"this"** pointer.

```
#include<iostream>
using namespace std;
class A{
    int a;
    public:
        void setData(int a){
            this->a = a;
        }

        void getData(){
            cout<<"The value of a is "<<a<<endl;
        }
};
```

#### Code Snippet 1: **"this"** Pointer Example Program

As shown in a code snippet 1,

1. We created a class **"A"**, which contains private data members **"a"**.
2. The class **"A"** contains two members functions **"setData"** and **"getData"**
3. The Function **"setData"** will take one parameter and assign the values of parameter to the private data members **"a"** using **"this"** pointer. As we know that one copy of member function is shared between all object. The use of **"this"** pointer helps to points to the object which invokes the member function.
4. The Function **"getData"** will print the values of private data members **"a"**

The code for the main program is shown below,

```
int main(){
    A a;
    a.setData(4);
    a.getData();
    return 0;
}
```

#### Code Snippet 2: Main Program

As shown in code snippet 2,

1. Object **"a"** is of data type **"A"** is created
2. The function **"setData"** is called using object **"a"** and the value **"4"** is passed to the function
3. The function **"getData"** is called using object **"a"**

The input and output of the following program is shown below,

The value of a is 4

Figure 1: Program Output

“**this**” pointer can be used to return a reference to the invoking object. An example program is shown below.

```
class A{
    int a;
    public:
        A & setData(int a){
            this->a = a;
            return *this;
        }

        void getData(){
            cout<<"The value of a is "<<a<<endl;
        }
};

int main(){
    A a;
    a.setData(4).getData();
    return 0;
}
```

Code Snippet 3: Return Reference to Invoking Object Example Program

As shown in Code Snippet 3,

1. In the function “**setData**” the reference of the object is returned using “**this**” pointer.
2. In the main program by using a single object we have made a chain of the function calls. The main thing to note here is that the function “**setData**” is returning an object on which we have used the “**getData**” function. So, we don’t need to call the function “**getData**” explicitly.

## Part: 54

### Polymorphism in C++

In this tutorial, we will discuss polymorphism in C++

### Polymorphism in C++

**“Poly”** means several and **“morphism”** means form. So, we can say that polymorphism is something that has several forms or we can say it as one name and multiple forms. There are two types of polymorphism:

- Compile-time polymorphism
- Run time polymorphism

#### Compile Time Polymorphism

In compile-time polymorphism, it is already known which function will run. Compile-time polymorphism is also called early binding, which means that you are already bound to the function call and you know that this function is going to run. There are two types of compile-time polymorphism:

##### 1. Function Overloading

This is a feature that lets us create more than one function and the functions have the same names but their parameters need to be different. If function overloading is done in the program and function calls are made the compiler already knows that which functions to execute.

##### 2. Operator Overloading

This is a feature that lets us define operators working for some specific tasks. For example, we can overload the operator **“+”** and define its functionality to add two strings. Operator loading is also an example of compile-time polymorphism because the compiler already knows at the compile time which operator has to perform the task.

#### Run Time Polymorphism

In the run-time polymorphism, the compiler doesn't know already what will happen at run time. Run time polymorphism is also called late binding. The run time polymorphism is considered slow because function calls are decided at run time. Run time polymorphism can be achieved from the virtual function.

##### 3. Virtual Function

A function that is in the parent class but redefined in the child class is called a virtual function. **“virtual”** keyword is used to declare a virtual function.

### Polymorphism in C++

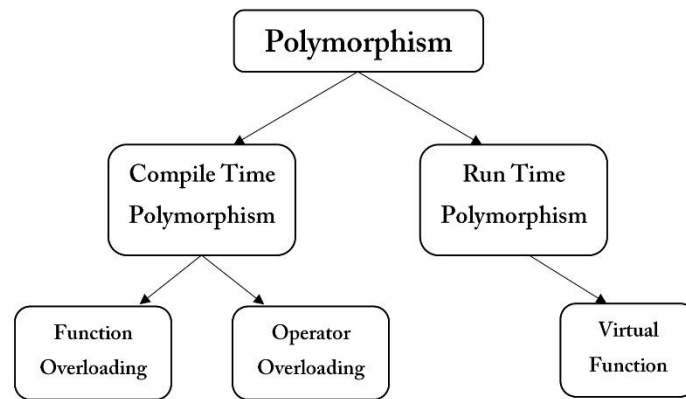
The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So, the same person possess different behaviour in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object-Oriented Programming.

