

# *Object Oriented Programming: Polymorphism*

Shahram Rahatlou

*Computing Methods in Physics*

<http://www.roma1.infn.it/people/rahatlou/cmp/>

*Anno Accademico 2018/19*



SAPIENZA  
UNIVERSITÀ DI ROMA

# Polymorphism

- ▷ 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!

# 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
  - **Gaussian, Breit-Wigner, polynomials, exponential** are all functions
  - A **Function** must have
    - **value(x)**
    - **integral(x1,x2)**
    - **primitive(x)**
    - **derivative(x)**
  - Can write a fit application that can handle existing or not-yet implemented functions using a base class **Function**

# Reminders about Inheritance

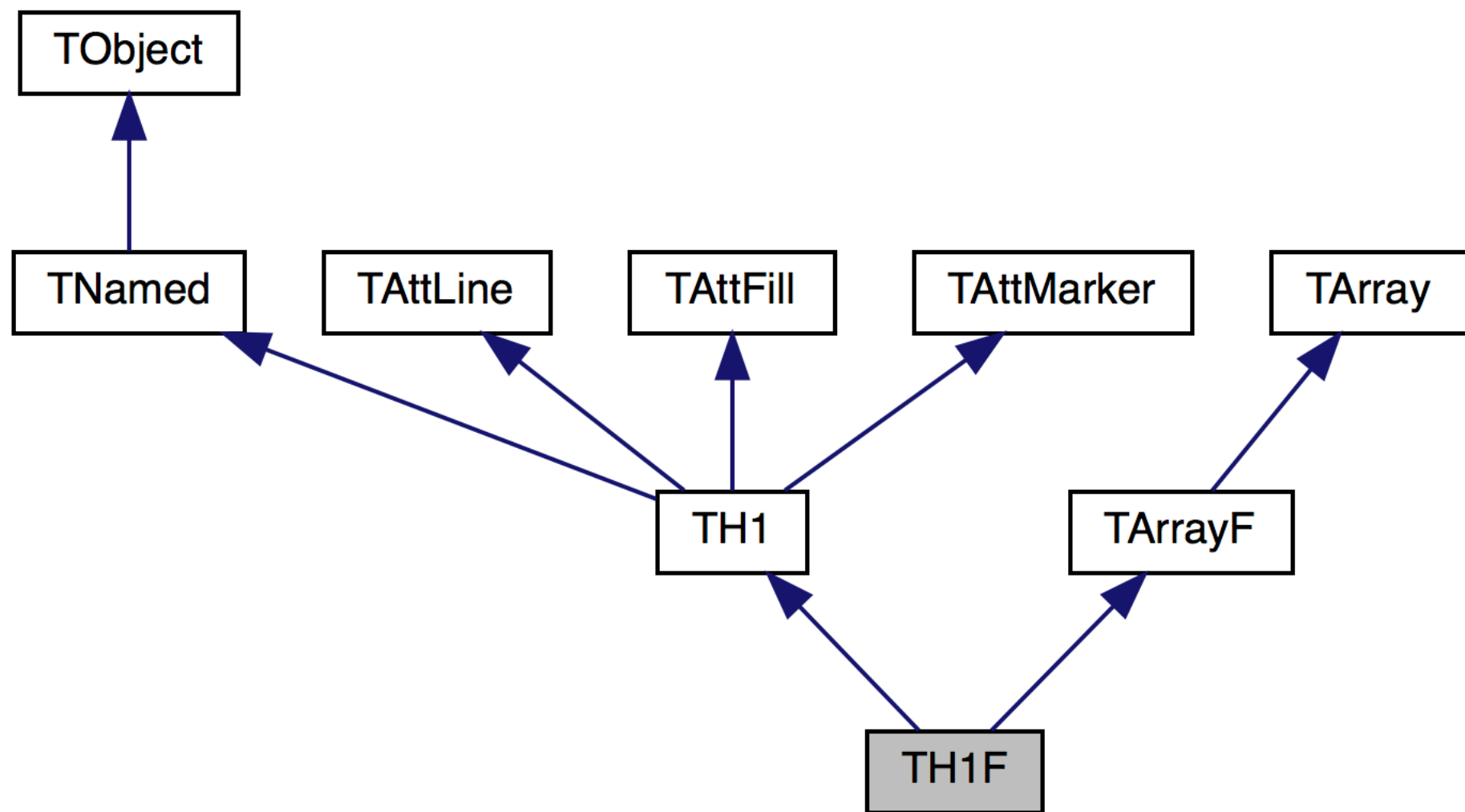
- Inheritance is a is-a relationship
  - Object of derived class 'is a' base class object as well
- Can treat a derived class object as a base class object
  - call methods of base class on derived class
  - can point to derived class object with pointer of type base class
- Base class does not know about its derived classes
  - Can not treat 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

# Example from ROOT: TH1F

<https://root.cern.ch/doc/master/classTH1F.html>

```
#include <TH1.h>
```

Inheritance diagram for TH1F:



# TObject and TNamed

ROOT » CORE » BASE » TObject

<https://root.cern.ch/root/html526/TObject.html>

## class TObject



### TObject

Mother of all ROOT objects.

