Agenda

- Runtime Polymorphism
- Virtual Functions
- vptr and vtable
- RTTI
- ~~Advanced Casting Operators~~
- ~~Virtual Destructor~~

RunTime Polymorphism

- During inheritance, members of base class inherit into derived class hence using derived class object, we can access members of base class as well as derived class.
- Members of derived class do not inherit into the base class hence using base class object we can access members of base class only.
- Members of base class inherit into derived class hence derived class object can be considered as base class object.
- Example : Employee object is-a Person object.
- Since Derived class object can be considered as Base class object, we can use it in place of Base class object.

```
Base b1;
Base b2 = b1;//OK
Derived d1;
b1 = d1;//OK
```

```
Base *ptr = NULL;
ptr = new Base(); // OK
ptr = new Derived(); // OK
```

- If we assign derived class object to the base class object then compiler copies state of base class portion from derived class object into base class object. It is called `Object slicing.`
- During Object slicing, mode of inheritance must be public.

```
class Base{
    public:
    int n1,n2;
    void printBase(){
        cout<<num1<<","<<num2<<endl;
    }
}
class Derived:public Base(){
    public:
    int n3;
    Derived(int n1,int n2,int n3){
```

SUNBEAM INFOTECH

```
        this->n1=n1;
        this->n2=n2;
        this->n3=n3;
    }
}
int main( void )
{
Base base;
Derived derived( 500,600,700);
base = derived; //OK : Object Slicing
base.printRecord(); //Base::printRecord() : 500,600
return 0;
}
```

- Members of derived class do not inherit into base class. Hence base class object, can not be considered as derived class object.
- Since base class object, can not be considered as derived class object, we can not use it in place of derived class object.

```
Derived *ptr = NULL;
ptr = new Derived();//Ok

ptr = new Base();//Not Ok
```

- Process of converting, pointer of derived class into pointer of base class is called upcasting.
- Upcasting represents object slicing.
- In case of upcasting, explicit type casting is optional.
- Main purpose of upcasting is to reduce object dependancy in the code.
- Process of converting pointer of base class into pointer of derived class is called downcasting.
- In Case of downcasting, explicit typecasting is mandatory.
- Note: Only in case of upcasting, we can do downcasting. Otherwise downcasting will fail.

```
int main( void )
{
    Base *ptrBase = new Derived( ); //Upcasting
    ptrBase->printRecord(); //Base::printRecord()

    Derived *ptrDerived = ( Derived*)ptrBase;//Downcasting
    ptrDerived->printRecord();

    return 0;
}
```

Virtual Function

SUNBEAM INFOTECH

- 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.

SUNBEAM INFOTECH

- 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;
};
```

SUNBEAM INFOTECH

- 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.

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.

SUNBEAM INFOTECH