Computational Intelligence on Automation Lab @ NCTU

UEE1303(1070) S'12 Object-Oriented Programming in C++

Lecture 09: *Polymorphism – Virtual Functions And Abstract Base Class*

Learning Objectives

- Understand the concept of polymorphism
  - –differences between static and dynamic bindings
  - –virtual functions for run-time polymorphism
- Use abstract base classes
- Understand the importance of virtual destructors
- Using polymorphism

Polymorphism

- Polymorphism is one of three keys in OOP and supports
  - –*function overloading* and *operator overloading* at *compile* time (see before)
  - –function *overriding* at *run-time*
  - ⇨ associate many meanings to one function
- Run-time polymorphism
  - –enable programmers to design a common interface that can be used on different but related objects
  - –reduce complexity and development time

Example for Polymorphism (1/3)

```
class CPoint   //base class: class CPoint
{
    double x, y;
public:
    CPoint(double a=0, double b=0):
           x(a), y(b) {}
    void SetPoint(double a=0, double b=0) {
        x=a; y=b;                       }
    double GetX() const { return x; }
    double GetY() const { return y; }
    friend ostream & operator << (ostream &,
                        const CPoint&);
    string ToString() const {
        return "CPoint";    }
};
```

lec9-1.cpp

Example for Polymorphism (2/3)

```
class CRect: public CPoint {
double lg, wd; //new private data members
public:     //add new member functions
    CRect(double a, double b,
        double c, double d) : lg(c), wd(d)
        { SetPoint(a,b); }
    void CRect(double a, double b,
        double c, double d)
        { SetPoint(a,b); lg=c; wd=d; }
    double GetL() const { return lg; }
    double GetW() const { return wd; }
    double Area() const { return lg*wd; }
    friend ostream & operator << (ostream &,
        const CRect &);
    string ToString() const {
        return "CRect";     }
};
```

lec9-1.cpp

Example for Polymorphism (3/3)

```
ostream & operator << (ostream & out,
            const CPoint& p) {
    out << p.x << " " << p.y; return out;}
ostream & operator << (ostream & out,
            const CRect& p) {
    out << p.GetX() <<" "<< p.GetY()
        << " area " <<p.Area(); return out;}
```

```
int main()
{
    CRect cr1(2,3,20,10);
    cout << "old: " << cr1
    cr1.SetRect(5,5,9,7);
    cout << "new: " << cr1 << endl;
    CPoint &rRf = cr1;
    cout << "rRf: " << rRf << endl;
    return 0;
}
```

```
old: 2 3 area 200
new: 5 5 area 63
rRf: 5 5
```

lec9-1.cpp

Another Example for Polymorphism (1/2)

```
class CCuboid : public CRect {
protected:
    double ht; //dp for depth
public:
    CCuboid(double a, double b,
            double c, double d, double e):
        CRect(a,b,c,d) { ht = e; }
    void SetHeight(double d=1.0){ ht = d; }
    void GetHeight() const { return ht;   }
    double Area() const {
        return (2*CRect::Area()+2*GetL()*ht
                +2*GetW()*ht);             }
    friend ostream & operator << (ostream &,
                const CCuboid &);
    string ToString() const {
        return "CCuboid";   }
};
```

lec9-2.cpp

Another Example for Polymorphism (2/2)

```
ostream & operator << (ostream & out,
            const CCuboid& p) {
    out << p.GetX() << " " << p.GetY()
        << " surface " << p.Area();
    return out;                              }
```

lec9-2.cpp

```
int main()
{
    CCuboid cu1(2,3,20,10,5);
    cout << "old: " << cu1 << endl;
    cu1.SetRect(1,1,8,6);
    cout << "new: " << cu1 << endl;
    CPoint &pRef = cu1;
    cout << "pRef: " << pRef << endl;
    CRect &rRef = cu1;
    cout << "rRef: " << rRef << endl;
    return 0;
}
```

What will show on screen??

