

OOP-2

Inheritance

- Inheritance is a technique of code reuse.
- Inheritance is the 2nd pillar of OOP
- It can be described as a process of creating new classes from existing classes.
- New classes inherit some of the properties and behavior of the existing classes. An existing class that is "parent" of a new class is called a **base class**. New class that inherits properties of the base class is called a **derived class**.
- It also provides possibility to extend existing classes by creating derived classes.

Inheritance - Syntax

- The basic syntax of inheritance is:

```
struct DerivedClass : BaseClass  
{  
};
```

- DerivedClass inherits methods and members of BaseClass
- DerivedClass can also implement some new methods or data members

Inheritance with Data - Example

```
struct A {  
    A() : x(0) {}  
    int x;  
};  
  
struct B : A    // B inherits from A  
{  
    B() : y(1) {}  
    int y;  
}  
  
int main()  
{  
    B b;  
    cout << b.y;    // object b has a data member y  
    cout << b.x;    // object b has a data member x  
    return 0;  
}
```

Inheritance with Methods - Example

```
struct A {  
    void foo() const { cout << "Hello from A\n"; };  
};  
  
struct B : A { // B inherits from A  
    void bar() const  
    {  
        foo();    // B has a method foo()  
        cout << "Hello from B\n";  
    }  
};  
  
int main()  
{  
    B b;  
    b.foo();    // b has a method foo  
    b.bar();    // b has a method bar  
    return 0;  
}
```

Inheritance with Methods - Example

```
struct A {  
    void foo() const { cout << "Hello from A\n"; };  
};  
  
struct B : A { // B inherits from A  
    void foo() const // same name as base class  
    {  
        A::foo(); // B has two methods foo(), here we want A::foo  
        cout << "Hello from B\n";  
    }  
};  
  
int main()  
{  
    B b;  
    b.foo(); // B::foo hides A::foo  
    return 0;  
}
```

Inheritance Constructors

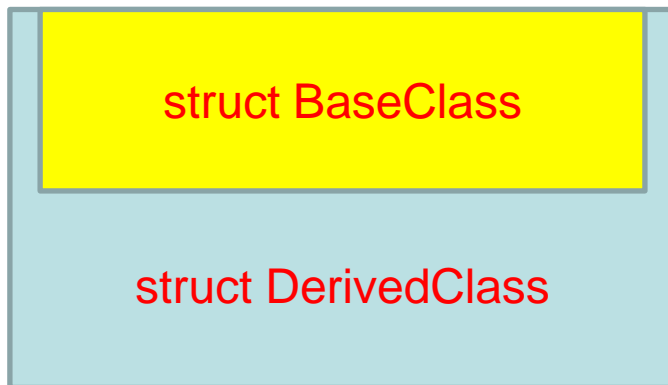
```
struct A
{
    A() : m_x(0) {}
    A(int x) : m_x(x) {}
    int m_x;
};

struct B : A    // B inherits from A
{
    B() {}                // this calls A()
    B(int y) : m_y(y) {}  // this calls A()
    B(int x, int y) : A(x), m_y(y) {} // this calls A(x)
    int m_y;
}
```

- The constructor of the base class is the first thing that is called

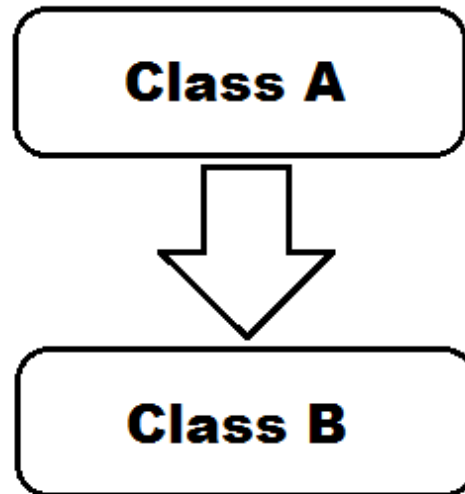
Inheritance Structure

- You can think about a derived class as built on top of the base class (i.e. it contains the base class)
- That explains why the base class needs to be constructed first



Single Inheritance

- Single inheritance represents a form of inheritance when there is only one base class and one derived class.



