3 Object-Oriented Programming

Class hierarchies and polymorphism

Preview

- inheritance, base classes, and derived classes
- using constructors and destructors in derived classes
- overriding base-class members in a derived class
- problems with slicing
- *virtual* destructors
- defining derived-class assignment
- *public*, *protected* and *private* inheritance
- pros and cons of object-oriented programming

Subclasses and late binding

Polymorphism

- can write reusable code for classes that are not known in advance (not written or even yet designed)
- two separate forms of polymorphism
  - *inclusion polymorphism* (the set of derived instances is a *subset* of the base instances)
  - *operation polymorphism* = late binding of methods

Late (dynamic) *binding*

- existing code can change their behavior to appropriately deal with new kinds of objects
  - object's exact type need not be know at compile time for a call of a polymorphic (*virtual*) operation
  - virtual call is matched at run time according to type of target object

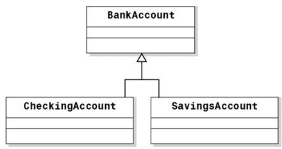
Basic terminology

- in C++, D is called a *derived class* of B
  - in Java, class D *extends* another class B
- B is called a *base class* of D
- B is also called a *superclass* for the subclass D

B

D

- e.g., BankAccount class has two derived classes: CheckingAccount and SavingsAccount



What is inheritance?

- new classes created (derived) from existing classes
  - defines a subtype relationship (also a *subset*)
  - enables code reuse: a derived class inherits data members and member functions from a previously defined base class
  - a derived object contains the parts defined in its super classes, as base-class subobjects (layered form)
- originally in Simula, Smalltalk: used inheritance / sub-classing for both classification and code reuse
  - these tasks can and often should be separated
- in C++, both *single* and *multiple inheritance*
  - multi-inheritance means multiple direct base classes
- plus different kinds of inheritance (*public*, *private*)

Example: hypothetical Shape library

```
class Shape {              // abstract class: interface only
public:
    virtual void draw () const = 0;        // note "= 0"
    virtual void rotate (double) = 0;
    virtual Point center () const = 0;
    //  no representation
};
class Circle : public Shape  {             // note "public"
    . . .
   virtual void draw () const;   . . .     // note "virtual"
private:
    Point c;  double r;  Color c;          // specific data . .
};
```

Handling objects polymorphically

```
void DrawAll (std::vector <Shape *> const& v)
{
    std::vector <Shape*>::const_iterator it;
    for (it = v.begin (); it != v.end (); ++it)
        (*it)->draw ();                // uses late binding
}
```

- STL vector v stores pointers to different Shape objects
- *const iterator* is used to traverse the data structure
- each specific shape knows how to draw itself
- can write general "polymorphic" code at base-class level that can adapt to any shape (even not yet existing)
- STL containers are discussed more later

Type fields vs. OOP

Problems with a *switch* statement (in C)

- e.g., a *switch* statement could determine which *draw* function to call based on which type in a hierarchy of different shapes
- action is based on the *type tag* stored in a member
- tracking down and updating multiple *switch* statements is time consuming and error-prone

Polymorphism and late binding

- programs can be written to uniformly process objects of classes derived from the classes in a hierarchy
- *virtual* functions and late binding of calls replace and hide low-level *switch* logic
- the mechanisms are built-in, and support designing and implementing extensible systems

Base and derived classes

Visibility and access of members with (public) inheritance

```
class Base {                 // potential base class
public:        . . .
protected:     . . .
private:       . . .
};
```

- *private* members of base class are not accessible from derived classes (unless a subclass is defined as *friend*)
- derived-class member functions can directly refer to inherited *protected* members (but only within *this* object, or objects of the same derived class)
- using *protected* data is convenient but often considered to break data abstraction (information hiding)
- however, *protected* member functions are OK


```
class Student {
public:
    explicit Student (long no, std::string const& name = "");
    long getNumber () const;
    std::string getName () const;
    void print () const;    // print student number and name
private:
    long number_;
    std::string name_;
};
class StudentWithAccount : public Student {      // silly . .
public:
    StudentWithAccount (long, std::string const&, double);
    double getBalance () const;
    void setBalance (double);
    void print () const;                   // print balance, too
private:
    double balance_;
};
```


