Object Oriented Programming: Polymorphism

Shahram Rahatlou

SAPIENZA
UNIVERSITÀ DI ROMA

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

Corso di Programmazione++

Roma, 19 May 2009

Today's Lecture

- Polymorphism with inheritance hierarchy
- virtual and pure virtual methods
  - When and why use virtual or/and pure virtual functions
- virtual destructors
- Abstract and Pure Abstract classes
  - Providing common interface and behavior

Polymorphism

- Ability to treat objects of an inheritance hierarchy as belonging to the base class
  - Focus on common general aspects of objects instead of specifics

- Polymorphism allows programs to be general and extensible with little or no re-writing
  - resolve different objects of same inheritance hierarchy at runtime
  - Recall videogame with polymorphic objects Soldier, Engineer, Technician of same base class Unit
  - Can add new ‘types’ of Unit without rewriting application

- Base class provides interface common to all types in the hierarchy
- Application uses base class and can deal with new types not yet written when writing your application!

Polymorphism in OOP (from Wikipedia)

Polymorphism in object-oriented programming - Wikipedia, the free encyclopedia - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming

Headlines News Sport CMS Shahram CERN TuttoCitta` Wikipedia Corso Particelle Calendar Yahoo! Cal Google Now: Sunny, 63° F

Sign in / create account

article | discussion | edit this page | history

Your continued donations keep Wikipedia running!

WIKIPEDIA
The Free Encyclopedia

navigation
- Main Page
- Community Portal
- Featured articles
- Current events
- Recent changes
- Random article
- Help
- Contact Wikipedia
- Donations

search

Go Search

toolbox
- What links here
- Related changes
- Upload file
- Special pages
- Printable version
- Permanent link
- Cite this article

Polymorphism in object-oriented programming

From Wikipedia, the free encyclopedia

In object-oriented programming theory, **polymorphism** is the ability of objects belonging to different types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behaviour. The programmer (and the program) does not have to know the exact type of the object in advance, so this behavior can be implemented at run time (this is called *late binding* or *dynamic binding*).

The different objects involved only need to present a compatible interface to the clients (the calling routines). That is, there must be public methods with the same name and the same parameter sets in all the objects. In principle, the object types may be unrelated, but since they share a common interface, they are often implemented as subclasses of the same parent class. Though it is not required, it is understood that the different methods will also produce similar results (for example, returning values of the same type).

Advantages of polymorphism

[edit]

Polymorphism allows client programs to be written based only on the abstract interfaces of the objects which will be manipulated (interface inheritance). This means that future extension in the form of new types of objects is easy, if the new objects conform to the original interface. In particular, with object-oriented polymorphism, the original client program does not even need to be recompiled (only relinked) in order to make use of new types exhibiting new (but interface-conformant) behaviour. (In C++, for instance, this is possible because the interface definition for a class defines a memory layout, the virtual function table describing where pointers to functions can be found. Future, new classes can work with old, precompiled code because the new classes must conform to the abstract class interface, meaning that the layout of the new class's virtual function table is the same as before; the old, precompiled code can still look at the same memory offsets relative to the start of the object's memory in order to find a pointer to the new function. It is only that the new virtual function table points to a new implementation of the functions in the table, thus allowing new, interface-compliant behavior with old, precompiled code.)

Since program evolution very often appears in the form of adding new types of objects (i.e. classes), this ability to cope with and localize change that polymorphism allows is the key new contribution of object technology to software design.

Done

Examples of Polymorphism

- Application for graphic rendering
  - Base class **`Shape`** with **`draw()`** and **`move()`** methods
  - Application expects all shapes to have such functionality

- **`Function`** in Physics
  - We'll study this example in detail
  - **`Guassian`**, **`Breit-Wigner`**, **`polynomials`**, **`exponential`** are all functions
  - A **`Function`** must have
    - **`value(x)`**
    - **`integral(x1,x2)`**
    - **`primitive()`**
    - **`derivative()`**
  - Can write a fit application that can handle existing or not-yet implemented functions using a base class Function