Single Inheritance - Example

- For example, a class describes a **Person**

```
struct Person
{
    string m_szLastName;
    int m_iYearOfBirth;

    Person(string szName, int iYear)
        : m_szLastName(szName), m_iYearOfBirth(iYear) {}

    void print()
    {
        cout << "Last name: " << szLastName << endl;
        cout << "Year of birth: " << iYearOfBirth << endl;
    }
};
```

Single Inheritance - Example

- We want to create a new class **Student** which should have the same information as the **Person** class plus one new information about university.
- In this case, we can create a derived class **Student**:

```
//derived class
struct Student: Person
{
    string m_szUniversity;
};
```

Single Inheritance - Example

- Since class Student does not have a constructor you can create one as below

```
Student(string szName, int iYear, string szUniversity)
    : Person(szName, iYear)    // we save code here
    , m_szUniversity( szUniversity )
{
}
```

- Person(szName, iYear) represents call of a constructor of the base class **Person**.
- The passing of values to the constructor of a base class is done via member initialization list.

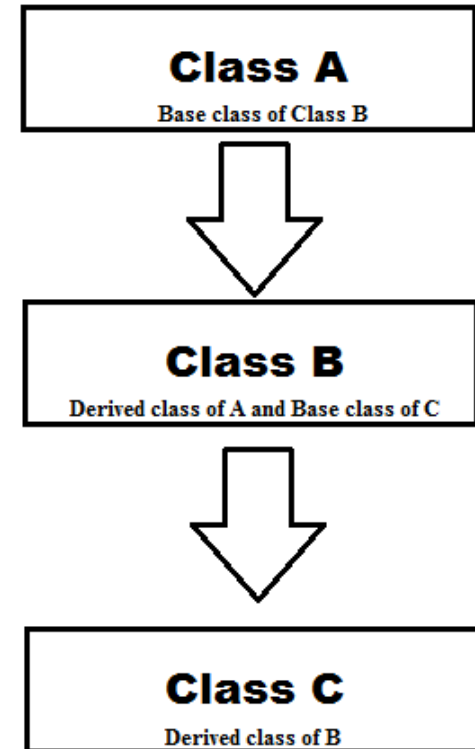
Single Inheritance - Example

- We can access member functions of a base class from a derived class. For example, we can create a new **print()** function in a derived class, that uses **print()** member function of a base class:
- If you want to call the member function of the base class which has the same name then you have to use the name of a base class

```
void print()
{
    //call function print from base class
    Person::print();    // we save code here
    cout << "University " << m_szUniversity << endl;
}
```

Multilevel Inheritance

- Multilevel inheritance represents a type of inheritance when a Derived class is a base class for another class.
- In other words, deriving a class from a derived class is known as multi-level inheritance.
- Simple multi-level inheritance is shown in below image where Class A is a parent of Class B and Class B is a parent of Class C



Multilevel Inheritance Example

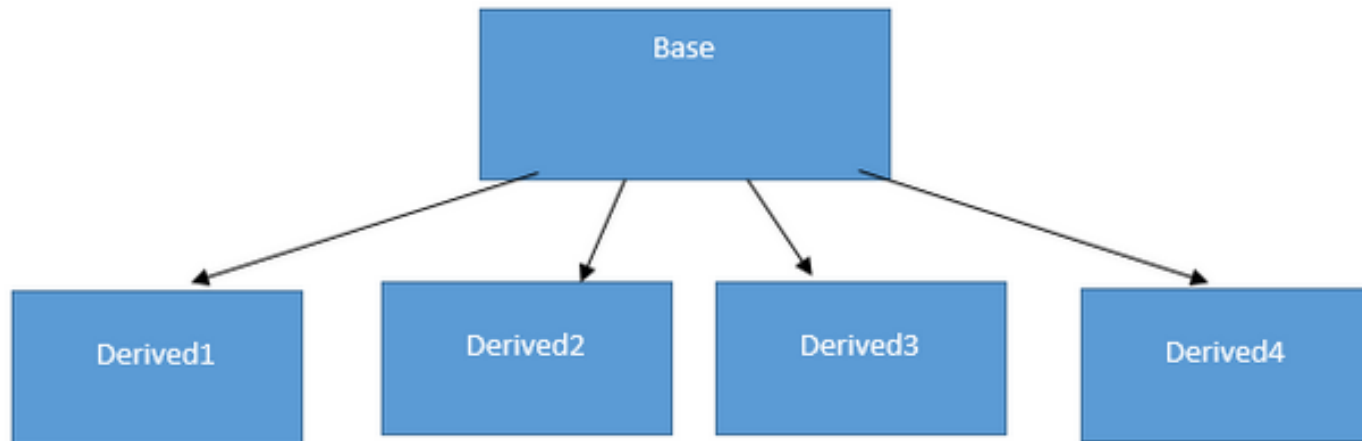
```
struct A {};    // this is a base class
```

```
struct B : A    {}; // this derives from A and is base for C
```

```
struct C : B    {}; // this derives from B
```

