An Introduction to Programming though C++

Bernard L. Menezes

About These Slides

- Based on Chapter 25 of the book

An Introduction to Programming Through C++

by Abhiram Ranade (Tata McGraw Hill, 2014)

- Original slides by Abhiram Ranade

Categories and subcategories of objects

- Programs often have to deal with categories of objects, with some category itself containing subcategories.
- Example 1:
  - Category: Bank accounts
  - Subcategories: Savings accounts, current accounts.
  - Instances: specific accounts, e.g. my account
- Example 2:
  - Category: Geometric Shapes
  - Subcategories: Rectangles, Circles
  - Subcategory of Rectangles: Squares
  - Instances: specific shapes, e.g. circle with center (10,20) and radius 5.

Should categories/subcategories be represented in programming?

- Natural to make Category = type/class.
  - Subcategory = type/class?
- Suppose we make subcategories also types.
  - Current accounts and Savings accounts will both have data member `balance`, and functions members `deposit` and `withdraw`.
  - The `deposit` function may have the same code in current accounts and savings accounts, but the withdraw function in current accounts may permit some overdraft, i.e. withdrawing more money than you have in your account.
- Key question: Can we give the common code just once?
  - Generally it is not desirable to replicate code, if the similarity is not accidental.

Classes and subclasses

- C++ allows you to define subclasses of any class.
- If A is a subclass of B, then B is said to be the superclass of A.
- Subclass inherits the data and function members defined of its superclass
  - class called `Account` has data member `balance`, and member functions `deposit` and `withdraw`.
  - `CurrentAccount`: subclass of `Account`. `balance,deposit, withdraw` will be inherited.
  - But we can override implementation of `Withdraw`.
- One important gain: definitions and code are not duplicated.
- Other gains: later.
- Subclasses of a class can also have subclasses.
- A class can be defined as a subclass of several classes, in which case it inherits from all its superclasses.

Another motivational example

- Design a class `mTurtle` which is like `Turtle`, but which in addition has a member function `distanceCovered` which will return the distance covered by the turtle since its creation.
- Here is an example of a main program that we would like to write.

```
int main(){
  initCanvas();
  mTurtle m;
  m.forward(100);
  m.right(90);
  m.forward(50);
  cout << m.distanceCovered() << endl;
}
```

- This program should print 150.

Implementation using “Composition”

```
class mTurtleC{
  Turtle t;
  double distance;
public:
  mTurtleC(){ distance = 0; }

  forward(double d){
    distance += d;
    t.forward(d);
  }
  double distanceCovered(){ return distance; }
  void right(double angle){ t.right(angle);  }
  void left(double angle){  t.left(angle);   }
  // similar forwarding code for other functions
  // allowed on Turtle..
};
```

Implementation using inheritance

```
class mTurtleI :
 public Turtle{
  float distance;
public:
  mTurtleI(){
    distance = 0;
  }
  void forward(float d){
    distance += d;
    Turtle::forward(d);
  }
  float distanceCovered(){
    return distance;
  }
};
```

- The definition of `mTurtleI` does not need to have code for `right`, this code is inherited from `Turtle`. All functions that are defined for `Turtle` are inherited. Also data members.
- The member function `forward` is defined explicitly in `mTurtleI`. This definition overrides the definition that would have been inherited from `Turtle`.
- The overridden member function can be accessed as `Turtle::forward` if needed.
- Detailed explanation of inheritance soon.

Defining a class B as a subclass of class A

- The general form for this is:

```
class B : type-of-inheritance A {
  // Body.
  // describes how B is different from A.
}
```

- `Type-of-inheritance`: described later.
- `B` will have all members of `A`.
- Additional members can be specified in `Body`.
- The function members in `A` can be overridden in the `Body`.

Accessibility of members

- Definition of members is in sections named public, private, protected.
- Members in private sections can be accessed only in the current function definition.
- Members in public sections can be accessed outside of the class definition, and also in the subclass definitions.
- Members in protected sections can be accessed only in the current definition and subclass definitions.
- Is the definition of class B correct as per these rules?

```
class A{
  int p;
protected:
  int q;
  int getp(){ return p; }
public:
  int r;
  void init(){p=1;q=2;r=3;}
};

class B: public A{
  double s;
public:
  void print(){
   cout << p << q << r << getp();
 }
};
```

Example (continued)

```
int main(){
  B b;
  b.init();
  cout << b.p      // error: p is private
       << b.q      // error: q is protected
       << b.r      // OK
       << b.getp()
               // error: getp is protected.
       << endl;
  b.print();       // OK
}
// All errors will be flagged by the compiler.
```

Action of constructors

- The constructors are not inherited.
- The constructor of a subclass B of class A can be defined as follows within the body of B:

```
B(constructor-arguments) :
  call-to-constructor-of-A, initialization-list
{ constructor body }
```

- `Execution order:`
  - `call-to-constructor-of-A`. This initializes the members inherited into B from A.
  - Initialization of the members in B ias per the initialization list.
  - After that the constructor body is executed.
- `call-to-constructor-of-A` can be omitted in the code above; if so, the default constructor of A is called.