**In C++ polymorphism is mainly divided into two types:**

- Compile time Polymorphism



- Runtime Polymorphism



1. **Compile time polymorphism:** This type of polymorphism is achieved by function overloading or operator overloading.
- **Function Overloading:** When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

## Rules of Function Overloading

```

// C++ program for function overloading
#include <bits/stdc++.h>
using namespace std;
class Geeks{
public:
    // function with 1 int parameter
    void func(int x){
        cout << "value of x is " << x << endl;
    }
    // function with same name but 1 double parameter
    void func(double x){
        cout << "value of x is " << x << endl;
    }
    // function with same name and 2 int parameters
    void func(int x, int y){
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};

int main() {
    Geeks obj1;
    // Which function is called will depend on the parameters passed
    // The first 'func' is called
    obj1.func(7);
    // The second 'func' is called
    obj1.func(9.132);
    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}
  
```

## Output:

```

value of x is 7
value of x is 9.132
value of x and y is 85, 64
  
```

In the above example, a single function named **func** acts differently in three different situations which is the property of polymorphism.

**Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So, a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

Example:

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0) {real = r;  imag = i;}
    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj){
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};
int main(){
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:

12 + i9

In the above example the operator '+' is overloaded. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit [this link](#).

**Runtime polymorphism:** This type of polymorphism is achieved by Function Overriding.

**Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding
#include <bits/stdc++.h>
using namespace std;
class base{
public:
    virtual void print ()
    { cout<< "print base class" <<endl; }
    void show ()
    { cout<< "show base class" <<endl; }
};
class derived:public base{
public:
    void print () //print () is already virtual function in derived class, we
    could also declared as virtual void print () explicitly
    { cout<< "print derived class" <<endl; }
```

```

        void show ()
        { cout<< "show derived class" <<endl;
        }
    };
    //main function
    int main()
    {
        base *bptr;
        derived d;
        bptr = &d;
        //virtual function, binded at runtime (Runtime polymorphism)
        bptr->print();
        // Non-virtual function, binded at compile time
        bptr->show();
        return 0;
    }

```

## Output:

```

print derived class
show base class

```

## Part: 55

### Pointers to Derived Classes in C++

In this tutorial, we will discuss pointer to derived class in C++

### Pointer to Derived Class in C++

In C++ we are provided with the functionality to point the pointer to derived class or base class. An example program is shown below to demonstrate the concept of pointer to a derived class in C++

```

#include<iostream>
using namespace std;
class BaseClass{
public:
    int var_base;
    void display(){
        cout<<"Dispalying Base class variable var_base "<<var_base<<endl;
    }
};
class DerivedClass : public BaseClass{
public:
    int var_derived;
    void display(){
        cout<<"Dispalying Base class variable var_base "<<var_base<<endl;
        cout<<"Dispalying Derived class variable var_derived
"<<var_derived<<endl;
    }
};

```

**Code Snippet 1:** Pointer to Derived Class Program Example

As shown in Code snippet 1,

1. We created a class **“BaseClass”** which contains public data member **“var\_base”** and member function **“display”**. The member function **“display”** will print the value of data member **“var\_base”**
2. We created another class **“DerivedClass”** which is inheriting **“BaseClass”** and contains data member **“var\_derived”** and member function **“display”**. The member function **“display”** will print the values of data members **“var\_base”** and **“var\_derived”**

The code for the main program is shown below,

