Lecture 3
Object Oriented Programming II

Jumping down the rabbit hole…..

Lecture Overview

- **Inheritance**
  - Motivation
  - Syntax
  - Overridden method
- **Polymorphism**
  - Static vs Dynamic Binding
- **Abstract Class**
  - Motivation & Syntax
  - Design implication

Inheritance

Like father, like son

Inheritance : Motivation

- Let's define a **saving account** class
  - **Data** :
    - account number, balance
    - **interest rate**
  - **Process**:
    - withdraw, deposit
    - **pay_interest**

- It is clear that:
  - **Saving Account** shares > 50% code with **Bank Account**

- Should we just cut and paste the code?

Inheritance : Motivation

- Duplicating code is undesirable:
  - Hard to maintain
    - Need to correct all copies if error is found
    - Need to update all copies if modification is needed
    - Etc
- Since the classes are logically unrelated:
  - Other code that work on one class cannot work on the other
  - Example:

```
void transfer(BankAcct& fromAcct,
              BankAcct& toAcct, double amt);
```

will **not** work on **saving account** objects
(compilation error due to incompatible data type)

Inheritance : Motivation

- Object oriented languages allow **inheritance**
  - Derive a new class from another class
  - The new class **inherits** most of the attributes and methods from the other class

- **Terminology:**
  - If `class B` is derived from `class A`, then
    - `class B` is called a **child (sub-class)** of `class A`
    - `class A` is called a **parent (super-class)** of `class B`

Saving Account : Inheritance example

```
class BankAcct {
protected:
    int _acctNum;
    double _balance;
    ... ...
};

class SavingAcct : public BankAcct {
protected:
    double _rate; // interest rate

public:
    SavingAcct(int anum, double bal, double rate)
    : BankAcct(anum, bal) {
        _rate = rate; // BankAcct does not have rate
    }

    void payInterest() {
        _balance += _balance * _rate;
    }
};
```

Changed from `private`

To indicate inheritance

Note that there is **no** declaration for account number and balance, they are inherited

Special syntax for initializing base class or object member

Observations

- Inheritance greatly reduces the amount redundant coding
  - No (re)definition of **`account number`** and **`balance`**
  - No (re)definition of **`withdraw()`** and **`deposit()`**

- Improve maintainability:
  - E.g. If the **`withdraw()`** function is modified in **`class BankAcct`**
    - no changes is needed in **`class SavingAcct`**
  - The code in **`class BankAcct`** remains untouched
    - Other programs using **`BankAcct`** are not affected

Saving Accounts : Sample Usage

```
class BankAcct { // definition not shown };
class SavingAcct { // definition not shown };

int main() {
    BankAcct ba1(1234, 500.00); // from Lecture 2
    SavingAcct sa1(8888, 999.99, 0.025);

    sa1.balance();
    sa1.deposit(0.01);
    sa1.balance();

    sa1.payInterest();
    sa1.balance();

    return 0;
}
```

Inherited methods from superclass

New method in `SavingAcct` class

Method Overriding

- Sometimes we need to modify the inherited method:
  - To change / extend the functionality
  - This is known as **method overriding**

- In the `SavingAcct` example:
  - The (print) **`balance()`** method should be modified to include the interest rate in output

- To override an inherited method:
  - Simply recode the method in the subclass using the same method header
  - Method header refers to the name and parameters type of the method (also known as **method signature**)

Method Overriding: Example

```
class SavingAcct : public BankAcct {
  // attributes and other methods not shown

  void balance() { // override balance method from BankAcct
      cout << "Acc: " << _acctNum
           << ", Bal: " << _balance
           << ", Int: " << _rate << endl;
  }
}
```

- The first two lines of code is exactly the same as **BankAcct**'s print **balance()**:
  - Can we reuse **BankAcct**'s print **balance()** instead of re-coding?

