Object Oriented Programming Using C++

Ketan G Kore
ketan.kore@sunbeaminfo.com

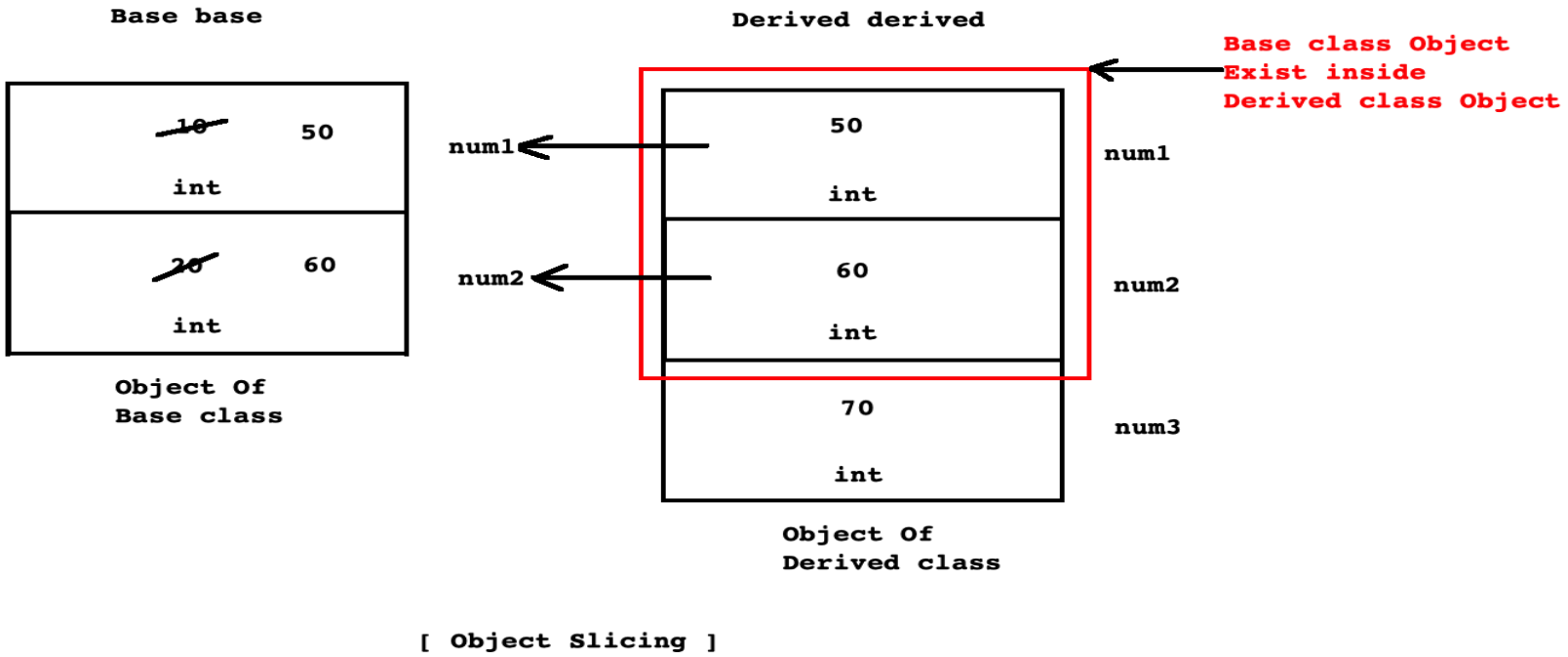
Inheritance Basics - Revision

- During inheritance, members(data member, member function, nested types) of derived class do not inherit into base class. Hence using base class object, we can access members of base class only.
- During inheritance, members(data member, member function, nested types) 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 base class inherit into derived class. Hence we can consider object of derived class as a object of base class.
- Since derived class object can be considered as base class object, we can use it in place of base class object.

