Lab 6: Inheritance and Maps

Diego Aguilar, Laura Cruz Castro, Cameron Brown

Spring 2024

Contents

Part I
Inheritance

1 Overview

In this part of the assignment, you're going to create a series of classes to represent bank accounts. While the classes themselves are all simple, you will be using inheritance and the concept of polymorphism to make them functional with the given main code. You will also create classes which make use of multiple inheritance, as well as private inheritance.

1.1 Description

The basic idea of inheritance is that you have a **base** class, and you create new classes which **derive** from that base class. One common use of inheritance is to reuse data and functions: if a base class defines those things, then the derived class doesn't have to; it inherits them, so in effect, you get "free" code in that class.

Another use of inheritance is to define a class which serves as what other programming languages would call an **interface**. In C++, the keyword interface doesn't exist, but some of the classes you create here will serve the same purpose. An interface defines how a class should look on the outside.

The first class shown here will serve as interfaces; all the other classes will derive from it. In this way, all classes in this assignment are bank accounts. Some types of inheritance model the "Is-a" relationship. A triangle IS A shape. A circle IS A shape as well.

Other types of inheritance model a "has-a" relationship. A car, for example, has an engine—but it isn't an engine. This type of relationship can still be created, but the process is a bit different.

1.2 Abstract Base Class

An **A**bstract **B**ase **C**lass (**ABC**) is a class you never want to create an instance of; it doesn't have enough information to be useful **on its own**. An ABC could have any number of functions or variables, whether public, protected, or private. You cannot create an instance of an abstract base class.

Perhaps you are creating a program that needs to store data about students and faculty members. They share similar data (name, age, email address, etc.), but have their own **unique** data as well. You might create 3 classes:

```
1 // Abstract class. "Just a Person" isn't enough data
2 class Person
3 class Student : public Person
4 class Faculty : public Person
```

To create an abstract base class, at least one function in the class must be a pure virtual function. A pure virtual function is one that:

1. Have the virtual keyword before the return type
2. Has = 0 at the end of the prototype
3. Has no function definition (i.e. no body) – the only exception to this is the destructor. You MUST have a body on a destructor, pure virtual or otherwise.

For example, our display function is a pure virtual function:

```
1 virtual void Display() const = 0;
```

1.3 Multiple Inheritance

While many languages do not allow multiple inheritance, C++ does! The basic concept is the same. You have multiple base classes, and a single class can derive from some or all of them, inheriting the data members and functions of each base class.

While that is all you really need to know, it may help in this lab to also be aware of how multiple inheritance works behind the scenes. If you do wish for some extra reading this article might be useful. This is a very complicated topic so take your time in understanding it!

1.4 The Diamond Problem

An issue with multiple inheritance is the dreaded "diamond problem." This comes about when a derived class inherits from two base classes, which each derive from the same parent base class. Consider this:

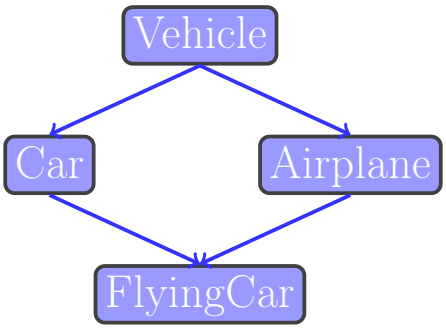

Figure 1.1: Diamond Problem

In this example, you want FlyingCar to inherit both driving and flying functionality... but you unfortunately inherit Vehicle functionality twice!

1.4.1 How to resolve the diamond problem

You can fix the issue of the diamond problem by using **virtual** inheritance. In the above example, the class declarations might look like this:

```
1 class Vehicle
2 class Car : public Vehicle
3 class Airplane : public Vehicle
4 class FlyingCar : public Car, public Airplane
```

With virtual inheritance, the "in-between" classes would add the virtual keyword to the inheritance:

```
1 class Car : virtual public Vehicle
2 class Airplane : virtual public Vehicle
```

This ensures that FlyingCar would only inherit ONE instance of anything Car and Airplane inherited from Vehicle. For more information:

https://en.wikipedia.org/wiki/Virtual_inheritance
https://isocpp.org/wiki/faq/multiple-inheritance#virtual-inheritance-where