Reminders about Inheritance

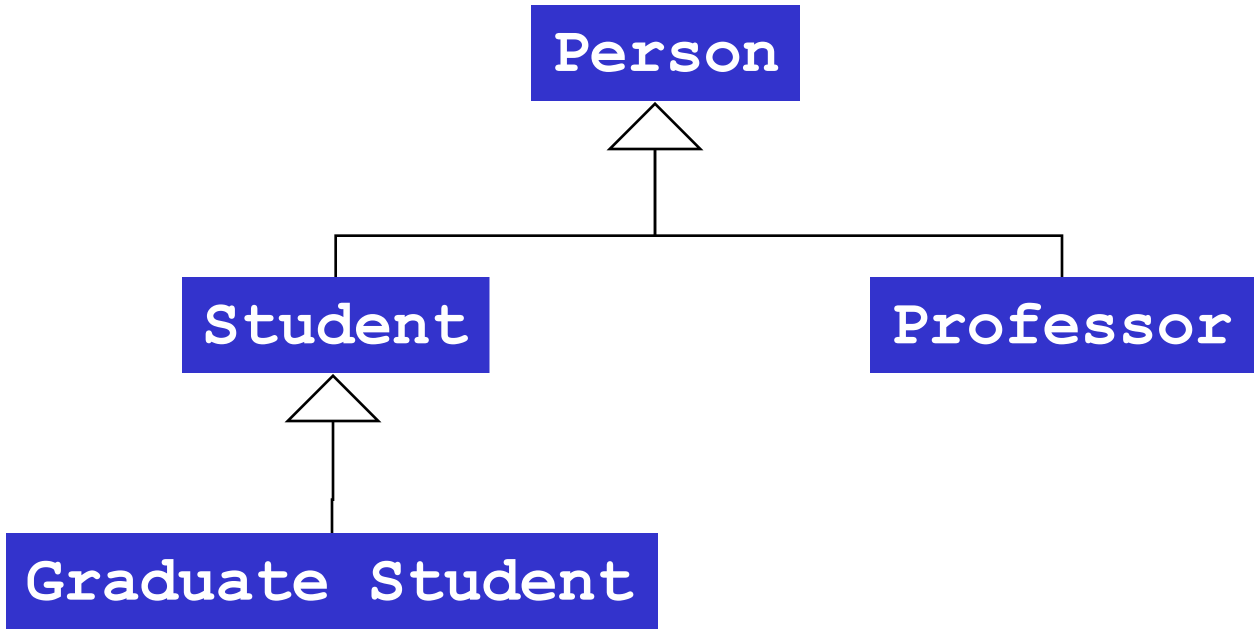
- Inheritance is a is-a relationship
  - Object of derived class 'is a' base class object as well

- Can treat a derived class object as a base class object
  - call methods of base class on derived class
  - can point to derived class object with pointer of type base class

- Base class does not know about its derived classes
  - Can not trat a base class object as a derived object

- Methods of base class can be redefined in derived classes
  - Same interface but different implementation for different types of object in the same hierarchy

`Person` Inheritance Hierarchy



Student and GraduateStudent

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

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

```
class Student : public Person {
  public:
    Student(const std::string& name, int id);
    ~Student();
    int id() const { return id_; }
    void print() const;

  private:
    int id_;
};
```

```
class GraduateStudent : public Student {
  public:
    GraduateStudent(const std::string& name, int id,
                    const std::string& major);
    ~GraduateStudent();
    std::string getMajor() const { return major_; }
    void print() const;

  private:
    std::string major_;
};
```

Example

```
// example1.cpp

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()

  GraduateStudent* paolo =
    new GraduateStudent("Paolo", 9856, "Physics");
  paolo->print();

  Person* p3 = paolo;
  p3->print();


  delete john;
  delete susan;

  return 0;
}
```

Can point to **Student** or **GraduateStudent** object with a pointer of type **Person**

Can treat **paolo** and **susan** as **Person**

Depending on the pointer different **print()** methods are called

```
$ g++ -Wall -o example1 example1.cpp *.cc
$ ./example1
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(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am GraduateStudent Paolo with id 9856 major in Physics
I am a Person. My name is Paolo
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```

Bad Mistake!
No delete for **paolo**!!

Memory Leak!

Problem with Previous Example

