Object Oriented Programming: Polymorphism

Shahram Rahatlou

Computing Methods in Physics

`http://www.roma1.infn.it/people/rahatlou/cmp/`

Anno Accademico 2019/20

SAPIENZA
UNIVERSITÀ DI ROMA

Abstract Class

Pure virtual Functions

- virtual functions with no implementation
  - All derived classes ***are required*** to implement these functions
- Typically used for functions that can't be implemented (or at least in an unambiguous way) in the base case
- Class with at least one pure virtual method is called an "Abstract" class

= 0 is called pure specifier

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;

  private:
    std::string name_;
};
```

```
#include "Function.h"

Function::Function(const std::string& name) {
  name_ = name;
}
```

ConstantFunction

```
#ifndef ConstantFunction_h
#define ConstantFunction_h

#include <string>
#include "Function.h"

class ConstantFunction : public Function {
  public:
    ConstantFunction(const std::string& name, double value);
    virtual double value(double x) const;
    virtual double integrate(double x1, double x2) const;

  private:
    double value_;
};
```

```
#include "ConstantFunction.h"

ConstantFunction::ConstantFunction(const std::string& name, double value) :
  Function(name) {
  value_ = value;
}

double ConstantFunction::value(double x) const {
  return value_;
}

double ConstantFunction::integrate(double x1, double x2) const {
  return (x2-x1)*value_;
}
```

Typical Error with Abstract Class

```
// func1.cpp
#include <string>
#include <iostream>
using namespace std;

#include "Function.h"

int main() {

  Function* gauss  = new Function("Gauss");

  return 0;
}
```

Cannot make an object of an Abstract class!

Pure virtual methods not implemented and the class is effectively incomplete

```
$ g++ -o /tmp/app func1.cpp Function.cc
func1.cpp:10:22: error: allocating an object of abstract class type 'Function'
  Function* gauss  = Function("Gauss");
                     ^
./Function.h:9:20: note: unimplemented pure virtual method 'value' in 'Function'
    virtual double value(double x) const = 0;
                   ^
./Function.h:10:20: note: unimplemented pure virtual method 'integrate' in 'Function'
    virtual double integrate(double x1, double x2) const = 0;
                   ^
1 error generated.
```

`virtual` and pure `virtual`

- No default implementation for pure virtual
  - Requires explicit implementation in derived classes

- Use pure virtual when
  - Need to enforce policy for derived classes
  - Need to guarantee public interface for all derived classes
  - You expect to have certain functionalities but too early to provide default implementation in base class
  - Default implementation can lead to error
    - User forgets to implement correctly a virtual function
    - Default implementation is used in a meaningless way

- Virtual allows polymorphism

- Pure virtual forces derived classes to ensure correct implementation

Abstract and Concrete Classes

- Any class with at least one pure virtual method is called an Abstract Class
  - Abstract classes are incomplete
    - At least one method not implemented
    - Compiler has no way to determine the correct size of an incomplete type
  - ***Cannot instantiate an object of Abstract class***

- Usually abstract classes are used in higher levels of hierarchy
  - Focus on defining policies and interface
  - Leave implementation to lower level of hierarchy

- Abstract classes used typically as pointers or references to achieve polymorphism
  - Point to objects of sub-classes via pointer to abstract class

Example of Bad Use of `virtual`

```
class BadFunction {
  public:
    BadFunction(const std::string& name);
    virtual double value(double x) const { return 0; }
    virtual double integrate(double x1, double x2) const { return 0; }

  private:
    std::string name_;
};
```

Default dummy implementation

```
class Gauss : public BadFunction {
  public:
    Gauss(const std::string& name, double mean, double width);

    virtual double value(double x) const;
    //virtual double integrate(double x1, double x2) const;

  private:
    double mean_;
    double width_;
};
```

Implement correctly **`value()`** but use default **`integrate()`**

We can use ill-defined **`BadFunction`** and wrongly use **`Gauss`**!

```
// func2
int main() {

  BadFunction f1  = BadFunction("bad");
  Gauss g1("g1",0.,1.);
  cout << "g1.value(2.): " << g1.value(2.) << endl;
  cout << "g1.integrate(0.,1000.): "
       << g1.integrate(0.,1000.) << endl;
  return 0;
}
```

```
$ g++ -o /tmp/app func2.cpp
        {BadFunction,Gauss,Function}.cc
$ /tmp/app
g1.value(2.): 0.0540047
g1.integrate(0.,1000.): 0
```

Function and BadFunction

```
class BadFunction {
  public:
    BadFunction(const std::string& name);
    virtual double value(double x) const { return 0; }
    virtual double integrate(double x1, double x2) const { return 0; }

  private:
    std::string name_;
};
```

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;

  private:
    std::string name_;
};
```

```
// func3.cpp
int main() {

  BadFunction f1  = BadFunction("bad");
  Function f2("f2");

  return 0;
}
```

