Object-Oriented Programming in C++

Pre-Lecture 8: Polymorphism

Prof Niels Walet (Niels.Walet@manchester.ac.uk)

Room 7.07, Schuster Building

Version: March 3, 2020

Prelecture 8

Outline

In this pre-lecture, we will discuss polymorphism, specifically concentrating on

- Base class pointers
- Virtual functions
- Pure virtual functions and abstract base classes

Polymorphism

what is it?

- Polymorphism: *poly* (many) and *morph* (form)
- It is the 3rd pillar of Object Oriented Programming
- It gives us the ability to create classes with the same structure (e.g. function names) but with different methods
- It makes use of inheritance and function overriding (not overloading)
- Of central importance is the concept of the base class pointer
- As usual, easiest to understand by example

Base class pointers

Base class pointers

an example

```
class particle
{
protected:
  double charge{};
public:
  particle(double q) : charge{q} {}
  void info(){std::cout<<"particle: charge="<<charge<<"e"<<std::endl;}
};

class ion : public particle
{
private:
  int atomic_number;
public:
  ion(double q, int Z) : particle{q}, atomic_number{Z} {};
  void info()
  {
    std::cout<<"ion: charge="<<charge
         <<"e, atomic number="<<atomic_number<<std::endl;
  }
};
```

Listing 1 : selection of PL8/baseclasspointer.cpp

Base class pointers

an example

- Base class: `particle` with one data member (`charge`)
- Derived class: `ion`, inherits `particle` and adds a 2nd data member (`atomic_number`)
- Both classes contain a member function called `info`
- Used function overriding: result of calling `info` depends on the object's type (`particle` or `ion`)

Base class pointers

an example

A simple example of using our classes

```
int main()
{
  particle particle_1{1}; // proton
  ion ion_1{2,2}; // helium nucleus
  particle_1.info();
  ion_1.info();
  return 0;
}
```

Listing 2 : selection of PL8/baseclasspointer.cpp

which produces

```
particle: charge=1e
ion: charge=2e, atomic number=2
```

Base class pointers

an example

- Can use a pointer to an object of the base-class type

```
int main()
{
  particle particle_1{1}; // proton
  ion ion_1{2,2}; // helium nucleus
  particle_1.info();
  ion_1.info();
  particle *particle_pointer; // pointer to particle
  particle_pointer=&particle_1; // point to particle_1
  particle_pointer->info();
  particle_pointer=&ion_1; // point to ion_1 (allowed!)
  particle_pointer->info();
  return 0;
}
```

Listing 3 : selection of PL8/baseclasspointer2.cpp

- Such a pointer is known as a base class pointer
- Base class pointers are special: also allowed to point to objects instantiated from derived class (`ion`) and call overridden member functions (`info`)
- In this case, what will the code output?

Base class pointers

an example

- Answer:

```
particle: charge=1e
ion: charge=2e, atomic number=2
particle: charge=1e
particle: charge=2e
```

- Last two lines are result of using base class pointer
- Annoyingly, both outputs print charge using the `info()` from the base class, even when pointing to an `ion`
- Can we make base class pointer use the appropriate version of `info`?

Virtual functions

Virtual functions

an example

- Yes: we modify the base class version of `info()`
- Originally this was

```
void info(){std::cout<<"particle: charge="<<charge<<"e"<<↩
 std::endl;}
```

- We now add the `virtual` modifier

```
virtual void info(){std::cout<<"particle: charge="<<↩
 charge<<"e"<<std::endl;}
```

- We now call `info` a virtual function
- The derived class version remains unchanged
- Our code output is now

```
particle: charge=1e
ion: charge=2e, atomic number=2
particle: charge=1e
ion: charge=2e, atomic number=2
```

- Base class pointer now accesses the appropriate function!

Virtual functions

- Powerful use: arrays of mixed types

```
// Array of base and derived objects, one particle and one ion
particle *particle_array[2];
particle_array[0] = new particle{2}; // He
particle_array[1] = new ion{1,2};    // He+
particle_array[0]->info(); // print info for particle
particle_array[1]->info(); // print info for ion
delete particle_array[0]; particle_array[0]=0;
delete particle_array[1]; particle_array[1]=0;
```

Listing 4 : selection of PL8/mixedarray.cpp

- Defines an array of base class pointers

```
particle *particle_array[2];
```

- Can then point to objects instantiated from base or derived classes
- We use `new` to create instances of each class
- Then `delete` individual objects when finished (to avoid memory leaks)
- Recall: for every `new` there should also be a `delete`

Virtual destructors:

a word of warning

- Recap: destructors are called whenever an object goes out of scope
- Usually happens at end of function
- Destructors should be used to `delete` memory when using dynamic arrays in classes
- Advice: when using base class pointers, make sure your base class destructor is virtual

```
virtual ~particle(){std::cout<<"Calling base class destructor"<<std::endl;}
```

```
~ion(){std::cout<<"Calling derived class destructor"<<std::endl;}
```

- That way, appropriate destructor is called when object (accessed with base class pointer) goes out of scope
- If base class destructor is not a virtual function, this will always be called in preference to any derived class destructor

