Object-Oriented Programming for Scientific Computing

Dynamic Polymorphism

Ole Klein

Interdisciplinary Center for Scientific Computing
Heidelberg University
ole.klein@iwr.uni-heidelberg.de

19. Mai 2015

Objects of Derived Classes as Function Arguments

Objects of a derived class can be used in functions that expect objects of the base class, as shown before, but

- only the part belonging to the base class is copied if the argument is passed call-by-value (slicing).
- only the part belonging to the base class will be accessible when the argument is passed by reference.

In particular: even if a function has been redefined in the derived class, it is always the function of the base class that will be called.

Problems with Inheritance

```
#include<iostream>

class A
{
    public:
        int work(int a)
        {
            return(a);
        }
};

class B : public A
{
    public:
        int work(int a)
        {
            return(a*a);
        }
};

int doWork(A& object)
{
    return(object.work(2));
}
```

```
int main()
{
    A objectA;
    B objectB;
    std::cout << objectA.work(2)
        << ",␣";
    std::cout << objectB.work(2)
        << ",␣";
    std::cout << doWork(objectA)
        << ",␣";
    std::cout << doWork(objectB)
        << std::endl;
}
```

Output:

```
2, 4, 2, 2
```

Virtual Functions

```
#include<iostream>

class A
{
    public:
        virtual int work(int a)
        {
            return(a);
        }
};

class B : public A
{
    public:
        int work(int a)
        {
            return(a*a);
        }
};

int doWork(A& object)
{
    return(object.work(2));
}
```

```
int main()
{
    A objectA;
    B objectB;
    std::cout << objectA.work(2)
        << ",␣";
    std::cout << objectB.work(2)
        << ",␣";
    std::cout << doWork(objectA)
        << ",␣";
    std::cout << doWork(objectB)
        << std::endl;
}
```

Output:

```
2, 4, 2, 4
```

Virtual Functions

- When a function is declared as `virtual` in the base class, then the function of the derived class will be called for objects of the derived class, even if the method is called via a reference or a pointer with the type of the base class.
- The function definition in the derived class has to match that of the base class, otherwise the function is overloaded as usual.
- The return value of the method may differ, if it is a class derived from the base class, e.g. for correct behavior of `operator+=` and similar.
- This is known as polymorphism ("many shapes", "many forms").

C++11: Override

- As mentioned above, if the function definition in the derived class doesn't exactly match the one of the base class, the function is simply overloaded.
- This is often the result of a typing error and not what was intended.
- Also writing `virtual` before the function in the derived class only has the consequence that this overloaded function is also virtual.
- In C++11 there is the additional keyword `override`. If this is written after the function header in a derived class, then a compiler error will occur if the function doesn't redefine a virtual method of the base class.
- It's advised to use this keyword whenever possible.

```
class B : public A
{
    public:
        int work(int a) override
        {
            return(a*a);
        }
};
```

Virtual Functions and Scoping

When a method is selected by explicit specification of the namespace (scoping), then the corresponding variant is directly invoked without polymorphism.

```
int doWork(A& object)
{
    return(object.A::work(2));
}
```

creates the output

```
2, 4, 2, 2
```

Typical Implementation of Dynamic Polymorphism

- To implement this kind of polymorphism, the compiler adds a hidden pointer to each object (the “virtual table pointer” or “vpointer”). This “vpointer” points to a global table (the “virtual table” or “vtable”).
- The compiler generates a vtable for each class that contains at least one virtual function. The vtable itself has a pointer for each virtual function of the class.
- During a call to a virtual function, the runtime system accesses the code of the method using the vpointer of the object and then the function pointer in the vtable.
- The overhead in terms of memory consumption is therefore a pointer per object containing virtual methods, plus a pointer for each virtual method. The runtime overhead are two additional memory accesses (for the vpointer and the address of the method).
- Inlining is not possible with virtual methods.

