Runtime Polymorphism

- the compiler resolves the object at run time and then it decides which function call should be associated with that object.
- It is also known as dynamic or late binding polymorphism.
- This type of polymorphism is executed through virtual functions and function overriding.
- the compiler resolves the object at run time and then it decides which function call should be associated with that object.
- It is also known as dynamic or late binding polymorphism.
- This type of polymorphism is executed through virtual functions and function overriding.

```
class Animal {
   public:
void eat(){cout<<"Eating...";}
};
class Dog: public Animal
{public:
  void eat(){ cout<<"Eating bread...";}
};
int main(void) {
Animal *a;
   Dog d;
   a = &d;
   a->eat();
}
```

Output □ “Eating...”

- The output is “Eating…” which is from the base class ‘Animal’
- So irrespective of what type object the base pointer is holding, the program outputs the contents of the function of the class whose base pointer is the type of. In this case, also static linking is carried out.
- In order to make the base pointer output, correct contents and proper linking, we go for dynamic binding of functions. This is achieved using Virtual functions mechanism.

Virtual Functions

- A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a **virtual** keyword in a base class.
- The resolving of a function call is done at runtime.

Rules for Virtual Functions

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be the same in the base as well as the derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have a virtual destructor but it cannot have a virtual constructor.
-
- Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class.
- Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer since the print() function is declared with the virtual keyword so it will be bound at runtime (output is *print derived class* as the pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is *show base class* as the pointer is of base type).
- ***Note:*** *If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.*

Working of Virtual Functions (concept of VTABLE and VPTR)

- if a class contains a virtual function then the compiler itself does two things.

1. If an object of that class is created then a **virtual pointer (VPTR)** is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
2. Irrespective of whether the object is created or not, the class contains as a member **a static array of function pointers called VTABLE**. Cells of this table store the address of each virtual function contained in that class.
3. Explanation(runpoly4): Initially, we create a pointer of the type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.
4. A similar concept of Late and Early Binding is used as in the above example. For the fun_1() function call, the base class version of the function is called, fun_2() is overridden in the derived class so the derived class version is called, fun_3() is not overridden in the derived class and is a virtual function so the base class version is called, similarly fun_4() is not overridden so base class version is called.
5. Note: fun_4(int) in the derived class is different from the virtual function *fun_4() in the base class as prototypes of both functions are different.*

Limitations of Virtual Functions

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

```
class Base {

public:

  // Initialization of virtual function

  virtual void fun(int x = 0)

  {

    cout << "Base::fun(), x = " << x << endl;

  }

};

 // Initialization of Derived class

class Derived : public Base {

public:

  // NOTE this virtual function will take an argument
```

```
        // But haven't initialized yet
        virtual void fun(int x)
        {
            cout << "Derived::fun(), x = " << x << endl;
        }
    };
// Driver Code
int main()
{
    Derived d1; // Constructor

    // Base class pointer which will
    // Edit value in memory location of
    // Derived class constructor
    Base* bp = &d1;

    bp->fun(); // Calling a derived class member function

    return 0; // Returning 0 means the program
            // Executed successfully
}
```

- Output Derived::fun(), x = 0
- If we take a closer look at the output, we observe that the '*fun()*' function of the derived class is called, and the default value of the base class '*fun()*' function is used.
- Default arguments do not participate in the signature(function's name, type, and order) of functions. So signatures of the '*fun()*' function in the base class and derived class are considered the same, hence the '*fun()*' function of the base class is **overridden**. Also, the default value is used at compile time. When the compiler sees an argument missing in a function call, it substitutes the default value given. Therefore, in the above program, the value of **x** is substituted at compile-time, and at run-time derived class's '*fun()*' function is called.
- **Can virtual functions be inlined?**

- Whenever a virtual function is called using a base class reference or pointer it cannot be inlined because the call is resolved at runtime, but whenever called using the object (without reference or pointer) of that class, can be inlined because the compiler knows the exact class of the object at compile time.

