Object-Oriented Programming: Polymorphism

Polymorphism: Overview

- The same entity (function or object) behaves differently in different scenarios
- Example: + operator
  - can add numbers or concatenate strings
  - its interpretation depends on context (here the operand type, but can be operand number)

1. **Compile-time polymorphism** (or early binding or static binding): a function is called at the time of program compilation

   ➡ Function/operator overloading

2. **Runtime polymorphism** (or late binding or dynamic binding): a function is called at the time of program execution

   ➡ Inheritance, function overriding, virtual functions

Polymorphism: Subtopics

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

Runtime Polymorphism

- We looked at the ability to treat objects of an inheritance hierarchy as belonging to the base class
  - Focus on common general aspects of objects instead of specifics
  - **Base class provides common interface** to all types in the hierarchy

- Polymorphism allows programs to be general and extensible with little/no re-writing, **resolving 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
  - Application uses base class and you can deal with new types not yet implemented 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 will study this case in more 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 one-way 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
    - Point to derived class object with pointer of type base class
  - Base class does not know about its derived classes
    - Cannot 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

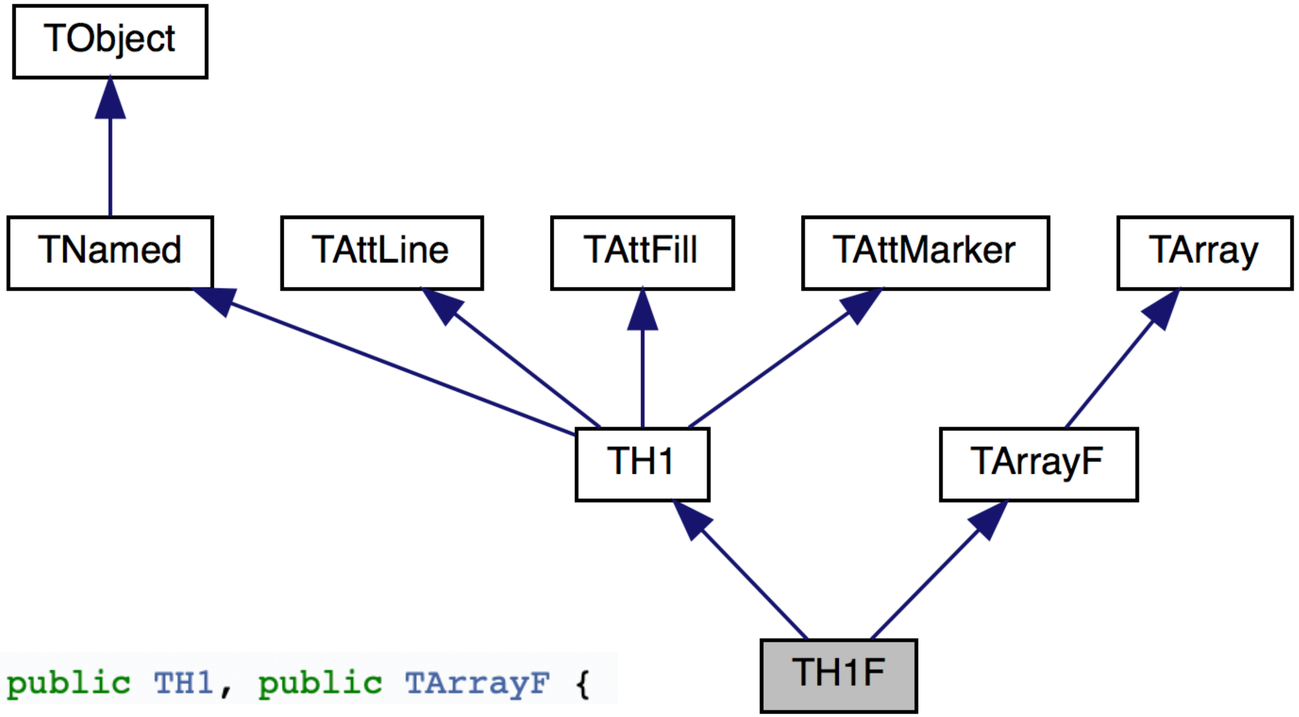
Example from ROOT: TH1F

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

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

1-D histogram with a float per channel

Inheritance diagram for TH1F:




```
575 class TH1F : public TH1, public TArrayF {
```

TObject and TNamed

https://root.cern.ch/root/html526/TObject.html
https://root.cern.ch/root/html526/TNamed.html

ROOT » CORE » BASE » TObject

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

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.