Cannot instantiate Function because abstract

Bad Function can be used

```
$ g++ -o /tmp/app func3.cpp {BadFunction,Function}.cc
func3.cpp:13:12: error: variable type 'Function' is an abstract class
  Function f2("f2");
           ^
./Function.h:9:20: note: unimplemented pure virtual method 'value' in 'Function'
    virtual double value(double x) const = 0;
                   ^
./Function.h:10:20: note: unimplemented pure virtual method 'integrate' in 'Function'
    virtual double integrate(double x1, double x2) const = 0;
                   ^
1 error generated.
```

Use of `virtual` in Abstract Class `Function`

```
class Function {
  public:
    Function(const std::string& name);
    virtual double value(double x) const = 0;
    virtual double integrate(double x1, double x2) const = 0;
    virtual void print() const;
    virtual std::string name() const { return name_; }

  private:
    std::string name_;
};
```

```
#include "Function.h"
#include <iostream>

Function::Function(const std::string& name) {
  name_ = name;
}

void
Function::print() const {
  std::cout << "Function with name "
            << name_ << std::endl;
}
```

Default implementation of name()

Unambiguous functionality: user will always want the name of the particular object regardless of its particular subclass

print() can be overriden in sub-classes to provide more details about sub-class but still a function with a name

Concrete Class Gauss

```
#include "Gauss.h"
#include <cmath>
#include <iostream>
using std::cout;
using std::endl;

Gauss::Gauss(const std::string& name,
           double mean, double width) :
  Function(name) {
  mean_ = mean;
  width_ = width;
}

double Gauss::value(double x) const {
  double pull = (x-mean_)/width_;
  double y = (1/sqrt(2.*3.14*width_)) * exp(-pull*pull/2.);
  return y;
}

double Gauss::integrate(double x1, double x2) const {
  cout << "Sorry. Gauss::integrate(x1,x2) not implemented yet..."
       << "returning 0. for now..." << endl;
  return 0;
}

void
Gauss::print() const {
  cout << "Gaussian with name: " << name()
       << " mean: " << mean_
       << " width: " << width_
       << endl;
}
```

```
#ifndef Gauss_h
#define Gauss_h

#include <string>
#include "Function.h"

class Gauss : public Function {
  public:
    Gauss(const std::string& name,
     double mean, double width);

    virtual double value(double x) const;
    virtual double integrate(double x1,
                    double x2) const;
    virtual void print() const;

  private:
    double mean_;
    double width_;
};
#endif
```

```
int main() {

  Function* g1  = new Gauss("gauss",0.,1.);
  g1->print();
  double x = g1->integrate(0., 3.);

  return 0;
}
```

```
$ g++ -o /tmp/app func4.cpp {Gauss,Function}.cc
$ /tmp/app
Gaussian with name: gauss mean: 0 width: 1
Sorry. Gauss::integrate(x1,x2) not implemented yet...returning 0. for now...
```

Problem with destructors

- We now properly delete the Gauss object

```
// func5.cpp
int main() {

  Function* g1  = new Gauss("gauss",0.,1.);
  g1->print();
  double x = g1->integrate(0., 3.);

  delete g1;

  return 0;
}
```

```
$ g++ -o /tmp/app func5.cpp {Gauss,Function}.cc
$ g++ -o /tmp/app func5.cpp {Gauss,Function}.cc
func5.cpp:15:3: warning: delete called on 'Function' that is abstract but has non-virtual destructor
      [-Wdelete-non-virtual-dtor]
  delete g1;
  ^
1 warning generated.
$ /tmp/app
Gaussian with name: gauss mean: 0 width: 1
Sorry. Gauss::integrate(x1,x2) not implemented yet...returning 0. for now...
Illegal instruction
```

- In general with polymorphism and inheritance it is a VERY GOOD idea to use virtual destructors

- Particularly important when using dynamically allocated objects in constructors of polymorphic objects

Revisit Person and Student

```
// example7.cpp
int main() {

 Person* p1  = new Student("Susan", 123456);
 Person* p2  = new GraduateStudent("Paolo", 9856, "Physics");


  delete p1;
  delete p2;

  return 0;
}
```

```
$ g++ -o /tmp/app example7.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
~Person() called for Susan
~Person() called for Paolo
```

Note that ~Person() is called and not that of the sub class!

We did not declare the destructor to be virtual

destructor called based on the pointer and not the object! Non-polymorphic behaviour

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

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

```
GraduateStudent::~GraduateStudent() {
  cout << "~GraduateStudent() called for name:" << name()
       << " id: " << id()
       << " major: " << major_ << endl;
}
```

virtual destructors

- Derived classes might allocate dynamically memory
  - Derived-class destructor (if correctly written!) will take care of cleaning up memory upon destruction
- Base-class destructor will not do the proper job if called for a sub-class object
- Declaring destructor to be virtual is a simple solution to prevent memory leak using polymorphism
- virtual destructors ensure that memory leaks don't occur when delete an object via base-class pointer

Simple Example of **`virtual`** Destructor

```
// noVirtualDtor.cc
#include <iostream>

using std::cout;
using std::endl;

class Base {
  public:
  Base(double x)  {
    x_ = new double(x);
    cout << "Base(" << x << ") called" << endl;
  }
  ~Base()  {
    cout << "~Base() called" << endl;
    delete x_;
  }
  private:
   double* x_;
};

class Derived : public Base {
  public:
  Derived(double x)  : Base(x){
    cout << "Derived("<<x<<") called" << endl;
  }
  ~Derived()  {
    cout << "~Derived() called" << endl;
  }
};

int main()  {
  Base* a = new Derived(1.2);
  delete a;
  return 0;
}
```