```
int main(){
    BaseClass * base_class_pointer;
    BaseClass obj_base;
    DerivedClass obj_derived;
    base_class_pointer = &obj_derived; // Pointing base class pointer to derived
class
    base_class_pointer->var_base = 34;
    // base_class_pointer->var_derived= 134; // Will throw an error
    base_class_pointer->display();
    base_class_pointer->var_base = 3400;
    base_class_pointer->display();
    DerivedClass * derived_class_pointer;
    derived_class_pointer = &obj_derived;
    derived_class_pointer->var_base = 9448;
    derived_class_pointer->var_derived = 98;
    derived_class_pointer->display();
    return 0;
}
```

#### Code Snippet 2: Main Program

As shown in code snippet 2,

1. We created a pointer **“base\_class\_pointer”** of the data type **“Baseclass”**.
2. Object **“obj\_base”** of the data type **“BaseClass”** is created.
3. Object **“obj\_derived”** of the data type **“DerivedClass”** is created
4. Pointer **“base\_class\_pointer”** of the base class is pointing to the object **“obj\_derived”** of the derived class
5. By using the pointer **“base\_class\_pointer”** of the base class we have set the value of the data member **“var\_base”** by **“34”**. The main thing to note here is that we cannot set the value of the derived class data member by using the base class pointer otherwise the compiler will throw an error.
6. The function **“display”** is called using a base class pointer. The main thing to note here is that the base class **“display”** function will run here.
7. Again by using the pointer **“base\_class\_pointer”** of the base class we have set the value of the data member **“var\_base”** by **“3400”** which will update the previous value and the function **“display”** is called.
8. We created a pointer **“derived\_class\_pointer”** of the data type **“DerivedClass”**
9. Pointer **“Derived\_class\_pointer”** of the derived class is pointing to the object **“obj\_derived”** of the derived class

10. By using pointer **“Derived\_class\_pointer”** of the derived class we have set the value of the data member **“var\_base”** of the base class by **“9448”**. The main thing to note here is that this will not throw an error because we can set the value of base class data member by using derived class pointer but we cannot set the value of derived class data member by using base class pointer
11. By using pointer **“Derived\_class\_pointer”** of the derived class we have set the value of the data member **“var\_derived”** of the derived class by **“98”**.
12. The function **“display”** is called using a derived class pointer. The main thing to note here is that the derived class **“display”** function will run here.

The output of the following program is shown in figure 1,

```

Displaying Base class variable var_base 34
Displaying Base class variable var_base 3400
Displaying Base class variable var_base 9448
Displaying Derived class variable var_derived 98

```

Figure 1: Program Output

## Part: 56

### Virtual Functions in C++

In this tutorial, we will discuss virtual functions in C++

### Virtual Functions in C++

A member function in the base class which is declared using virtual keyword is called virtual functions. They can be redefined in the derived class. To demonstrate the concept of virtual functions an example program is shown below

```

#include<iostream>
using namespace std;

class BaseClass{
public:
    int var_base=1;
    virtual void display(){
        cout<<"1 Displaying Base class variable var_base "<<var_base<<endl;
    }
};

class DerivedClass : public BaseClass{
public:
    int var_derived=2;
    void display(){
        cout<<"2 Displaying Base class variable var_base "<<var_base<<endl;
    }
};

```

```

        cout<<"2 Displaying Derived class variable var_derived
"<<var_derived<<endl;
    }
};

```

## Code Snippet 1: Virtual Function Example Program

As shown in code snippet 1,

1. We created a class **“BaseClass”** which contains public data member **“var\_base”** which has the value **“1”** and member function **“display”**. The member function **“display”** will print the value of data member **“var\_base”**
2. We created another class **“DerivedClass”** which is inheriting **“BaseClass”** and contains data member **“var\_derived”** which has the value **“2”** and member function **“display”**. The member function **“display”** will print the values of data members **“var\_base”** and **“var\_derived”**

The code for the main program is shown below

```

int main(){
    BaseClass * base_class_pointer;
    BaseClass obj_base;
    DerivedClass obj_derived;

    base_class_pointer = &obj_derived;
    base_class_pointer->display();
    return 0;
}

```

## Code Snippet 2: Main Program

As shown in code snippet 2,

1. We created a pointer **“base\_class\_pointer”** of the data type **“Baseclass”**
2. Object **“obj\_base”** of the data type **“BaseClass”** is created.
3. Object **“obj\_derived”** of the data type **“DerivedClass”** is created
4. Pointer **“base\_class\_pointer”** of the base class is pointing to the object **“obj\_derived”** of the derived class
5. The pointer **“base\_class\_pointer”** is pointed to the object **“obj\_derived”** of the derived class.
6. The function **“display”** is called using the pointer **“base\_class\_pointer”** of the base class.

The main thing to note here is that if we don't use the **“virtual”** keyword with the **“display”** function of the base class then beside of the point that we have pointed our base call pointer to derived class object still the compiler would have called the **“display”** function of the base class because this is its default behaviour as we have seen in the previous tutorial.

But we have used the **“virtual”** keyword with the **“display”** function of the base class to make it **virtual function** so when the display function is called by using the base class pointer the display function of the derived class will run because the base class pointer is pointing to the derived class object.

The output of the following program is shown in figure 1

