SFU CMPT-212 2008-1 Topic: Public Inheritance

Ján Maňuch

E-mail: jmanuch@sfu.ca

Friday 22nd February, 2008

Class inheritance

Class inheritance is one of the main *code reusability* mechanisms in C++, and is achieved through *class derivation*.

When a class is derived from another (*base class*), we can do the following:

- use the data of the base class
- use the public methods of the base class
- add new data
- add new methods (member functions)
- redefine existing class methods

Derived class is also called a *subclass* of the base class and the base class is called a *superclass* of the derived class.

Example of a public derivation:

```
1  class BaseClass {
2    int a;
3   public:
4    void seta(int);
5    int get();
6  };
7
8  class DerivedClass : public BaseClass {
9    int b;           // add data
10  public:
11   int get();      // redefine the method
12   void setb(int); // add a method
13 };
```

Example: derived.cpp

Public derivation

- models an *is-a* relationship: object of a derived class should be also a particular object of the base class
- an object of the derived class:
  - contains data members of the base class
  - can use the *interface* (public members) of the base class
- private members of the base class cannot be accessed
  (**Note:** private data members are contained in an object of derived class but cannot be accessed directly, only through the interface of the base class)
- public members of the base class become public members of the derived class

Constructors

- a constructor of derived class always calls a constructor of the base class
- if not specified by the programmer which constructor to call, *the default constructor* is called
- to specify a different constructor, you have to use the *"member initializer list"* `: BaseClass(...)`

Example:

```
1 class DerivedClass : public BaseClass {
2  public:
3   DerivedClass(int);
4   // calls the default constructor of BaseClass
5   DerivedClass(double d) : BaseClass(d) {}
6   // explicitly calls unary constructor of BaseClass
7 };
```

Example: constructor.cpp

Remarks:

- the base class constructor is called first — it initializes the part of object corresponding to the base class
- the derived class constructor should
  - pass required data to the base class constructor (using the member initializer list)
  - initialize added data members

Destructors

- destructor of a derived class should clean up only after data members *added* to derived class (as destructor for the base class *is automatically called*)
- they are called in *reversed* order: first destructor of the derived class, then destructor of the base class

Example: destructor.cpp

Relaxation for pointers and references

- so far, the pointers and references had to *match* the assigned type (no conversions were allowed)
- *inheritance* introduces an *exception* to this rule:
  - a base class pointer can point to a derived class object (not the other way!)
  - a base class reference can refer to a derived class object (not the other way!)

  this is called *upcasting*

Example:

```
1 DerivedClass obj;
2 BaseClass &ref = obj;   // allowed
3 BaseClass *pnt = &obj;  // allowed
4 DerivedClass &dref = ref; // not allowed
5 DerivedClass *dpnt = pnt; // not allowed
```

Why this exception?

- expressing the *is-a relationship*
- *make sense:* after *upcasting* we can access the public methods of the base class on the object using the base class reference or pointer — that's ok since derived class inherits all public methods
- **Usage:**
  - functions defined with base class reference or pointer argument can be used also for derived class objects
  - allows us to initialize a base class object to a derived object
  - allows us to assign a derived object to a base class object

  Example: exception.cpp

More practical example:

```
1  class Point {
2    int x,y;
3   public:
4    Point(int ix=0,int iy=0) { x=ix; y=iy; }
5    friend move(Point &p);
6  };
7
8  void move(Point &p) {
9    p.x += 2; p.y +=1;
10 }
11
12 class ColoredPoint : public Point {
13   char *color;
14  public:
15   ColoredPoint(int ix=0,int iy=0,char *icolor="black")
16     : Point(ix,iy) { color=icolor; }
17 };
18 ...
19 ColoredPoint p(10,20,"red");
20 move(p); // upcasting reference ColoredPoint& to Point&
```

Static and dynamic binding

- *binding* = for a function call, determining which function to execute
- **in C:** easy — the name of function determines the function
- **in C++:** more complicated because of function overloading — still this is done during compilation: *static binding*
- **in C++:** in addition, C++ allows *dynamic binding*: which function to call is decided during runtime (only for virtual methods)

Example of static binding: static.cpp

Example of dynamic binding: dynamic.cpp

Virtual methods

- declared with the keyword `virtual`
- a virtual method of the base class stays virtual in all subclasses (hence, the keyword `virtual` for the method in a subclass is optional)
- will cause a *dynamic binding* if a virtual method is called:
  - makes difference for references or pointers to an object
  - the method which will be called does not depend on type of pointer or reference, but on the actual type of the object!
- **Note:** using virtual methods, we can not only reuse the old code (e.g., use the code for class `Point` to define class `ColoredPoint`), but we can make the old code to call the new code!

