POLYMORPHISM AND DYNAMIC BINDING

CHAPTER 7

KABIN DEVKOTA

UDITA BISTA

TABLE OF CONTENTS

01 Introduction

02 Need of Virtual Functions

03 Pointer to Derived Class

04 Array of pointers to base class

05 Pure Virtual Functions and Abstract Class

06 Virtual Destructor

07 Reinterpret_cast Operator

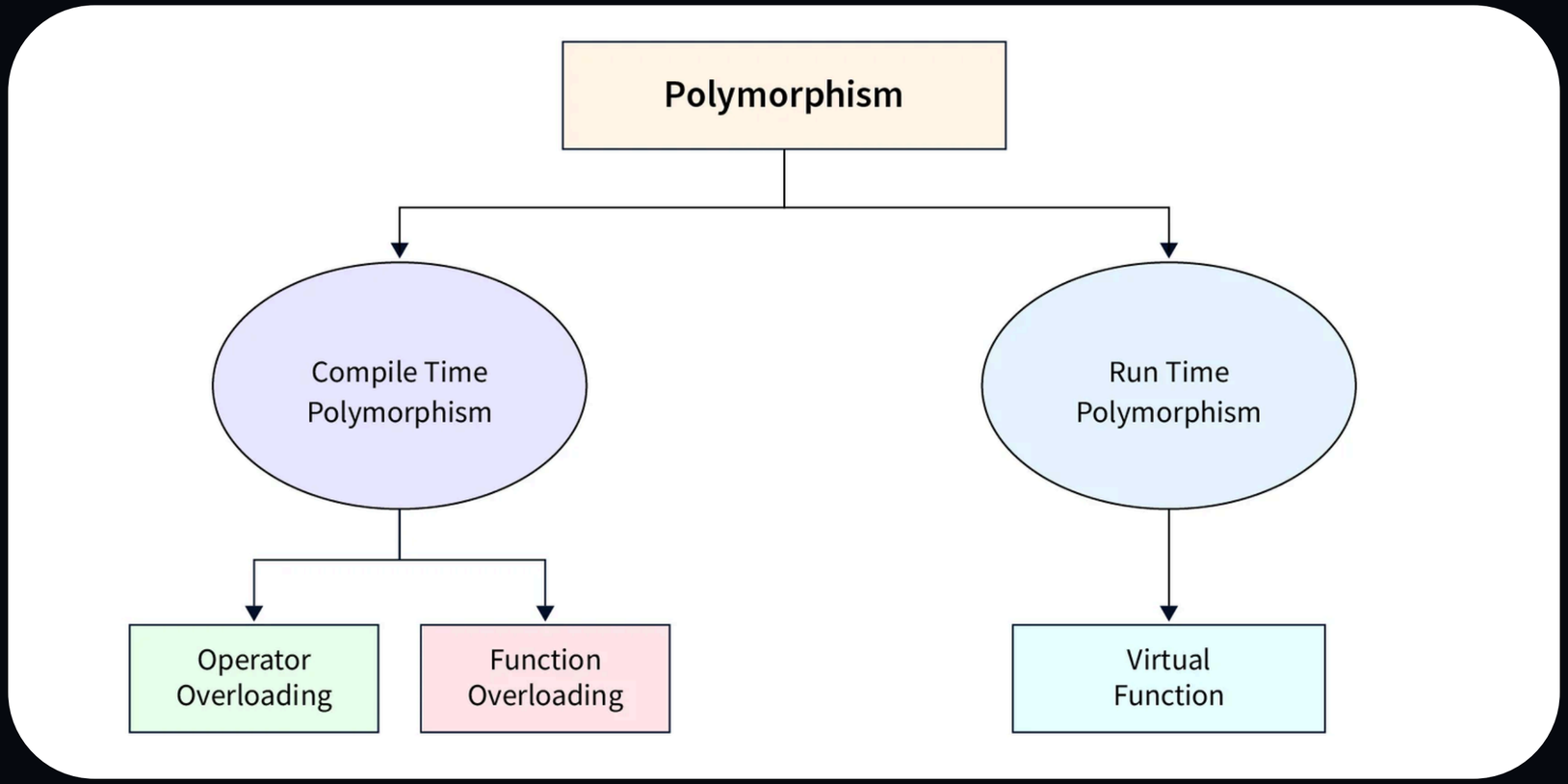
08 Run-Time Type Information(RTTI)

.....