1.5 Public and Private Inheritance

Perhaps the most common type of inheritance is public inheritance. It represents an **"is-a"** relationship between classes. The derived class is a base class.
For example:

```
1 class Car : public Vehicle
2 class SavingsAccount : public BankAccount
3 class Button : public UIControl
```

We would say that the Car (derived class) IS A Vehicle (base class). A SavingsAccount IS A BankAccount. A Button IS A UIControl, etc. If class inherits from another class using **private** inheritance, however, it models a "has a" relationship. This is also sometimes referred to as **composition**. For example:

```
1 // Private inheritance -- the car "has a" Engine
2 class Car : private Engine
```

The only thing the Car class can access of the Engine object is *public* members and functions - here's where the *accessor* functions of a class would come into play.

1.5.1 Accessing members of base class(es)

To access the functions or variables of a base class, you must specify the name of the class, and then the thing you are trying to access. So in the case of the car and its engine, you would do something like this:

```
1 void Car::SomeFunction() {
2     string maker = Engine::GetManufacturer();
3 }
```

Reference: https://isocpp.org/wiki/faq/private-inheritance

1.6 Virtual

We've gone over different use cases of virtual, that it's necessary for solving the diamond problem, that ABC's must have at least one pure virtual function, etc. So what is it?

Let's start with virtual inheritance. This is specifically when you specify a virtual parent class. It would look something like this: "`Derrived : virtual public Base`". We've already gone over the fact that this is necessary for solving the diamond problem. But why? The reason is that virtual inheritance tells the compiler to only ever include one instance of the class within derived classes. More accurately, it tells the compiler to check if the parent class has already been inherited and to use that instance as the parent rather than making a new one.

Let's move on to virtual methods. In this case we are telling the compiler to make what's called a vtable for any class with this method. You don't have to know much about what the vtable does but I do recommend looking into it as it helps with a lower-level understanding. **With that said**, what you should know is that the result of using a vtable is the ability to override and call overriden behavior from a downcasted parent declaration. This means running the code snippet below will result in the child behavior not the parents!

```
1 ((Parent*)child)->someVirtualOverridenMethod()
```

As a take away, the virtual keyword seems ensure that there is only one instance of something in a class. Whether that means one definition for a function (regardless the static dispatch) or one instance of a parent within the child class.

1.7 Polymorphism

One important concept in object-oriented programming is that a **base-class pointer** can point to any object which **derives** from that base class. For example:

```
class Car{};
class Mustang : public Car{};
// Base class
// Derived class
// Set a base class pointer to an instance of the derived
class Car *someCar = new Mustang; // Valid due to inheritance
```

Following that same concept, we could extend that to something like the following:

```
Car *cars[3];
cars[0] = new Car;
cars[1] = new SportsCar;
cars[2] = new LuxurySedan;
```

What polymorphism allows you to do is call a function from one of those objects and **depending on what the pointer is pointing to** the correct function will be called.

```
class Car {
public:
    virtual void Identity()
    { cout << "I'm just a car!"; }
};
class SportsCar : public Car {
public:
    void Identity()
    { cout << "I'm a sports car!"; }
};
class LuxurySedan : public Car {
public:
    void Identity()
    { cout << "I'm a luxury car!"; }
};
```

Given the previous array, code like the main function bellow would print out "I'm a sports car!".

```
int main() {
    cars[1]->Identity();
    return 0;
}
```

Polymorphism is the concept that allows you to write code like that, without knowing exactly what type of object you are pointing to. Since cars[1] points to a SportsCar, the call to `Identity()` will determine that the SportsCar version of the function should be used. The **virtual** keyword is required on the base class version of the function in order to make this functionality possible.

For more on polymorphism: http://www.cplusplus.com/doc/tutorial/polymorphism/

2 Assignment Specifics

2.1 Bank Account

BankAccount is an abstract base class. Because at least one of the functions in this class are **pure virtual functions** (virtual functions with no definition). This base says that all BankAccounts can calculate an amount, display their data, and recieve a deposit. Each derived class can define HOW they do each of these. This class will not require a virtual destructor. A virtual destructor is used to make sure child destructors (if necessary) are called even if the deleted variable is upcasted as a parent class.