  Example: point.cpp

When to use virtual methods?

- methods which are expected to be modified in the derived classes should be declared as virtual
  (if the programmer who wants to derive a new class from old class doesn't have access to the code of old class, there is no way how he can make a non-virtual method of the old class virtual)
- hence, if it's expected that a class is going to have subclasses, it's good idea to declared the *destructor* as virtual
  Example: destructor2.cpp
- *Q.* why not make all functions virtual?
  - a bit less efficient
  - each class with virtual methods has a *virtual function table*
  - dynamic binding (during runtime): 1) find the type of the object, 2) access the corresponding virtual table 3) call the method using a pointer from the table

Sample questions

- What's the problem with the following code:

```
1  class A {
2    int x;
3   public:
4    A(int ix) { x=ix;}
5  };
6  class B : public A {
7   public:
8    void set(int nx) { x=nx; }
9  };
```

- Consider the following program:

```
1   class A {
2    public:
3     A() { cout <<1; }
4     A(int v) { cout <<2; }
5     ~A() { cout <<3; }
6   };
7   class B : public A {
8    public:
9     B() : A(10) { cout <<4; }
10    B(int v) { cout <<5; }
11    ~B() { cout <<6; }
12  };
13  int main() {
14    { B x; }
15    { B x(100); }
16    return 0;
17  }
```

What is the output of the above program after executing

1. the first block
2. the second block?

- What is upcasting? Why is it safe to use upcasting?
- What is dynamic binding? When and how is it used?

  Show an example in which both static and dynamic binding are used (your example will probably contain: definitions of base and derived classes and a short code which exhibits these two types of binding).
- Show an example of a code with two classes `B` and `D`, where `D` is derived from `B` in which an "old code" (a method in class `B`) is calling a "new code" (a method in class `D`).

Inheritance and dynamic memory allocation

- assume that the base class uses dynamic memory allocations

Example: array1.cpp

```
1  class intArray { // base class
2    int *data;
3    long size;
4   public:
5    intArray(long s);
6    intArray(const intArray &); // copy constructor
7    virtual ~intArray();
8    intArray & operator=(const intArray & a);
9    // assignment operator
10   int get(long index) const; // get a value
11   void set(long index, int value); // set a value
12 };
```

- we want to *derive* a new class from this base class
- *Question.* Do we always have to *redefine* the copy constructor and the assignment operator?

Case 1 — derived class doesn't use operator new

review:

- *copy constructor* — if not provided, the default copy constructor performs *memberwise copying*
- any constructor of the derived class will call some constructor of the base class (if not specified, the default base class constructor)

however

- the default copy constructor of the derived class calls the *copy constructor of the base class* instead! (exception to the above rules)
- *memberwise copying* is applied only to added data members

hence

- the copy constructor doesn't have to be redefined
- similarly, the *default assignment operator* uses the base class assignment operator for the base class data, and *memberwise assignment* for the added data, hence, doesn't need to be redefined

Example: let's derive class `MinMaxIntArr` from `intArray` which keeps the minimal and the maximal values stored in the array

- add new data: `minval` and `maxval`
- since the values of array are initialized to zeros, we can initialize `minval` and `maxval` to zeros as well
- the only way how to modify an element in the array is to use method `set(long,int)`
  hence, this method should be declared *virtual* in the base class and modified in `MinMaxIntArr` to keep the values of `minval` and `maxval` correct

Example: array2.cpp

Case 2 — derived class uses operator new

Example:

```
1  class NamedIntArr : public intArray {
2   private:
3    char *name;
4   public:
5    NamedIntArr(long size,const char* name)
6      : intArray(size)
7    {
8      name = new char[strlen(array.name)+1];
9      strcpy(name,array.name);
10   }
11   ...
12 };
```

