Roxana Dumitrescu

C++ in Financial Mathematics

What have we learnt?

Classes with non-trivial destructor

Study case: YourVector class

```
class YourVector{
int size;
int* Name;

public: YourVector(); //default constructor
YourVector(int size); //constructor with
   parameters
YourVector(const YourVector&); //copy
   constructor
~YourVector(); //destructor
YourVector& operator=(const YourVector& v);
   //assignment operator
```

What have we learnt?

Classes with non-trivial destructor

Study case: YourVector class

Rule of three: You have to write the destructor, the copy constructor and the assignment operator. The copy constructor and the assignment operator provided by default by the compiler are **not OK**!

Remark: We have seen during the practical a more complex class YourVector which has an additional member variable **capacity**. A benefit of this class is represented by the fact that we can write an efficient function **push_back()**.

Plan

- Inheritance
- Polymorphism

Plan

Inheritance

Inheritance

Motivating example

- Consider different types of bank accounts
  - Savings accounts
  - Checking accounts
  - Time withdrawal accounts, which are like saving accounts, except that only the interest can be withdrawn
- If you were designing C++ classes to represent each of these, what **member functions might be repeated** among the different classes? What **member functions would be unique** to a given class?

Inheritance

Motivating example

- **To avoid repeating** common member functions and member variables, we will creat a **class hierarchy**, where the common member functions and variables are placed in a **base class** and specialized ones are placed in **derived classes**.

Inheritance

Accounts Hierarchy We will use the accounts class hierarchy as a working example.

- **Account** is the *base class* of the hierarchy.
- **SavingsAccount** is a *derived* class from **Account**. **SavingsAccount** has inherited member variables and functions and ordinarily-defined member variables and functions.
- The member variable **balance** in base class **Account** is **protected**, which means:
  - **balance** is not publicly accessible outside the class, but it is accesible in the derived classes.
  - if **balance** was declared as **private**, then **SavingsAccount** member functions could not access it.

Inheritance

- When using objects of type **SavingsAccount** the inherited and derived members are treated exactly the same and are not distinguishable.
- **CheckingAccount** is also derived from base class **Account**.
- **TimeAccount** is derived from **SavingsAccount**. **SavingsAccount** is its base class and **Account** is its indirect base class.

Inheritance

The Accounts Class Hierarchy

Base class: Account

```
class Account{
protected:
double balance; //account balance
public:
Account(): balance(0.0){};
Account(double bal): balance(bal){};
void deposit(double amt){balance+=amt;};
double get_balance() const {return balance;};
};
```

Inheritance

The Accounts Class Hierarchy

Derived class: SavingsAccount from the Base class Account

```
class SavingsAccount: public Account{
protected:
double rate; // periodic interest rate
public:
SavingsAccount(): Account(){rate=0.0;};
SavingsAccount(double bal, double rate):
   Account(bal){this->rate=rate;};
double compound() {// deposit interest
double interest=balance*rate;
balance+=interest;
return interest;
}
```

Inheritance

The Accounts Class Hierarchy

Derived class: SavingsAccount from the Base class Account

```
double withdraw(double amt) {// if overdraft
    return 0, else return amount
if (amt>balance) { return 0.0;}
else{
  balance-=amt;
  return amt;
       }
}
};
```

Inheritance

The Accounts Class Hierarchy

Derived class: CheckingAccount from the Base class Account

```
class CheckingAccount: public Account{
protected:
double limit;  //lower limit for free checking
double charge;   //per check charge
public:
CheckingAccount(): Account(bal){ limit=lim;
   charge=chg;};
CheckingAccount(double bal, double lim ,
   double chg): Account(bal), limit(lim),
   charge(chg){};
```

Inheritance

```
double cash_check(double amt){
assert(amt>0);
if (balance<limit && (amt+charge<=balance)){
balance-=amt+charge;
return amt;
}
else if (balance>=limit && amt<=balance){
balance-=amt;
return amt;
} else { return 0.0;}}
}
};
```

Inheritance

The Accounts Class Hierarchy

Derived class: TimeAccount from the Base class SavingsAccount

```
class TimeAccount: public SavingsAccount{
        protected:
double funds_avail; //amount available for
   withdrawal
public:
TimeAccount():
   SavingsAccount(){funds_avail=0.0;};
TimeAccount(double bal, double rate):
   SavingsAccount(bal, rate),
   funds_avail(0.0){};
// redefines 2 member functions from
   SavingsAccount
```