Upcasting and slicing

- inheritance creates layered objects, with subparts that represent its inheritance levels
- a cast from a derived class to a base class is allowed for a public inheritance

```
void printInfo (Student s) {   // uses a value parameter
    s.print ();  . . .
}
StudentWithAccount stud (30, "barbara", 10);  . . .
printInfo (stud);      // stud is upcasted:  (Student)stud
Student s1 = stud;       // here, too: s1 = (Student)stud
```

- results in a (base) slice of the object being copied
- calls Student::print, not StudentWithAccount::print
- for OOP, must use *virtual*s and pointers (see later)

Constructors and destructors in derived class

- a derived-class constructor must first call the constructor for its base class, to initialize the base-class members
- the compiler makes sure (some) base-class constructor always gets called
  - e.g., if derived-class constructors are omitted, a compiler-generated default constructor calls the base-class' default constructor (with zero arguments)
- destructors are always called in the reverse order of constructors: a derived-class destructor is called before its base-class destructor
  - the derived-class destructor may assume that the base-class members are still existing with a valid state
  - the base-class destructor does not know about any derived classes

Calling base-class constructor

The constructor of a derived class calls the constructor of the base class Base using the member initializer list

```
. . : Base (arguments), dataMember_( . . ),   // note order
```

Base class initializer item

- uses normal constructor call syntax with appropriate arguments
- is called before the initialization of local data members
- calling the correct constructor is the programmer's responsibility
  - otherwise, the base class's default (zero-args) constructor is called (whether right or not)

Creating objects

- derived constructors take care of the initialization of its base part, using the member initializer list

```
StudentWithAccount::StudentWithAccount (long no,
      std::string const& name, double balance)
 :  Student (no, name),           // base class' constructor
    balance_(balance)             // member list of derived
{ }
```

- note how the construction of base-class and derived-class layers proceeds, one layer or subobject at a time

Copy constructors in derived classes

- similar syntax used to define a copy constructor

```
Student::Student (Student const& s);

StudentWithAccount (StudentWithAccount const& swa)
  : Student (swa)           // swa is also a Student reference
    balance_(swa.balance_)
{}
```

calls base class' copy constructor

Warning. If a call of the base copy constructor is omitted, the compiler calls (here erroneously) the default (zero-args) constructor for Student

=> the base part is not copied in copy construction

Early vs. late binding of methods

- in Java, operations that are not *final* employ late binding
  - if the value of the variable is an object of a derived class that redefines an operation defined in a base class, then this redefined operation is invoked
  - for *final* operations, Java uses early binding: the static type of the variable, and not its value, determines which operation is invoked
- C++ uses an *opposite* philosophy:
  - by default, early binding is used ~ identical to C calls
  - late binding is used only with pointers or references, and then only if the member function is explicitly specified as *virtual*

Early vs. late binding (cont.)

- by default, the C++ compiler implements early (static) binding:

```
Base base;                // declared (local) object
base.print ();            // function determined at
                          // compile time
```

- note that the base variable cannot hold but Base values
- for *virtual* functions, the compiler can implement dynamic binding

```
Base * basePtr = new Derived;    // a dynamic object
. . .
basePtr->print ();          // virtual function determined
                            // at execution time
```


Using virtual functions

- suppose a hierarchy of shape classes such as Circle, Triangle, etc.
  - define a base class Shape with a virtual draw method
  - different shapes have unique draw operations so we must override draw in each of the derived classes
- call them by calling on the draw function on the base class Shape
  - the target object is provided through a pointer or a reference
  - the program determines dynamically (i.e., at run time) which function is actually executed
- objects actually carry along some kind of compiler-generated hidden internal type info (*vtable* pointer)

virtual functions (cont.)

- virtual declaration has the keyword *virtual* before a function prototype in base-class

```
virtual void draw () const;
```

- a base-class pointer to a derived class object will call the correct draw function

```
Shape * shape = new Triangle;
. . .
shape->draw ();
```

- if a derived class does not override a virtual function, the base-class function is used (if any)

Early (static) binding

- suppose draw is *not a virtual* function

```
Shape s, * sPtr = &s;
Circle c, * cPtr = &c;
sPtr->draw ();              // calls base-class draw
cPtr->draw ();              // calls derived-class draw
sPtr = &c;                  // allowable implicit conversion
sPtr->draw ();              // still calls base-class draw
```

- the original definition of Student::print () was not virtual

```
Student * ptrStd = &stud; // StudentWithAccount object
ptrStd->print ();           // calls Student::print
```

- even if a pointer is used, employs early (compile-time) binding for non-virtual functions

