More Polymorphism

Tiziana Ligorio

Details

There is a lot of detail one needs to pay attention to when using Polymorphism

The following slides are for those of you who wish to dig a little deeper into the topic but will not be on exams

These are marked with 

Need to pay **extra** attention to **destructors**!!!

With Polymorphism destructor **MUST** always be **virtual**!!!

```
class BaseClass()
{
public:
    BaseClass();
    ~BaseClass();

}; //end BaseClass
```

```
class DerivedClass:
            public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();

private:
    char* myString;
}; //end DerivedClass
```

```
DerivedClass::DerivedClass()
{
    //allocate some memory
    myString = new char[128];
}

DerivedClass::~DerivedClass()
{
    //deallocate memory
    delete[] myString;
}
```

```
main()

BaseClass* myClass = new DerivedClass;
delete myClass; //PROBLEM!!!
```

BaseClass destructor is invoked.
Need to allow late binding for destructor!!!

Fix

```
class BaseClass()
{
public:
    BaseClass();
    virtual ~BaseClass();

}; //end BaseClass

class DerivedClass:
            public BaseClass
{
public:
    DerivedClass();
    ~DerivedClass();

private:
    char* myString;
}; //end DerivedClass
```

```
DerivedClass::DerivedClass()
{
    //allocate some memory
    myString = new char[128];
}

DerivedClass::~DerivedClass()
{
    //deallocate memory
    delete[] myString;
}
```

```
main()
BaseClass* myClass = new DerivedClass;
delete myClass; // both destructors
                //invoked
```

Problem fixed! BOTH destructors invoked

Virtual Functions in Constructors and Destructors

Recall

- `BaseClass` constructor invoked before `DerivedClass`'
- `DerivedClass` destructor invoked before `BaseClass`'

If virtual function in constructor/destructor is called polymorphically could try to access uninitialized/deallocated data

C++ prevents this by calling virtual functions in constructors/destructors **non-polymorphically**

```
class BaseClass()
{
public:
    BaseClass()
    {
        someVirtualFunction();
    }
    virtual void someVirtualFunction()
    {
        cout << “Base” << endl;
    }
}; //end BaseClass
```

```
class DerivedClass: public BaseClass
{
public:

    virtual void someVirtualFunction()
    {
        cout << “Derived” << endl;
    }

}; //end DerivedClass
```

```
main()

DerivedClass myDerivedClas;
```

Standard output:
Base

Invoking Virtual Members Non-Virtually

Sometimes may need to call the `BaseClass` version of a virtual function from a `DerivedClass`

```
void DerivedClass::someFunction()
{
    BaseClass::someVirtualFunction(); // no polymorphism
    //do more stuff
}
```

Copy Constructors and Assignment Operators with Inheritance

Can become complicated beasts with inheritance!!!

Must **always call explicitly** BaseClass within DerivedClass

```
class Base()
{
public:
    Base();
    Base(const Base& other);
    Base& operator=(const Base& other);
    virtual ~Base();
    //other public and protected members here that will be inherited

}; //end BaseClass
```

```
class Derived: public Base
{
public:
    Derived();
    Derived(const Derived& other);
    Derived& operator=(const Derived& other);
    virtual ~Derived();
private:
    char* theString;  //a C string
    //generic helper functions
    void copyOther(const Derived& other);
    void clear();
}; //end DerivedClass
```

Derived Implementation

```
//generic “copy other” private member function
void Derived::copyOther(const Derived& other)
{
    theString = new char[strlen(other.theString)+1];
    strcpy(theString, other.theString);
}

// clear out private member function
void Derived::clear()
{
    delete[] theString; //deallocate memory
    theString = NULL;    //avoid dangling pointer
}
```

