Virtual Function

- In case of upcasting, if we want to call function, depending on type of object rather than type of pointer then we should declare function in base class virtual.
- If class contains, at least one virtual function then such class is called polymorphic class.
- If signature of base class and derived class member function is same and if function in base class is virtual then derived class member function is by default considered as virtual.
- If base class is polymorphic then derived class is also considered as polymorphic.
- Process of redefining, virtual function of base class, inside derived class, with same signature, is called function overriding.
- Rules for function overriding
    1. Function must be exist inside base class and derived class(different scope)
    2. Signature of base class and derived class member function must be same(including return type).
    3. At least, Function in base class must be virtual.
- Virtual function, redefined in derived class is called overriden function.
- Definition 1: In case of upcasting, a member function, which gets called depending on type of object rather than type of pointer, is called virtual function.
- Definition 2: In case of upcasting, a member function of derived class which is designed to call using pointer of base class is called virtual function.
- We can call virtual function on object but it is designed to call on Base class pointer or reference.

Early Binding And Late Binding

- If Call to the function gets resolved at compile time then it is called early binding.
- If Call to the function gets resolved at run time then it is called late binding.
- If we call any virtual/non virtual function on object then it is considered as early binding.
- If we call any non virtual function on pointer/reference then it is considered as early binding.
- If we call any virtual function on pointer/reference then it is considered as late binding.

v-ptr and v-table

- Size of object = size of all the non static data members declare in base class and derived class + 2/4/8 bytes(if Base/Derived class contains at least one virtual function).
- If we declare member function virtual then to store its address compiler implicitly create one table(array/structure). It is called virtual function table/vf-table/v-table.
- In other words, virtual function table is array of virtual function pointers.
- Compiler generates V-Table per class.
- To store address of virtual function table, compiler implicitly declare one pointer as a data member inside class. It is called virtual function pointer / vf-ptr / v-ptr.
- v-ptr get space once per object.
- ANSI has not defined any specification/rule on position of v-ptr hence compiler vendors are free to decide its position in object. But generally it gets space at the start of the object.
- The vptr is managed by the compiler and is automatically set up during object construction. It is not something that you need to initialize or manage explicitly in your code. It's a mechanism provided by the compiler to enable polymorphic behavior and dynamic dispatch of virtual function calls.
- V-Table and V-Ptr inherit into derived class.

- Process of calling member function of derived class using pointer/reference of base class is called Runtime Polymorphism.
- According to client's requirement, if implementation of Base class member function is logically 100% complete then we should declare Base class member function non virtual.
- According to client's requirement, if implementation of Base class member function is logically incomplete / partially complete then we should declare Base class member function virtual.
- According to client's requirement, if implementation of Base class member function is logically 100% incomplete then we should declare Base class member function pure virtual.

Pure Virtual Function:

- If we equate, virtual function to zero then such virtual function is called pure virtual function.
- We can not provide body to the pure virtual function.
- If class contains at least one pure virtual function then such class is called abstract class.
- If class contains all pure virtual functions then such class is called pure abstract class/interface.

```
//Pure Abstract class or Interface
class A
{
    public:
    virtual void f1( void ) = 0;
    virtual void f2( void ) = 0;
};

//Pure abstract class / Interface
class B : public A
    //Interface Inheritance
    {
    public:
    virtual void f3( void ) = 0;
};
```

- We can instantiate concrete class but we can not instantiate abstract class and interface.
- We can not instantiate abstract class but we can create pointer/reference of it.
- If we extend abstract class then it is mandatory to override pure virtual function in derived class otherwise derived class can be considered as abstract.
- Abstract class can contain, constructor as well as destructor.
- An ability of different types of object to use same interface to perfrom different operation is called Runtime Polymorphism.

Runtime Type Information/Identification[RTTI]

- It is the process of finding type(data type/ class name) of object/variable at runtime.
- It is in standard C++ Header file(/usr/include). It contains declaration of std namespace. std namespace contains declaration of type_info class.
- Since copy constructor and assignment operator function of type_info class is private we can not create copy of it in our program.
- If we want to use RTTI then we must use typeid operator.

