Object Oriented Programming: Inheritance

Shahram Rahatlou

STVDIVM VRBIS

SAPIENZA
UNIVERSITÀ DI ROMA

`http://www.roma1.infn.it/people/rahatlou/programmazione++/`

Corso di Programmazione++

Roma, 18 May 2009

Today's Lecture

- Introduction to elements of object oriented programming (OOP)
  - Inheritance
  - Polymorphism

- Base and Derived Classes

- Inheritance as a mean to provide common interface

What is Inheritance?

- Powerful approach to reuse software without too much re-writing

- Often several types of object are in fact special cases of a basic type
  - keyboard and files are different types of an input stream
  - screen and file are different types of output stream
  - Resistors and capacitors are different types of circuit elements
  - Circle, square, ellipse are different types of shapes
  - In StarCraft, engineers, builders, soldiers are different types of units

- Inheritance allows to define a “base” class that provides basic functionalities to “derived” classes
  - Derived classes can extend the base class by adding new data members and functions

Inheritance: Student “is a” Person

```
// example1.cpp
#include <string>
#include <iostream>
using namespace std;


class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name
           << ") called" << endl;
    }

    ~Person() {
      cout << "~Person() called for "
           << name_ << endl;
     }

    string name() const { return name_; }

    void print() {
      cout << "I am a Person. My name is “
           << name_ << endl;
    }

  private:
    string name_;
};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) :
      Person(name) {
      id_ = id;
      cout << "Student(" << name
           << ", " << id << ") called"
           << endl;
    }

    ~Student() {
      cout << "~Student() called for name:"
           << name() << " and id: " << id_
           << endl;
     }

     int id() const { return id_; }

  private:
    int id_;
};
```

A more compact mode equivalent to

```
Student(const string& name, int id) {
      Person(name);
      id_ = id;
}
```

Example of Inheritance in Use

```cpp
// example1.cpp

int main() {

  Person* john = new Person("John");
  john->print();

  Student* susan = new Student("Susan", 123456);

  susan->print();
  cout << "name: " << susan->name() << " id: " << susan->id() << endl;

  delete john;
  delete susan;

  return 0;
}
```

```
$ ./example1
Person(John) called
I am a Person. My name is John
Person(Susan) called
Student(Susan, 123456) called
I am a Person. My name is Susan
name: Susan id: 123456
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```

Student "behaves as" Person

```
  Person* john = new Person("John");
  john->print();

  Student* susan = new Student("Susan", 123456);
  susan->print();
  cout << "name: " << susan->name()
       << " id: " << susan->id()
       << endl;

  delete john;
  delete susan;

  return 0;
}
```

print() and name() are methods of Person

id() is a method of Student

- Methods of **Person** can be called with an object of type **Student**
  - Functionalities implemented for **Person** available for free
  - No need to re-implement the same code over and over again
  - If a functionality changes, we need to fix it just once!

Student is an "extension" of Person

```
class Student : public Person {
  public:

     int id() const { return id_; }

  private:
    int id_;
};
```

id() is a method of Student

```
  Person* john = new Person("John");
  john->print();

  Student* susan = new Student("Susan", 123456);
  susan->print();
  cout << "name: " << susan->name()
       << " id: " << susan->id()
       << endl;

  delete john;
  delete susan;

  return 0;
}
```

- Student provides all functionalities of Person **and more**
- Student has additional data members and member functions
- Student is an extension of Person but not limited to be the same

Typical Error: **Person** is not **Student**!

```
// bad1.cpp

int main() {

  Person* susan = new Student("Susan", 123456);
  cout << "name: " << susan->name() << endl;
  cout << "id: " << susan->id() << endl;

  delete susan;

  return 0;
}
```

susan is a pointer to **Person** but initialized by a **Student**!

OK… because a **Student** is also a **Person**!
elements of polymorphism

```
$ g++ -o bad1 bad1.cpp
bad1.cpp: In function `int main()':
bad1.cpp:53: error: 'class Person' has no member named 'id'
```

- You can not use methods of **Student** on a **Person** object
  - Inheritance is a one-way relation
  - **Student** knows to be derived from **Person**
  - **Person** does not know who could be derived from it

- You can treat a **Student** object (***susan**) as a **Person** object

Student cannot Access Everything in Person

```
class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name
           << ") called" << endl;
    }

    ~Person() {
      cout << "~Person() called for "
           << name_ << endl;
     }

    string name() const { return name_; }

    void print() {
      cout << "I am a Person. My name is “
           << name_ << endl;
    }

  private:
    string name_;
};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) :
      Person(name) {
      id_ = id;
      cout << "Student(" << name
           << ", " << id << ") called"
           << endl;
    }

    ~Student() {
      cout << "~Student() called for name:"
           << name() << " and id: " << id_
           << endl;
     }

     int id() const { return id_; }

  private:
    int id_;
};
```

Student can use only public methods and data of **Person** like anyone else

No special access privilege… as usual access can be granted not taken

public and private in public inheritance

- Student is derived from Person through public inheritance

```
class Student : public Person {
  public:

  private:
};
```

private and protected inheritance are possible but rare and will not be discussed here

- All public members of Person become public members of Student as well
  - Both data and functions

- Private members of Person REMAIN private and not accessible directly by Student
  - Access provided only through public methods (getters)
- You don’t need to access source code of a class to inherit from it!
  - Use public inheritance and add new data members and functions

protected members

- **protected** members become **protected** members of derived classes
  - **Protected** is somehow between **public** and **private**