```cpp
// example1.cpp

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()

  GraduateStudent* paolo =
    new GraduateStudent("Paolo", 9856, "Physics");
  paolo->print();

  Person* p3 = paolo;
  p3->print();


  delete john;
  delete susan;

  return 0;
}
```

```
$ g++ -Wall -o example1 example1.cpp *.cc
$ ./example1
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(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am GraduateStudent Paolo with id 9856 major in Physics
I am a Person. My name is Paolo
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
```

- Call to method `print()` is resolved base on the type of the pointer
  - `print()` methods is determined by pointer not the actual type of object
- Desired feature: use generic `Person*` pointer but call appropriate `print()` method for `paolo` and `susan` based on ACTUAL TYPE of these objects

Desired Feature: Resolve Different Objects at Runtime

- We would like to use the same Person* pointer but call different methods based on the type of the object being pointed to
- We DO NOT want to use the scope operator to specify the function to call

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

Student* susan = new Student("Susan", 123456);
Person* p2 = susan;
p2->print(); // Person::print()

GraduateStudent* paolo =
  new GraduateStudent("Paolo", 9856, "Physics");

Person* p3 = paolo;
p3->print();
```

Same **`Person*`** pointer used for three different types of object in the same hierarchy

Same code used by types solved at runtime

```
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

Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am GraduateStudent Paolo with id 9856 major in Physics
```

Polymorphic Behavior

```
int main() {

 vector<Person*> people;

  Person* john = new Person("John");
  people.push_back(john);

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

  GraduateStudent* paolo = new GraduateStudent("Paolo", 9856, "Physics");
  people.push_back(paolo);

  for(int i=0;
     i< people.size(); ++i) {
    people[i]->print();
  }

  delete john;
  delete susan;
  delete paolo;

  return 0;
}
```

vector of generic type Person
No knowledge about specific types

Different derived objects stored in the vector of Person

Generic call to print()

```
$ g++  -o example2 example2.cpp *.cc
$ ./example2
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for John
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~GraduateStudent() called for name:Paolo id: 9856 major: Physics
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
```

Different functions called based on the real type of objects pointed to!!

How? virtual functions!

`virtual` functions

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

  private:
    int id_;
};
```

```
class GraduateStudent : public Student {
  public:
    GraduateStudent(const std::string& name, int id, const std::string& major);
    ~GraduateStudent();
    std::string getMajor() const { return major_; }
    virtual void print() const;

  private:
    std::string major_;
};
```

- Virtual methods of base class are overridden NOT redefined by derived classes
  - if not overriden base class function called
- Type of objects pointed to determine which function is called
- Type of pointer (also called handle) has no effect on the method being executed
- `virtual` allows polymorphic behavior and generic code without relying on specific objects

Dynamic (or late) binding

- Choosing the correct derived class function at run time based on then type of the object being pointed to, regardless of the pointer type, is called dynamic binding or late binding

- Dynamic binding works only with pointers and references not using dot-member operators
  - static binding: function calls resolved at compile time

```
// example3.cpp

int main() {

  Person john("John");
  Student susan("Susan", 123456);
  GraduateStudent paolo("Paolo",
      9856, "Physics");

  john.print();
  susan.print();
  paolo.print();

  return 0;
}
```

static binding

```
$ g++ -o example3 example3.cpp *.cc
$ ./example3
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
~GraduateStudent() called for name:Paolo id: 9856 major: Physics
~Student() called for name:Paolo and id: 9856
~Person() called for Paolo
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for John
```

Another Example of Dynamic Binding

```
// example4.cpp

  Person* john = new Person("John");
  Person* susan = new Student("Susan", 123456);
  Person* paolo = new GraduateStudent("Paolo", 9856, "Physics");

  (*john).print();
  (*susan).print();
  (*paolo).print();

  john->print();
  susan->print();
  paolo->print();
```

```
$ ./example4
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for John
~Person() called for Susan
~Person() called for Paolo
```

Example: `virtual` Function at Runtime