TH1

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

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

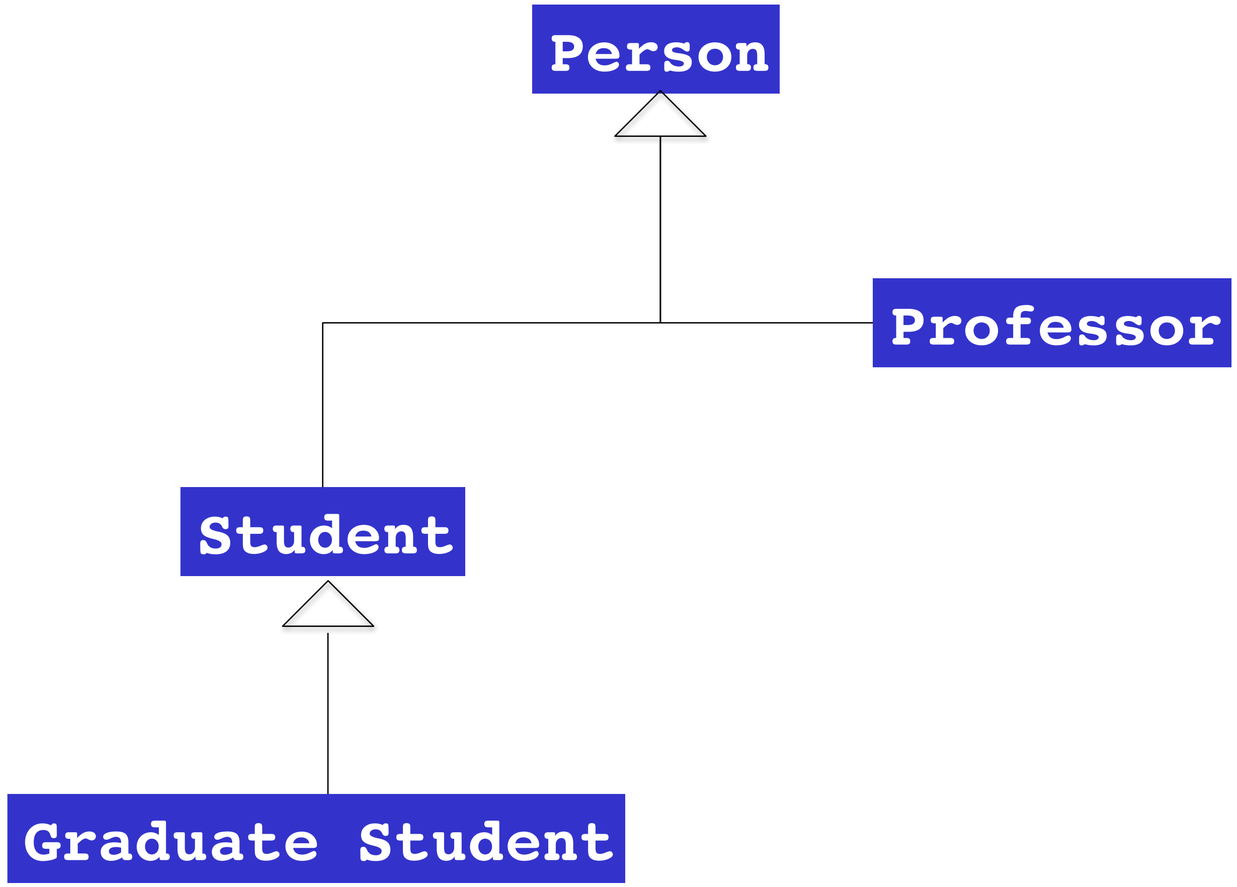
The Histogram classes

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

`Person` Inheritance Hierarchy



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

Proper Constructor

Based on `examples/11/Student.cc`, `GraduateStudent.cc` and `person.cc`

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

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

```
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)` calls `Person::Person(name)` and assigns `id_`
- `GraduateStudent(name, id, major)` calls `Student::Student(name, id)`, which calls `Person::Person(name)`, and assigns `major_`

Example with Desired Polymorphic Behaviour

Based on `examples/11/Polymorphism1.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();

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

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


  delete john;
  delete susan;

  return 0;
}
```

A type `Person` pointer can point to `Student` object

A type `Person` pointer can point to a `GraduateStudent` object

No delete for `paolo`!! Memory Leak!

