

# *Object Oriented Programming:*

## *Inheritance*

## *Polymorphism*

Shahram Rahatlou

*Computing Methods in Physics*

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

*Anno Accademico 2019/20*



SAPIENZA  
UNIVERSITÀ DI ROMA

# 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 way to reuse software without too much re-writing
- ▷ Often several types of object are in fact special cases of a basic object
  - 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_;
};
```

# Example of Inheritance in Use

```
// 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;
}
```

```
$ g++ -o /tmp/app example1.cpp
$ /tmp/app
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() << endl;  
cout << "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 ctor!

OK... because a Student is also a Person!  
elements of polymorphism

```
$ g++ -o /tmp/app bad1.cpp
bad1.cpp:53:28: error: no member named 'id' in 'Person'
    cout << "id: " << susan->id() << endl;
                        ~~~~~^
1 error generated.
```

- ▷ 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



# Public Inheritance

```
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 (public inheritance)

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 also possible but 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

```
// example2.cpp
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**

# Constructors of Derived Classes

- ▷ Compiler calls default constructor of base class in constructors of derived class **unless** you call explicitly a specific constructor
  - NB: constructors are not inherited by constructed by compiler
- ▷ Necessary to insure data members of the base class **always** initialised 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 initialised 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_;
};
```

```
$ g++ -o /tmp/app bad2.cpp
bad2.cpp:32:5: error: constructor for 'Student' must explicitly initialize
the base class 'Person' which does not
    have a default constructor
    Student(const string& name, int id) {
    ^
bad2.cpp:7:7: note: 'Person' declared here
class Person {
    ^
1 error generated.
```

```
// bad2.cpp

int main() {

    Person anna("Anna");

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

    return 0;
}
```

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

# 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 /tmp/app bad3.cpp
$ /tmp/app
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
  - ▷ Ask yourself: why name is not used in the constructor?
- ▷ Code compiles, links, 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... bad things can happen!
- ▷ 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
```

# Undesired limitation

```
// example3.cpp
int main() {

    Person john("John");
    john.print(); // Person::print()

    Student susan("Susan", 123456);
    susan.print(); // calls Student::print()
    susan.Person::print(); // explicitly call Person::print()

    // using base class pointer
    cout << "-- using base class pointer" << endl;
    Person* p2 = &susan;
    p2->print(); // calls Person::print()

    //using derived class pointer
    cout << "-- using derived class pointer" << endl;
    Student* sp = &susan;
    sp->print(); // calls Student::print()

    // using base class reference
    cout << "-- base class reference" << endl;
    Person& p3 = susan;
    p3.print(); // calls Person::print()

    // behavior of print() depends on the type
    // of pointer determined at compilation time

    return 0;
}
```

```
$ g++ -o /tmp/app example4.cpp
$ /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
-- using base class pointer
I am a Person. My name is Susan
-- using derived class pointer
I am Student Susan with id 123456
-- base class reference
I am a Person. My name is Susan
~Student() called for name:Susan and id: 123456
~Person() called for Susan
~Person() called for John
```

# Polymorphism

# 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

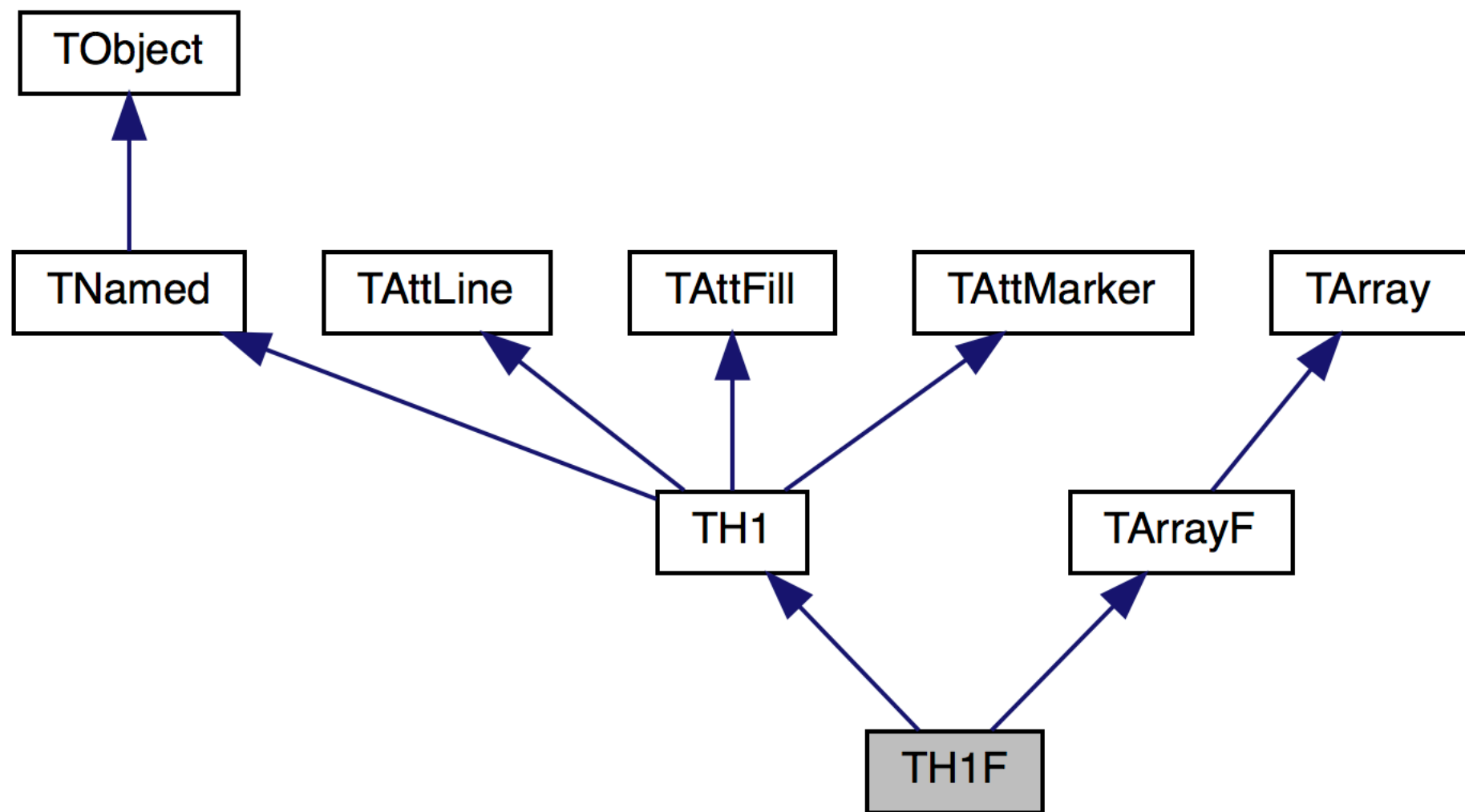
- Inheritance is a is-a relationship
  - Object of derived class 'is a' base class object as well
- Can treat a derived class object as a base class object
  - call methods of base class on derived class
  - can point to derived class object with pointer of type base class
- Base class does not know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