Recap 1

Polymorphism

- We have just demonstrated polymorphism in action!
- Used inheritance to create base and derived classes
  - Used function overriding to change the action of `info` in derived class
  - Defined a base class pointer to point to either type of object
  - Made `info` a virtual function to access correct version of `info` with pointer
  - This is runtime polymorphism: only while running the code can we decide what version of `info` to call
- Note: polymorphism relies on overridden virtual members (otherwise base class pointer always refers to base class member function)
- Summary: action depends on which object base class pointer is pointing to in hierarchy
- Classes used in this way (with virtual functions) are known as polymorphic classe

Abstract base classes

Abstract base classes

- In the previous example, objects could be created from either the base or derived class
- But base class is special: it contains the virtual functions and its type is used when declaring base class pointer
- We can take this further: we can (should?) use the base class as an interface only:
  - 1 Use base class to declare[1] virtual functions only (**pure virtual function**)
  - 2 In the derived class we now **must** override the virtual functions and define their action–otherwise the derived class is also abstract (which may be what you want...)
  - 3 The derived classes can still contain their own data and member functions
- A base class that **only** declares existence of virtual functions is known as an abstract base class
- Formally: base class becomes abstract base class when converting at least one virtual function to a pure virtual function
- Let's see how ...

[1] i.e. name and parameter list, not what they do

Abstract base classes

- Modified base class (now abstract)

```
class particle
{
public:
  virtual ~particle(){} // Need this!
  virtual void info()=0; // pure virtual function
};
```

- Base class becomes abstract base because it contains a pure virtual function

```
  virtual void info()=0; // pure virtual function
```

- Pure virtual functions have no method in base class: must be implemented in derived classes
- We use `particle` to declare what functions common to all derived classes (and as name of base class pointer)
- All objects can be accessed using a base class pointer through `particle` - known as an interface

Abstract base classes

- Our derived classes could then be

```
class electron : public particle
{
private:
  int charge;
public:
  electron() : charge{-1} {}
  ~electron() {std::cout<<"Electron destructor called"<<std::endl;}
  void info(){std::cout<<"electron: charge="<<charge<<"e"<<std::endl;}
};
class ion : public particle
{
private:
  int charge,atomic_number;
public:
  // Note constructor short-hand!
  ion(int q, int Z) : charge{q},atomic_number{Z} {}
  ~ion() {std::cout<<"Ion destructor called"<<std::endl;}
  void info(){std::cout<<"ion: charge="<<charge
              <<"e, atomic number="<<atomic_number<<std::endl;}
};
```

- Derived classes define members specific to each particle type!

Abstract base classes

- Look at an application such as

```
int main()
{
  particle *particle_pointer = new ion{1,2};
  particle_pointer->info();
  delete particle_pointer;
  //
  particle_pointer = new electron;
  particle_pointer->info();
  delete particle_pointer;
  //
  return 0;
}
```

Listing 5 : selection of PL8/abstract.cpp

- Both types of particle (which have their own version of `info`) are accessed using a single base class pointer
- This is the power of polymorphism: *one interface, multiple methods*

Abstract base classes

polymorphic arrays

- We can define polymorphic arrays as arrays of base class pointers

```cpp
{
  // Array of 2 base class pointers
  particle **particle_array = new particle*[2];
  particle_array[0] = new ion{1,2};
  particle_array[1] = new electron;
  particle_array[0]->info(); // print info for electron
  particle_array[1]->info(); // print info for ion
  // clean-up
  delete particle_array[0];
  delete particle_array[1];
  delete[] particle_array;
  return 0;
}
```

Listing 6 : selection of PL8/polymorphicarray.cpp

- We need to define new objects individually since they are of different derived types
- We use base class pointer to access members
- We can then delete one object at a time at the end (and array) ; see the output

```
ion: charge=1e, atomic number=2
electron: charge=-1e
Ion destructor called
Electron destructor called
```

Abstract base classes

polymorphic arrays

- Alternative: use vectors

```
int main()
{
  std::vector<particle*> particle_vector;
  particle_vector.push_back(new ion{1,3});
  particle_vector.push_back(new electron);
  particle_vector[0]->info();
  particle_vector[1]->info();
  std::cout<<"particle_vector has size "<<particle_vector.size()<<std::endl;
  for (auto particle_vectorit=particle_vector.begin();
       particle_vectorit<particle_vector.end();
       ++particle_vectorit) delete *particle_vectorit;
  particle_vector.clear();
  std::cout<<"particle_vector now has size "<<particle_vector.size()<<std::endl;
  return 0;
```

Listing 7 : selection of PL8/polymorphicvector.cpp

- Again, must `delete` each object then clear vector itself
- Output:

```
ion: charge=1e, atomic number=3
electron: charge=-1e
particle_vector has size 2
Ion destructor called
Electron destructor called
particle_vector now has size 0
```

Summary

Polymorphism

summary and buzzwords

- We want to design classes for a set of related objects
- We create a base class that contains members (data and functions) applicable to all objects within the set
- We make those functions we wish to override (same name/parameters different method) virtual functions
- If we do not want to create objects of the base class (and use it solely as an interface), we make our virtual functions pure virtual functions, assigning them to zero in the base class
- Our base class is now known as an abstract base class; only accessible to derived classes
- We can call each object's virtual member functions with a single base class pointer