Static v.s. Dynamic Bindings

- Binding
  - compiler reserves a space in memory for all user-defined functions and keeps track of memory addresses for each function
  - a function name is bound with function address ⇨ the starting location in memory for the function code
- Static binding (a.k.a. *early binding*)
  - compiler binds all function calls to the addresses of function code at compile-time
- Dynamic binding (a.k.a. *late binding*)
  - function calls are resolved at run-time
  - order of function calls depends on the action taken by the user

Realizing Polymorphism

- Compile-time polymorphism
  - apply static binding
  - advantage of fast speed
  - realized by function overloading and operator overloading
- Run-time polymorphism
  - apply dynamic binding
  - advantage of enhanced flexibility
  - realized by *inheritance* + *virtual functions*
- In C++, redefining a virtual function in a derived class is called overriding a function

```
virtual 〈datatype〉 〈fname〉 (〈para_list〉);
```


Example for Polymorphism (w/o Virtual)

- Given class `CPoint`, `CRect`, and `CCuboid`

```
void DisplayObject(const CPoint & p) {
    cout << p.ToString() << endl;       }
```

lec9-3.cpp

```
//in main()
    CPoint o1(5,7);
    CRect o2(2,4,5,7);
    CCuboid o3(1,3,5,7,9);
    DisplayObject(&o1);
    DisplayObject(&o2);
    DisplayObject(&o3);
```

```
CPoint
CPoint
CPoint
```


Example for Polymorphism (w/ Virtual)

```
virtual string ToString() const { ... }
```

```
void DisplayObject(const CPoint * p) {
    cout << p->ToString() << endl;       }
```

lec9-3.cpp

```
//in main()
    CPoint o1(5,7);
    CRect o2(2,4,5,7);
    CCuboid o3(1,3,5,7,9);
    DisplayObject(&o1);
    DisplayObject(&o2);
    DisplayObject(&o3);
```

```
CPoint
CRect
CCuboid
```

Virtual Functions (1/2)

- Casting between the base class and the derived class
  - can assign a derived-class *object* to a base-class *object*
  - can copy the *address* of a derived-class object to a *pointer* of a base-class object
  - a derived-class object can be a *reference* to base-class object

  ⇨ can only access members in the base class, not members in the derived class
- Virtual functions makes that a pointer or reference of a base-class object can be applied onto a derived-class object

Virtual Functions (2/2)

- Virtual functions tell the compiler
  - don't know how function is implemented
  - wait until used in program
  - get implementation from object instance
  - call dynamic (late) binding
- If class `C1`, `C2`, … are derived from `C0` which has a *virtual function* `f()` (public or protected)
  - `f()` can be *redefined* in `C1`, `C2`, and etc
  - call by the base-class object or pointer to the base-class object
  - decide which `f()` to call during run-time

More on Virtual Functions

- If a function `f()` in the base class is virtual, then all `f()`'s in the derived classes are virtual
  - all *redefined* `f()`'s in the derived class `C1`, `C2`, and etc have the same prototype ⇨ overriding ≠ overloading
  - a virtual function must be a member function of a class ⇨ cannot be *global*, *static* or *friend*
  - *destructors* can be virtual but constructors cannot be virtual
- Major disadvantage: *more storage overhead* + *running slower* ⇨ used only if necessary

Example of Virtual Functions (1/2)

lec9-4.cpp

```
class B0 {
public: void ShowFun() { //not virtual
        cout << "B0::ShowFun()" << endl; }
};
class C0 : public B0 {
public: virtual void ShowFun() { //virtual
          cout << "C0::ShowFun()" << endl; }
};
class C1 : public C0 {
public: void ShowFun() { //virtual
        cout << "C1::ShowFun()" << endl; }
};
class C2 : public C1 {
public: void ShowFun() { //virtual
        cout << "C2::ShowFun()" << endl; }
};
```