Our content is
divided into the given topics.
Each part will be described in the slides
that follow

INTRODUCTION

Polymorphism is one of the crucial features of OOP which simply means “ one name, multiple forms”. We have already seen how the concept of polymorphism is implemented in the function overloading and operator overloading. There are two types of polymorphism namely, **compile time polymorphism** and **run-time polymorphism**.



Compile Time Polymorphism

At compile time, the compiler at the compile time knows all the matching arguments, therefore the compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early binding or static binding or static linking. Also known as compile time polymorphism. Early binding simply means that an object is bound to its function call at compile time.

Run Time Polymorphism

If appropriate member functions are chosen at run time rather than compile time, this is known as runtime polymorphism or late binding or runtime binding or dynamic binding. Dynamic binding(also known as runtime binding) means that the code associated with a given procedure call is not known until the time of the call at run-time. Since an appropriate function is called during the runtime it needs the use of a pointer by making base class pointer points at different objects even that of derived class we can execute different versions of virtual function. It is associated with polymorphism and inheritance.

NEED OF VIRTUAL FUNCTIONS

```
#include<iostream.h>
#include<conio.h>
class base
{
    public:
    void show()
    {
        cout<<"you have called base class function";
    }
};
class derived: public base
{
    public:
    void show()
    {
        cout<<"you have called derived class function";
    }
};
int main()
{
    base *b;
    derived d;
    b=&d;//base pointer to derived object
    b->show();
    getch();
    return 0;
}
```

Output:

you have called base class function

In the above program, even though the base pointer points to a derived class object, it cannot access the unique public function of the derived class. This is because **the compiler ignores the content of the pointer and chooses a member function that matches the type of pointer**. Here, *b is a pointer of base class "base" so b->show() only invokes the base class member function even though it is pointing to the derived class object d.

In order to resolve this issue, we need a virtual function. Here, A virtual function can be used to call appropriate derived class functions by using keyword virtual.

Virtual Function

A C++ virtual function is a member function in the base class and is re-defined(overridden) in a derived class. It is declared using the virtual keyword. It is used to tell the compiler to perform runtime polymorphism or dynamic linkage or late binding on the function.

Syntax:

virtual void func_name() { } OR,

void virtual func_name() { }

When there is a necessity to use the single pointer to refer to all the objects of the different classes, we create the pointer to the base class that refers to all the derived objects. But, when the base class pointer contains the address of the derived class object, it always executes the base class function.

This issue can only be resolved by using the 'virtual' function. A 'virtual' is a keyword preceding the normal declaration of a function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Virtual Function

```
#include<iostream.h>
#include<conio.h>
class Base
{
public:virtual void show()
{cout<<"you have called Base class function";}
};
class Derived1:public Base
{
public:void show()
{cout<<"you have called derived 1 class function"<<endl;}
};
class Derived2:public Base
{
public:void show()
{
cout<<"you have called derived 2 class function"<<endl;
}};
```

```
int main()
{
Base *b;
Derived1 d1;
Derived2 d2;
b=&d1;
b->show();
b=&d2;
b->show();
getch();
return 0;
}
```

Output:

you have called derived 1 class function
you have called drived 2 class function

Virtual Function

In the above program, base class pointer b is pointing to the objects d1 and d2 of classes Derived1 and Derived2 respectively. when b->show() is evaluated it actually refers to the show function of Base class i.e. Base::show(); but this show() function is a virtual function and it instructs the compiler to go and see another derived version of this function. According to the object pointed by the base pointer, it looks into those derived classes and calls the appropriate function. In the first case, the base pointer points to object ‘d1’ of Derived1 class so it calls show() function of Derived1 class and in second case, the base pointer points to object ‘d2’ of Derived2 class so calls the show() function of Derived2 class.