Calling SuperClass Method

- We can call a super class's method from any sub class:
  - Useful when the method is overriden
- **Syntax:**

```
superclass_name::method( parameter… )
```

```
class SavingAcct: public BankAcct {

  // attributes and other methods not shown
  void balance() {
     BankAcct::balance();
     cout << "Interest: " << _rate << endl; // addition
  }
}
```

Make use BankAcct's balance() method

Subclass Substitutability

- An added advantage for inheritance is that:
  - Whenever a super class object is expected, a sub class object **is acceptable as substitution**
    - **Caution:** the **reverse is NOT true**
  - Hence, all existing functions that works with the super class objects will work on sub class objects with **no modification**!
- Analogy:
  - We can drive a car
  - Honda is a car (Honda is a subclass of car)
  - We can drive a Honda car

Subclass Substitution: Example

```
class BankAcct { …… }  // not shown
class SavingAcct { …… }  // not shown

void transfer(BankAcct& fromAcct,
              BankAcct& toAcct, double amt) {
   fromAcct.withdraw(amt);
   toAcct.deposit(amt);
}

int main() {
    BankAcct ba1(1234, 500.00);
    SavingAcct sa1(8888, 1025.00);

    transfer(ba1, sa1, 75.00);

    ba1.balance();
    sa1.balance();
    return 0;
  }
}
```

`transfer()` can work with `SavingAcct` object!

Pitfalls and Rules of thumb

- Beware:
  - Do not overuse inheritance
  - Do not overuse `protected`
    - Make sure it is something inherent for future sub class

- To determine whether it is correct to inherit:
  - Use the "**is-a**" rules of thumb
    - If "B is-a A" sounds right,
      then ***B is a subclass of A***
  - Frequently confused with the "**has-a**" rule
    - If "B has-a A" sounds right,
      then ***B should have an A attribute***

Rules of thumb: “is-a” and “has-a”

```
class BankAcct {
    ... ...
};

class SavingAcct : public BankAcct {
      ... ...

};
```

Inheritance: Saving Account IS-A Bank Acccount

```
class BankAcct {
    ... ...
};

class Person {

    BankAcct _customerAcct;
};
```

Attribute: Person HAS-A Bank Acccount

Polymorphism

Poly = Many

Morphism = Forms

Overview

- Substitution Principle
  - A more in depth discussion

- Method Binding:
  - Static Binding
  - Dynamic Binding (Polymorphism)
    - Syntax on virtual method

Interacting with Objects in C++

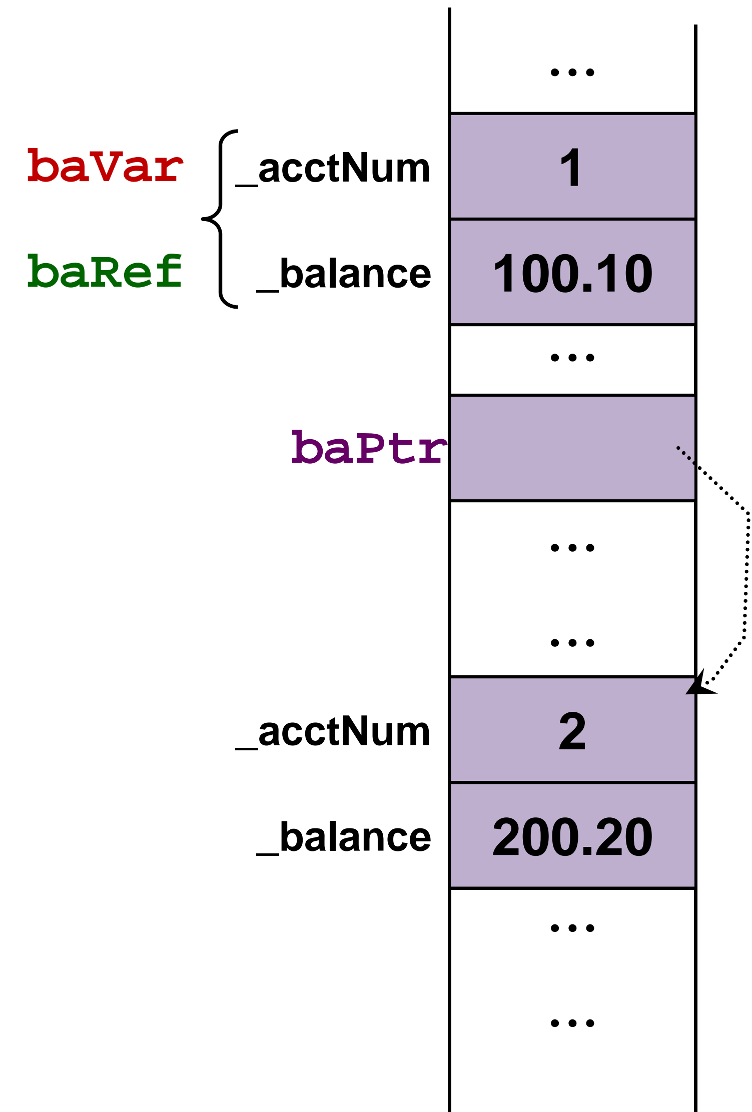
- There are 3 ways to refer to an object in C++:

```
class BankAcct { //...definition not shown };

int main() {

    BankAcct baVar(1, 100.10);

    BankAcct *baPtr;
    baPtr = new BankAcct(2, 200.20);

    BankAcct& baRef = baVar;
    return 0;
}
```