Example of Virtual Functions (2/2)

```
void FunPtr(C0 *ptr) {
    ptr->ShowFun();
}
```

lec9-4.cpp

```
//in main()
    B0 w, *p; C0 x, *q;
    C1 y; C2 z;
    p = &w; p->ShowFun();
    p = &y; p->ShowFun(); //what happen??
    q = &x; FunPtr(q);
    q = &y; FunPtr(q);
    q = &z; FunPtr(q);
```

```
B0::ShowFun()
B0::ShowFun()
C0::ShowFun()
C1::ShowFun()
C2::ShowFun()
```


More Example of Virtual Functions

```
class A {  //case 1
public: virtual void ShowFun() { //virtual
        cout << "A::ShowFun()" << endl; } };
class C : public A {
public: void ShowFun(int i) { //not virtual
        cout << "C::ShowFun()" << endl; } };
```

```
//in main()
    C c;
    A *pa = &c, &ra = c, a = c;
    a.ShowFun();
    pa->ShowFun();
    ra.ShowFun();
```

lec9-5.cpp

```
A::ShowFun()
A::ShowFun()
A::ShowFun()
```


More Example of Virtual Functions

```
class B {  //case 2
public: virtual void ShowFun(char c) {
        cout << "B::ShowFun()" << endl; } };
class D : public B {
public: void ShowFun(int i) { //not virtual
        cout << "D::ShowFun()" << endl; } };
```

```
//in main()
    D d;
    B *pb = &d, &rb = d, b = d;
    b.ShowFun(0);
    pb->ShowFun(0);
    rb.ShowFun(0);
```

lec9-5.cpp

```
B::ShowFun()
B::ShowFun()
B::ShowFun()
```


More Example of Virtual Functions

```
class B { public:
    void f() { cout << "Bf "; }
    virtual void g(){ cout << "Bg "; }
    void h() { g(); f(); }
    virtual void m(){ g();  f(); }
};
class D : public B { public:
    void f() { cout << "Df "; }
    void g() { cout << "Dg "; }
    void h() { f(); g(); }
};
```

lec9-6.cpp

```
//in main()
    D d; B *pB = &d;
    pB->f(); pB->g(); pB->h(); pB->m();
```

```
Bf Dg Dg Bf Dg Bf
```

Virtual Destructors

- If a base-class pointer to derived-class object is deleted ⇨ base-class destructor will act on such object ⇨ what's wrong?

```
CBase *pB = new CDerived;
...
delete pB;
```

  – deletion may not be not thorough
  – point to `CDerived` object but not free `CDerived` members
- Good to always have base-class destructors as virtual destructors
  – then appropriate destructors will be called

Example of Virtual Destructors

```
class A { public:
    ~A() { cout << "A::~A()\n";}
};
class C : public A {
    int * iary;
public:
    C(int i) { iary = new int [i]; }
    ~C() {
        delete [] iary;
        cout << "C::~C()\n";  }
};
```

lec9-7.cpp

```
//in main()
    A *pa = new C(10);
    delete pa;
```

```
A::~A()
```


Example of Virtual Destructors

```
class B { public:
    virtual ~B() { cout << "B::~B()\n";}
};
class D : public B {
    int * iary;
public:
    D(int i) { iary = new int [i]; }
    ~D() {
        delete [] iary;
        cout << "D::~D()\n";  }
};
```

lec9-7.cpp

```
//in main()
    B *pB = new D(10);
    delete pB;
```

```
D::~D()
B::~B()
```


Pure Virtual Functions

- Base class might not have meaningful definition for some of its members!
  – only for other classes to derive from
- Recall class `CPoint`
  – all other figures are objects of derived classes
  – rectangles, circles, triangles, etc.
  – class `CPoint` has no idea how to calculate area ⇨ a pure virtual function

```
virtual void Area()=0;
```

Abstract Base Classes

- Pure virtual functions require no definition
  - force each derived classes to define its own version
- Class with one or more pure virtual functions ⇨ abstract base class
  - can only be used as base class
  - no objects can ever be created from it because it doesn't include complete definitions of all its members
- If one derived class fails to define all pure virtual functions,
  - also an abstract base class