Hierarchical Inheritance

- When there is a need to create multiple Derived classes that inherit properties of the same Base class is known as Hierarchical inheritance



Multilevel Inheritance Example

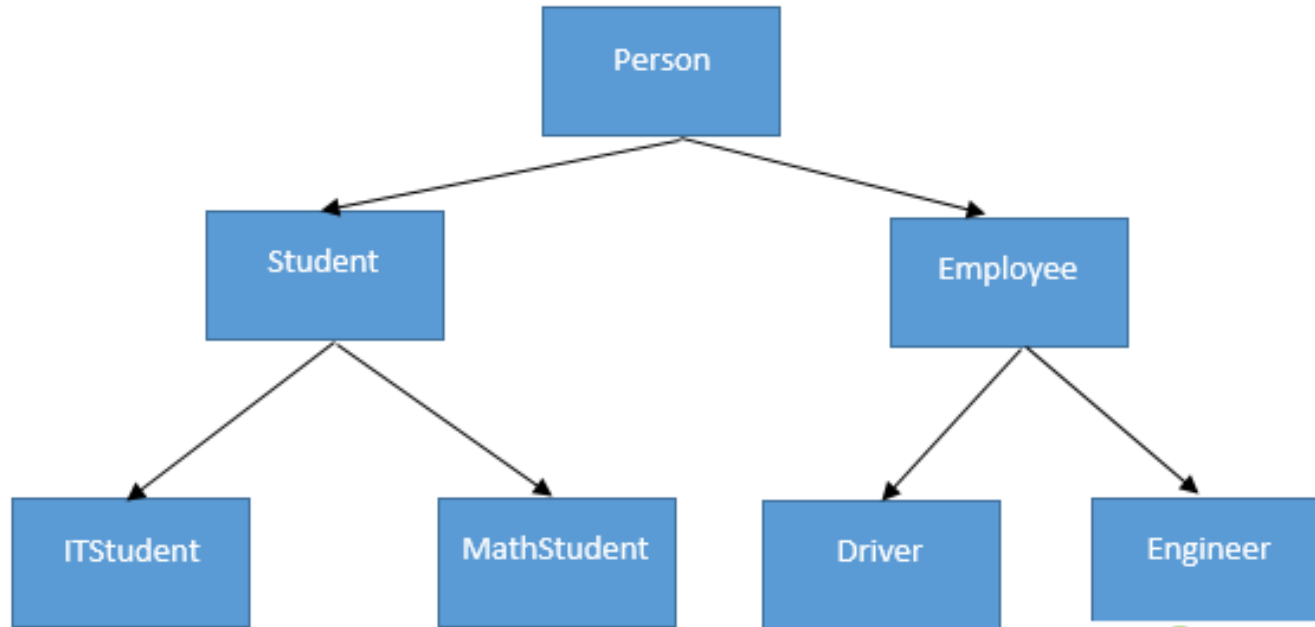
```
struct A {};    // this is a base class
```

```
struct B : A    {}; // this derives from A
```

```
struct C : A    {}; // this derives from A
```

```
struct D : A    {}; // this derives from A
```