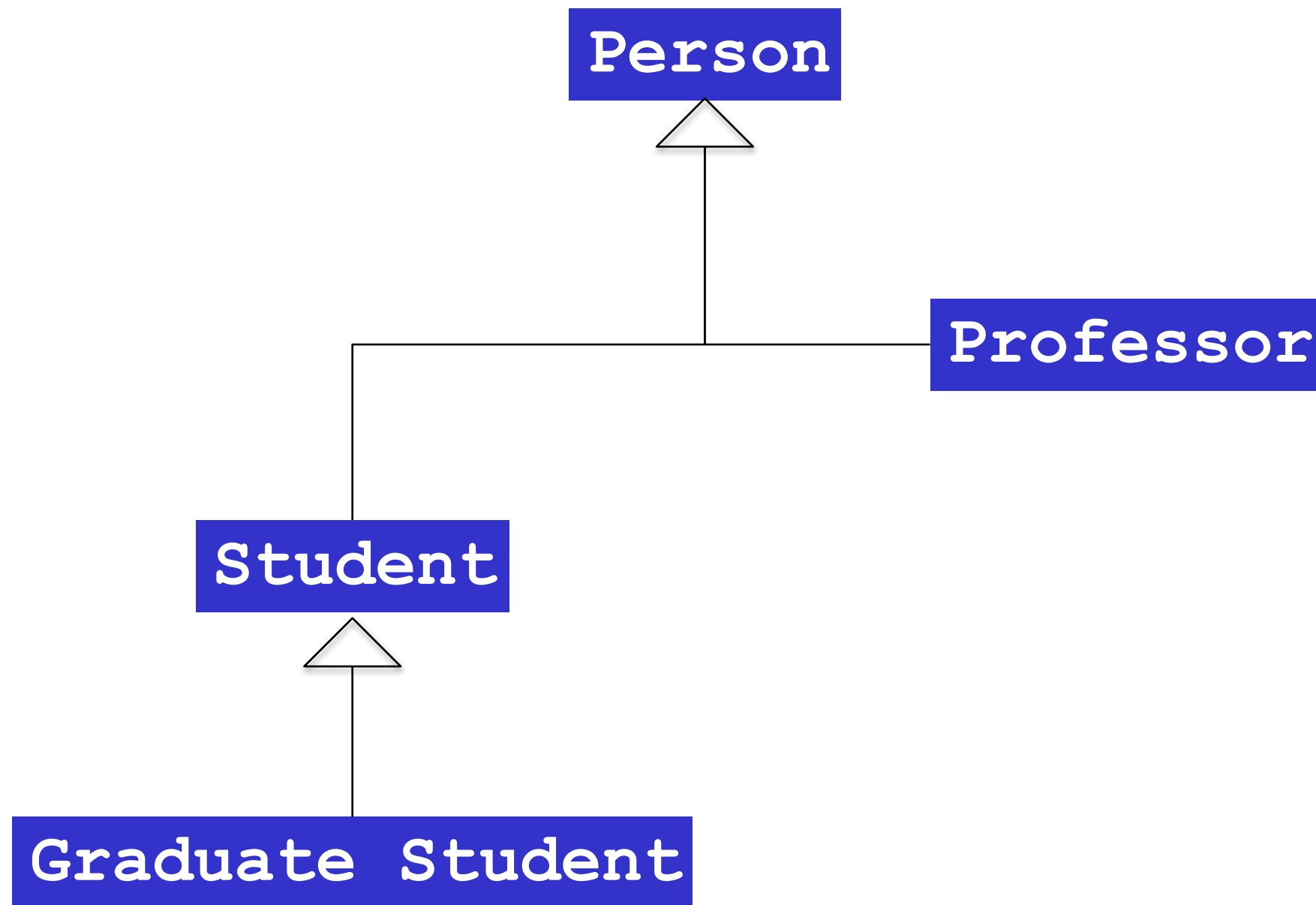
ROOT supports the following histogram types:

- 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();
    }
```

Generic call to 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

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

How? virtual functions!

```
$ 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 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
```

# 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**