Destructor Not virtual

```
$ g++ -Wall -o /tmp/noVirtualDtor noVirtualDtor.cc
$ /tmp/noVirtualDtor
Base(1.2) called
Derived(1.2) called
~Base() called
```

```
// virtualDtor.cc
#include <iostream>

using std::cout;
using std::endl;

class Base {
  public:
  Base(double x)  {
    x_ = new double(x);
    cout << "Base(" << x << ") called" << endl;
  }
  virtual ~Base()  {
    cout << "~Base() called" << endl;
    delete x_;
  }
  private:
   double* x_;
};

class Derived : public Base {
  public:
  Derived(double x)  : Base(x){
    cout << "Derived("<<x<<") called" << endl;
  }
  virtual ~Derived()  {
    cout << "~Derived() called" << endl;
  }
};

int main()  {
  Base* a = new Derived(1.2);
  delete a;
  return 0;
}
```

Virtual Destructor

```
$ g++ -Wall -o /tmp/VirtualDtor VirtualDtor.cc
$ /tmp/VirtualDtor
Base(1.2) called
Derived(1.2) called
~Derived() called
~Base() called
```

Revised Class **Student**

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    ~Student();
    void addCourse(const std::string& course);
    virtual void print() const;

    int id() const { return id_; }
    const std::vector<std::string>* getCourses() const;
    void printCourses() const;

  private:
    int id_;
    std::vector<std::string>* courses_;
};
```

```
void Student::addCourse(const std::string&
course) {
  courses_->push_back( course );
}

void
Student::printCourses() const {
  cout << "student " << name()
       << " currently enrolled in following
courses:"
       << endl;

  for(int i=0; i<courses_->size(); ++i) {
    cout << (*courses_)[i] << endl;
  }
}

const std::vector<std::string>*
Student::getCourses() const {
  return courses_;
}
```

```
Student::Student(const std::string& name,
int id) :
  Person(name) {
  id_ = id;
  courses_ = new
std::vector<std::string>();
  cout << "Student(" << name << ", " << id
<< ") called"
       << endl;
}

Student::~Student() {
  delete courses_;
  courses_ = 0; // null pointer
  cout << "~Student() called for name:" <<
name()
       << " and id: " << id_ << endl;
}

void Student::print() const {
  cout << "I am Student " << name()
       << " with id " << id_ << endl;
  cout << "I am now enrolled in "
       << courses_->size() << " courses."
<< endl;
}
```

Example of Memory Leak with **Student**

```
// example8.cpp

int main() {

  Student* p1  = new Student("Susan", 123456);
  p1->addCourse(string("algebra"));
  p1->addCourse(string("physics"));
  p1->addCourse(string("Art"));
  p1->printCourses();

  Student* paolo   = new Student("Paolo", 9856);
  paolo->addCourse("Music");
  paolo->addCourse("Chemistry");

  Person* p2 = paolo;

  p1->print();
  p2->print();

  delete p1;
  delete p2;

  return 0;
}
```

Memory leak when deleting paolo because nobody deletes courses_

Need to extend polymorphism also to destructors to ensure that object type not pointer determine correct destructor to be called

```
$ g++ -o /tmp/app example8.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(Susan) called
Student(Susan, 123456) called
student Susan currently enrolled in following courses:
algebra
physics
Art
Person(Paolo) called
Student(Paolo, 9856) called
I am Student Susan with id 123456
I am now enrolled in 3 courses.
I am Student Paolo with id 9856
I am now enrolled in 2 courses.
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for Paolo
```

virtual Destructor for `Person` and `Student`

```
class Person {
  public:
    Person(const std::string& name);
    virtual ~Person();
    std::string name() const { return name_; }
    virtual void print() const;

  private:
    std::string name_;
};
```

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    virtual ~Student();
    void addCourse(const std::string& course);
    virtual void print() const;

    int id() const { return id_; }
    const std::vector<std::string>* getCourses() const;
    void printCourses() const;

  private:
    int id_;
    std::vector<std::string>* courses_;
};
```

Correct destructor is called using the base-class pointer to Student

```
// example9.cpp

int main() {

  Student* p1  = new Student("Susan", 123456);
  p1->addCourse(string("algebra"));
  p1->addCourse(string("physics"));
  p1->addCourse(string("Art"));
  p1->printCourses();

  Student* paolo   = new Student("Paolo", 9856);
  paolo->addCourse("Music");
  paolo->addCourse("Chemistry");
  Person* p2 = paolo;

  delete p1;
  delete p2;

  return 0;
}
```

```
$ g++ -o /tmp/app example9.cpp
                    {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(Susan) called
Student(Susan, 123456) called
student Susan currently enrolled in following courses:
algebra
physics
Art
Person(Paolo) called
Student(Paolo, 9856) called
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
```