1st Way: Object Variable

2nd Way: Object Pointer

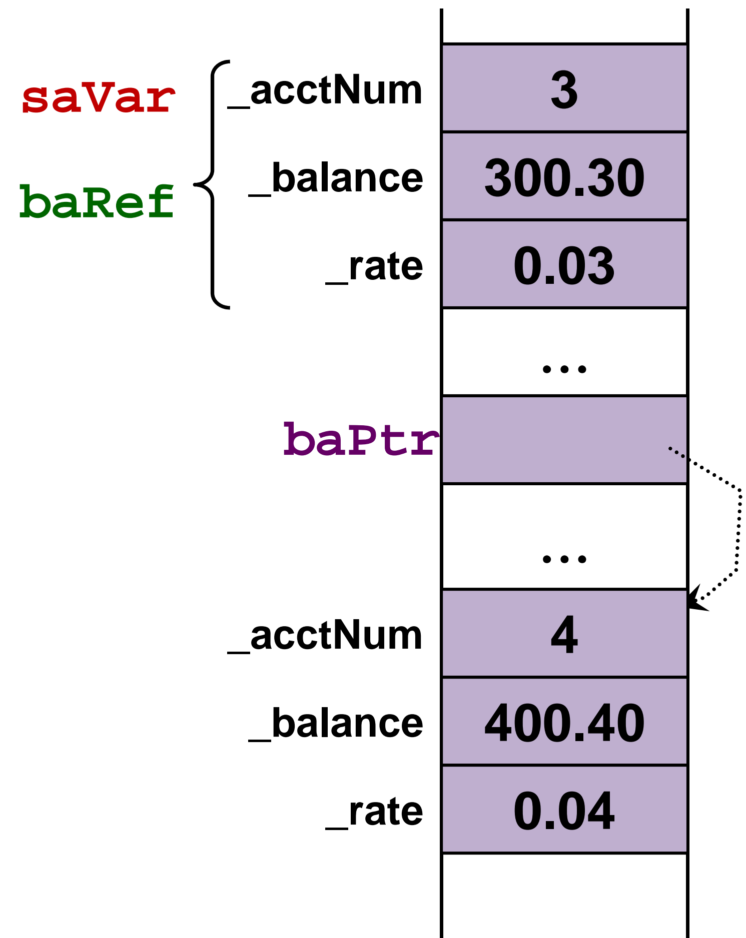
3rd Way: Object Reference



Subclass Substitution Principle

- Object pointer and reference of class type **A**:
  - Can refer to objects of type A
  - **Can also refer to objects of subclass of A**

```
class BankAcct { ... };
class SavingAcct: public BankAcct { ... };

int main() {
    SavingAcct saVar(3, 300.30, 0.03);

    BankAcct *baPtr;
    baPtr = new
          SavingAcct(4, 400.40, 0.04);

    BankAcct& baRef = saVar;

    return 0;
}
```



Polymorphism: **Basic Idea**

- Since we know:
  1. A superclass pointer/reference **can refer to an object of subclass**
  2. A method implementation can be overridden in the subclass, resulting in multiple versions
  - **Question:**
    - If we invoke a method using pointer/reference, which version of the method should be invoked?
- C++ provides **two possibilities:**
  - **Static Binding**
  - **Dynamic Binding**:
    - Also known as **polymorphism**

Static Binding

- Use **the class type of the pointer/reference** to determine which version of method to call:
  - This information is known during compilation time

```
class BankAcct {
void balance() {
    // Print out Acct No
    // and Balance
}
};
```

```
class SavingAcct : public BankAcct {
void balance() {
    // Print out Acct No, Balance
    // and interest rate
}
};
```

```
int main() {
    BankAcct *baPtr;
    baPtr = new SavingAcct(1, 100.10, 0.01);

    baPtr->balance();
    ......
}
```