Inheritance

```
double compound(){
double interest=SavingsAccount::compound();
funds_avail+=interest;
return interest;};

      double withdraw(double amt){
if (amt<=funds_avail){
funds_avail-=amt;
balance-=amt;
return amt;
}  else { return 0.0;};
}

double get_avail() const {return funds_avail;};

};
```

Inheritance

Constructors and Destructors

- **Constructors** of a derived class *call the base class constructor* immediately, **before** doing **anything** else. The only thing you can control is which constructor is called and what the arguments will be.
  When a **TimeAccount** is created 3 constructors are called in the following order:
  - The **Account** constructor
  - The **SavingsAccount** constructor
  - The **TimeAccount** constructor.

Inheritance

Constructors and Destructors

- The **reverse is true for destructors**: derived class destructors do their job first and then base class destructors are called automatically.

  In our particular example: when an object of the class **TimeAccount** goes out of scope, the destructors are called in the following order:

  - The **TimeAccount** destructor.
  - The **SavingsAccount** destructor.
  - The **Account** destructor.

Inheritance

Overriding Members

- A derived class can redefine member functions in the base class. The function prototype must be identical, not even the use of **const** can be different (otherwise both functions will be accessible).
- For example, in the class **TimeAccount** we have **TimeAccount::compound** and **TimeAccount::withdraw**
- Once a function is redefined it is not possible to call the base class function, unless it is explicitely called as in **SavingsAccount::compound**.

Inheritance

Overriding Members

Example

```
TimeAccount obj;
obj.compound(); // It is called the function
   compound defined in the class TimeAccount;
obj.SavingsAccount::compound(); // It is
   called the function compound defined in the
   class SavingsAccount;
```

Inheritance

Public, Private and Protected Inheritance

- Note the line **class Savings_Account: public Account** This specifies that the member functions and variables from **Account** do not change their *public, protected* or *private* status in SavingsAccount. This is called *public* inheritance.
- *protected* inheritance: public members in the base class become protected and other members are unchanged.
- *private* inheritance: all members become private.

Inheritance

Public Inheritance

- Note the line **class Savings_Account: public Account** This specifies that the member functions and variables from **Account** do not change their *public, protected* or *private* status in SavingsAccount. This is called *public* inheritance.

There also exist:

- ***protected* inheritance**: public members in the base class become protected and other members are unchanged.
- ***private* inheritance**: all members become private.

In this course, we'll only use public inheritance.

Inheritance

The most important aspect of **inheritance** is the **relationship expressed between the new class and the base class**.

After encapsulation, **Inheritance** is the second essential feature of an object-oriented programming language.

Inheritance

Relationships Among Classes

- If **"C1 is a C2"** class, then C1 should be a derived class (a subclass) of C2. For example, "a savings account is an account". This relationship corresponds in our case to the **public inheritance**.
- If **"C1 has a C2"**, then class C1 should have a member variable of type C2. For example, "a cylinder has a circle as its base". Or "a circle **has** a Point as its center". This relationship corresponds to a design technique called **composition**.
- In the case of **"C1 is implemented as a C2"**, then C1 should be derived from C2, but with **private inheritance**. For example, "the stack is implemented as a list".

Inheritance

Relationships Among Classes: Composition/Inheritance

Inheritance and composition allow to create a new type from existing types, and both embed subobjects of the existing types inside the new type. The main difference between them is the following:

- You use composition to reuse existing types as part of the underlying implementation of the new type.
- You use inheritance when you want to force the new type to be the same type as the base class.

Polymorphism

Polymorphism

Let us consider the following **Musician hierarchy**.

1. Base class

```
class Musician{
public:
void greet();
};

void Musician::greet()   {cout<<"musician says:
   hello\n":}
```

Polymorphism

2. A first derived class: Trumpeter

```
class Trumpeter: public Musician{
public:
void greet();
};

void Trumpeter::greet()   {cout<<"trumpeter
    says:hello\n":}
```

Polymorphism

3. A second derived class: Pianist

```
class Pianist: public Musician{
public:
void greet();
};
void Pianist::greet()   {cout<<"pianist says:
   hello\n":}
```