Object Slicing

- If we assign, derived class object to the base class object then compiler consider only base class portion from derived class object and copy it into base class object. This process is called object slicing.
- **During object slicing, mode of inheritance must be public.**

Mode



[Object Slicing]

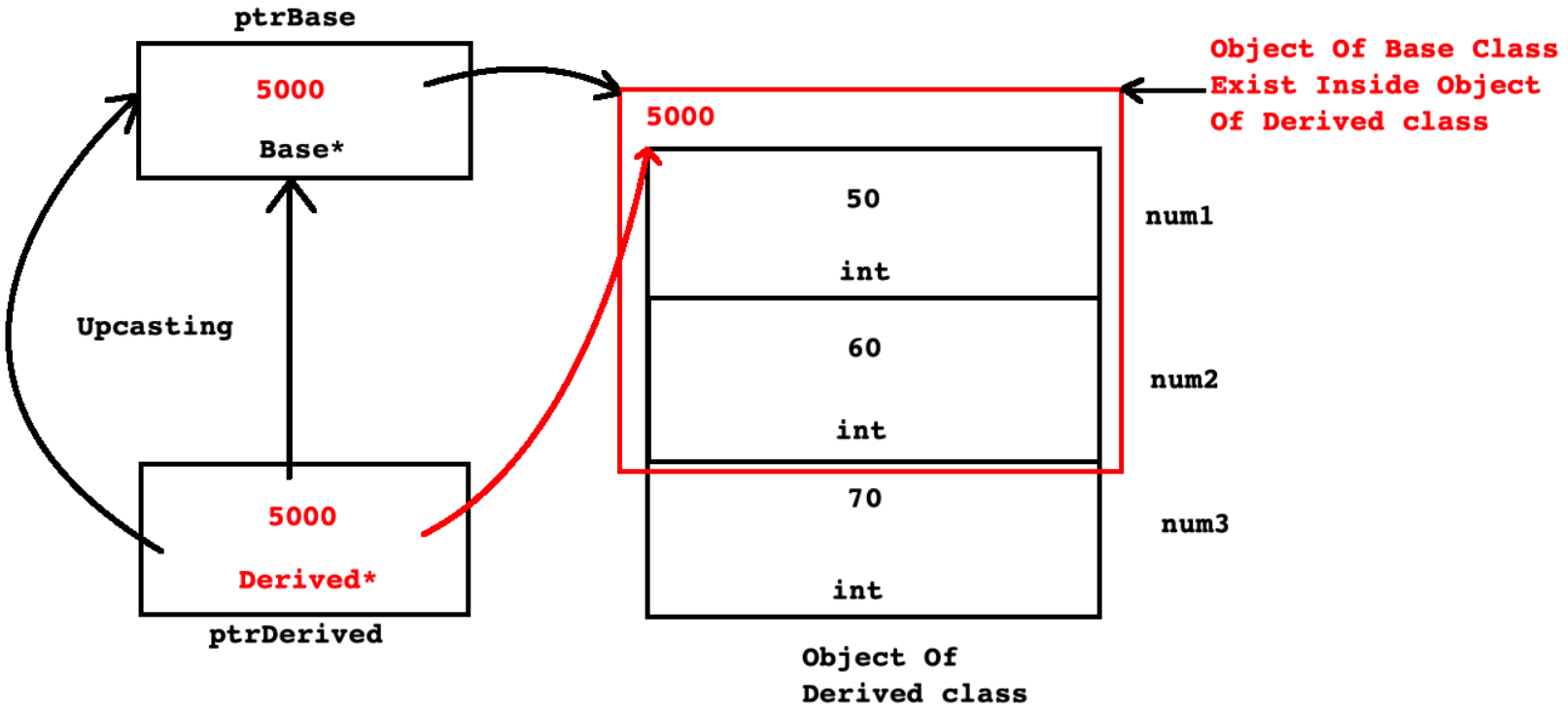
Upcasting

- Process of converting, pointer of derived class into pointer of base class is called upcasting.

```
Derived derived(50, 60, 70);
Derived *ptrDerived = &derived;
//ptrDerived->printRecord( ); //Derived::printRecord
//Base *ptrBase = ( Base* )ptrDerived;    //Upcasting
Base *ptrBase = ptrDerived;    //Upcasting
//ptrBase->printRecord( ); //Base::printRecord
```

- We can store address of derived class object into pointer of base class. It is also called as upcasting.

