CSE 165/ENGR 140 Intro to Object Orient Program

Lecture 9.5 – Inheritance/Derivation

Quiz

- 10 mins to complete the quiz on CATCourses

Access control for inheritance

```
class Rect
{
    public :
    float xa, ya, xb, yb;
};

class RoundedRect : public Rect // 1) we are deriving with public
                                //    access to Rect's members
{
    public :
    float cornerDist;
};

void main ()
{
   RoundedRect r;
   r.xa = 1.0;            // 2) we have public access to the
                          //    members of the base class
   r.cornerDist = 0.2;    // 3) we also have public access to the
                          //    members of RoundedRect
}
```

Access control

```
class Rect
{
    float xa, ya, xb, yb; // 1) => these members are now private
};

class RoundedRect : public Rect // 2) we are still deriving with
                                //    public access
{
   public :
    float cornerDist;
};

void main ()
{
   RoundedRect r;
   r.xa = 1.0;                // 3) error: public derivation will not
                              //    break access control of the base class
   r.cornerDist = 0.2;
}
```

Access control

```
class Rect
{
   protected :
    float xa, ya, xb, yb; // 1) these members are now protected
};

class RoundedRect : public Rect // 2) we are still deriving with
                                //    public access
{
   public :
    float cornerDist;
    void setXa ( float f ) { xa = f; } // 3) => set gives access to xa
};

void main ()
 {
   RoundedRect r;
   r.xa = 1.0;             // 4) error: public derivation will not
                           //    break access control of the base class
   r.setXa ( 1.0 );        // 5) ok
   r. cornerDist = 0.2;
 }
```

Inheritance Protection

class derived_class_name: **public** base_class_name {...}

public members of base are **public** in derived

protected members of base are **protected** in derived

class derived_class_name: **protected** base_class_name {...}

public members of base are **protected** in derived

protected members of base are **protected** in derived

class derived_class_name: **private** base_class_name {...}

public members of base are **private** in derived

protected members of base are **private** in derived

Access control: private derivation

```
class X
{
   private:       int privx;
   protected:     int protx;
   public :       int publx;
};

class Y : private X // private derivation
{
   public :
    void protset ( int i ) { protx=i; } // 2) Ok
    void publset ( int i ) { publx=i; } // 3) Ok
};

void main ()
{
   Y y;
   y.privx=1; // 4) error
   y.protx=2; // 5) error
   y.publx=3; // 6) error
}
```

Everything in X is private through Y now!

Inheritance access matrix

Access	Public	Protected	Private
Same class member	Yes	Yes	Yes
Derived class member	Yes	Yes	No
Non-member	Yes	No	No

Constructors of Derived Classes

```
// Example of typical constructors in a class:
class Rect
{
   public :
    float x, y, w, h; // rectangle upper-left corner (x,y) and size (w,h)

    Rect () { x=y=w=h=0; }  // 1) Default constructor declared in-line

    Rect ( const Rect& r )  // 2) Copy constructor, takes in an object of same type
     { x=r.x; y=r.y; w=r.w; h=r.h; }

    Rect ( float rx, float ry, float rw, float rh ) // 3) Another constructor
     { x=rx; y=ry; w=rw; h=rh; }
};
```

Constructors of Derived Classes

```
// Constructors in a derived class must call
// the correct constructors of the base class:
class RoundedRect : public Rect
 {
   public :
    float cornerLen; // how much to round on each corner

    RoundedRect () { cornerLen=0; }       // 1) Default constructor of base class
                                                automatically called

    RoundedRect ( const RoundedRect& r ) // 2) Copy Constructor declaration
     :Rect(r)                            // 3) Calling copy constructor of Rect
     { cornerLen=r.cornerLen; }


    RoundedRect ( float rx, float ry, float rw, float rh, float len )
     :Rect(rx,ry,rw,rh),   // 4) Calling constructor of base class
      cornerLen(len)       // 5) Calling float "pseudo-constructor"
      { }
 };
```

Constructors of Derived Classes

- The parenthesis syntax for constructors can be used in several ways:

```
// 1) Example of "pseudo-constructors" :
int i(100); // same as int i=100;
int* ip = new int(47); // different than new int[47]!

// 2) Default constructor of an object automatically called:
Rect r; // no need to use ()

// 3) Primitive types do not have default constructors!
int i; // no initialization done here

// 4) Object initialization will call the copy constructor:
Rect a;    // will call default constructor
Rect b=a;  // will call copy constructor, same as Rect b(a)
Rect c(a); // will call copy constructor, same as Rect c=a
```