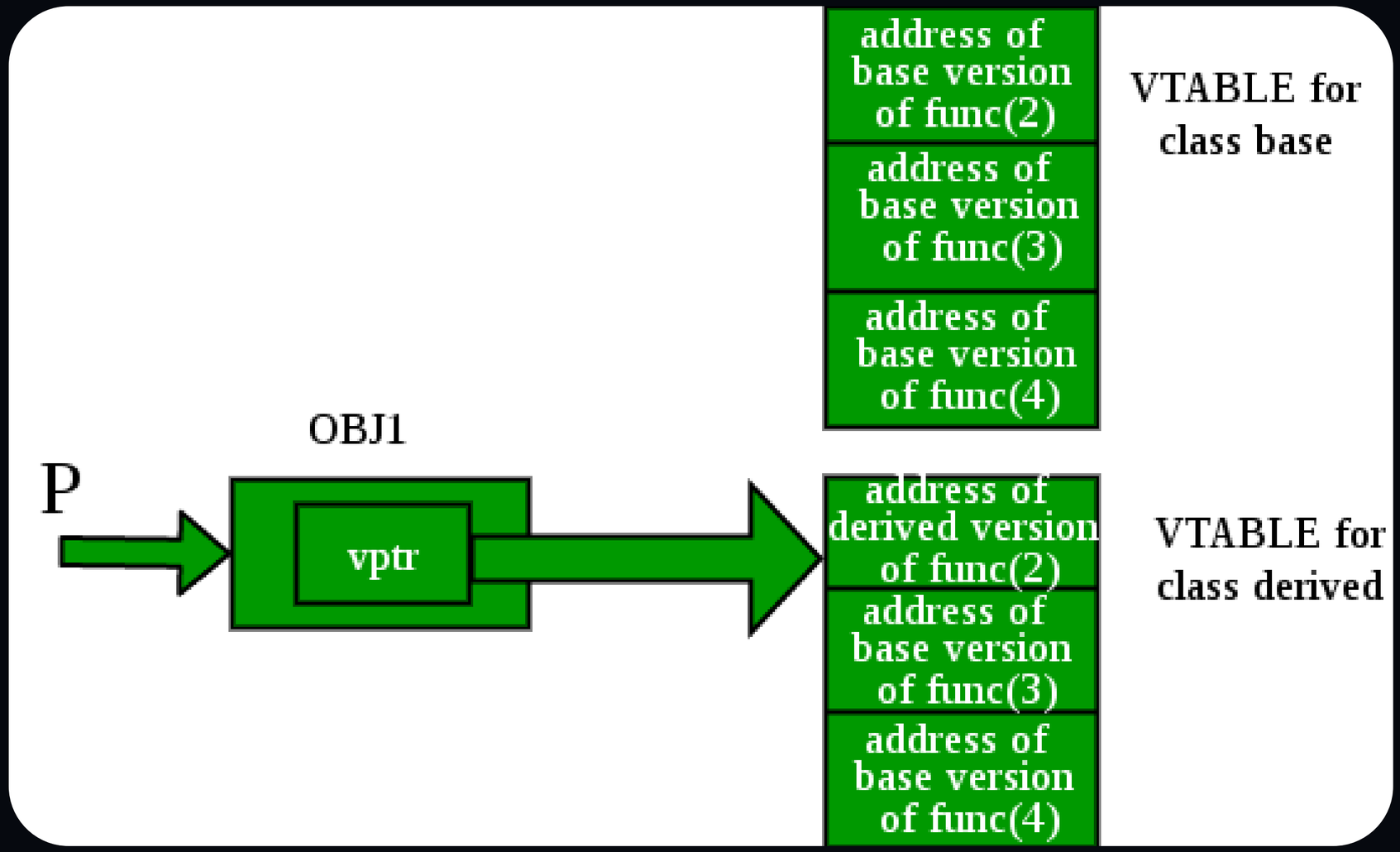
POINTER TO DERIVED CLASS

- Whenever we create an object of a class, another pointer called virtual pointer or vptr is created.
- Each object created has its unique vptr associated with it, which points to the vtable of the corresponding class.
- The vptr is also not visible to the programmer.
- From the vtable, the run-time system can find out the address of the function of the class. Thus, a link between the specific object and the address of the function is clearly established.
- When we implement the virtual functions, we first assign one of the derived class objects to the base class pointer. When this is carried out, the vptr of the derived object will in turn be linked to the base class pointer. Next, when we call the function with the base class pointer. the exact function can be linked through the respective vptr.

POINTER TO DERIVED CLASS

- The pointer to the object of base class or the derived class can be created.
- As derived classes are the type of the base class, the derived class pointer(address) is also the type of base class pointer.
- That is base class pointers are type compatible with derived class pointers, allowing derived class pointers to be used as base class pointers. So a base class pointer can hold the address of the derived class but the reverse is not true.
- The runtime polymorphism or dynamic binding in C++ is possible because the base class pointer can hold the address of its own class as well as the address of its derived class.

POINTER TO DERIVED CLASS



ARRAY OF POINTERS TO BASE CLASS

- The array of pointers to base class objects is used to store pointers to objects of different derived classes of that base class.
- The common interface function in all the classes is declared as virtual in the base class and defined as a normal function in all other derived classes.
- With virtual functions in base class when accessing the same member function through elements of base class pointer pointing to different objects, different functions related to those objects are called.
- So the base class pointer responds to the same function call differently.