Polymorphism

Invoking the functions through pointers/ instances

```
int main(){
Trumpeter t; // a Trumpeter instance
Pianist p; // a Pianist instance
Musician m, *pm; // a Musician instance and a
   Musician pointer
// 1: invoking through a Musician instance
m.greet();  // prints "musician says: hello"

// 2: invoking through  Trumpeter and Pianist
   instances
t.greet();  // prints "trumpeter says: hello"
p.greet();  // prints "pianist says: hello"
}
```

Polymorphism

Invoking the functions through pointers/ instances

```
// 3: invoking through a Musician pointer on a
   Musician instance
pm=&m;  // points to Musician
pm->greet();  // prints "musician says: hello"
// 4: invoking through a Musician pointer on a
   Trumpeter instance and on a Pianist instance
pm=&t;   // points to Trumpeter
pm->greet();  // prints "musician says: hello"
pm=&p;  // points to Pianist
pm->greet(); // prints "musician says:hello"
}
```

Polymorphism

Remark: Taking the address of an object of a derived class (either using a pointer or a reference) and treating it as the address of the base type is called **upcasting**

In Group 4, we declare a pointer to a **Musician**, and without complaint we can initialise it with the address of a **Trumpeter** or **Pianist** (because **Trumpeter** or **Pianist** are derived from **Musician**) (this is an example of upcasting). Note that the interface of **Musician** must exist in **Trumpeter** or **Pianist**, because **Trumpeter** and **Pianist** are publicly inherited from **Musician**.

Polymorphism

Remark 2:

- The results obtained for the first 3 Groups are as expected. In **Group 4**, invoking the function greet through a Musician pointer has in all the cases as result "musician says: hello" (even if the Musician is actually a Trumpeter or a Pianist): **the result is not the one that we expect!**

- Which should be the **correct result**?
  If the pointer pm points to a Musician which is a Trumpeter (as in Group 4) we should obtain as message: "Trumpeter says: hello"!, instead of "Musician says: hello"!

- Which is the **solution**?

Polymorphism

The solution is: the keyword "virtual"!

Basing the behavior of an object on its run-time is a task that C++ takes on with *virtual functions*. A virtual function is a special kind of member function. You declare it with the keyword **virtual**.

We now re-write our example of Musician hierarchy using the keyword virtual.

Polymorphism

1. Base class

```
class Musician{
public:
virtual void greet();    // virtual function
};

void Musician::greet()   {cout<<"musician says:
   hello\n":}
```

We declare **greet()** to be a virtual function.

Polymorphism

We have created a base class **Musician** with **a virtual function greet()**. Which is the next step?

- We derive the classes where we override the virtual functions to behave in ways specific to each class.

Polymorphism

2. A first derived class: Trumpeter

```
class Trumpeter: public Musician{
public:
virtual void greet();    // overriding
   Musician's virtual function
};

void Trumpeter::greet()   {cout<<"trumpeter
   says: hello\n":}
```

Polymorphism

3. A second derived class: Pianist

```
class Pianist: public Musician{
public:
virtual void greet();    // overriding
   Musician's virtual function
};

void Pianist::greet()  {cout<<"pianist says:
   hello\n":}
```

Polymorphism

Invoking the functions through pointers/ instances

```
int main(){
Trumpeter t; // a Trumpeter instance
Musician m, *pm; // a Musician instance and a
    Musician pointer
// 1: invoking through a Musician instance
m.greet();  // prints "musician says: hello"
// 2: invoking through a Trumpeter instance
t.greet();  // prints "trumpeter says: hello"
}
```

Polymorphism

Invoking the functions through pointers/ instances

```
// 3: invoking through a Musician pointer on a
   Musician instance
pm=&m;  // points to Musician
pm->greet();  // prints "musician says: hello"

// 4: invoking through a Musician pointer on a
   Trumpeter instance
pm=&t;   // points to Trumpeter
pm->greet();  // prints "trumpeter says: hello"
}
```

Polymorphism

Group 1 - We call the function **greet()** on a **Musician** instance - it acts like a **Musician**

Group 2 - We call the function **greet()** on a **Trumpeter** instance - acts like a **Trumpeter**

Group 3 - We make **pm** - a **Musician** pointer -point to **m**. Invoking the member function **greet()**, we see that the object acts like a **Musician**.

Group 4 - we point **pm** at *t*. When we invoke **greet()**, we see the **Trumpeter** greeting!

Polymorphism

Difference between non-virtual and virtual functions.