Can treat `paolo` and `susan` as `Person`

Example with Desired Polymorphic Behaviour

Based on `examples/11/Polymorphism1.cpp`

```
int main() {

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

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

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

  GraduateStudent* paolo = new Gradua
  paolo->print();

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


  delete john;
  delete susan;

  return 0;
}
```

A type Pers
point to Stu

A type Pers
to a Gradua

No delete for paol
Memory Leak!

```
$ g++ -o Polymorphism1 Polymorphism1.cpp {Student,GraduateStudent,Person}.cc
$ ./Polymorphism1
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
```

Different `print()` methods called based on the object type the `Person` pointer pointed to!

The `print()` method is resolved at runtime!

How? **VIRTUAL FUNCTIONS**

Virtual Functions: Declaration

Based on `examples/11/Student.h`, `GraduateStudent.h` and `Person.h`

```
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 Functions: Implementation

Based on `examples/11/Student.cc`, `GraduateStudent.cc` and `Person.cc`

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

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

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

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

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

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

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

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

void GraduateStudent::print() const {
  cout << "I am GraduateStudent " << name() << " with id " << id()
       << " major in " << major_ << endl;
}
```

Another Example with Polymorphic Behaviour

Based on `examples/11/Polymorphism2.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()`

Another Example with Polymorphic Behaviour

Based on `examples/11/Polymorphism2.cpp`

```
$ g++ -o Polymorphism2 Polymorphism2.cpp {Student,GraduateStudent,Person}.cc
$ ./Polymorphism2
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
```

- `vector` of generic type `Person`: no knowledge about specific types
- Different derived objects stored in the `vector` of `Person`
- Generic call to `print()`

`virtual` functions

Based on `examples/11/Student.h`, `GraduateStudent.h` and `Person.h`

```
class Person {
  public:
    // ...
    virtual void print() const;
 private:
    std::string name_;
};
```

```
class Student : public Person {
  public:
    // ...
    virtual void print() const;
 private:
    int id_;
};
```

```
class GraduateStudent : public Student {
  public:
    // ...
    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 object pointed to determines which function is called
- Type of pointer (also called **handle**) has no effect on the method being executed
- `virtual` allows polymorphic behaviour and generic code without relying on specific objects

Static and Dynamic (or late) binding

Based on `examples/11/Polymorphism3.cpp`

- Choosing the **correct derived class function at runtime 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

```
int main() {

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

  // static binding at compile time
  john.print();
  susan.print();
  paolo.print();

  return 0;
}
```

Static binding

```
$ g++ -o Polymorphism3 Polymorphism3.cpp {Student,GraduateStudent,Person}.cc
$ ./Polymorphism3
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

Based on `examples/11/Polymorphism4.cpp`

```
int main() {

  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();

  delete john;
  delete susan;
  delete paolo;

  return 0;
}
```

Dynamic binding

```
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

Based on `examples/11/Polymorphism5.cpp`

```
int main() {

  Person* p = 0;

  int value = 0;
  while(value<1 || value>10) {
    cout << "Give me a number [1,10]: ";
    cin >> value;
  }
  cout << flush;

  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 Polymorphism5 Polymorphism5.cpp {Student,GraduateStudent,Person}.cc
$ ./Polymorphism5
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

$./Polymorphism5
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

Based on `examples/11/Polymorphism6.cpp` and `Professor.*`

```
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_;
};
#endif
```

`print()` not overriden in `Professor`

```
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 Polymorphism6 Polymorphism6.cpp {Student,GraduateStudent,Person,Professor}.cc
$ ./Polymorphism6
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