ARRAY OF POINTERS TO BASE CLASS

```
#include<iostream>
using namespace std;
class CPolygon
{
    protected:
    int width, height;
    public:
    void set_values(float a, float b)
    {
        width=a;
        height=b;
    }
    virtual int area()
    {
        return 0;
    }
};
```

```
class CRectangle: public CPolygon
{
    public:
    int area()
    {
        return (width*height);
    }
};
class CTriangle: public CPolygon
{
    public:
    int area()
    {
        return ((width*height)/2.0);
    }
};
```

ARRAY OF POINTERS TO BASE CLASS

```
int main()
{
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;
    CPolygon *cpoly[] = {&poly, &trgl, &rect};
    for(int j = 0; j < 3 ; j++)
        cpoly[j]->set_values(4,5);
   cout<<”Figure drawn by base pointer are:”<<endl;
    for(int j = 0; j < 3 ; j++)
       cout<<cpoly[j]->area()<<endl;
    return 0;
}
```

Output:

Figure drawn by base pointer are:

0

10

20

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASS

A **pure virtual function** (or abstract function) in C++ is a virtual function in the base class for which there is no implementation(no body). A pure virtual function is one with an initialize of = 0 in its declaration. All the derived classes must override the base class pure virtual function and provide implementations of those functions. A class containing pure virtual functions cannot be used to declare an object of its own. such classes are called **abstract classes**. The difference between virtual function and pure virtual function is that a virtual function has an implementation and gives derived class an option of overriding the virtual functions whereas the pure virtual function doesn't provide implementation and requires the derived class to override those functions.

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

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASS

- A class containing at least one pure virtual function is known as an abstract class.
- We cannot create objects of abstract classes. But, we can create pointers for abstract classes which is required for runtime polymorphism.
- We must override the pure virtual function of an abstract class in the derived class, otherwise, the derived class will also become an abstract class.
- Normally, when creating a class hierarchy with virtual functions, in most of the cases it seems that the base class pointers are used but the objects of base class are rarely created. The concept of abstract classes seems useful in such scenarios.
- Abstract classes mostly exist to act as a parent of the derived class.

PURE VIRTUAL FUNCTIONS AND ABSTRACT CLASS

```
// C++ Program to illustrate the abstract class and
//virtual functions
#include <iostream>
using namespace std;
class Base
{
    protected:
    int x;
    public:
        virtual void sum() = 0;   // pure virtual function
        virtual void input()
        {
            cout<<“Enter the value of x: “;
            cin>>x;
         }
};
class Derived: public Base
{
    int y;
    public:
        void input() {
                Base::input();
                cout<<“Enter the value of y: “;
                cin>>y;   }
        void sum() {  //overriding pure virtual function
                cout << "The sum is: "<< x+y<<endl;  }
};
int main()
{
    Derived d;
    Base *bptr;
    bptr = &d;
    bptr -> input();
    bptr -> sum();
     return 0;
}
```

Output:

```
Enter the value of x: 5
Enter the value of y: 4
The sum is: 11
```

VIRTUAL DESTRUCTORS

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.

```
// CPP program without virtual destructor causing undefined behavior
#include <iostream>
using namespace std;
class Base {
public:
 Base()
 { cout << "Constructing base\n"; }
 ~Base()
 { cout<< "Destructing base\n"; }
};
```

```
class Derived: public Base {
public:
 Derived()
 { cout << "Constructing derived\n"; }
 ~Derived()
 { cout << "Destructing derived\n"; }
};
```

VIRTUAL DESTRUCTORS

```
int main()
{
Base * b = new Base;                    //calls Base constructor
delete b;                               //calls Base destructor
cout<<"----------------------\n";
Derived * d = new Derived;        // calls  Derived constructor
delete d;                               //calls Derived destructor
cout<<"----------------------\n";
Base * b = new Derived;           // calls Derived Destructor
delete b;                               // calls  Base Destructors
getch();
return 0;
}
```

Output:

```
Constructing base
Destructing base
----------------------------
Constructing base
Constructing derived
Destructing derived
Destructing base
----------------------------
Constructing base
Constructing derived
Destructing base
```

VIRTUAL DESTRUCTORS

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.

```
// A program with virtual destructor
#include <iostream>
using namespace std;
class Base {
public:
 Base()
 { cout << "Constructing base\n"; }
 virtual ~Base()
 { cout << "Destructing base\n"; }
};
class Derived : public Base {
public:
 Derived()
 { cout << "Constructing derived\n"; }
 ~Derived()
 { cout << "Destructing derived\n"; }
};
int main()
{
Base *b = new Derived;
delete b;
getch();
return 0;
}
```

Virtual destructors are useful when you might potentially delete an instance of a derived class through a pointer to base class. 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).