Multilevel Hierarchical Inheritance



protected

- If in the base class something is **private**, it **cannot** be accessed in a derived class
- If in the base class something is **public**, it **can** be accessed in a derived class
- The **protected** keyword is some way in between public and private
- if in the base class something is **protected**, it **can** be accessed in a derived class, but it cannot be accessed from outside the class hierarchy

Inheritance Access Rules

- Inheritance can override access rights
- Access rights can only become stricter. E.g. something public in the base class can become private in the derived class, but something private in the base class cannot become public in the derived class

- Syntax

```
struct DerivedClass : accessRights BaseClass  
{  
}
```

- **accessRights** can be any of: public, private, protected

public inheritance

```
struct DerivedClass : public BaseClass
{
}
```

This inheritance mode is used mostly. In this the protected member of Base class becomes protected members of Derived class and public becomes public.

```
class DerivedClass : public BaseClass
```

Accessing Base class members	public	protected	private
From Base class	Yes	Yes	Yes
From object of a Base class	Yes	No	No
From Derived classes	Yes (As Public)	Yes (As Protected)	No
From object of a Derived class	Yes	No	No
From Derived class of Derived Classes	Yes (As Public)	Yes (As Protected)	No

protected inheritance

```
struct DerivedClass : protected BaseClass
{
}
```

In protected mode, the public and protected members of Base class becomes protected members of Derived class.

```
class DerivedClass : protected BaseClass
```

Accessing Base class members	public	protected	private
From Base class	Yes	Yes	Yes
From object of a Base class	Yes	No	No
From Derived classes	Yes (As Protected)	Yes (As Protected)	No
From object of a Derived class	No	No	No
From Derived class of Derived Classes	Yes (As Protected)	Yes (As Protected)	No

private inheritance

```
struct DerivedClass : private BaseClass  
{  
}
```

In private mode the public and protected members of Base class become private members of Derived class.

```
class DerivedClass : private BaseClass
```

```
class DerivedClass : BaseClass // By default inheritance is private
```

Accessing Base class members	public	protected	private
From Base class	Yes	Yes	Yes
From object of a Base class	Yes	No	No
From Derived classes	Yes (As Private)	Yes (As Private)	No
From object of a Derived class	No	No	No
From Derived class of Derived Classes	No	No	No

class vs struct

- If the inheritance accessRights is omitted
 - the default for a struct is public inheritance
 - the default for a class is private inheritance

```
struct B : A {}; // B inherits public from A
```

```
class C : A {}; // C inherits private from A
```


is-a, vs has-a

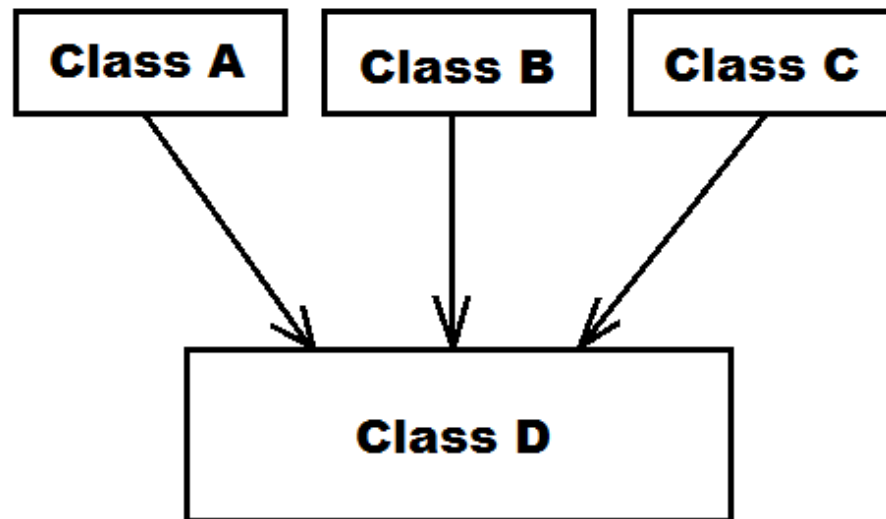
- Correct use of inheritance is to describe relations between object of type is-a.
- A common design mistake is to use it for relationship of type has-a, which should be represented via aggregation or composition
- Example:
 - A square **is a** polygon (ok to use inheritance)
 - A polygon **is a** shape (ok to use inheritance)
 - A circle **is a** shape (ok to use inheritance)
 - A cylinder **has a** circle face, but it **is not a** circle (incorrect to use inheritance)

Multiple Inheritance

- Consider the following problem
 - A Cricket is a Jumper
 - A fish is a Swimmer
 - A frog is both a Jumper and a Swimmer
- How can we solve this?