- typeid operator return reference of constant object of type_info class.
- To get type name we should call name() member function on type_info class object.

```
#include<iostream>
#include<string>
#include<typeinfo>
using namespace std;
int main( void )
{
float number = 10;
const type_info &type = typeid( number );
string typeName = type.name();
cout<<"Type Name : "<<typeName<<endl;
return 0;
}
```

- In case of upcasting, if we want to find out type of object then we should use RTTI.
- In case of upcasting, if we want to find out true type of object then base class must be polymorphic.
- Using NULL pointer, if we try to find out true type of object then typeid throws std::bad_typeid exception.

Virtual Destructor

- A destructor is implicitly invoked when an object of a class goes out of scope or the object's scope ends to free up the memory occupied by that object.
- Due to early binding, when the object pointer of the Base class is deleted, which was pointing to the object of the Derived class then, only the destructor of the base class is invoked
- It does not invoke the destructor of the derived class, which leads to the problem of memory leak in our program and hence can result 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.
- to make a virtual destrictor use virtual keyword preceded by a tilde(~) sign and destructor name inside the parent class.
- It ensures that first the child class's destructor should be invoked and then the destructor of the parent class is called.
- Note: There is no concept of virtual constructors in C++.

Advanced Typecasting Operators:

1. dynamic_cast
2. static_cast
3. const_cast
4. reinterpret_cast

1. dynamic_cast operator

- In case of polymorphic type, if we want to do downcasting then we should use dynamic_cast operator.

- dynamic_cast operator check type conversion as well as inheritance relationship between type of source and destination at runtime.
- In case of pointer if, dynamic_cast operator fail to do downcasting then it returns NULL.
- In case of reference, if dynamic_cast operator fail to do downcasting then it throws std::bad_cast exception.

2. static_cast operator

- If we want to do type conversion between compatible types then we should use static_cast operator.
- In case of non polymorphic type, if we want to do downcasting then we should use static_cast operator.
- In case of upcasting, if we want to access non overriden members of Derived class then we should do downcasting.
- static_cast operator do not check whether type conversion is valid or invalid. It only checks inheritance between type of source and destination at compile time.
- Risky conversion not be used, should only be used in performance-critical code when you are certain it will work correctly.
- The static_cast operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe.

```
int main( void )
{
double num1 = 10.5;
//int num2 = ( int )num1;
int num2 = static_cast<int>( num1 );
cout<<"Num2:"<<num2<<endl;
return 0;
}
```

3. const_cast operator

- Using constant object, we can call only constant member function.
- Using non constant object, we can call constant as well as non constant member function.
- If we want convert pointer to constant object into pointer to non constant object or reference to constant object into reference to non constant object then we should use const_cast operator.
- Used to remove the const, volatile, and __unaligned attributes.
- const_cast<class *> (this)->membername = value;

4. reinterpret_cast operator.

- If we want to convert pointer of any type into pointer of any other type then we should use reinterpret_cast operator.
- The reinterpret_cast operator can be used for conversions such as char* to int*, or One_class* to Unrelated_class*, which are inherently unsafe.

Exception Handling

- Following are the operating system resources that we can use in application development

    1. Memory
    2. File
    3. Thread
    4. Socket
    5. Network connection
    6. IO Devices etc.
- Since OS resources are limited, we should use it carefully.
- If we make syntactical mistake in a program then compiler generates error.
- Without definition, if we try to access any member then linker generates error.
- Logical error / syntacticaly valid but logicaly invalid statements represents bug.
- If we give wrong input to the application then it generates runtime error/exception.
- Exception is an object, which is used to send notification to the end user of the system if any exceptional situation occurs in the program.
- If we want to manage OS resources carefully then we should use exception handling mechanism.
- Need of exception Handling:
    1. To avoid resource leakage.
    2. To handle all the runtime errors(exeption) centrally.
- If we want to handle exception then we should use 3 keywords:
    1. try
    2. catch
    3. throw

1. try:

- try is keyword in C++.
- If we want to inspect exception then we should put statements inside try block/handler.
- try block must have at least one catch block/handler

2. throw:

- throw is keyword in C++.
- If we want to generate exception explicitly then we should use throw keyword.
- "throw statement" is a jump statement.