Example for Abstract Base Class (1/2)

lec9-8.cpp

```
class CFig {
protected:
    double x, y;
public:
    void SetDim(double a=0, double b=0) {
        x=a; y=b;                             }
    virtual void Area()=0; //pure virtual!
};
class CRec: public CFig {
public: void Area(int i) {
            cout<<"Rec:"<<x*y<<"\n"; }
};
class CTri: public CFig {
public: void Area() {
            cout<<"Tri:"<<x*y/2<<"\n"; }
};
```


Example for Abstract Base Class (2/2)

lec9-8.cpp

```
//in main()
    CFig *pF;
    CFig  f1; //what happen?
    CRec  r2; //what happen?
    CTri  t3;
    t3.SetDim(20,10);
    pF = &t3;
    pF->Area();
    CFig &rF = t3;
    rF.SetDim(5,4);
    rF.Area();
```

```
Tri:100
Tri:10
```


Extended Type Compatibility

- Given: CDerived is a derived class of CBase
  - CDerived objects can be assigned to objects of type CBase
  - But NOT the other way!
- Example:

```
class Pet {
public:
    string name;
    virtual void print() const;      };

class Dog : public Pet {
public:
    string breed;
    virtual void print() const;      };
```

Classes Pet and Dog

- Now given declarations:

```
Dog vdog;
Pet vpet;
```

  – member variables name and breed are public ⇨ for example, not typical
- Anything that "is a" dog "is a" pet:

```
vdog.name = "Tiny";
vdog.breed = "Maltese";
vpet = vdog;
```

  – above are allowable
  – a pet "is not a" dog (not necessarily)

Slicing Problem

- The value assigned to variable `vpet` loses its `breed` field
  – Ex: `cout << vpet.breed;`
  ⇨ produce ERROR!
  – called the slicing problem
- Might seem appropriate
  – `Dog` was moved to `Pet` variable, so it should be treated like a `Pet`
  – therefore not have `Dog` properties
  – make for interesting philosophical debate

Example for Slicing Problem

lec9-9.cpp

```
class Pet {
public:
    string name;
    virtual void print() const;      };

class Dog : public Pet {
public:
    string breed;
    void print() const;      };
```

```
Pet *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Maltese";
ppet = pdog;
cout << ppet->breed; //what happens?
//cannot access breed => slicing problem
```


Example of Casting

- Consider

```
Pet vpet; //base-class object
Dog vdog; //derived-class object
vdog = static_cast<Dog>(vpet);  //ILLEGAL!
```

- Cannot (down)cast a pet object to be a dog object, but

```
vpet = vdog;   // Legal!
vpet = static_cast<Pet>(vdog); //legal!
```

- Upcasting is OK and safe
  – from descendant type to ancestor type
  – but not the other way (downcast) around

Downcasting & dynamic_cast

- Downcasting is dangerous! (self-study)
  - casting from the ancestor type to the descended type
  - assume more *additional* information
  - can be done with `dynamic_cast`:

```
Pet *ppet;
ppet = new Dog;
Dog *pdog = dynamic_cast<Dog*>(ppet);
```

  - Legal, but dangerous!
- Downcasting rarely done due to pitfalls
  - must track all information to be added
  - all member functions must be *virtual*

Inner Workings of Virtual Functions

- Don't need to know how to use it!
  - principle of information hiding
- Virtual function table
  - compiler creates it
  - has pointers for each virtual member function
  - points to location of correct code for that function
- Objects of such classes also have a pointer
  - point to virtual function table

Summary (1/2)

- Late binding delays decision of which member function is called until runtime
  - in C++, virtual functions use late binding
- Pure virtual functions have no definition
  - classes with at least one are abstract
  - no objects can be created from abstract class
  - used strictly as base for others to derive

Summary (2/2)

- Derived class objects can be assigned to base class objects
  - Base-class members are lost ⇨ slicing problem
- Pointer assignments and dynamic objects
  - allow "fix" to slicing problem
- Make all destructors virtual
  - good programming practice
  - ensure memory correctly de-allocated