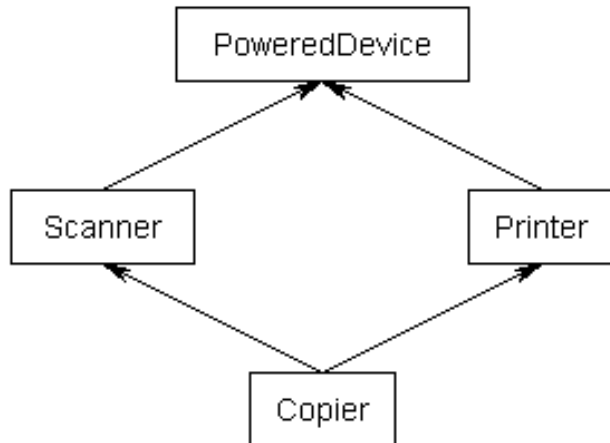
Multiple Inheritance

- Multiple inheritance represents a kind of inheritance when a derived class inherits properties of **multiple** classes. For example, there are three classes A, B and C and derived class is D as shown below



Multiple Inheritance

- Things get messy very soon and there is a big risk of overdesign
- I almost never use it. Consider the diamond problem:



- Does Copier contains twice the data of Powered Device?
- If your Copier paper jams, you might want to tell it to halt() to prevent damage. Fortunately, you don't have to write that method because it's already implemented in Printer. Unfortunately, because inheritance usually is implemented via depth-first search of the inheritance tree, you've called the PoweredDevice.halt() method, removing power to the Copier altogether, losing all of the other queued jobs. Awesome!

My Advice

- There is a high risk of overdesign and create horribly complicated hierarchies
 - Keep it simple
 - Avoid multiple inheritance
 - Avoid deep inheritance
 - Avoid override method names


Problem

- Create a struct Point with data members of type double x and y
- Overload the operator- for Point to return the distance between two point
- Create a base struct Quadrilateral with data members $p1$, $p2$, $p3$, $p4$, and methods *perimeter*, *side12*, *side23*, *side,34*, *side41*
- Create two derived structures: *Rectangle* and *Trapeze*, with an *area* method

Up Casting

- Any pointer (or reference) to a derived class can be up-casted to a pointer of the base class
- The casting is done automatically by the compiler
- After casting, only methods of the base class are accessible

Up Casting (pointer) - Example

```
struct A {  
    void foo() { cout << "Hello from A\n"; }  
};  
  
struct B : A {  
    void foo() { cout << "Hello from B\n"; }  
};  
  
int main()  
{  
    B b;                // b is an instance of type B  
    B *bp = &b;         // B* pointer   
    A *ap = bp;         // up-cast B* to A*  
    cout << bp->foo();   // calls B::foo  
    cout << ap->foo();   // calls A::foo  
    return 0;  
}
```


Up Casting (reference) - Example

```
struct A {  
    void foo() { cout << "Hello from A\n"; }  
};  
  
struct B : A {  
    void foo() { cout << "Hello from B\n"; }  
};  
  
int main()  
{  
    B b;                // b is an instance of type B  
    A& a = b;           // up-cast B& to A&  
    cout << b.foo();     // calls B::foo  
    cout << a.foo();     // calls A::foo  
    return 0;  
}
```

Polymorphism

- It allows to specialise an expected behaviour for different objects and later refer to it with no knowledge of the specific type of the object referred to
- What problem does polymorphism solves:
 - A shape is expected to implement a perimeter method
 - A circle is a shape
 - A square is a shape
 - Square and circle implement the method perimeter in different way
 - Imagine we have an **up-casted** reference (or a pointer) to an object of type Shape, we the actual instance pointed is a circle or a square (and we do not want to know), but we want to the perimeters