Example: Typical Implementation

```
class B1
{
public:
  void f0() {}
  virtual void f1() {}
  int int_in_b1;
};

class B2
{
public:
  virtual void f2() {}
  int int_in_b2;
};

class D : public B1, public B2
{
public:
  void d() {}
  void f2() {}  // override
      B2::f2()
  int int_in_d;
};
```

Example memory layout for an object of type `D`:

```
d:
  +0: pointer to virtual method
      table of D (for B1)
  +4: value of int_in_b1
  +8: pointer to virtual method
      table of D (for B2)
 +12: value of int_in_b2
 +16: value of int_in_d

virtual method table of D (for
    B1):
  +0: B1::f1()  // B1::f1() is
      not overridden

virtual method table of D (for
    B2):
  +0: D::f2()   // B2::f2() is
      overridden by D::f2()
```

Interface Base Classes

- The purpose of an interface base class (abstract base class) is to provide a common interface for the derived classes.
- Interface base classes usually have no attributes (and contain no data).
- The functions of the interface base class are typically purely virtual, which means the functionality is only implemented in the derived classes. This is indicated by adding `= 0` after the function declaration.
- Classes that contain at least one purely virtual function are called abstract base classes.
- There is no way to create objects of an abstract base class, but there may be references and pointers of this type (which then point to objects of a derived class).
- Objects of a derived class can only be created if *all* purely virtual functions of the base class have been implemented. This ensures that the class complies with the complete interface.

Interface Base Classes

```
class BaseClass
{
    public:
        virtual int functionA(double x) = 0;
        virtual void functionB(int y) = 0;
        virtual ~BaseClass() // will be explained later
        {};
}

class DerivedClass : public BaseClass
{
    public:
        int functionA(double x); // has to exist
        void functionB(int y);   // has to exist
}
```

Function Objects (Functors)

Definition:
A *function object* (functor) is each object which can be called like a function.[1]

- In C++ a function is of the form `return_type foo(Type1 arg1, Type2 arg2);`.
- An object which defines the parenthesis operator `operator()` can be used as a function, e.g.

```
class Foo
{
  public:
    return_type operator()(Type1 arg1, Type2 arg2);
};
```

---

[1] D. Vandevoorde, N. M. Josuttis: C++ Templates - The Complete Guide, p. 417

Benefits of Functors

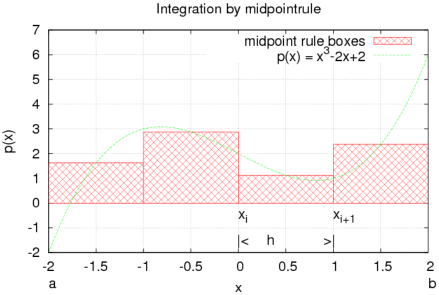
- Functors are “intelligent functions”. They can
  - provide other functions in addition to `operator()`.
  - have an internal state.
  - be pre-initialized.
- Every functor has its own type.
  - The functions (or function pointers to) `bool less(int,int)` and `bool greater(int,int)` would have the same type.
  - The functors `class less` and `class greater` have different type.

Numerical Integration (Quadrature)

As an example of this concept we want to implement a class for the numerical integration of arbitrary functions with the composite midpoint rule.

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f((i+\frac{1}{2})\cdot h)\cdot h$$

with $h = \frac{b-a}{n}$.



Application of the midpoint rule with $n = 4$ for $p(x) = x^3 - 2x + 2$.

Example: Numerical Integration of $\cos(x-1)$

The example program integrates $\cos(x-1)$ with the composite midpoint rule. The files used are:

- `functor.h`: contains the interface base class for a functor.
- `cosine.h`: contains the definition of a concrete functor: $\cos(ax+b)$
- `midpoint.h`: contains the definition of a function that takes a functor as argument and integrates it with the composite midpoint rule.
- `integration.cc`: contains the main program that uses the integrator to integrate $\cos(x-1)$ over the interval $[1:\frac{\pi}{2}+1]$.