Some other important features of the BankAccount class are the name and id that each bank account must hold. ID is something specific and unique to an account. However, in the specific case of the investment account, there may be multiple amounts and therefore balance can't be stored in this interface. The private variables you make should also have getters defined here!

```
class BankAccount {
private: // variables must be defined with these names for check 2 to work properly!
    int id;
    std::string name;
public:
    // Parameterized constructor (makes your life easier but not required)
    // Necessary getters: getName(), getID()
    // The following pure virtual functions: Display(), Deposit(float), getAmount()
};
```

2.2 Checking Account

CheckingAccount derives from the BankAccount class (a CheckingAccount **IS A** BankAccount). This class, however, is not an abstract base class. This means we now must define any previous virtual functions.

All CheckingAccount must have `amount` and `setAmmount()`. Privacy of variables is easy to overlook and see as just being "good practice". However, forgetting or deciding not to care about privacy can lead to very important vulnerabilities! For this reason we will have the variable amount as private. With that in mind, and the knowledge that this class will have a child class, we still may need to modify it **within the class hierarchy**. This is where protected comes in. protected is a class member privacy that allows it to be visible to other classes within the hierarchy. For these reasons `setAmount()` will be protected.

There is also the need to consider unique behavioral functions specific to the checking account class. In this case it is one simple function: `Withdraw()`. This function is fairly straight forward, except for the fact that it will need to be used in a child class. How you implement it, including whether you use virtual or not, is up to you. You may have a virtual definition here and a overriden one in the child class, you may decide to have the child simply inherit a definition that works for it too, you may even decide to have overriden behaviour without use of virtual. In any case, it's important that this function returns a boolean based on whether the withdraw is successful **within the checking account**.

```
class CheckingAccount : virtual public BankAccount {
private:
    // this variable must have this name for the implementation check to work properly!
    float amount;
protected:
    // this function must have this name for the implementation check to work properly!
    void setAmount(float amount);
public:
    // Parameterized constructor (float, int, string)
    // Behavioral function(s): Withdraw(float)
    // all three overrides (see 'Bank Account' section)
};
```

2.3 Savings Account

The savings account class will be treated similarly to the Checking account. All the same virtual functions must be overridden, a new private variable `amount` must be added as well as a new protected method `setAmount()`.

There is also a new behavioral function to add: `CompoundEarnings()`, which will return a float (the earned amount). This method should take in a positive float and set the amount in the account to amount*(1+percent). This is how the account will earn interest. While this seems like the same situation as with the Checking account, this function will change in the future investment account.

In addition, you must also implement `Transfer()`. This function will return a boolean representing the operations success. It will take in a checking account object and a float then transfer 'float' amount from the savings account to the passed in checking account (modifying parameters... where have I heard that before).

```
1  class SavingsAccount : virtual public BankAccount {
2  private:
3      // this variable must have this name for the implementation check to work properly!
4      float amount;
5  protected:
6      // this function must have this name for the implementation check to work properly!
7      void setAmount(float amount);
8  public:
9      // Parameterized constructor (float, int, string)
10     // Behavioral function(s): Transfer(CheckingAccount, float), CompoundEarnings(float)
11     // all three overrides
12 };
```

2.4 Investment Account

Now it's time for the more complicated account class of the 4. The premise of this class is an account that for the most part **is a** checking account but **has a** sort of savings account. This savings account will be the investment side of the account. The idea is for any *earnings* from the investment side to be immediately available in a checking account format as to not have the user worry about transferring money at all.

With that said here are a few things to look out for:

- getAmount should give the total amount in both!
- initial values and any deposits go to the savings account not to the checking!
- You will have to override quite a few functions here!
- Transfer should only take one 'float percent' and transfer from this savings to this checking!

```
1  class InvestmentAccount : public CheckingAccount, private SavingsAccount {
2      // This is up to you to figure out!
3  };
```

2.5 Tips

A few tips about this assignment:

- **MOST IMPORTANT:** One very necessary distinction to make is that of `getAmount()` and `amount`. Specifically because `getAmount()` is virtual calling it will get the definition of the **invoking object** which is not necessarily the current class. That is to say calling it within the CheckingAccount class will return the total amount within an InvestmentAccount object if that is the invoking object.
- Start with one class at a time. It is important to keep future classes in mind but trying to work from the bottom up can mean a lot of back and forth.
- For the Display function use the following formatting (replace [...] with corresponding information/variables):

```
1         std::cout << [Account name] << " (" << [Account ID] << "):" << std::endl;
2         std::cout << "\t[Account type] Account: \$" << std::fixed << std::
              setprecision(2) << [Amount in Account] << std::endl;
```

  For example a savings account would replace [Account type] with: `Savings`.