3. catch:

- If we want to handle exception then we should use catch block/handler.
- Single try block may have multiple catch block.
- Catch block can handle exception thrown from try block only.
- With the help of function, we can throw exception from outside try block.
- For thrown exception, if we do not provide matching catch block then C++ runtime gives call the std::terminate function which implicitly give call the std::abort function.
- A catch block, which can handle any type of exception is called generic catch block / catch-all handler.
- Generic catch block must appear after all specific catch block.

```
try
{
```

```
}
catch(...)
{
}
```

Nested Exception Handling

- We can write try catch block inside another try block as well as catch block. It is called nested try catch block.
- Outer catch block can handle excpetion's thrown from inner try block.
- Inner catch block, cannot handle exception thrown from outer try block.
- If information, that is required to handle exception is incomplete inside inner catch block then we can rethrow that exception to the outer catch block.

```
class ArithmeticException{
private:
    string message;
public:

    ArithmeticException( string message ) : message( message ){}
    void printStackTrace( void )const{
        cout<<this->message<<endl;
    }
};
int main( void ){
    try{
        try{
            throw ArithmeticException("/ by zero");
        }
        catch( ArithmeticException &ex)
        {
            cout<<"Inside inner catch"<<endl;
            throw; //throw ex;
        }
    }
    catch( ArithmeticException &ex){
       cout<<"Inside outer catch"<<endl;
    }
    catch(...){
        cout<<"Inside generic catch block"<<endl;
    }
    return 0;
}
```

Stack Unwinding

- During execution of function if any exception occurs then process of destroying FAR and returning control back to the calling function is called stack unwinding.
- During stack unwinding, destructor gets called on local objects(not on dynamic objects).

Template

- If we want to write generic program in C++ then we should use template.
- Using template we can not reduce code size or execution time but we can reduce developers effort.
- It is designed for implementing generic data structure and algorithms
- Types of template:
    1. Function Template
    2. Class Template

1. Function Template

```
//template<typename T>//T : Type Parameter
template<class T> //T : Type Parameter
void swap_number( T &o1, T &o2 )
{
    T temp = o1;
    o1 = o2;
    o2 = temp;
}
int main( void )
{
    int num1 = 10;
    int num2 = 20;
    swap_number<int>( num1, num2 );
    //Here int is type argument
    cout<<"Num1 : "<<num1<<endl;
    cout<<"Num2 : "<<num2<<endl;
    return 0;
}
```

- Type inference : It is ability of compiler to detect type of argument at compile time and passing it as a argument to the function.

```
template<class X, class Y>
void swap_number( X &o1, Y &o2 )
{
    X temp = o1;
    o1 = o2;
    o2 = temp;
}
int main( void )
{
    float num1 = 10.5f;
    double num2 = 20.5;
    swap_number<float, double>(num1,num2 );
    cout<<"Num1 : "<<num1<<endl;
    cout<<"Num2 : "<<num2<<endl;
    return 0;
}
```

- We can pass multiple type arguments to the function.
- Using template argument list, we can pass data type as a argument to the function.
- Using template we can write type safe generic code.

2. Class Template

- In C++, by passing data type as a argument, we can write generic code hence parameterized type is called template.

```
template<class T>
class Array // Parameterized type
{
    private:
    int size;
    T *arr;
    public:
    Array( void ) : size( 0 ), arr( NULL )
    {
    }
    Array( int size )
    {
    this->size = size;
    this->arr = new T[ this->size ];
    }
    void acceptRecord( void ){
    }
    void printRecord( void ){
    }
    ~Array( void ){ }
};
int main( void )
{
    Array<char> a1( 3 );
    a1.acceptRecord();
    a1.printRecord();
    return 0;
}
```

Friend function & class

- If we want to access private members inside derived class
- Either we should use member function(getter/setter).
- Or we should declare a facilitator function as a friend function.
- Or we should declare derived class as a friend inside base class.
- Friend function is non-member function of the class, that can access/modify the private members of the class.
- It can be a global function.
- Or member function of another class.

- Friend functions are mostly used in operator overloading.
- If class C1 is declared as friend of class C2, all members of class C1 can access private members of C2.
- Friend classes are mostly used to implement data struct like linked lists.