Example

```
class A{
public:
  int p;
  A(int x){ p = x;}
};

class B: public A{
public:
  int q,r;
  B(int x) : A(x), q(x*x){
    r = 10;
  }
};
int main(){
  B b(5);
  cout << b.p << b.q << b.r << endl;
}

// Will print 52510.
```

Destructors

- The destructor is not inherited.
- The destructor of a class is called, and it destroys the data members defined in the class by calling their destructors.
- After this the destructor of the superclass is called.
- As always, destructors should not be called explicitly.

Other operations on subclass objects

- Assignment operators are also not inherited.
- As always a default assignment operator that does member-by-member copy is defined by the compiler. You may override this.

Polymorphism: motivation

- Suppose I want to perform operation f on all objects of a certain category
  - Add interest to all accounts
  - Draw a set of shapes on the screen
- Natural implementation strategy
  - Store the objects in a list L
  - Apply f to each element of L
- Is L a list of class associated with the category or the subcategories?

Types and inheritance

- If object x is of class X which is a subclass of Y, then x has type both X and Y!
- An object of a subclass can be assigned to a variable of its superclass. In this case only the members of the superclass get assigned (unless the assignment operator is explicitly overridden).
- The address of a subclass object can be stored in a pointer variable of the superclass. In this case only the superclass members of the object are accessible through the pointer by default.

Example

```
class A{
  void f(){
   cout <<“a”;
  }
};
class B: public A{
  void f(){
   cout <<“b”;
  }
};
```

```
int main(){
  A *L[10];
  A a;
  B b;
  L[0] = &a;
  L[1] = &b;
  // allowed!
  for(int i=0;i<2;i++)
    L[i]->f();
  // Will print “aa”.
}
```

Motivation: Virtual functions

- The example shows that addresses of objects of a subclass can be stored in pointers of type superclass.
- When dereferenced, the definitions in the superclass get used.
- What if we want the definitions in the subclass to be used? This will often be convenient.
- Can be done by prefixing the function definition by the keyword `virtual`.

Example - 2

```
class A{
  virtual
  void f(){
   cout <<“a”;
  };
class B: public A{
  void f(){
   cout <<“b”;
  }
};
```

```
int main(){
  A *L[10];
  A a;
  B b;
  L[0] = &a;
  L[1] = &b;
  // allowed!
  for(int i=0;i<2;i++)
    L[i]->f();
  // Will print “ab”.
}
```

Another example

```
class Flower{
public:
  void whoAmI(){ cout << name() << endl; }
  virtual string name(){ return "Flower"; }
};

class Rose: public Flower{
public:
  string name(){ return "Rose"; }
};

int main(){
  Flower a;
  Rose b;
  a.whoAmI();  // will print “Flower”
  b.whoAmI();  // will print “Rose”
}
```

Remarks

- A pointer variable that can hold pointers to objects of a class or its subclasses is said to be polymorphic.
- If a reference can refer to objects of a class or its subclasses, it is also polymorphic.
- Because of polymorphism, we can consider the type of an object to be either its class, or its subclass.
- Because of polymorphism, we can effectively place objects of a class or its subclasses into a single queue.
  - Note: the queue must hold pointers to objects, not objects themselves.

Abstract classes

- Sometimes we might define a class, which will have subclasses, but of which we do not expect to create instances.
- Example:
  - We may create a class `Account` and create subclasses of it called `currentAccount` and `savingsAccount`.
  - We may not wish to create instances of class `Account`; it may make sense only to create instances of classes `currentAccount` and `savingsAccount`.
- Classes of which we do not create instances are said to be abstract.
  - We can declare a class to be abstract by specifying at least one virtual member function definition as "`= 0;`".

```
class Account{ virtual double withdraw(double amount) = 0;}
```

- If we just declare a member function in a class definition, it leaves open the possibility that its implementation will be specified outside the definition.
  - If such a definition is not given outside, then the linker will complain.
  - Defining a member function as `= 0;` tells the linker: the definition of this function will not be given outside of the body either. Knowing this case the linker will not complain.
  - Such a function must be virtual.

Abstract classes - 2

- We can declare a class to be abstract by specifying at least one virtual member function definition as "`= 0;`".

```
class Account{
virtual double withdraw(double amount) = 0;
}
```

- "= 0" seems ugly. Why not just omit the implementation?
- Omitting the implementation is interpreted as: "implementation will be specified outside the definition".
  - If such a definition is not given outside, then the linker will complain.
  - Defining a member function as `= 0;` tells the linker: the definition of this function will not be given outside of the body either. Knowing this case the linker will not complain.
  - Such a function must be virtual.

Types of Inheritance

- `public` : What has been discussed so far.
- `protected`: The public and protected members of the superclass become protected members of the subclass.
- `private`: The public and protected members become private members of the subclass.
- Public inheritance is by far most common, so we will not discuss others in more detail.

Concluding remarks

- Whenever your program deals with objects belonging to categories and subcategories, use classes and subclasses.
- If a member belongs to several subclasses, consider creating a superclass of them and declare the member in that superclass. If the same definition will work for all subclasses, then put the definition also in the superclass.
- The definition of a member function can be overridden in subclasses.
- Polymorphism can be obtained using virtual classes. This enables us to think of an object as belonging to its class, or its superclass, depending upon our convenience.
- The book gives more examples, and also discusses ideas such as multiple inheritance and virtual destructors.
- Chapter 26 also gives more examples.