Part II
Maps... and other

3 Overview

For this assignment, you must write main(int argc, char** argv) and the functor, `AccountReader`. As long as you implemented everything correctly in part 1 this part will come a lot easier!

3.1 Maps in C++

Thankfully the map class is not one we have to right ourselves! It's included in the STL specifications meaning all we have to do is `#include<map>`. There is also the unordered_map which uses hashing functions to have a slightly better performance, but to produce consistent output with options 1 and 2 we will be using the map class. To help get you started here is an example of looping through a map:

```
map<string, int> myMap;
// ... (inserts)
for (auto it = myMap.begin(); it != myMap.end(); it++) {
    cout << "Key: " << it->first << endl;
    cout << "Value: " << it->second << endl << endl;
}
```

3.2 Upcasting

Upcasting is the process of converting an object which is declared as a parent class into one declared as the child class. Notice that this changes what's called the static dispatch, you can think of this as what you declare a variable. However, without checking that this object actually is the child class we want to cast to, this is a dangerous thing to do. To safely upcast we will use `dynamic\_cast` in the following format:

```
ChildClass* myChild = dynamic_cast<ChildClass*>(parentPtr);
if (myChild) { // equivalent to myChild != nullptr
    // code assuming myChild was successfully upcasted
}
else {
    // code to handle the case that it didn't
}
```

3.3 Functors

We will be using functors in this lab! Functors are a neat way to generalize functions. For example, if you need to have a function that calculates gravitational force you might need different constants for gravity. So instead of having to pass a parameter every time we call the function, we can use a functor! The following code is for this example:

```
struct GForce {
    float acceleration; // You CAN make a constructor for this
    float operator()(float mass) { // the return type and parameter(s) can be changed!
        return mass*acceleration;
    }
};
int main() {
    GForce earth, moon, mars;
    earth.acceleration = 9.81f;
    moon.acceleration = 1.62f;
    mars.acceleration = 3.71f;
    cout << "Earth: " << earth(70) << endl;
    cout << "Moon: " << moon(70) << endl;
    cout << "Mars: " << mars(70) << endl;
}
```

4 Assignment Specifics

4.1 Description

You are given a csv file (**C**omma **S**eparated **V**alues) with rows representing different bank accounts. You must read from these rows and store their information within a **new** object of it's respective account class. Then store that object within two maps. You will have store it as a pointer to ensure polymorphism is maintained. The keys you will need to look up with are the account owner's name and the account's ID. You will code a functor that takes in the file path of the csv file.

To make things easier you should have the following 2 maps in your functor:

- Map 1: ID lookup map. This map will use the owners id as the key and the owners Bank Account as a value (BankAccount*)
- Map 2: Name lookup map. This map will use the owners name as the key and the owners Bank Account as a value (BankAccount*)

The format of the file is as follows:

Type	Name	Amount	ID
2	Derek Pearce	1773	1009
1	Ria Marshall	3288	1010
3	Ronan Walker	8971	1011
3	Eric Carr	4762	1012
etc...			

The type number will be one of the ints 1, 2, or 3 representing Checking, Savings, and Investment accounts respectively

The options you will have to read (cin) and implement within your call operator:

1. Display all accounts
2. Display all of account type
   2.1. Checking Accounts
   2.2. Savings Accounts
   2.3. Investment Accounts
3. Find account
   - Integer input for ID search (use stoi!)
   - String input for name search **(HAS TO BE FULL NAME)**
   - Not Found

4.2 Additional Notes

- ****IMPORTANT**** You are expected to use CLI to get the file path from the main and pass it to the functor. The program will be called in the following format:
  - ./main.out path/to/accounts.csv
- We are using stdin (aka cin) for option selection. HOWEVER, note that `cin >> name` is **NOT** the same as `getline(cin, name)`... this important for when the input has spaces!
- Because this part requires you to create **new** variables and store their pointers in maps, we are working with dynamic memory. As such you are expected to delete the pointers also!