Polymorphism Syntax

- In the base class a method is declared as **virtual**, which means that it will be customised in derived class
- The function head needs to be identical in base class and derived class
- When the method is invoked from a reference to the base class, the program will identify at runtime (on the fly) the actual type of the object and call the correct implementation

Polymorphism - Example

```
struct Shape {  
    virtual double perimeter() const = 0;    // pure virtual method  
};  
  
struct Circle : Shape {  
    double radius;  
    Circle( double r) : radius(r) {}  
    virtual double perimeter() const { return 2*3.14*radius; }  
};  
  
struct Square : Shape {  
    double side;  
    Square(double l) : side(l) {}  
    virtual double perimeter() const { return 4*side; }  
};
```

Polymorphism - Example

```
int main()
{
    // create a vector of 2 pointers to shape and initialize it
    // with a circle and a square
    Shape *s[2] = { new Square(3), new Circle(2) };

    // we use Shape* to access the implementation of perimeter
    // in each specific class
    for (int i = 0; i < 2; ++i)
        cout << s[i]->perimeter() << endl;

    // remember we need to release the objects created with new
    for (int i = 0; i < 2; ++i)
        delete s[i];

    return 0;
}
```

- Full code **Shape.cpp**

Polymorphism - Example

```
struct Shape {  
    virtual double perimeter() const = 0;    // pure virtual method  
};
```

```
struct Circle : Shape {  
    double radius;  
    Circle( double r) : radius(r) {}  
  
    // we omit the keyword virtual  
    double perimeter() const { return 2*3.14*radius; }  
};
```

- The keyword `virtual` can be omitted in derived classes.
- In my opinion it is bad practice, as it makes harder to remember which methods are virtual and which ones are not

Polymorphism - Constructors

- virtual methods cannot be called from constructors
- Without going too technical, this is because the resolution of virtual methods relies on low level data structures initialized by the constructor
- When the constructor is still running, such structures are not available yet

Virtual Destructor

- Consider the following problem

```
struct A {};  
  
struct B : A {  
    B() : x(10, 0) {} // allocate 10 elements  
    vector<int> x;  
};
```

- The default destructor of B automatically calls the destructor of x and release the memory

```
void foo() {  
    A *a = new B{}; // Create object B and up-cast.  
    delete a;       // This calls the destructor of A but not the  
                    // one of B, hence the memory leaks  
}
```


Virtual Destructor

- The solution is to declare the destructor as **virtual** in class A

```
struct A { virtual ~A() {} }; // virtual destructor which does nothing

struct B : A {
    B() : x(10, 0) {} // allocate 10 elements
    vector<int> x;
};
```

- The default destructor of B automatically calls the destructor of x and release the memory

```
void foo() {
    A *a = new B{}; // Create object B and up-cast
    delete a;       // This calls the destructor of B
}
```

Default Behaviour

- Sometimes a certain default behaviour is good for most derived classes, but not all
- It is declared as virtual, but instead of being pure it has a body with a default implementation
- Derived classes can choose if to override the method or not
- If they do not, they get the default behaviour

Default Behaviour

```
struct Shape {  
    virtual double foo() const { cout << "I am a polygon\n"; };  
};  
  
struct Square : Shape {};  
  
struct Hexagon : Shape {};  
  
struct Circle : Shape {  
    virtual double foo() const { cout << "I am a NOT a polygon\n"; };  
};
```

- In my opinion it is bad practice to define default behaviours which are overridden.
- I always use only pure virtual method declarations

Default behaviour

```
struct Shape {  
    virtual double foo() const = 0;    // pure virtual  
};  
  
struct Polygon: Shape {    // add an intermediate class Polygon  
    virtual double foo() const { cout << "I am a polygon\n"; };  
};  
  
struct Square : Polygon {};  
  
struct Hexagon : Polygon {};  
  
struct Circle : Shape {  
    virtual double foo() const { cout << "I am a NOT a polygon\n"; };  
};
```

- This is how I would implement this case, to allow re-use of Polygon::foo. Now there is more no confusion with overrides.