Defining virtual functions

- to use late binding, print must be made *virtual*

```
class Student {
public: . . .
    virtual void print () const;
};
void Student::print () const {        // no "virtual" here
    std::cout << "Number: " << number_ << "; name: "
              << name_ << std::endl;
}
class StudentWithAccount : public Student {
public:
    virtual void print () const;    // could omit "virtual"
    . . .                           // but once virtual, always virtual
```


Defining virtuals (cont.)

```
void StudentWithAccount::print () const {    // error version
    std::cout << "Number: " << number_ << "; name: "
      << name_ << "; balance: " << balance_ << std::endl;
}
```

- but a derived class cannot access *private* base members
- often, the subclass version is meant to *extend* the original service with some new behaviour
- so, the correct and more modular version

```
void StudentWithAccount::print () const {   // ok version
    Student::print ();                      // first print the base part
    std::cout << "; balance: " << balance_ << std::endl
}
```

- no danger of unwanted recursion because the scope operator :: turns off late binding

Overloading vs. overriding

1. Overriding enables run-time matching of type

- a virtual operation can be overridden in a derived class
- the same call can invoke *different* member functions
- C++ has no *final*, thus: once *virtual*, always *virtual*
- but, a class can be made "final" (e.g., private ctors/dtor)

2. Overloading works statically; e.g., for "operator <<"

```
std::cout << "name is " << stud.getName ();
```

- the actual operation is determined at compile-time
- in a given source position, the *same function* is called
- for members, works only within the scope of a single class: operations with the same name in a derived class hides operations from the base class
  - the *using* keyword makes them available (next slide)

Overloading vs. overriding (cont.)

```
class B {
public:
    int f (int i) { return i; }  . . .
};
class D : public B {
public:
    using B::f;      // makes every f from B available
    double f (double d) { return d; }  . . .
};
D d;
std::cout << pd.f (1) << '\n';          // prints 1
std::cout << pd.f (2.3) << '\n';        // prints 2.3
```

Abstract and concrete classes

Abstract classes

- provide base classes for other classes
- no objects of an abstract classes can be instantiated
  - too "general" to define real objects (e.g., Shape)
- but can have pointers and references declared
- declare a virtual functions as pure with a *zero clause*

```
virtual void draw () const = 0;  // pure member function
```

- must be defined in a derived *concrete* class

Concrete classes

- classes that can instantiate objects
- provide all missing definitions to make real objects (e.g., Square, Circle)

Problems with slicing

- if a parameter is passed by value, and the type of the formal parameter is a base class, then slicing of an argument may create problems
- consider the following class hierarchy

```
class Shape {
public:
    virtual void draw () const;  . . .
};
class SpecificShape : public Shape {
public:
    virtual void draw () const;  . . .
};
```


Problems with slicing (cont.)

- consider the function using a parameter of the type Shape:

```
void display (Shape s) {         // uses value parameter
    . . s.draw ();  . .
}
SpecificShape ss;
display (ss);                    // the parameter s is sliced
```

- only the base part (layer) is copied into the parameter s
- even if draw is virtual, the call within display invokes Shape::draw() rather than SpecificShape::draw ()
- passing parameters by reference avoids the problems

```
void display (Shape const&);
```

- if Shape is abstract, the compiler would complain

Potential problems with virtuals

1. Late binding doesn't work with sliced (upcasted) objects.
2. When a virtual operation is invoked from the base class' constructor, then it is not yet really virtual, i.e., early binding is used; technically:
   - the object has a layered structure, and only the base object is constructed at this stage: the derived-class version of a virtual function does not yet exist
   - early binding is used for calls of virtuals in a destructor, too
3. A derived-class virtual operation must have an identical signature
   - with the exception of covariant return types, e.g. copy: see [Stroustrup, p. 425]

Destructing objects

- destructors are "inherited" in the sense that the destructor of the derived class lastly (implicitly) calls the base destructor
  - should never explicitly call a destructor of a base class in the definition of a destructor of the derived class
- must always start from the most specific destructor
  - note that the base-class destructor cannot call the derived-class destructor - which one of them to call?
- calling a destructor only for the base class would be a grave mistake when the derived class has resources that need to be released

Virtual destructors

- a base class has a destructor Shape::~Shape () that is called by *delete*

```
delete ShapePtr;     // refers to some specific shape
```