```
int main() {

  Person* p = 0;
  int value = 0;
  while(value<1 || value>10) {
    cout << "Give me a number [1,10]: ";
    cin >> value;
  }
  cout << flush; // write buffer to output
  cout << "make a new derived object..." << endl;
  if(value>5) p = new Student("Susan", 123456);
  else        p = new GraduateStudent("Paolo", 9856, "Physics");

  cout << "call print() method ..." << endl;

  p->print();

  delete p;
  return 0;
}
```

Type of object decided at runtime by user.

Compiler does not know what object will be used

```
$ ./example6
Give me a number [1,10]: 3
make a new derived object...
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
call print() method ...
I am GraduateStudent Paolo with id 9856 major in Physics
~Person() called for Paolo

$ ./example6
Give me a number [1,10]: 9
make a new derived object...
Person(Susan) called
Student(Susan, 123456) called
call print() method ...
I am Student Susan with id 123456
~Person() called for Susan
```

Virtual methods allow dynamic binding at runtime

Default for Virtual Methods

```
class Professor : public Person {
  public:
    Professor(const std::string& name,
              const std::string& department);
    ~Professor();
    std::string department() const { return department_; }
    //virtual void print() const;

  private:
    std::string department_;
};
```

`print()` not overriden in `Professor`

`Person::print()` used by default

```
// example5.cpp

int main() {

  Person john("John");
  Student susan("Susan", 123456);
  GraduateStudent
   paolo("Paolo", 9856, "Physics");
  Professor
      bob("Robert", "Biology");

  john.print();
  susan.print();
  paolo.print();
  bob.print();

  return 0;
}
```

```
$ g++ -o example5 example5.cpp *.cc
$ ./example5
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
Person(Paolo) called
Student(Paolo, 9856) called
GraduateStudent(Paolo, 9856,Physics) called
Person(Robert) called
Professor(Robert, Biology) called
I am a Person. My name is John
I am Student Susan with id 123456
I am GraduateStudent Paolo with id 9856 major in Physics
I am a Person. My name is Robert
```

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

```
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_;
};
```

= 0 is called pure specifier

```
#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

```
// bad1.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 bad1 bad1.cpp Function.cc
bad1.cpp: In function `int main()':
bad1.cpp:10: error: cannot allocate an object of type `Function'
bad1.cpp:10: error:   because the following virtual functions are abstract:
Function.h:10: error:  virtual double Function::integrate(double, double) const
Function.h:9: error:  virtual double Function::value(double) const
```

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 class

```
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_;
};
```

= 0 is called pure specifier

```
#include "Function.h"

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

`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`**!

```
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 func2 func2.cpp *.cc
$ ./func2
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_;
};
```

```
int main() {

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

  return 0;
}
```

```
$ g++ -o func3  func3.cpp
func3.cpp: In function `int main()':
func3.cpp:13: error: cannot declare variable `f2' to be of type `Function'
func3.cpp:13: error:   because the following virtual functions are abstract:
Function.h:10: error:  virtual double Function::integrate(double, double) const
Function.h:9: error:  virtual double Function::value(double) const
```

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.);

  delete g1;

  return 0;
}
```

```
$ g++ -o func5 func5.cpp *.cc
$ ./func5
Gaussian with name: gauss mean: 0 width: 1
Sorry. Gauss::integrate(x1,x2) not implemented yet...returning 0. for now...
```

Bad Programming in Previous Example

- When using **–Wall** option of **g++** we get following warning

```
$ g++ -Wall  -c Gauss.cc
In file included from Gauss.h:5,
                 from Gauss.cc:1:
Function.h:6: warning: `class Function' has virtual functions but
non-virtual destructor
In file included from Gauss.cc:1:
Gauss.h:7: warning: `class Gauss' has virtual functions but
non-virtual destructor
```

- 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

Destructor of `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;
}
```

```
$ ./example7
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! Not polymorphic

```
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 noVirtualDtor noVirtualDtor.cc
$ ./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 virtualDtor virtualDtor.cc
$ ./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;
  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

```
$ ./example8
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;
}
```

```
$ ./example9
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
```