- In our **first example**: the function **greet()** is **non-virtual**. In this case, the function is invoked based on the *apparent* type of the object. As the object of type **Trumpeter** is accessed thorugh a pointer to a **Musician**, it is apparently a **Musician**; so the **Musician** greeting is invoked.
- In our **second example**: the function **greet()** is **virtual**. In this function, the function is invoked based on the *actual* type of the object. In the final group above, we made **pm** point to a **Trumpeter**; so the **Trumpeter** player is invoked.

Polymorphism

Concept of binding.

- Connecting a function call to a function body is called *binding*.
- When binding is performed before the program is run (by the compiler and linker), it's called *early binding*. In our first example, the problem related to the function **greet** is caused by the *early binding*: the compiler cannot know the correct function to call when it has only the Musician address!
- The solution is called *late binding*, which means that the binding occurs at runtime, based on the type of the object. In the last example, the **late binding occurs for the function greet thanks to virtual keyword!**
- C compilers have only one kind of function call, that's *early binding*.

Polymorphism

To retain

- Creating *late binding*, we get the desired behavior of an object!
- *Late binding* occurs only with **virtual functions** and only when you're using **pointers to the base class** where those virtual functions exist.

Polymorphism

Polymorphism is the third essential feature of an object-oriented programming language, after encapsulation and inheritance.

Polymorphism means to use different objects in the same code, only manipulating pointers to the base class Musician! **With polymorphism we can write code in terms of generic Musician and make the code work correctly for any actual Musician**.

The generic Musician is polymorphic - *many forms* - because at any particular point during program execution it can be a Trumpeter, a Pianist etc. **Polymorphism** requires the generic Musician to behave differently at run-time depending on the *actual* type of Musician it is.

Late binding is used in order to implement **Polymorphism**!

Polymorphism

Polymorphism

What we do is create a common base class for all types we want to use together polymorphically. The common **interface** of all the classes is implemented using virtual functions. In our examples above, we used the **Musician** class as the base class holding the common interface.

Polymorphism

Example of polymorphic code only using pointers to the base class Musician.

```
void Musiciangreet(Musician* pm)
{
cout<<"introducing....\n";
pm->greet();
}
```

This routine will work on an instance of Musician or any class derived from Musician because **greet() is virtual**!

Polymorphism

We can create an entire orchestra in a single data structure!

```
int main(){
vector<Musician*> orchestra; // a vector which
   holds the entire orchestra
orchestra.push_back(new Trumpeter);
orchestra.push_back(new Pianist);
orchestra.push_back(new Violonist);
for (int i=0; i<orchestra.size(); i++)
Musiciangreet(orchestra[i]);
for (int i=0; i<orchestra.size(); i++)
delete v[i];
}
```

Polymorphism

Object slicing There is a **distinct** difference between passing

the address of objects (using pointers) and passing objects by value when using polymorphism. All the examples that we have seen pass addresses and not values!

What happens if you try to upcast without using pointers?

The object is "sliced" until all that remains is the subobject that corresponds to the destination type of your cast.

Polymorphism

Object slicing

```
void Musiciangreet (Musician pm)
{
cout<<"introducing....\n";
pm.greet();
}
```

The function **Musiciangreet()** is passed an object of type **Musician** *by value*. It then calls the virtual function **greet()** for the **Musician**.

Polymorphism

```
Trumpeter t;
Musiciangreet(t);
```

You might expect that the above code produce "Trumpeter says...". The result will be "Musician says.."

When a virtual function is invoked through an object, there is no doubt about which version of the function is invoked.

Polymorphism

Rule: Use virtual function+pointers to the base class!

- With greet() defined as virtual in the base class, you can add as many new types as you want without changing the Musiciangreet( ) function. In a well-designed OOP program, most or all of your functions will follow the model of Musiciangreet( ) and communicate only with the base-class interface. Such a program is **extensible** because you can add new functionality by inheriting new data types from the common base class. The functions that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

Polymorphism

Invoking virtual functions dynamically

We have seen that virtual functions are invoked dynamically, that is, the version of the function invoked is based on the run-time type of the object. This allows us to support polymorphism by using pointers to the base class. **But pointers aren't the only way we can use virtual functions in order to obtain a polymorphic behaviour. We can also use virtual functions and references together.**

Polymorphism

Invoking virtual functions dynamically

```
void Musiciangreeting(Musician &rm)
    {rm.greet()}    // invokes a virtual function
int main() {
Musician m;
Trumpeter t;
Pianist p;

Musiciangreeting(m);     // will result in
    invocation of Musician::play()
Musiciangreeting(t);  // will result in
    invocation of Trumpeter::play()
Musiciangreeting(p);  // will result in
    invocation of Pianist::play()
return 0;
}
```

Polymorphism

Conclusion: Calling a virtual function through an object - rather than through a pointer or reference to an object - always results in the same version being invoked.