Derived Incorrect Implementation

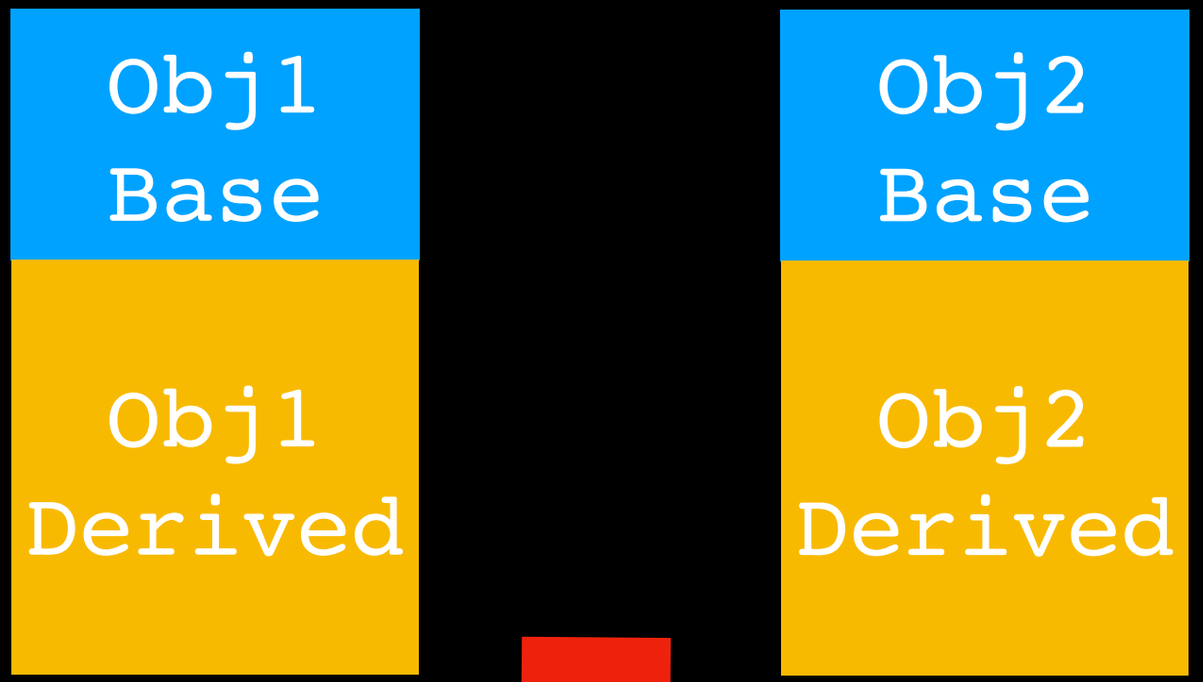
```
//copy constructor
Derived::Derived(const Derived& other)
{
    copyOther(other);
}

//assignment operator
Derived& Derived::operator=(const Derived& other)
{
    if(this != other)
    {
        clear();
        copyOther(other);
    }
    return *this;
}
```

Derived Incorrect Implementation

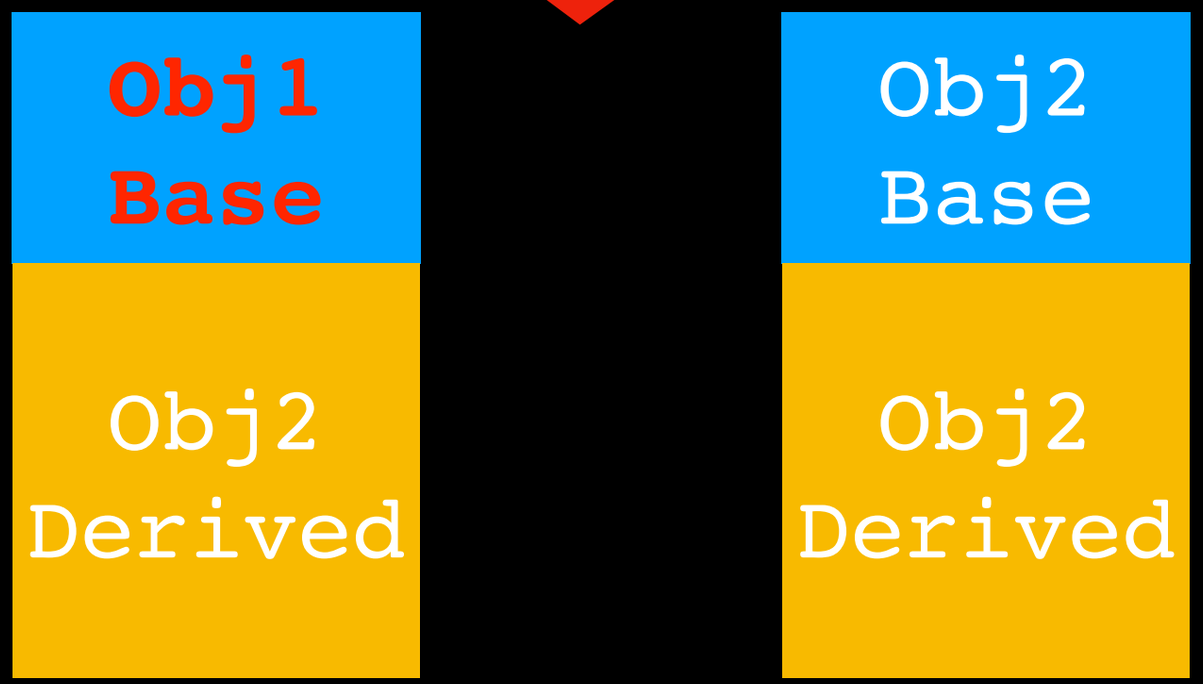
```
//copy constructor
Derived::Derived(const Derived& other)
{
    copyOther(other);                          //WRONG!!!
}

//assignment operator
Derived& Derived::operator=(const Derived& other)
{
    if(this != other)
    {
        clear();
        copyOther(other);                      //WRONG!!!
    }
    return *this;
}
```



After invoking copy constructor
or assignment operator

PROBLEM!!!