functor.h

```
#ifndef FUNCTORCLASS_H
#define FUNCTORCLASS_H

// Base class for arbitrary functions with one double parameter

class Functor
{
  public:
    virtual double operator()(double x) = 0;
    virtual ~Functor()
    {}

};

#endif
```

cosine.h

```
#ifndef COSINECLASS_H
#define COSINECLASS_H

#include <cmath>
#include "functor.h"

// realization of a function cos(a*x+b)
class Cosine : public Functor
{
  public:
    Cosine(double a=1.0, double b=0.0) : a_(a), b_(b)
    {}

    double operator()(double x) override
    {
        return cos(a_*x+b_);
    }

  private:
    double a_,b_;
};

#endif
```

midpoint.h

```
#include "functor.h"

double MidpointRule(Functor& f, double a=0.0, double b=1.0, size_t
    n=1000)
{
    double h = (b-a)/n; // length of a single interval

    // compute the integral boxes and sum them
    double result = 0.0;
    for (size_t i=0; i<n; ++i)
    {
        // evaluate function at midpoint and sum integral value
        result += f(a + (i+0.5)*h);
    }

    return h*result;
}
```

integration.cc

```
// include system headers
#include <iostream>
// own headers
#include "midpoint.h"
#include "cosine.h"

int main()
{
  Cosine cosine(1.0,-1.0);
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣
      "
            << MidpointRule(cosine,1.0,M_PI_2+1.0) << std::endl;

  return 0;
}
```

integrator.h

In an alternative implementation, it is also possible to generalize the integrator:

```
#ifndef INTEGRATORCLASS_H
#define INTEGRATORCLASS_H

#include "functor.h"

class Integrator
{
  public:
    virtual double operator()(Functor& f) = 0;
    virtual ~Integrator()
    {}
};

#endif
```

midpoint_class.h

```
#include "integrator.h"

class MidpointRule : public Integrator
{
    double a_,b_;
    size_t n_;
  public:
    MidpointRule(double a, double b, size_t n) : a_(a), b_(b), n_(n)
    {}
    double operator()(Functor& f) override
    {
        double h = (b_-a_)/n_; // length of a single interval

        // compute the integral boxes and sum them
        double result = 0.0;
        for (size_t i=0; i<n_; ++i)
        {
            // evaluate function at midpoint and sum integral value
            result += f(a_ + (i+0.5)*h);
        }

        return h*result;
    }
};
```

simpson_class.h

```
#include "integrator.h"

class SimpsonRule : public Integrator
{
    double a_,b_;
    size_t n_;
  public:
    SimpsonRule(double a, double b, size_t n) : a_(a), b_(b), n_(n)
    {}
    double operator()(Functor& f) override
    {
        double h = (b_-a_)/n_; // length of a single interval

        // compute the integral boxes and sum them
        double result = f(a_)+f(b_);
        for (size_t i=1; i<n_; i+=2)
            result += 4. * f(a_ + i*h);
        for (size_t i=2; i<n_; i+=2)
            result += 2. * f(a_ + i*h);

        return (h*result)/3.;
    }
};
```

integration_class.cc

```
#include <iostream>
#include <memory>
#include "midpoint_class.h"
#include "simpson_class.h"
#include "cosine.h"

int main()
{
  Cosine cosine(1.0,-1.0);
  std::unique_ptr<Integrator> integrate(new
      MidpointRule(1.0,M_PI_2+1.0,10));
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣
      "
            << (*integrate)(cosine) << std::endl;
  SimpsonRule simpson(1.0,M_PI_2+1.0,10);
  std::cout << "Integral␣of␣cos(x-1)␣in␣the␣interval␣[1:Pi/2+1]␣is␣
      "
  << simpson(cosine) << std::endl;
  return 0;
}
```

Arrays of Objects

- It is often necessary to create an array of objects of a common interface base class, e.g. the parameter functions for various materials which are used in a simulation.
- Since references must already be initialized when they are created, only an array of base class pointers may be used, with the pointers being pointed at the various objects.
- The pointer should be initialized with 0 or in C++11 with `nullptr`, or (even better) `std::unique_ptr` or `std::shared_ptr` should be used.

```
std::vector<std::unique_ptr<Functor> > Function(4);
Function[0] = std::unique_ptr<Functor>(new Cosine(1.0,-1.0));
// alternative
Function[0].reset(new Cosine(1.0,-1.0));
...
```