Remark: If you are invoking a virtual function through a pointer or reference, you still need to ensure which version is invoked.

```
int main(){
Musician *pm=new Trumpeter;  // really a
   Trumpeter
pm->greet();  // invokes Trumpeter::greet()
pm->Musician::greet(); // invokes
   Musician::greet()
pm->Trumpeter::greet(); // error
delete pm;
return 0;
}
```

Polymorphism

Conclusion: Calling a virtual function through an object - rather than through a pointer or reference to an object - always results in the same version being invoked.

If you want a function to behave polymorphically: use the virtual keyword + pointers or references to the base class.

Polymorphism

Relaxed Overriding Rules

You must always match signatures to override a virtual function. You do have some freedom with the return type. For example, if the original function returns a pointer to some class, the overriding function can return a pointer to a derived class.

Polymorphism

Abstract base classes and pure virtual functions

Often in a design, you want the base class to present *only* an interface for its derived classes. In other words, you don't want anyone to actually create an object of the base class, only to upcast to it, so that its interface can be used. This is accomplished by making that class *abstract* which happens if you give it at least one *pure virtual function*.

What is a pure virtual function?

A pure virtual function is a function which uses the **virtual** keyword and it is followed by **=0**.

If anyone tries to make an object of an abstract class, the compiler prevents them. This is a tool that allows you to enforce a particular design.

Polymorphism

Abstract base classes and pure virtual functions

When an abstract class is inherited, all pure virtual functions must be implemented.

An interface class (or a pure abstract class) is a class which contains only pure virtual functions and no member variables. It can be seen as a contract between the designer of the class and the users, in the sense that any class implementing the interface class provides the functionality announced in the interface class.

Polymorphism

The constructor can not be made virtual!

Destructors and Virtual destructors What happens if you want to manipulate an object through a pointer to its base class (that is, manipulate the object through its generic interface)? The problem occurs when you want to **delete** a pointer of this type for an object that has been created on the heap with **new**. If the pointer is to the base class, the compiler can only know to call the base-class version of the destructor during **delete**. This is the same problem that virtual functions were created to solve the general case.

Virtual functions work for destructors as they do for all other functions except constructors.

Polymorphism

Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
   destructor
class Base1{
public:
  ~Base1(){ cout<<"~Base1()\n ";}
};

class Derived1: public Base1{
public: ~Derived1(){cout<<"~Derived1()\n";}

};
```

Polymorphism

Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
   destructor

class Base2{
public:
 virtual ~Base2(){ cout<<"~Base2()\n ";}
};

class Derived2: public Base2{
public: ~Derived2(){cout<<"~Derived2()\n";}
};
```

Polymorphism

Virtual versus non-virtual destructor!

```
//Behavior of virtual vs. non-virtual
   destructor
int main()
{
Base1 *bp=new Derived 1; // Upcast
delete bp;
Base2 * bp2=new Derived2; // Upcast
delete bp2;
}
```

Polymorphism

Virtual versus non-virtual destructor!

When you run the program you'll see:

- **delete bp** only calls the base-class destructor;
- **delete b2p** calls the derived-class destructor followed by the base class destructor, which is the behavior we desire.

Note that:

- Forgetting to make a destructor **virtual** is an insidious bug because it often doesn't directly affect the behavior of your program, but it can quietly introduce a memory leak.
- Even though the destructor, like the constructor, is an "exceptional" function, it is possible for the destructor to be virtual because the object already knows what type it is.

Summing up

We have learnt about:

- **Inheritance** Create a base class and derived classes, where the base class contains the common member variables and functions.
- **Polymorphism**- *Many forms*. You want a function of the base class to behave in a specific way to each of the derived classes. In order to get the polymorphic behaviour: use the keyword **virtual** and **pointers or reference**s to the base class.

Summing up

- Constructors can not be virtual. The destructor of a class that you intend to use as a base class should be a **virtual destructor**.
- In conclusion; the base class contains the common non-virtual functions (that have the same behaviour for each derived class) and virtual functions (same name, but different behaviour depending on the derived class)!