```
2 Displaying Base class variable var_base 1
2 Displaying Derived class variable var_derived 2
```

Figure 1: Program Output

## Part: 57

### Virtual Functions Example + Creation Rules in C++

In this tutorial, we will discuss virtual functions example and its creation rules in C++

### Virtual Functions Example in C++

As we have seen in the previous tutorial that how virtual functions are used to implement run-time polymorphism. In this tutorial, we will see an example of virtual functions.

```
class CWH{
protected:
    string title;
    float rating;
public:
    CWH(string s, float r){
        title = s;
        rating = r;
    }
    virtual void display(){}
};
```

### Code Snippet 1: Code with Harry Class

As shown in a code snippet 1,

1. We created a class **“CHW”** which contains protected data members **“title”** which has a **“string”** data type and **“rating”** which has a **“float”** data type.

2. The class **“CWH”** has a parameterized constructor which takes two parameters **“s”** and **“r”** and assign their values to the data members **“title”** and **“rating”**
3. The class **“CHW”** has a virtual function void **“display”** which does nothing

```
class CWHVideo: public CWH
{
    float videoLength;
public:
    CWHVideo(string s, float r, float vl): CWH(s, r){
        videoLength = vl;
    }
    void display(){
        cout<<"This is an amazing video with title "<<title<<endl;
        cout<<"Ratings: "<<rating<<" out of 5 stars"<<endl;
        cout<<"Length of this video is: "<<videoLength<<" minutes"<<endl;
    }
};
```

### Code Snippet 2: Code with Harry Video Class

As shown in a code snippet 2,

1. We created a class **“CHWVideo”** which is inheriting the **“CWH”** class and contains private data members **“videoLength”** which has a **“float”** data type.
2. The class **“CWHVideo”** has a parameterized constructor which takes three parameters **“s”**, **“r”** and **“vl”**. The constructor of the base class is called in the derived class and the values of the variables **“s”** and **“r”** are passed to it. The value of the parameter **“vl”** will be assigned to the data members **“videoLength”**
3. The class **“CHWVideo”** has a function void **“display”** which will print the values of the data members **“title”**, **“rating”** and **“videoLength”**

```
class CWHText: public CWH
{
    int words;
public:
    CWHText(string s, float r, int wc): CWH(s, r){
        words = wc;
    }
    void display(){
        cout<<"This is an amazing text tutorial with title "<<title<<endl;
        cout<<"Ratings of this text tutorial: "<<rating<<" out of 5 stars"<<endl;
        cout<<"No of words in this text tutorial is: "<<words<<" words"<<endl;
    }
};
```

### Code Snippet 3: Code with Harry Text Class

As shown in a code snippet 3,

1. We created a class **“CHWText”** which is inheriting the **“CWH”** class and contains private data members **“words”** which has an **“int”** data type.
2. The class **“CWHText”** has a parameterized constructor which takes three parameters **“s”**, **“r”** and **“wc”**. The constructor of the base class is called in the derived class and the values of the variables **“s”** and **“r”** are passed to it. The value of the parameter **“wc”** will be assigned to the data members **“words”**



3. The class **“CWHText”** has a function void **“display”** which will print the values of the data members **“title”, “rating”** and **“words”**

```
int main(){
    string title;
    float rating, vlen;
    int words;

    // for Code With Harry Video
    title = "Django tutorial";
    vlen = 4.56;
    rating = 4.89;
    CWHVideo djVideo(title, rating, vlen);

    // for Code With Harry Text
    title = "Django tutorial Text";
    words = 433;
    rating = 4.19;
    CWHText djText(title, rating, words);