- if a derived object is deleted through a base-class pointer, the *default static binding* will cause
  - the base-class destructor to be called and act on the object
  - the potential derived-class resources remain unreleased
- must declare a *virtual* base-class destructor to ensure that the right (= most-specific) destructor will be called
  - that destructor will then call the base destructors

```
class AccountForStudent {                // silly but illustrative
public:  . . .
    virtual ~AccountForStudent ();  . . .
protected:
    Student * stud_;
    double balance_;
};
AccountForStudent::~AccountForStudent () {
    delete stud_; stud_= 0;              //  or  "=  INVPTR_VAL"
}
class Bank;                              // forward declaration
class BankAccount : public AccountForStudent {
public:  . . .
    virtual ~BankAccount ();  . . .
protected:
    Bank * bank_;
};
BankAccount::~BankAccount () {
    delete bank_;  bank_= 0;             //  or  "=  INVPTR_VAL"
}
```


Destructing objects

```
AccountForStudent * ba = new BankAccount ("John", 100,
                    1200.50, new Bank ("Royal Bank"));
```

- ba maintains two resources: a Student and a Bank
- in the deallocation:

  delete ba; ba = 0;

  the actual value of ba is used (late binding), and so calls the destructor of the most specific derived class
- when the destructor of a derived class completes, it calls the destructor of its base class (and so on)

Note. This example uses dynamic (heap-allocated) objects only in order to illustrate memory management.

Idiom: *virtual destructor*

- when you design a class (potentially) used as a base class, always make the destructor *virtual*
- if the base class has no resources to release, make the destructor's body empty (you must still implement it)
  - the derived-class destructor calls it anyway
  - but the compiler can often optimize away unnecessary calls of empty (inline) blocks
- note the destructors are *not* defined *virtual* in "plain" data types, e.g. std::string does not have *virtual* destructor
  - a polymorphic object must carry along some extra info, namely a pointer to its class' *vtable*

Assignment in derived class

- copy constructor and base-class assignment are not automatically called by derived classes (destructors are)
- so assignment must call base assignment; otherwise no assignment is done for the base part

```
Derived& Derived::operator = (Derived const& rhs) {
    if (this == &rhs) return *this; // can often omit check
    Base::operator = (rhs);        //  assign the base part
    . . .                          //  assign the derived part
    return *this;
}
```

- another form for a base-class assign (expression syntax)

```
static_cast <Base&> (*this) = rhs;     // uses base part
```

Note. Sometimes should consider *private* assignment.

Problems with derived assignment

- above, the copying of the derived class part may fail - *after* the base part has already been assigned
  - thus, the object is left with some changed and some unchanged parts => the object is messed up
- better and more generally, can often implement assignment based on an already existing copy constructor

```
Derived& Derived::operator = (Derived const& rhs)  {
    Derived tmp (rhs);             // copy ctor may fail
    swap (*this, tmp);             //  swap contents safely
    return *this;
}                                  // tmp is destructed here at exit
```

- the called swap function is here assumed to never fail
- note that no self-check is required here (code is cleaner)
- generally: using a copy constructor can provide exception safety (discussed more later)

Kinds of inheritance in C++

Public inheritance implies *type conformance* ("*is-a*")

```
class D : public B {  . .            // default is "private"
```

- an instance of a subclass is also an instance of its base class: can assign, pass as parameters, and so on

Private inheritance implements pure *code reuse*

```
class D : private B {  . .  // can omit keyword "private"
```

- no *is-a-relationship*: no assignment, no base ptrs
- but can use already defined parts and code from B (*has-a-relationship*), and override virtual functions if required

(1) *public*: inherited members remain as such (~ Java-style)
(2) *protected*: inherited public become protected
(3) *private*: inherited public/protected become private

Multi-inheritance in C++

- a class can be derived from multiple base classes
  - any name conflicts are resolved with qualification (::): which base class and its member is meant
- a situation where the same base class is inherited via multiple paths is called "*diamond inheritance*"
- such a common-base object may be either *duplicated* or *shared* - and both ways are possible in C++
  - when two *separate* class libraries or frameworks are combined, no such common bases exist => no problems
  - within *same* framework, duplication may create conflicts
  - but when *virtual inheritance* is used, only one common-base object will be created in derived classes
- multi-inheritance is actually used in C++ *iostream* library

  basic_iostream is derived from both basic_istream and basic_ostream that are derived from basic_ios


```
class File {                    // Case: hypothetical abstract file class
public:
    File (std::string const& s = "") : name_(s) {
        if (name_!= "") { /* open File */ } }
    std::string const& getName () const { return name_; }
    virtual void close () = 0;                    // pure function
    virtual ~File ()  = 0 { }                     // pure but implemented
private:
    std::string name_;    . . .                   // possibly other data
};
class InFile : virtual public File {              // File part is shared
public:
    InFile (std::string const& s = ""): File (s) {  . .  } // overridable
    virtual char read () { return ' '; }          // do some reading..
    virtual void close () { /* close Infile */ }  // make concrete
    virtual ~InFile () { close (); }  . . .       // or is already closed
}
```