Upcasting



- We can convert, pointer of derived class into pointer of base class. It is called upcasting.
- Using upcasting, we can reduce object dependancy in the code. Which helps developer to reduce maintenance of the code.
- Here using ptrBase we can access:
  1. num1 and num2
  2. showRecord and printRecord of base class.
- Here using ptrDerived we can access:
  1. num1, num2 and num3
  2. showRecord, printRecord as well as displayRecord

Down casting

- Process of converting, pointer of base class into pointer of derived class is called down casting.

```
Base *ptrBase = new Derived( 50, 60, 70 );   //Upcasting
ptrBase->printRecord( );     //Base::printRecord
Derived *ptrDerived = ( Derived* )ptrBase;   //Downcasting
ptrDerived->printRecord( );     //Derived::printRecord
```

- During upcasting, if we want to access:
    1. Data members of derived class
    2. Non overridden function of derived class

then we should use down casting.

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.
- **Process of redefining, virtual function of base class, inside derived class with same signature is called function overriding.**
- Virtual function redefined inside derived class is called overrided function.
- For function overriding:
    1. Functions must be exist inside base class and derived class.
    2. Signature of base class and derived function must be same.
    3. At least, function in base class must be virtual.

Virtual Function

- Definition of virtual function:
  1. *In case of upcasting, a function, which gets called depending on type of object rather than type of pointer is called virtual function.*
  2. *A member function of derived class, which is designed to call using pointer of base class is called virtual function.*
- **Process of calling member function of derived class using pointer of base class is called runtime Polymorphism.**

```
if( ptr != NULL ){
    ptr->acceptRecord();     //Runtime Polymorphism
    ptr->printRecord();      //Runtime Polymorphism
    delete ptr;
}
```

- Only in case of upcasting, it is necessary to declare function in base class virtual. In other words, **virtual member functions are designed to call using base class pointer/reference.**