Output:
`Acc: 1, Bal: 100.1`

Explanation:
"`baPtr`" is of `BankAcct` type, the `BankAcct`'s `method`() is used

Dynamic Binding (Polymorphism)

- Use **the actual class type of the object** to determine which version of method to call:
  - This information is known only during run time

```
class BankAcct {
virtual void balance() {
    // Print out Acct No
    // and Balance
}
};
```

```
class SavingAcct : public BankAcct {
void balance() {
    // Print out Acct No, Balance
    // and interest rate
}
};
```

```
int main() {
    BankAcct *baPtr;
    baPtr = new SavingAcct(1, 100.10, 0.01);

    baPtr->balance();
    ......
}
```

Output:

```
Acc: 1, Bal: 100.1
Int: 0.01
```

Explanation:

"**baPtr**" points to a **SavingAcct** object, the **SavingAcct**'s **balance**() method is used

Dynamic Binding: **Syntax**

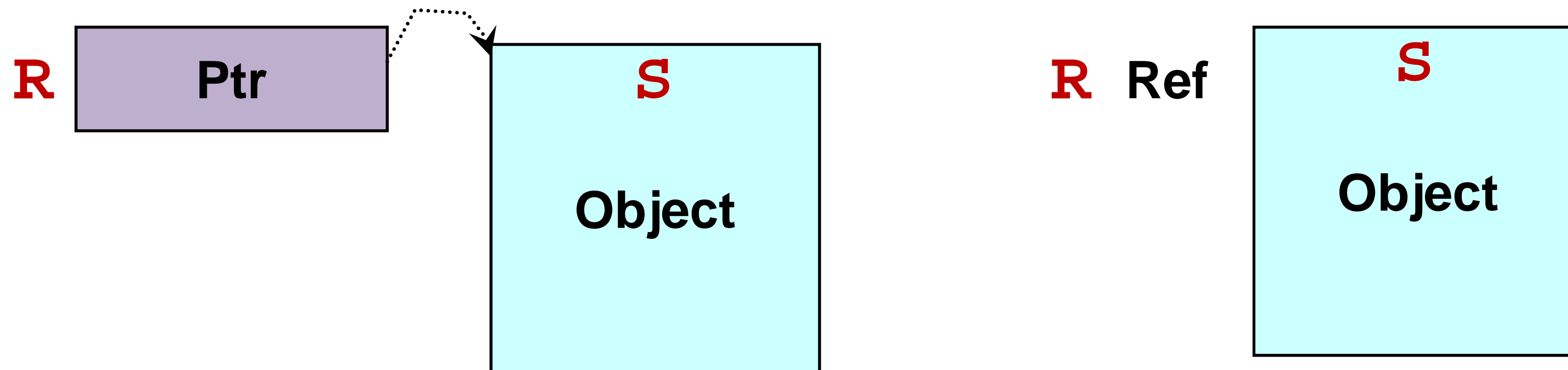
- To enable dynamic binding of a method in C++, add the "virtual" keyword before the method declaration

Syntax

```
virtual return_type method_name( [parameters] )
```

- Once a method is declared as virtual:
  - it will remain so in all descendant classes
  - → no need to restate the "`virtual`" keyword

Static VS Dynamic Binding: **illustration**



- **Static Binding:**
  - The class type `R` of pointer or reference is used to determine the method to call
- **Dynamic Binding:**
  - The class type `S` of object is used to determine the method to call

Polymorphism: Example

```
int main() {
    BankAcct *baPtr;
    int input;

    cout << "Account Type (1:Normal, 2:Saving):";
    cin >> input;

    if (input == 1 ){
        baPtr = new BankAcct(1234, 100);
    } else {
        baPtr = new SavingAcct(1234, 100, 0.03);
    }

    // balance() method should be declared
    // virtual in BankAcct class
    baPtr->balance();

}
```