The `TObject` class provides default behaviour and protocol for all objects in the ROOT system. It provides protocol for object I/O, error handling, sorting, inspection, printing, drawing, etc. Every object which inherits from `TObject` can be stored in the ROOT collection classes.

ROOT » CORE » BASE » TNamed

<https://root.cern.ch/root/html526/TNamed.html>

## class TNamed: public TObject



### TNamed

The `TNamed` class is the base class for all named ROOT classes. A `TNamed` contains the essential elements (name, title) to identify a derived object in containers, directories and files. Most member functions defined in this base class are in general overridden by the derived classes.

<https://root.cern.ch/root/html526/TH1.html>

**class TH1: public TNamed, public TAttLine, public TAttFill, public TAttMarker**



## The Histogram classes

ROOT supports the following histogram types:

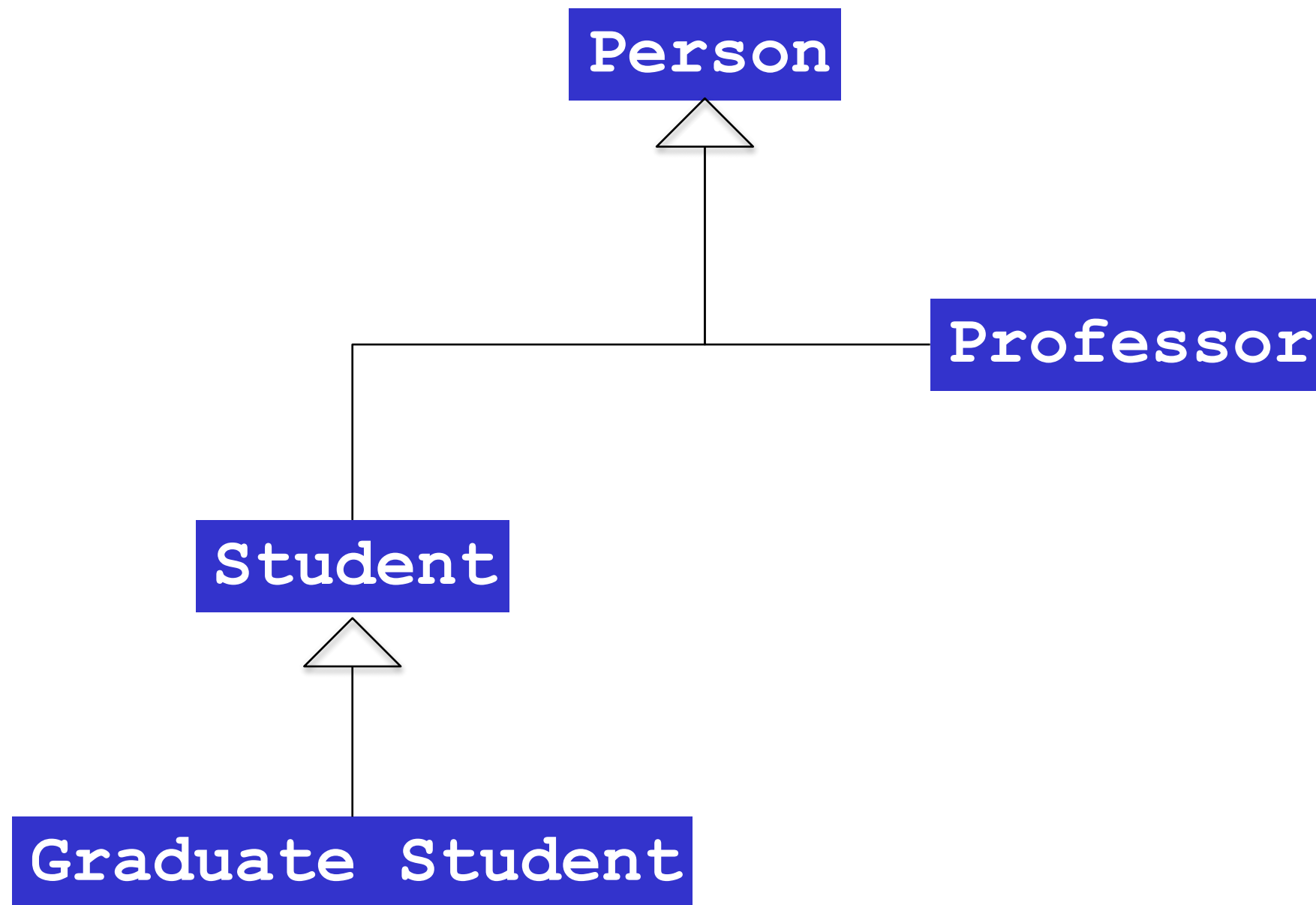
- 1-D histograms:
  - TH1C : histograms with one byte per channel. Maximum bin content = 127
  - TH1S : histograms with one short per channel. Maximum bin content = 32767
  - TH1I : histograms with one int per channel. Maximum bin content = 2147483647
  - TH1F : histograms with one float per channel. Maximum precision 7 digits
  - TH1D : histograms with one double per channel. Maximum precision 14 digits
- 2-D histograms:
  - TH2C : histograms with one byte per channel. Maximum bin content = 127
  - TH2S : histograms with one short per channel. Maximum bin content = 32767
  - TH2I : histograms with one int per channel. Maximum bin content = 2147483647
  - TH2F : histograms with one float per channel. Maximum precision 7 digits
  - TH2D : histograms with one double per channel. Maximum precision 14 digits
- 3-D histograms:
  - TH3C : histograms with one byte per channel. Maximum bin content = 127
  - TH3S : histograms with one short per channel. Maximum bin content = 32767
  - TH3I : histograms with one int per channel. Maximum bin content = 2147483647
  - TH3F : histograms with one float per channel. Maximum precision 7 digits
  - TH3D : histograms with one double per channel. Maximum precision 14 digits
- Profile histograms: See classes TProfile, TProfile2D and TProfile3D. Profile histograms are used to display the mean value of Y and its RMS for each bin in X. Profile histograms are in many cases an elegant replacement of two-dimensional histograms : the inter-relation of two measured quantities X and Y can always be visualized by a two-dimensional histogram or scatter-plot; If Y is an unknown (but single-valued) approximate function of X, this function is displayed by a profile histogram with much better precision than by a scatter-plot.