- we have to modify the copy constructor, the assignment operator and the destructor of the derived class

- *destructor* — destructor for the base class will be called automatically, so we only need to clean-up after added data members
  Example:

```
1 NamedIntArr::~NamedIntArr() { delete [] name; }
```

- *copy constructor* — we can use the initializer list to invoke copy constructor of the base class, and then copy the added data members
  Example:

```
1 NamedIntArr::NamedIntArr(const NamedIntArr & array)
2    : intArray(array)
3 // calls base class copy constructor
4 {
5   name = new char[strlen(array.name)+1];
6   strcpy(name,array.name);
7 }
```

- *assignment operator* — we should call the assignment operator for the base class to copy the base class data, and then copy the added data

  Example:

```
1  NamedIntArr & NamedIntArr::operator=
2    (const NamedIntArr & array)
3  {
4    if (this == &array)
5      return *this;
6    intArray::operator=(array);
7    // we have to use this notation
8    // to use the scope operator
9    delete [] name;
10   name = new char[strlen(array.name)+1];
11   strcpy(name,array.name);
12   return *this;
13 }
```

Example: array3.cpp

Keyword protected

- members of the class can be declared also in `protected` section:

```
1 class X {
2  private:
3   int a;
4  protected:
5   int b;
6   void f();
7  public:
8   void g();
9 };
```

- outside of the class `X` protected members behave as *private members*:

```
1   X x;
2   x.b=2; // compiler error
3   x.f(); // compiler error
```

- but in methods of the derived class they behave as *public members*:

```
1   class Y : public X {
2     void h(); };
3   void Y::h() {
4     a=1; // compiler error
5     b=2; // ok
6     f(); // ok }
```

- **Remark:** Protected methods from the base class will stay protected in a derived class.
- why to use `protected`?— simplifies the code of derived class and speeds up access
- why not to use `protected`?— less *secure*: derived class can manipulate with data of the base class in inappropriate manner
- when to use `protected`?— basically, should be used for auxiliary functions of the base class, which are not supposed to be part of *public interface*, but can be useful for derived class

Abstract base classes

- abstract class contains at least one pure virtual function: it has `=0` in the end of its declaration
  Example:

```
1   class X {
2     virtual void f() =0; // pure virtual function
3   };
```

- a pure *virtual function* does not need to be defined in the base class
- it's not possible to create objects of an *abstract class*
- a derived class of an *abstract class* becomes "*concrete*" if it *(re)defines* all *pure virtual functions*
- hence, if somebody wants to derive class which can be used (to create objects), he has to define all *pure virtual functions*
- **Remark:** you can still create pointers and references to the abstract classes

usage:

- consider classes representing circle, ellipse, triangle, rectangle, star, etc.
- for all of them we can implement (add to the public interface) functions like: `draw()`, `move()`, `rotate()`, `area()` etc.
- hence, it make sense to define one superclass `Shape` and derive all classes representing particular shape from it
- however, the class `Shape` is too *abstract*
  - we don't know how to rotate an object of class `Shape`,
  - we just know that every concrete subclass of `Shape` should be able to rotate its content
- **solution:**
  make class `Shape` *abstract* and functions (like `rotate()`, etc.) *pure virtual functions*

Example: shape.cpp

Dynamic casting

Question: What if we have an array of pointer to objects derived from `Shape` and we want to perform specific task for every object of type `Circle` in the array?

Answer: We can use dynamic (downcasting) casting:

```
1 for (int i=0; i<size; i++) {
2   Circle* p = dynamic_cast<Circle*>(array[i]);
3   if (p!=NULL) {
4     cout <<"Radius: "<<p->getRadius()<<endl;
5   }
6 }
```

- **Note:** if it doesn't succeed the `NULL` pointer is returned.

Sample questions

- Assume that the base class uses dynamic memory allocations. Does it affect the decision whether you need to provide destructor, copy constructor and assignment operator for the derived class?
- Explain access rules for `protected` members of a class. Compare them with access rules for `public` and `private` members.
- Show an example of a class which does not allow to create its objects. Consider a derived class of this class. When is it possible to create objects of the derived class?
- How can you declare a pure virtual function? How are the classes containing pure virtual functions called? What is special about such classes (i.e., how they differ from ordinary classes)?