Virtual Destructors

- If only base class pointers are available to objects of the derived class, then only the destructor of the base class can be called.
- Since derived classes could use allocated resources that must be released in the destructor, it makes sense to give the base class a (typically empty) virtual destructor.
- This has the consequence that the correct destructor is called for each derived class object even when using a base class pointer.
- The destructor can not be purely virtual.

```
class Functor
{
  public:
    virtual double operator()(double x) = 0;
    virtual ~Functor()
    {};
};
```

- In C++11 this is done via the keyword `default`:

```
    virtual ~Functor() = default;
```

Dynamic Cast

- In a running program it may be desirable to find out whether a pointer to an object can be converted into a pointer to another class (e.g. because one of the pointers is a pointer to a base class of the object).
- This can be achieved using a `dynamic_cast`. `func = dynamic_cast<Functor*>(&f)` converts the pointer f into a pointer to a functor, if this is allowed.
- This also works the other way around in the class hierarchy:

```
Cosine* cos = dynamic_cast<Cosine*>(func);
```

- A `dynamic_cast` returns either a converted pointer or a null pointer if the conversion is not possible.
- `dynamic_cast` also works with references:

```
Cosine& cos = dynamic_cast<Cosine&>(f);
```

  If the conversion can't be done, an exception of the type `std::bad_cast` will be thrown.
- In C++11 there is the free function `std::dynamic_pointer_cast` that provides this functionality for `std::shared_ptr`s.

Dynamic Cast

```
#include<cstdlib>
#include "midpoint.h"
#include "simpson.h"
#include "cosine.h"

double Integrate(Functor& f, double a, double b)
{
    Cosine* cos = dynamic_cast<Cosine*>(&f);
    if (cos == 0)
        return MidpointRule(f,a,b);
    else
        return SimpsonRule(f,a,b);
}
```

Virtual Constructors

If you only have a base class pointer to an object, then it is usually impossible to create an object of the same type (the derived class) or to copy the entire object (only the base class part is copied). Using so-called “virtual constructors” this can be done anyway:

```
class Functor
{
  public:
    ...
    virtual Functor* create() = 0;
    virtual Functor* clone() = 0;
}
```

```
class Cosine : public Functor
{
  public:
    ...
    Cosine* create()
    {
        return new Cosine();
    }

    Cosine* clone()
    {
        return new Cosine(*this);
    }
}
```

Virtual Base Classes

- Sometimes it is useful to derive a class from several other classes that are derived from the same base class.
- One possible application is to let the derived classes execute various aspects of a problem, that can be implemented in different ways in each case, but need to access the same data from the base class.
- A class derived from these classes using multiple inheritance then combines the functionality into a certain overall process.

Example: Newton Method from DUNE-PDELab

- Example: the Newton method is to be used to solve a nonlinear system of equations. A Newton method consists of:
  - a basic algorithm
  - steps that must be performed at the start of each Newton iteration (e.g. the reassembly of the Jacobi matrix)
  - a test whether the process has converged
  - optionally a linesearch to enlarge the convergence area

  Each of these intermediate steps is outsourced to a separate class, so you can replace all the components independently. The common data and the virtual functions are placed in a base class.
- Normally each class has its own base class, but this would mean that the data exists multiple times. Virtual inheritance prevents this.

```
class NewtonSolver : public virtual NewtonBase
{
...
};

class NewtonTerminate : public virtual NewtonBase
{
...
};

class NewtonLineSearch : public virtual NewtonBase
{
...
};

class NewtonPrepareStep : public virtual NewtonBase
{
...
};

class Newton : public NewtonSolver, public NewtonTerminate,
               public NewtonLineSearch, public NewtonPrepareStep
{
...
};
```

Summary

If there are several objects that embody a fundamental principle (such as circle, triangle, rectangle … are special realizations of a geometric object) or a specific functionality (such as the trapezoidal rule and Simpson's rule being a special variant of the integration process), then it is good style in C++ to define a common interface that is implemented by each specific realization in its own way.

Dynamic polymorphism

- uses abstract base classes and virtual functions for this.
- employs special language constructs to ensure that each class actually implements the full interface functionality (pure virtual functions).
- allows variant selection at runtime.
- requires additional work (virtual function table).
- prevents some optimizations (inlining, loop unrolling).