All histogram classes are derived from the base class TH1

class  
library: I  
#include  
Display  
☐ Show  
☒ Show  
[ ↑ Top



# 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_;
};
```

# Proper Constructor

```
// Person.cc
```

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

```
// Student.cc
```

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

```
// GraduateStudent.cc
```

```
GraduateStudent::GraduateStudent(const std::string& name, int id, const std::string& major) :  
    Student(name, id) {  
    major_ = major;  
    cout << "GraduateStudent(" << name << ", " << id << ", " << major << ") called" << endl;  
}
```

- ▷ **Person::Person(name)** assigns value to name\_
- ▷ **Student(name, id)** assigns id\_ and calls **Person::Person(name)**
- ▷ **GraduateStudent(name, id, major)** assigns major, calls **Student::Student(name, id)**, which calls **Person::Person(name)**

# Example

```
// example11.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;
}
```

No delete for paolo!!  
Memory Leak!

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

Can treat paolo and susan as Person

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

```
$ g++ -o /tmp/app example11.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
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 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
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
```

# Problem with Previous Example

```
// example11.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++ -o /tmp/app example11.cpp {Person,Student,GraduateStudent}.cc  
$ /tmp/app  
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 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  
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
```

- ▷ 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

```
// example12.cpp
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 /tmp/app example12.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
Person(John) called
Person(Susan) called
Student(Susan, 123456) called
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 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 overridden, 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



# Static and Dynamic (or late) binding

- Choosing the **correct derived class function at run time** based on the 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

```
// example13.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 /tmp/app example13.cpp {Person,Student,GraduateStudent}.cc  
$ /tmp/app  
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
```

# Example of Dynamic Binding

```
// example14.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();
```

```
$ g++ -o /tmp/app example14.cpp {Person,Student,GraduateStudent}.cc
$ /tmp/app
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

```
$ g++ -o /tmp/app example15.cpp {Person,Student,GraduateStudent,Professor}.cc  
$ /tmp/app  
Give me a number [1,10]: 6  
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  
  
$ /tmp/app  
Give me a number [1,10]: 2  
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
```

Virtual methods allow dynamic  
binding at runtime

# Default for Virtual Methods

```
// Professor.h
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; // will use Person::Print()

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

**print() not overridden in Professor**

```
// example16.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 /tmp/app example16.cpp {Person,Student,GraduateStudent,Professor}.cc
$ /tmp/app
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
~Professor() called for name:Robert and department: Biology
~Person() called for Robert
~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
```

**Person::print() used by default**

# 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

```
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

```
// 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 overridden 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
```

```
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;
}
```

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



# 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_;
};
```

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

```
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_;
};
```

```
// 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
```