```
class Person {
  public:
    Person(const string& name, int age) {
      name_ = name;
      age_ = age;
      cout << "Person(" << name << ", "
           << age << ") called" << endl;
    }
    ~Person() {
      cout << "~Person() called for "
           << name_ << endl;
     }

    string name() const { return name_; }
    int age() const { return age_; }
    void print() {
      cout << "I am a Person. name: " << name_
           << " age: " << age_ << endl;
    }

  private:
    string name_;

  protected:
    int age_;

};
```

```
class Student : public Person {
  public:
    Student(const string& name, int age,
            int id) :
                      Person(name,age) {
      id_ = id;
      cout << "Student(" << name << ", "
           << age  << ", " << id
           << ") called"
           << endl;
    }

    ~Student() {
      cout << "~Student() called for name:"
           << name()
           << " age: " << age_ << " and id: "
           << id_ << endl;
    }

    int id() const { return id_; }

  private:
    int id_;
};
```

protected members can be used by derived classes

Don't Abuse `protected`!

- Bad habit to make everything protected
  - Transfers responsibility for proper initialization and data handling to derived classes

- Base class should be complete and self-sufficient

- If something must be protected in base class for your derived class to work then almost always there is a mistake or bad design

- **`Person::name_`** has no reason to be protected!
  - Proper implementation of derived class must correctly use base class constructors

Constructors of Derived Classes

- Compiler calls default constructor of base class in constructors of derived class UNLESS you call explicitly a specific constructor

- Necessary to insure data members of the base class ALWAYS initialized when creating instance of derived class

```
class Student : public Person {
  public:
    Student(const string& name, int id) {
      id_ = id;
      cout << "Student(" << name << ", "
           << id << ") called" << endl;
    }

private:
    int id_;
};
```

Bad Programming!

Constructor of Student does not call constructor of Person

Compiler is forced to call Person() to make sure name_ is intialized correctly

Bad: we rely on default constructor to do the right thing

Common Error with Missing Constructors

```
class Person {
  public:
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name
           << ") called" << endl;
    }
    ~Person() {
      cout << "~Person() called for "
          << name_  << endl;
     }

private:
    string name_;
};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) {
      id_ = id;
      cout << "Student(" << name << ", "
           << id << ") called" << endl;
    }

  private:
    int id_;
};
```

```
// bad2.cpp

int main() {

  Person anna("Anna");

  Student* susan =
        new Student("Susan", 123456);
  susan->print();
  delete susan;

  return 0;
}
```

```
$ g++ -o bad2 bad2.cpp
bad2.cpp: In constructor
          `Student::Student(const std::string&, int)':
bad2.cpp:32: error: no matching function for call to
           `Person::Person()'
bad2.cpp:7: note: candidates are:
           Person::Person(const Person&)
bad2.cpp:9: note:  Person::Person(const std::string&)
```

No default constructor implemented for **`Person`**

Compiler can use a default one to make **`anna`**

But gives error dealing with derived classes.
You need to provide a default constructor or call one of the implemented constructors

Default Constructors are Crucial

- Very often you wondered why bother implementing the default constructors

- They play a crucial role for polymorphic objects

- Derived classes rely heavily on base-class constructors to initialize objects

- Empty default constructors are a bad habit. Use constructors for what they are meant: initialize properly all data members

Bad Working Example

```
class Person {
  public:
    Person() { } // default constructor
    Person(const string& name) {
      name_ = name;
      cout << "Person(" << name << ") called‟
           << endl;
    }

};
```

```
class Student : public Person {
  public:
    Student(const string& name, int id) {
      id_ = id;
      cout << "Student(" << name << ", ‟
           << id << ") called" << endl;
    }

};
```

```
// bad3.cpp

int main() {

  Student* susan =
       new Student("Susan", 123456)
  susan->print();

  delete susan;

  return 0;
}
```

```
$ g++ -o bad3 bad3.cpp
$ ./bad3
Student(Susan, 123456) called
I am a Person. My name is
~Student() called for name: and id: 123456
~Person() called for
```

- Default constructor is called by compiler
- No name assigned to student by default
- Code compiles and runs but bad behavior

Destructors

- Similar to constructors
- Compiler calls the default destructor of base class in destructor of derived class
- No compilation error if destructor of base class not implemented
    - Default will be used but…
- Extremely important to implement correctly the destructors to avoid memory leaks!

Member Functions of Derived Classes

- Derived classes can also overload functions provided by the base class
    - Same signature but different implementation

```
class Person {
  public:
    void print() {
      cout << "I am a Person. My name is "
           << name_ << endl;
    }

  private:
    string name_;
};
```

```
class Student : public Person {
  public:
    void print() {
      cout << "I am Student "
           <<  name()
           << " with id " << id_
           << endl;
    }

  private:
    int id_;
};
```

Overloading Methods from Base Class

```
// example3.cpp
#include <string>
#include <iostream>
using namespace std;

int main() {

  Person* john = new Person("John");
  john->print();  // Person::print()

  Student* susan = new Student("Susan", 123456);
  susan->print(); // Student::print()
  susan->Person::print(); // Person::print()

  Person* p2 = susan;
  p2->print(); // Person::print()

  delete john;
  delete susan;

  return 0;
}
```

Compiler calls the correct version of **`print()`** for **`Person`** and **`Student`**

We can use **`Person::print()`** implementation for a **`Student`** object by specifying its scope

Remember: a function is uniquely identified by its namespace and class scope

```
$ g++ -o example3 example3.cpp
$ ./example3
Person(John) called
I am a Person. My name is John

Person(Susan) called
Student(Susan, 123456) called
I am Student Susan with id 123456
I am a Person. My name is Susan

I am a Person. My name is Susan

~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```