Abstract Classes

```
struct Shape {  
    virtual double foo() const = 0;  
    ~Shape() {} // virtual destructor  
};  
  
int main()  
{  
    Shape a; // compile error: can't instantiate abstract class  
}
```

- A class containing pure virtual methods is said to be abstract
- It cannot be instantiated, because it is incomplete
- Only derived classes which implements foo() can be instantiated

Pure Interfaces

```
struct IShape {  
    virtual double foo() const = 0;  
};
```

- A class containing only pure virtual methods and nothing else (no methods with a body, no data members) is said to be a **pure interface**

Polymorphism and Containers

- The typical use of polymorphism is with containers
- A container contains homogenous types (e.g. int, double, Square, Circle)
- It cannot contain a Square and a Circle because they have different types
- But it can contain a Shape pointer, because it is the same type. Then Circle and Square pointers can be up-casted to it
- E.g.
 - A vector of **pointers to Shape**, which could be of various types
 - A vector of **pointers to Trade**, which could be of various types

Container of Pointers

- A `vector<Circle>` will call the destructor for each `Circle` object
- A `vector<Circle *>` will call the destructor for each `Circle *`. But a `Circle *` is just a pointer and has no destructor
- There is difference between destroying a `Circle` object and a pointer to a `Circle` object
- Because of this, in a container of pointers we must remember to de-allocate each individual object

Smart Pointers

- Smart Pointers are classes which wrap internally a raw pointer.
- It overload the usual pointer operators (->) and (*) so that we can use it indistinguishably from a raw pointer
- There are many variations of smart pointer
- When a smart pointer is destroyed, its destructor usually calls the destructor of the object pointed by the raw pointer

Smart Pointer - Example

```
#include <memory>

struct A
{
    int m_v[1000];
};

int main()
{
    // A contains quite a bit of memory
    // allocate heap memory for A and assign the pointer in
    // a smart pointer
    unique_ptr<A> p(new A);

    return 0;
} // we do not need to delete, the destructor of the smart
// pointer will take care of that
```

Polymorphism - Example

```
int main()
{
    // create a vector of 2 pointers to shape and initialize it
    // with a circle and a square
    vector<unique_ptr<Shape>> s = { new Square(3), new Circle(2) };

    // we use Shape* to access the implementation of perimeter
    // in each specific class
    for (int i = 0; i < 2; ++i)
        cout << s[i]->perimeter() << endl;

    return 0;

    // the vector destructor here calls the destructor of vector,
    // which calls the destructor of each unique_ptr<Shape> object
    // which calls the destructor of the pointed object
}
```

- Full code **ShapeSmartPtr.cpp**

Problem

- Design a polymorphic hierarchy with the class vanilla put option, vanilla call option, digital put option, digital put option
- A base class contains the *notional* data member and a method

```
double Price( double fwd, double vol, double ir)
```
- and a pure virtual method `undiscounted_price`

```
double undiscounted_price( double fwd, double vol,  
double ir)
```
- Store some trades in a vector
- Given some market parameters, compute the total PV of the portfolio

Down Cast

- This is the opposite of up-cast
- Given a pointer to a base class we want to convert it to a pointer to a derived class
- If we know for sure what is the actual type, but, if we are wrong, the program will probably crash

```
DerivedClass *pd = reinterpret_cast<DerivedClass *>(pb);
```

- Otherwise we can give it a try

```
DerivedClass *pd = dynamic_cast<DerivedClass *>(pb);  
if (pd != NULL)  
    // do something with pd
```

- See: <https://www.tutorialcup.com/cplusplus/upcasting-downcasting.htm>

My Advice

- When a polymorphic hierarchy is needed, first the design the pure interfaces
- Multiple inheritance from multiple pure interfaces is ok
- Do not use default implementations for virtual methods, only use pure virtual methods
- Add virtual destructor to base classes of polymorphic hierarchies
- Always define virtual methods in pure interfaces
- Use smart pointers for your containers
- Do not use downcast