    CWH* tuts[2];
    tuts[0] = &djVideo;
    tuts[1] = &djText;

    tuts[0]->display();
    tuts[1]->display();

    return 0;
}
```

## Code Snippet 4: Main Program

As shown in a code snippet 4,

1. We created a string variable **“title”**, float variables **“rating”, “vlen”** and integer variable **“words”**
2. For the code with harry video class, we have assigned **“Django tutorial”** to the string **“title”**, **“4.56”** to the float **“vlen”** and **“4.89”** to the float **“rating”**.
3. An object **“djVideo”** is created of the data type **“CWHVideo”** and the variables **“title”, “rating”** and **“vlen”** are passed to it.
4. For the code with harry text class, we have assigned **“Django tutorial text”** to the string **“title”**, **“433”** to the integer **“words”** and **“4.19”** to the float **“rating”**.
5. An object **“djText”** is created of the data type **“CWHText”** and the variables **“title”, “rating”** and **“words”** are passed to it.
6. Two pointers array **“tuts”** is created of the **“CWH”** type
7. The address of the **“djVideo”** is assigned to **“tuts[0]”** and the address of the **“djText”** is assigned to **“tuts[1]”**
8. The function **“display”** is called using pointers **“tuts[0]”** and **“tuts[1]”**

The main thing to note here is that if we don't use the **“virtual”** keyword with the **“display”** function of the base class then the **“display”** function of the base class will run.

But we have used the “**virtual**” keyword with the “**display**” function of the base class to make it a **virtual function** so when the display function is called by using the base class pointer the display function of the derived class will run because the base class pointer is pointing to the derived class object.

The output of the following program is shown in figure 1

```
This is an amazing video with title Django tutorial
Ratings: 4.89 out of 5 stars
Length of this video is: 4.56 minutes
This is an amazing text tutorial with title Django tutorial Text
Ratings of this text tutorial: 4.19 out of 5 stars
No of words in this text tutorial is: 433 words
```

**Figure 1: Program Output**

#### Rules for virtual functions

1. They cannot be static
2. They are accessed by object pointers
3. Virtual functions can be a friend of another class
4. A virtual function in the base class might not be used.
5. If a virtual function is defined in a base class, there is no necessity of redefining it in the derived class

## Part: 58

### Abstract Base Class & Pure Virtual Functions in C++

In this tutorial, we will discuss abstract base class and pure virtual functions in C++

#### Pure Virtual Functions in C++

Pure virtual function is a function that doesn't perform any operation and the function is declared by assigning the value 0 to it. Pure virtual functions are declared in abstract classes.

#### Abstract Base Class in C++

Abstract base class is a class that has at least one pure virtual function in its body. The classes which are inheriting the base class must need to override the virtual function of the abstract class otherwise compiler will throw an error.

To demonstrate the concept of abstract class and pure virtual function an example program is shown below.

```
class CWH{
protected:
    string title;
    float rating;
public:
    CWH(string s, float r){
        title = s;
        rating = r;
    }
    virtual void display()=0;
};
```

## Code Snippet 1: Code with Harry Class

As shown in code snippet 1,

1. We created a class **“CHW”** which contains protected data members **“title”** which has **“string”** data type and **“rating”** which has **“float”** data type.
2. The class **“CWH”** has a parameterized constructor which takes two parameters **“s”** and **“r”** and assign their values to the data members **“title”** and **“rating”**
3. The class **“CHW”** has a pure virtual function void **“display”** which is declared by 0. The main thing to note here is that as the **“display”** function is a pure virtual function it is compulsory to redefine it in the derived **classes**.

```
class CWHVideo: public CWH
{
    float videoLength;
public:
    CWHVideo(string s, float r, float vl): CWH(s, r){
        videoLength = vl;
    }
    void display(){
        cout<<"This is an amazing video with title "<<title<<endl;
        cout<<"Ratings: "<<rating<<" out of 5 stars"<<endl;
        cout<<"Length of this video is: "<<videoLength<<" minutes"<<endl;
    }
};
```

## Code Snippet 2: Code with Harry Video Class

As shown in code snippet 2,

1. We created a class **“CHWVideo”** which is inheriting **“CWH”** class and contains private data members **“videoLength”** which has **“float”** data type.
2. The class **“CWHVideo”** has a parameterized constructor which takes three parameters **“s”**, **“r”** and **“vl”**. The constructor of the base class is called in the derived class and the values of the variables **“s”** and **“r”** are passed to it. The value of the parameter **“vl”** will be assigned to the data members **“videoLength”**
3. The class **“CHWVideo”** has a function void **“display”** which will print the values of the data members **“title”**, **“rating”** and **“videoLength”**