Test Run 1:

```
Account Type …: 1
Acc: 1234, Bal: 100
```

Test Run 2:

```
Account Type …: 2
Acc: 1234, Bal: 100
Int: 0.03
```

Polymorphism: **Advantage**

- Make code reuse easier:
  - If a code is written to use a virtual method of a class A, the code can work with all future subclass of A with no modification
  - Furthermore, new/extended behavior of subclass of A can be incorporated by overriding the virtual method implementation
- For example:
  - Code that uses the virtual method **`balance()`** in **`BankAcct`** can work with all subclasses of **`BankAcct`** even when the **`balance()`** is overridden

Common Mistake

- A common error is to assume the actual type of the object is used to determine **validity of method invocation**

```
int main() {
    BankAcct *baPtr;
    baPtr = new SavingAcct(1, 100.10, 0.01);

    baPtr->payInterest();
    ......
}
```

Compilation Error! Why?

- The data type of the pointer/reference is used to determine validity of method invocation
  - The **`baPtr`** pointer has the type of **`BankAcct`**
  - **`BankAcct`** does not have ***`payInterest()`*** method ➔ error!

Polymorphism: **Summary**

- When we see the statement:
  - **`refR.`*`methodM`*`();`** OR
  - **`ptrR->`*`methodM`*`();`**

- **At compile time:**
  - If the class of **`refR/ptrR`** does not have a method **`methodM()`** ➔ **Compilation Error**
  - If **`methodM()`** is not a virtual method ➔ static binding ➔ **`methodM()`** in class of refR/ptrR is called

- **At run time:**
  - If **`methodM()`** is a virtual method ➔ dynamic binding ➔ **`methodM()`** in class of the actual object is called

Let's go meta……

ABSTRACT CLASS

Abstract Class: Motivation

- With inheritance and polymorphism, we gained the ability to **prepare for future expansion**:
  - e.g. code working with base class can work for future subclasses as well as overridden methods

- A new design possibility:
  - Design a base class that contains all essential methods of future subclasses
  - Sometimes, this base class is substantial enough to be a normal class
  - But, what if you want to define a base class that is simply a **placeholder (a mold)** for future subclasses?

Abstract Class & Method: **Syntax**

- In C++, an **abstract method** is a method without definition:
  - i.e. intended to be overridden in future subclasses
  - Also known as **pure virtual method**

Syntax

```
virtual return_type method_name([parameters]) = 0;
```

- A class with at least one abstract method is known as **abstract class** in C++
  - **Cannot** have object instantiated
  - Otherwise similar to a normal class definition:
    - Can have normal methods, constructors etc

Example: Redesigning Bank Account

- New design:
  - Bank account is now an **abstract class**
  - Two subclasses, the **saving account** and the **overdraft account**
  - Highlight only a few key changes

- Overdraft account:
  - Allow withdrawal to exceed balance (known as overdraft) up to a certain limit (overdraft limit)
  - Highlight different implementation of core functionalities

New Definition of BankAcct

```
class BankAcct {
protected:
   int _acctNum;
   double _balance;
public:
   // no change to constructors
   BankAcct(int aNum) {
      // no change ... }
   BankAcct(int aNum, double amt) {
      // no change ... }

   virtual int withdraw(double amount) = 0;
      // notice that method withdraw has no implementation

   virtual void deposit(double amount) {
      // no change ... }

   virtual void balance() {
      // no change ... }
   }
};
```

Question:

- What is the impact of these changes?
- Notice the various form of methods:
  - Normal, Virtual, Pure Virtual

Outline of SavingAcct & OverdraftAcct

```
class SavingAcct : public BankAcct {

protected:
  double _interest;

public:
  int withdraw(double amount) {
    // MUST implement to make
    // SavingAcct a normal class
  }

  void balance() {
    // Override example
    // Print out Acct No, Balance
    // and interest rate
  }
};
```

```
class OverdraftAcct : public BankAcct {

protected:
  double _limit; // Overdraft limit

public:
  int withdraw(double amount) {
    // MUST implement to make
    // OverdraftAcct a normal class
  }

  void balance() {
    // Override example
    // Print out Acct No, Balance
    // and overdraft limit
  }
};
```

- Note the implication of design of the base class in the subclasses
  - More details in ADT lecture (L5)

Summary

C++ Elements

```
- Inheritance
      superclass and subclass
      method overriding
      subclass substitutability

- Polymophism
      Dynamic Binding
```

Reference

- Carrano' s Book
  - Chapter 8: Advanced C++Topics