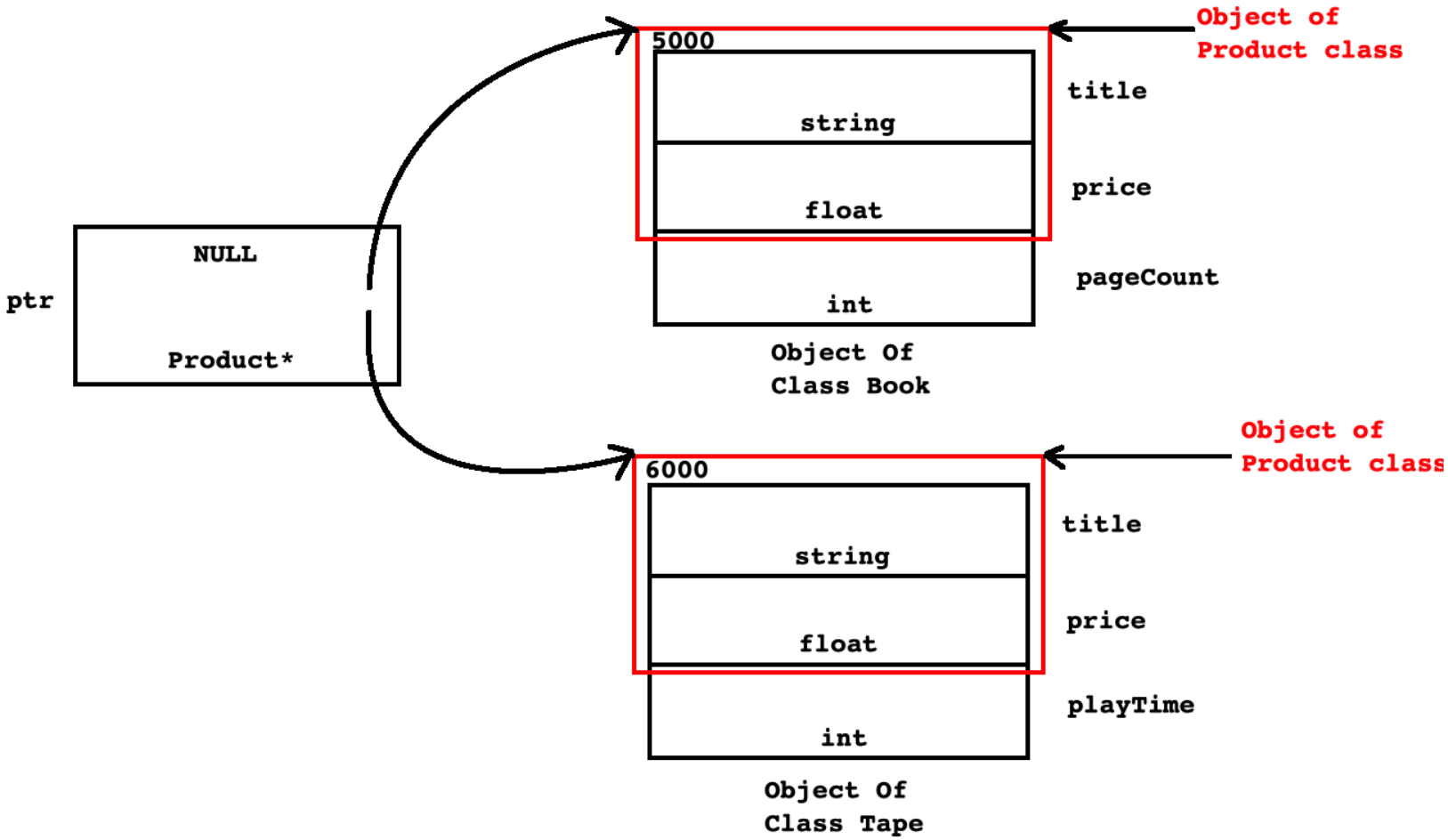
Virtual Function

- Can we declare static member function virtual?
  - Static member functions are designed to call on class name.
  - Virtual member function is designed to call on base class pointer/reference.
  - Since static member function is not designed to call on base class pointer/reference, we can not declare static member function virtual.

- Since we can not declare static member function virtual, we can not override it inside derived class.

- In case of modular approach, virtual keyword should appear in declaration part only.

Book and Tape assignment



Virtual Function

- **Why we can not declare constructor virtual?**
  - Virtual function is designed to call on base class pointer/reference
  - We can not call constructor on object/pointer/reference explicitly. It is designed to call implicitly.
  - Since we can not call constructor on pointer/reference, we can not declare it virtual.
- **Points to remember**
  1. According to problem statement / client's requirement, if implementation of base class member function is logically 100% complete(no need to override) then we should declare it non virtual.
  2. According to problem statement / client's requirement, if implementation of base class member function is logically incomplete / partially complete (need to override) then we should declare it virtual.
  3. According to problem statement / client's requirement, if implementation of base class member function is logically 100% incomplete(must override) then we should declare it pure virtual.

Pure Virtual Function

- **If we equate virtual function to zero then it is called pure virtual function.**
- **Pure virtual function do not contain body.**
- In oops, a member function, which do not contain body is called abstract method. In short, in C++, pure virtual function is also called as abstract method.

```
virtual void f1( void ) = 0;      //OK

virtual void f2( void ) = 0{     }     //NOT OK

void f3( void ) = 0{     }     //NOT OK

void f4( void ) = 0;    //NOT OK
```

Abstract Class

- **If class contains at least one pure virtual function then such class is called abstract class.**
- **If class contains all pure virtual function then such class is called pure abstract class / interface.**
- We can not create object of abstract class and interface. But we can create pointer and reference of abstract class.
- **In short, we can instantiate concrete class but we can not instantiate abstract class and interface.**

Interface Inheritance

```
class A{     //Pure abstarct class / interface
public:
    virtual void f1( ) = 0;
    virtual void f2( ) = 0;
};
//Interface inheritance
class B : public A{     //Pure abstract class / interface
public:
    virtual void f3( ) = 0;
};
```

Abstract Class

- Overriding virtual function is optional but overriding pure virtual function is mandatory.

- If we do not override pure virtual function inside derived class then derived will be considered as abstract.

- Since implementation of abstract class is logically incomplete, we can not instantiate abstract class.

RTTI

- type_info is a class declared in std namespace and std namespace is declared in <typeinfo> header file.
- **RTTI is a process of getting information(data type name) of any object at runtime.**
- If we want to use RTTI in a program then we should use typeid operator.
- typeid operator returns reference of constant object of type_info class which contains type information.

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

Thank you