Derived Correct Implementation

```
//copy constructor
Derived::Derived(const Derived& other): Base(other) //CORRECT!!!
{
    copyOther(other);
}

//assignment operator
Derived& Derived::operator=(const Derived& other)
{
    if(this != other)
    {
        clear();
        Base::operator= (other);//CORRECT!!!Invoke Base operator=
                                //explicitly
        copyOther(other);
    }
    return *this;
}
```

Slicing

Copy ONLY `BaseClass` portion of object

Opposite of previous case

```
Base* ptr1;
Base* ptr2 = new Derived; // pointer of type Base that points to type Derived
//do stuff

*ptr1 = *ptr2; //copy value pointed to by ptr2 into variable pointed to by
                //ptr1
```

Note potential problem!!!

The above expands into

```
ptr1->operator= (*ptr2);
```

Invoking the `operator=` of the `Base` loosing all data of `Derived` portion

*ptr1
Base

*ptr1
Derived

*ptr2
Base

*ptr2
Derived

*ptr1 = *ptr2

PROBLEM!!!

*ptr2
Base

Nothing copied here

*ptr2
Base

*ptr2
Derived

Slicing via Copy Constructor

```
void doSomething(Base baseObject)
{
   //do something
}
```

```
Derived myDerived;
doSomething(myDerived);
```

PROBLEM!!! Parameter `baseObject` will be initialized using `Base` copy constructor

Slicing
Ever more insidiously!!!

```
vector<Base> myBaseVector;
Base* myBasePtr = someFunction(); //pointer to Base
//ATTENTION myBasePtr could point to Derived object
myBaseVector.push_back(*myBasePtr);
```

If `someFunction` returns a pointer to an object of type `Derived` calling `push_back` on object of type `Derived` will likely **slice** the object storing only its `Base` data

Possible solution: store pointers in myBaseVector instead of objects

Casting

Forcing one datatype to be converted into another

Up-casting (`Derived` to `Base`) automatically available through inheritance

```
Base* basePtr;
Derived* derivedPtr;
//do stuff
basePtr = derivedPtr; //automatic conversion Derived is-a Base
```

Down-casting (`Base` to `Derived`)

```
Base* basePtr = new Derived; // pointer of type Base points to
Derived
//do stuff
Derived* derivedPtr = (Derived*) basePtr;
```

Casting

Classic C++ cast too powerful => no checks.
Could write something totally nonsensical

```
Base* basePtr;
vector<double>* myVectorPtr = (vector<double>*) basePtr;
//PROBLEM!! Makes no sense, BUT no compiler error


const Base* basePtr = new Derived;
// do stuff
Derived* derivedPtr = (Derived*) basePtr;
//PROBLEM!!! Lost constness of Base object
//derivedPtr  is now free to modify it
```

static_cast

`static_cast` checks at compile time that cast "makes sense”

Allows:

- Converting between primitive types (e.g. `int` to `float`)
- Converting pointers or references of `Derived` type to pointers or references of `Base` type (e.g. `Derived*` to `Base*`) where target is at least as `const` as the source
- Converting pointers or references of `Base` type to pointers or references of `Derived` type (e.g. `Base*` to `Derived*`) where target is at least as `const` as the source

```
Base* basePtr = new Derived;
// do stuff
Derived* derivedPtr = static_cast<Derived*>(basePtr);
```

dynamic_cast

If `Base*` did not point to `Derived` object, `static_cast` would succeed

=> runtime problems

e.g. access `Derived` data members not present in `Base`

```
Base* basePtr = new Base;
Derived* derivedPtr1 = (Derived*)basePtr; //BAD!!!
Derived* derivedPtr2 = static_cast<Derived*>(basePtr); //BAD!!!
Derived* derivedPtr3 = dynamic_cast<Derived*>(basePtr); //GOOD!!!
```

Will return a NULL pointer

Conclusion

Polymorphism is easy, Just put `virtual` everywhere and the compiler will take care of the rest!

Conclusion

Polymorphism is easy, Just put `virtual` everywhere and the compiler will take care of the rest!

Real Conclusion

Overhead! Use it only when useful/necessary

Carefully craft **constructors**

Always make **destructor** **virtual**

Beware of **Slicing** (**in all its forms**)

Beware of **casting** and use level most appropriate and safe for your situation