Virtual Destructor

- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.
- Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
- As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Pure Virtual Functions

- A **pure virtual function** (or abstract function) in C++ is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class.
- A pure virtual function is declared by assigning 0 in the declaration.

```
// An abstract class
class Test {
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;
    /* Other members */
};
```

- The class which has at least one pure virtual function that is called an "**abstract class**". We can never instantiate the abstract class i.e. we cannot create an object of the abstract class.
- This is because we know that an entry is made for every virtual function in the VTABLE (virtual table). But in case of a pure virtual function, this entry is without any address thus rendering it incomplete. So the compiler doesn't allow creating an object for the class with incomplete VTABLE entry.

Abstract Classes in C++

- Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **abstract class**.

- For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw().
- Similarly, an Animal class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move.
- We cannot create objects of abstract classes.

```
class Test {
   // private member variable
   int x;

public:
   // pure virtual function
   virtual void show() = 0;

   // getter function to access x
   int getX() { return x; }
};
int main(void)
{
   // Error: Cannot instantiate an abstract class
   Test t;

   return 0;
}
```

- **A class is abstract if it has at least one pure virtual function.**
- **We can have pointers and references of abstract class type.**
- **If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.**
- **An abstract class can have constructors.**
- An abstract class can have some implementation like properties and methods along with pure virtual functions.

```
class Base {
public:
   // pure virtual function
   virtual void show() = 0;
};
class Derived : public Base {
};
int main(void)
{
   // creating an object of Derived class
   Derived d;
   return 0;
}
```

Compiler Error: cannot declare variable 'd' to be of abstract type

Interfaces

- **Interfaces** are nothing but a way to describe the behavior of a class without committing to the implementation of the class.
- In C++ programming there is no built-in concept of interfaces
- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
- A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration
- **Importance of Interfaces**
- Any class derived from the pure abstract class (Interface) must implement all of the methods of the base class i.e. Interface.
- Interface pointers can be passed to functions and classes thereby we can call the functions of the derived class from there itself.

Type Casting

- Casting is a technique by which one data type to another data type. The operator used for this purpose is known as the **cast operator**. It is a **unary operator** which forces one data type to be converted into another data type.

```
int a, b;
```

```
    float c;
    a = 7;
    b = 2;
    // Typecasting
    c = (float)a / b;
```

- **C++ supports four types of casting:**

Static Cast
Dynamic Cast
Const Cast
Reinterpret Cast

Static Cast

- This is the simplest type of cast that can be used. It is a compile-time cast. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.
- static_cast <dest_type> (source);
- The return value of static_cast will be of **dest_type.**

```
int main()
{
    float f = 3.5;

    // Implicit type case
    // float to int
    int a = f;
    cout << "The Value of a: " << a;

    // using static_cast for float to int
    int b = static_cast<int>(f);
    cout << "\nThe Value of b: " << b;
}
```

Dynamic Cast

- A cast is an operator that converts data from one type to another type.

- In C++, dynamic casting is mainly used for safe downcasting at run time.
- To work on **dynamic_cast** there must be one virtual function in the base class.
- A **dynamic_cast** works only polymorphic base class because it uses this information to decide safe downcasting.
- dynamic_cast <new_type>(Expression)
- **Downcasting:** Casting a base class pointer (or reference) to a derived class pointer (or reference) is known as downcasting. In figure 1 casting from the Base class pointer/reference to the “derived class 1” pointer/reference showing downcasting (Base ->Derived class).
- **Upcasting:** Casting a derived class pointer (or reference) to a base class pointer (or reference) is known as upcasting. In figure 1 Casting from Derived class 2 pointer/reference to the “Base class” pointer/reference showing Upcasting (Derived class 2 -> Base Class).
- As we mention above for dynamic casting there must be one Virtual function. Suppose If we do not use the virtual function, then what is the result?
- In that case, it generates an error message “Source type is not polymorphic”.