Output:

Constructing base
Constructing derived
Destructing derived
Destructing base

REINTERPRET_CAST OPERATOR

reinterpret_cast is a type of casting operator used in C++. It is used to convert a pointer of some data type into a pointer of another data type, even if the data types before and after conversion are different. It does not check if the pointer type and data pointed by the pointer is the same or not.

For example, it can be used to change pointer type objects to integer type and vice versa. It can be used for casting inherently incompatible pointer types. The general form is

data_type *var_name = reinterpret_cast <data_type *>(pointer_variable);

Here, <data_type *> specifies the target type of the case and the pointer_variable being cast into the new type. The reinterpret_cast converts the object from one type to another type without checking the bit pattern or size of source and destination type. If the number of bits of source and destination is same, then the conversion in vice versa takes place properly but if there is difference in bit then it doesn’t.

REINTERPRET_CAST OPERATOR

```
// CPP program to demonstrate working of reinterpret_cast
#include <iostream>
using namespace std;

int main()
{
 int* p = new int(65);
 char* ch = reinterpret_cast<char*>(p);
 cout << *p << endl;
 cout << *ch << endl;
 cout << p << endl;
 cout << ch << endl;
 return 0;
}
```

Output:

```
65
A
0x1609c20
A
```

RUN-TIME TYPE INFORMATION(RTTI)

In C++. it is possible to find out information about an object's type and even change the type of the object at runtime. This feature is called run-time type information.
RTTI (Run-time type information) is a mechanism that exposes information about an object's data type at runtime. It allows the type of an object to be determined during program execution.
The operators ***dynamic_cast*** and ***typeid*** provide us the run-time type information.

dynamic_cast Operator

- Usually, the runtime type information is used in situations where a variety of classes are derived from the base class.
- For dynamic_cast operator to work, the base class must be polymorphic, that is it must have a virtual function.
- The dynamic_cast is used to convert types between objects of derived class and base class.

dynamic_cast Operator

```
// C++ program to demonstrate RTTI successfully with virtual function
#include<iostream>
using namespace std;
class Base{
 public:
 virtual void display()
 {
 cout<<"This is base class."<<endl;
 }
};
class Der1: public Base{
 public:
 void display()
 {
 cout<<"This is Derived 1 class."<<endl;
 }
};
int main()
{
 Base *b;
 Der1 *d, de;
 b = &de;
 d = dynamic_cast<Der1*>(b);
 if(d!=NULL)
 {
 cout<<"Base1 converted to Der1."<<endl;
 }
 else
 {
 cout<<"Cannot convert"<<endl;
 }
  return 0;
}
```

Output:

Base1 converted to Der1

typeid Operator

typeid is an operator in C++. It is used where the dynamic type or runtime type information of an object is needed. It is included in the <typeinfo> library. Hence in order to use typeid, this library should be included in the program.

typeid(type);

OR

typeid(expression);

typeid Operator

- Parameters: typeid operator accepts a parameter, based on the syntax used in the program:
  a. type: This parameter is passed when the runtime type information of a variable or an object is needed. In this, there is no evaluation that needs to be done inside type and simply the type information is to be known.
  b. expression: This parameter is passed when the runtime type information of an expression is needed. In this, the expression is first evaluated. Then the type information of the final result is then provided.

- Return value: This operator provides the runtime type information of the specified parameter and hence that type information is returned, as a reference to an object of class type_info.

- Usage: typeid() operator is used in different ways according to the operand type.

```
#include<iostream>
#include<typeinfo>
using namespace std;
class Base
{
 public:
 virtual void display()
 {
 cout<<"This is base class."<<endl;
 }
};
class Der1: public Base
{
 public:
 void display()
 {
 cout<<"This is Derived 1 class."<<endl;
 }
};
int main()
{
 Base *b;
 Der1 *d, de;
 b = &de;
 d = &de;
 int *j;
 cout << typeid(b).name() << endl;
 cout << typeid(de).name() << endl;
 cout << typeid(d).name() << endl;
 cout << typeid(j).name() << endl;
cout<< typeid(1*3.1).name()<<endl;
 if(typeid(*b)== typeid(*d))
 cout<<"Objects of same class."<<endl;
 else
 cout<<"Objects of different class."<<endl;
 return 0;
}
```

Output:

```
Base *
Der1
Der1 *
int *
double
Objects of same class
```

THANK YOU