Order of Constructors

- The constructor of a base class is always called before the constructor of its derived class.
- The same rule applies to long chains of derivation:

```
class A
 { };

class B : public A
 { };

class C : public B
 { };

 ...
```

Upcasting

- Casting an object type to the type of its base class (as a pointer or reference)
- So that classes can work with objects of known behavior (methods), even if an object may actually be of a derived type

```
class A  {  };

class B : public A {  };

void main ()
 {
   B bobject;
   B* bpt = &bobject; // 1) get a pointer to bobject

   A* a = (A*) bpt;   // 2) upcast bpt to a pointer to A:
                         always ok since "B is also A"
 }
```

Redefining versus overriding methods

- Redefinition of Methods
  - Methods with same name in a base and derived classes are disambiguated by the type of the object
- Overriding Methods
  - The ***virtual*** keyword allows to call a descendant method even if the object being used is of the base class type
  - Makes sense only when upcasting is used
- Polymorphism
  - The use of virtual methods is a key concept behind polymorphism
  - To be covered when we get to Chapter 15

Redefining versus overriding methods

```
class Animal
 { public:
    void eat () { cout<<"I eat generic food\n"; }
    virtual void fur () { cout<<"I have fur\n"; }
 };

class Cat : public Animal
 { public:
    void eat () { cout<<"I eat cat food\n"; } // 1) method redefined
    void fur () { cout<<"I have fluffy cat fur\n"; } // 2) overrided!
  };

void main ()
 {
   Cat cat;
   cout<<cat.eat();
   Animal* animal = (Animal*) &cat; // 3) upcast cat to a pointer to Animal
   cout<< animal->eat();   // 4) will print: "I eat generic food"
   cout<< animal->fur();   // 5) will print: "I have fluffy cat fur"
 }
```

Wake up

Pointers to functions

- A function pointer is a variable that stores the address of a function
- It allows a function to change its behavior when it is called separately
  - The same sort of function can either sort in an ascending or descending way
  - A compare function can be passed as an argument
- It enables “callback functions” or “event listener”
  - Passing a function to another function as an argument
  - In a graphic user interface, a function is called when a mouse click takes place

Pointers to functions

```
int add(int a, int b){
  return a + b;
}
int subtract(int a, int b){
  return a - b;
}

int main ()
{
  int num1, num2;
  char addOrSubtract = ‘a’;

  int (*myMath)(int, int);
  if(addOrSubtract == ‘a’){
    myMath = add;
  }
  else{
    myMath = subtract;
  }
  int answer = myMath(num1, num2);
  cout<<"Answer is: "<<answer<<endl;
}
```

References

- References are always tied to someone else's storage
  - When you change the value of a reference you are always changing the value of someone else's variable/object
- Similar to pointers, BUT:
  - References always manipulate someone else's storage
    - References cannot be null
  - References must be initialized
    - You cannot declare a reference without initialization
  - A reference cannot be changed to refer to something else
    - Assignment will assign contents, not make the reference to reference another object

References

```cpp
//: C11:FreeStandingReferences.cpp
  #include <iostream>
  using namespace std;

  // Ordinary free-standing reference:
  int y;
  int& r = y;          // (1) When a reference is created, it must
                       // be initialized to an existing object.

  const int& q = 12;   // (2) This is valid (note the const)

  // References are always tied to someone else's storage:
  int x = 0;
  int& a = x;          // (3) a is a reference to x

  int main() {
    a++;               // (4) we are actually incrementing x here
  }
```

References in functions

- References are commonly used as function arguments and return values
  - Any modification to the reference inside the function will cause changes to the argument outside the function

```
//: C11:Reference.cpp - Simple C++ references

int* f(int* x) { // 1) pointer passed to a function
  (*x)++;
  return x;      // Safe, x is outside this scope
}

int& g(int& x) { // 2) reference passed to a function
  x++;           // Same effect as in f()
  return x;      // Safe, x is outside this scope
}
```

References in functions

```
//: C11:Reference.cpp - continue

int& h() {          // 3) function returning a reference
  int q;
  return q;         // 3.1) this would generate an error since q is local

  static int x;     // 3.2) static makes x become a global variable
  return x;         // Safe, x lives outside this scope (even if not visible)
}

int g(int& a){
}

int main() {
  int a = 0;
  g(&a);            // Sending a pointer to a to f: ugly (but explicit)
  g(a);             // Sending a reference to a to g: clean (but hidden)
}
```

Passing a pointer by reference

```
//: C11:ReferenceToPointer.cpp

#include <iostream>
using namespace std;

void increment(int*& i) { i++; }  //Passing the reference of a pointer

int main() {
  int* i = 0; //i is a pointer
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

Output:

```
i = 0
i = 0x4
```