const_cast

- The **const_cast** operator is used to modify the const or volatile qualifier of a variable.
- It allows programmers to temporarily remove the constancy of an object and make modifications.
- Caution must be exercised when using const_cast, as modifying a const object can lead to undefined behavior.
- const_cast <new_type> (expression);

reinterpret_cast

- The reinterpret_cast operator is used to convert the pointer to any other type of pointer. It does not perform any check whether the pointer converted is of the same type or not.
- reinterpret_cast <new_type> (expression);

```
int number = 10;
  // Store the address of number in numberPointer
  int* numberPointer = &number;
  // Reinterpreting the pointer as a char pointer
  char* charPointer
    = reinterpret_cast<char*>(numberPointer);
  // Printing the memory addresses and values
```

```
cout << "Integer Address: " << numberPointer << endl;
cout << "Char Address: "
    << reinterpret_cast<void*>(charPointer) << endl;
```

- In the example, we have defined an int variable **'number'** and then store the address of 'number' in **'numberPointer'** of the int type after that we have converted the **'numberPointer'** of the int type into char pointer and then store it into **'charPointer'** variable. To verify that we have printed the address of both numberPointer and charPointer. To print the address stored in 'charPointer' **reinterpret_cast<void*>** is used to bypass the type-checking mechanism of C++ and allow the pointer to be printed as a generic memory address without any type-specific interpretation.
- ***Note:*** *const_cast and reinterpret_cast are generally not reccommended as they vulnerable to different kinds of errors.*

Exception Handling in C++

- In C++, exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution.
- The process of handling these exceptions is called exception handling.
- Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.
- So basically using exception handling in C++, we can handle the exceptions so that our program keeps running.

What is a C++ Exception?

- An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).
- **Types of C++ Exception**

1. **Synchronous:** Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as dividing a number by zero.
2. **Asynchronous**: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.
3. In C++, exception is an event or object which is thrown at runtime.
4. All exceptions are derived from std::exception class.
5. It is a runtime error which can be handled.
6. If we don't handle the exception, it prints exception message and terminates the program.

7. In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:

C++ try and catch

- C++ provides an inbuilt feature for Exception Handling. It can be done using the following specialized keywords: try, catch, and throw with each having a different purpose.

```
try {
   // Code that might throw an exception
   throw SomeExceptionType("Error message");
}
catch( ExceptionName e1 )  {
   // catch block catches the exception that is thrown from try block
}
```

- The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.
- The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.
- An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.
- Note: Multiple catch statements can be used to catch different type of exceptions thrown by try block.
- The try and catch keywords come in pairs: We use the try block to test some code and If the code throws an exception we will handle it in our catch block.
- **Why do we need Exception Handling in C++?**
- The following are the main advantages of exception handling over traditional error handling:
- **Separation of Error Handling Code from Normal Code***:* There are always if-else conditions to handle errors in traditional error handling codes. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.
- **Functions/Methods can handle only the exceptions they choose***:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller.

In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

- **Grouping of Error Types***:* In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, and categorize them according to their types.

Properties of Exception Handling in C++

- **Property 1 -** ***There is a special catch block called the 'catch-all' block, written as catch(…), that can be used to catch all types of exceptions.***
- **Property 2 -** ***Implicit type conversion doesn't happen for primitive types.***
- **Property 3 -** ***If an exception is thrown and not caught anywhere, the program terminates abnormally.***
- ***Note****: A derived class exception should be caught before a base class exception.*
- Like Java, the C++ library has a standard exception class which is the base class for all standard exceptions. All objects thrown by the components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type.
- **Property 4 -** ***In C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not. So, it is not necessary to specify all uncaught exceptions in a function declaration. However,exception-handling it's a recommended practice to do so.***

Limitations of Exception Handling in C++

- Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
- If exception handling is not done properly can lead to resource leaks as well.
- It's hard to learn how to write Exception code that is safe.
- There is no C++ standard on how to use exception handling, hence many variations in exception-handling practices exist.
- Exception handling in C++ is used to handle unexpected happening using "try" and "catch" blocks to manage the problem efficiently. This exception handling makes our programs more reliable as errors at runtime can be handled separately and it also helps prevent the program from crashing and abrupt termination of the program when error is encountered.