```
class CWHText: public CWH
{
    int words;
public:
    CWHText(string s, float r, int wc): CWH(s, r){
        words = wc;
    }
    void display(){
        cout<<"This is an amazing text tutorial with title "<<title<<endl;
        cout<<"Ratings of this text tutorial: "<<rating<<" out of 5 stars"<<endl;
        cout<<"No of words in this text tutorial is: "<<words<<" words"<<endl;
    }
};
```

## Code Snippet 3: Code with Harry Text Class

As shown in code snippet 3,

1. We created a class **“CHWText”** which is inheriting **“CWH”** class and contains private data members **“words”** which has **“int”** data type.
2. The class **“CWHText”** has a parameterized constructor which takes three parameters **“s”, “r”** and **“wc”**. The constructor of the base class is called in the derived class and the values of the variables **“s”** and **“r”** are passed to it. The value of the parameter **“wc”** will be assigned to the data members **“words”**
3. The class **“CHWText”** has a function void **“display”** which will print the values of the data members **“title”, “rating”** and **“words”**

```
int main(){
    string title;
    float rating, vlen;
    int words;

    // for Code With Harry Video
    title = "Django tutorial";
    vlen = 4.56;
    rating = 4.89;
    CWHVideo djVideo(title, rating,
vlen);

    // for Code With Harry Text
    title = "Django tutorial Text";

    words = 433;
    rating = 4.19;
    CWHText djText(title, rating,
words);

    CWH* tuts[2];
    tuts[0] = &djVideo;
    tuts[1] = &djText;

    tuts[0]->display();
    tuts[1]->display();

    return 0;
}
```

## Code Snippet 4: Main Program

As shown in code snippet 4,

1. We created a string variable **“title”**, float variables **“rating”, “vlen”** and integer variable **“words”**
2. For the code with harry video class we have assigned **“Django tutorial”** to the string **“title”**, **“4.56”** to the float **“vlen”** and **“4.89”** to the float **“rating”**.
3. An object **“djVideo”** is created of the data type **“CWHVideo”** and the variables **“title”, “rating”** and **“vlen”** are passed to it.
4. For the code with harry text class we have assigned **“Django tutorial text”** to the string **“title”**, **“433”** to the integer **“words”** and **“4.19”** to the float **“rating”**.
5. An object **“djText”** is created of the data type **“CWHText”** and the variables **“title”, “rating”** and **“words”** are passed to it.
6. Two pointers array **“tuts”** is created of the **“CWH”** type
7. The address of the **“djVideo”** is assigned to **“tuts[0]”** and the address of the **“djText”** is assigned to **“tuts[1]”**
8. The function **“display”** is called using pointers **“tuts[0]”** and **“tuts[1]”**

The main thing to note here is that if we don’t override the pure virtual function in the derived class the compiler will throw an error as shown in figure 1.

```
tut58.cpp:14:22: note: 'virtual void CWH::display()'
virtual void display()=0; // do-nothing function --> pure virtual function
~~~~~
```

## Figure 1: Program Error

The output of the following program is shown in figure 2

```
This is an amazing video with title Django tutorial
Ratings: 4.89 out of 5 stars
Length of this video is: 4.56 minutes
This is an amazing text tutorial with title Django tutorial Text
Ratings of this text tutorial: 4.19 out of 5 stars
No of words in this text tutorial is: 433 words
```

Figure 2: Program Output

## Part: 59

### File I/O in C++: Working with Files

In this tutorial, we will discuss file input and output in C++

The file is a patent of data which is stored in the disk. Anything written inside the file is called a patent, for example: **“#include”** is a patent. The text file is the combination of multiple types of characters, for example, semicolon **“;”** is a character.

The computer read these characters in the file with the help of the ASCII code. Every character is mapped on some decimal number. For example, ASCII code for the character **“A”** is **“65”** which is a decimal number. These decimal numbers are converted into a binary number to make them readable for the computer because the computer can only understand the language of **“0”** and **“1”**.

The reason that computers can only understand binary numbers is that a computer is made up of switches and switches only perform two operations **“true”** or **“false”**.