Case: hypothetical abstract file class


```
class OutFile : virtual public File {           // File part is shared
public:
    OutFile (std::string const& s = ""): File (s) { . . } // overridable
    virtual void write (char c) { c; }          // do some writing..
    virtual void close () { /* close OutFile */ }  // make concrete
    virtual ~OutFile () { close (); }  . . .     // or is already closed
};

class IOFile : public InFile, public OutFile {   // no virtual here
public:
    IOFile (std::string const& s = "") : File (s) { }   // overridden
    virtual void close () { /* close IOFile */ }   // also overridden
    virtual ~IOFile () { close (); }  . . .        // or is already closed
};
```

- note that all classes InFile, OutFile, and IOFile are concrete classes with "full file services" (can be opened, closed..)

Notes on multi-inheritance example

```
int main () {
  IOFile ioFile ("myfile.txt"); . . // construct a multi-derived one
} // calls: File(), InFile(), Outfile(), IOFile(), ~IOFile(), ~OutFile(),..
```

- the virtual base-class part File appears once in IOFile objects
- the *most derived class* can override ctor calls of the virtual base class, and any ambiguous definitions of functions (definitions that are given in classes between)
  - note that the multi-derived class IOFile both (1) defines construction of the File part and (2) resolves the conflict of *differing* definitions of the close function
- *virtual* inheritance is not done by default: it may involve some overheads
- Scott Meyers recommends defining virtual base classes as *interface* classes (resembling Java-style multi-inheritance)
  - but that really doesn't seem to solve all problems: where to put the data name_ - and is *it* shared or duplicated?

Downcasting base to derived

- objects of a derived class can be treated as objects of its (public) base class, using the subtype relationship
- of course, reverse is not true - base class objects are not derived-class objects: no value assignment to derived
  - but a base-class *pointer* or *reference* may refer to a derived-class object

Downcasting a pointer (also called: "type recovery")

- use an explicit cast to convert a base-class pointer to a derived-class pointer; of course, the type of the object should match the type of the pointer

```
derivedPtr = dynamic_cast <Derived *> (basePtr);
             // otherwise this cast returns zero (0)
```

- resembles *switch* statement: inspects type of an object (code is not really polymorphic and reusable)

Prohibiting instantiation of a class

1. Make constructors *private* (or sometimes *protected*).
2. Specify at least one operation as *pure virtual*

```
. . virtual T foo () = 0;          // defined in subclass
```

- or can define just *pure virtual* destructor
- then must provide a default implementation that is (implicitly) called from derived class destructors

```
class Base {
public:  . .
    virtual ~Base () = 0; . .  // pure virtual destructor
}; . .
// define an empty default body (called by Derived)
inline Base::~Base () { }
```

Potential problems with OOP

- must divide a phenomenon into objects and classes
  - separate transitory states ("attributes") vs. fixed inherent "essence" (class)
- create useful and conceptually valid class hierarchies
  - don't violate the *Liskov substitution principle* (LSP)
  - but don't overuse inheritance but prefer composition and black-box reuse when sufficient
- distinguish entity types from states and relationships
  - don't inherit Man and Woman from Person (why?)
- when needed, define relationships as independent entities with their own attributes and life cycles, e.g.:
  - have-account is relationship between Person and Bank
- how to use OOP: *design patterns* (discussed later)

Pros and cons of OOP

- useful way to create intuitive conceptual hierarchies
  - consider Shape, Circle, Triangle, etc.
- supports code reuse, extensibility, and run-time flexibility
- can sometimes cause (unnecessary) overhead
  - indirection (*vtable*) and general function call overhead
  - pointer downcasts: type recovery (used in old Java)
- in many cases, templates provide as flexible and more secure way to achieve (static) polymorphism
  - STL: type-parameterized containers and algorithms
  - late binding is avoided in STL; compile-time checking supports early and more secure error detection
  - optimizes performance (static binding, inlining, etc.)
- can be combined: generics & late binding (also in Java)