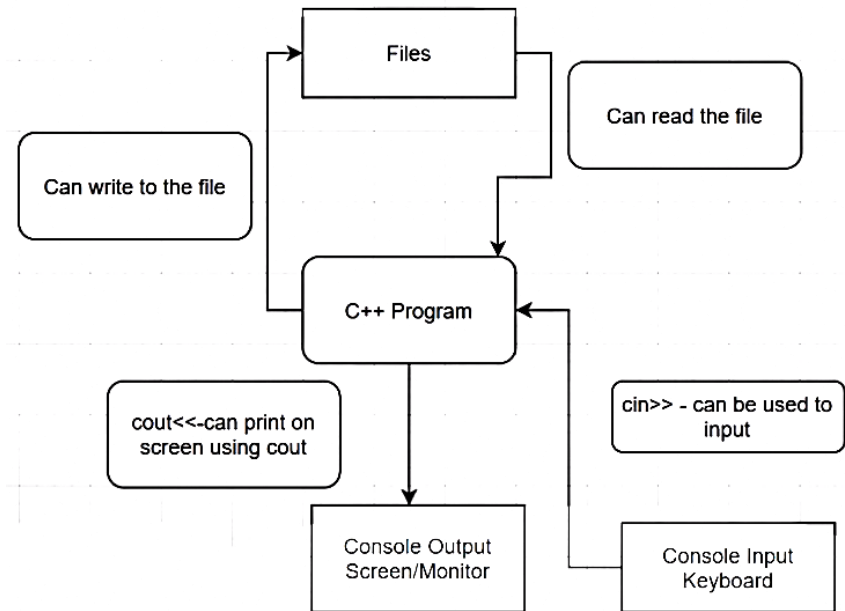
### File Input and Output in C++

The file can be of any type whether it is a file of a C++ program, file of a game, or any other type of file. There are two main operations which can be performed on files

- Read File

- Write File

An image is shown below to show the process of file read and write.



**Figure 1: File Read and Write Diagram**

As shown in figure 1,

1. The user can provide input to the C++ program by using keyboard through “**cin>>**” keyword
2. The user can get output from the C++ program on the monitor through “**cout<<**” keyword
3. The user can write on the file
4. The user can read the file

## Part: 60

### File I/O in C++: Reading and Writing Files

In this tutorial, we will discuss File I/O in C++: Reading and Writing Files

### File I/O in C++: Reading and Writing Files

These are some useful classes for working with files in C++

- **fstreambase**
- **ifstream** --> derived from **fstreambase**
- **ofstream** --> derived from **fstreambase**

In order to work with files in C++, you will have to open it. Primarily, there are 2 ways to open a file:

- Using the constructor
- Using the member function `open()` of the class

An example program is shown below to demonstrate the concept of reading and writing files

```

#include<iostream>
#include<fstream>
using namespace std;
int main(){
    string st = "Harry bhai";
    // Opening files using constructor and writing it
    ofstream out("sample60.txt"); // Write operation
    out<<st;
  
```

```
    return 0;
}
```

## Code Snippet 1: Writing Files Example Program

As shown in a code snippet 1,

1. We have created a string **“st”** which has a value **“harry Bhai”**
2. Object **“out”** is created of the type `ofstream` and the file **“sample60.txt”** is passed to it
3. The string **“st”** is passed to object **“out”**

The output of the following program is shown in figure 1



```
sample60.txt
1 Harry bhai
```

Figure 1: Writing File Operation Output

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

int main(){
    string st2;
    // Opening files using constructor and reading it
    ifstream in("sample60b.txt"); // Read operation
    in>>st2;
    getline(in, st2);
    cout<<st2;

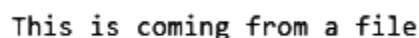
    return 0;
}
```

## Code Snippet 2: Reading Files Example Program

As shown in a code snippet 1,

1. We have created a string **“st2”** which is empty
2. We have made a text file **“sample60b.txt”** and written **“This is coming from a file”** in it
3. Object **“in”** is created of the type `istream` and the file **“sample60b.txt”** is passed to it
4. The function **“getline”** is called and the object **“in”** and the string **“st2”** are passed to it. The main thing to note here is that the function **“getline”** is used when we want to read the whole line
5. String **“st2”** is printed

The output of the following program is shown in figure 2



```
This is coming from a file
```

Figure 2: Reading File Operation Output