CSCI 104
Inheritance

Mark Redekopp
David Kempe
Files for Today

- $ mkdir inh
- $ cd inh
- $ wget http://ee.usc.edu/~redekopp/cs104/inh.tar
- $ tar xvf inh.tar
- $ make
C++ constructors often have a bunch of assignments and initializations to the data members.
• Rather than writing many assignment statements we can use a special initialization list technique for C++ constructors
  – Constructor(param_list) : member1(param/val), ..., memberN(param/val)
    { ... }

• We are really calling the respective constructors for each data member
You can still assign values in the constructor but realize that the **default constructors** will have been called already.

So generally if you know what value you want to assign a data member it's **good practice** to do it in the initialization list.
Constructor Initialization Lists

Person::Person() { }
Person::Person(string myname)
{ name_ = myname;
  id_ = -1;
}
Person::Person(string myname, int myid)
{ name_ = myname;
  id_ = myid;
}
...

String Operator=() Called

Initialization using assignment

Person::Person() { }
Person::Person(string myname) : name_(myname), id_(-1)
{ }
Person::Person(string myname, int myid) :
  name_(myname), id_(myid)
{ }
...

String Copy Constructor Called

Memory is allocated before the '{' ...
...then values copied in when assignment performed

Memory is allocated and filled in "one-step"

Initialization List approach

String name_
int id_
name_ = myname
id_ = myid
name_ = myname
id_ = myid
name_ = myname
id_ = myid
INHERITANCE
Object Oriented Design

• Encapsulation
  – Combine data and operations on that data into a single unit (e.g. a class w/ public and private aspects)

• Inheritance
  – Creating new objects (classes) from existing ones

• Polymorphism
  – Using the same expression to denote different operations
Inheritance

• A way of defining interfaces, re-using classes and extending original functionality
• Allows a new class to inherit all the data members and member functions from a previously defined class
• Works from more general objects to more specific objects
  – Defines an “is-a” relationship
  – Square is-a rectangle is-a shape
  – Square inherits from Rectangle which inherits from Shape
  – Similar to classification of organisms:
    • Animal -> Vertebrate -> Mammals -> Primates
Base and Derived Classes

- Derived classes inherit all data members and functions of base class
- Student class inherits:
  - `get_name()` and `get_id()`
  - `name_` and `id_` member variables

```cpp
class Person
{
    public:
        Person(string n, int ident);
        string get_name();
        int get_id();
    private:
        string name_; int id_;}

class Student: public Person
{
    public:
        Student(string n, int ident, int mjr);
        int get_major();
        double get_gpa();
        void set_gpa(double new_gpa);
    private:
        int major_; double gpa_;}
```
Base and Derived Classes

- Derived classes inherit all data members and functions of base class
- Student class inherits:
  - `get_name()` and `get_id()`
  - `name_` and `id_` member variables

```cpp
class Person {
public:
    Person(string n, int ident);
    string get_name();
    int get_id();
private:
    string name_; int id_;  
};
class Student : public Person {
public:
    Student(string n, int ident, int mjr);
    int get_major();
    double get_gpa();
    void set_gpa(double new_gpa);
private:
    int major_; double gpa_;  
};

int main()
{
    Student s1("Tommy", 1, 9);
    // Student has Person functionality
    // as if it was written as part of
    // Student
    cout << s1.get_name() << endl;
}
```
Inheritance Example

- **Component**
  - Draw()
  - onClick()
- **Window**
  - Minimize()
  - Maximize()
- **ListBox**
  - Get_Selection()
- **ScrollView**
  - onScroll()
- **DropDownBox**
  - onDropDown()
Constructors and Inheritance

• How do we initialize base class data members?
• Can't assign base class members if they are private

```cpp
class Person {
  public:
    Person(string n, int ident);
    ...
  private:
    string name_;  
    int id_;  
};
class Student : public Person {
  public:
    Student(string n, int ident, int mjr);
    ...
  private:
    int major_;  
    double gpa_;  
};
Student::Student(string n, int ident, int mjr)  
{
  name_ = n;  // can't access name_ in Student
  id_ = ident;
  major_ = mjr;
}  
```
Constructors and Inheritance

• Constructors are only called when a variable ‘enters scope’ (i.e. is created) and cannot be called directly
  – How to deal with base constructors?
• Also want/need base class or other members to be initialized before we perform this object's constructor code
• Use initializer format instead
  – See example below

```cpp
class Person {
    public:
    Person(string n, int ident);
    ...
    private:
    string name_;  
    int id_; 
};
class Student : public Person {
    public:
    Student(string n, int ident, int mjr);
    ...
    private:
    int major_; 
    double gpa_; 
};
Student::Student(string n, int ident, int mjr) 
{
    // How to initialize Base class members?  
    Person(n, ident); // No! can’t call Construc. as a function 
}
Student::Student(string n, int ident, int mjr) : Person(n, ident) 
{
    cout << "Constructing student: " << name_ << endl; 
    major_ = mjr;  
    gpa_ = 0.0;
}
```
Constructors & Destructors

• Constructors
  – A Derived class will automatically call its Base class constructor **BEFORE** it's own constructor executes, either:
    • Explicitly calling a specified base class constructor in the initialization list
    • Implicitly calling the default base class constructor if no base class constructor is called in the initialization list

• Destructors
  – The derived class will call the Base class destructor automatically **AFTER** it's own destructor executes

• General idea
  – Constructors get called from base->derived (smaller to larger)
  – Destructors get called from derived->base (larger to smaller)
### Constructor & Destructor Ordering

```cpp
class A {
    int a;
public:
    A() { a=0; cout << "A:" << a << endl; }
    ~A() { cout << "~A" << endl; }
    A(int mya) { a = mya;
        cout << "A:" << a << endl; }
};
class B : public A {
    int b;
public:
    B() { b = 0; cout << "B:" << b << endl; }
    ~B() { cout << "~B "; }
    B(int myb) { b = myb;
        cout << "B:" << b << endl; }
};
class C : public B {
    int c;
public:
    C() { c = 0; cout << "C:" << c << endl; }
    ~C() { cout << "~C "; }
    C(int myb, int myc) : B(myb) {
        c = myc;
        cout << "C:" << c << endl; }
};
```

```cpp
int main()
{
    cout << "Allocating a B object" << endl;
    B b1;
    cout << "Allocating 1st C object" << endl;
    C* c1 = new C;
    cout << "Allocating 2nd C object" << endl;
    C c2(4,5);
    cout << "Deleting c1 object" << endl;
    delete c1;
    cout << "Quitting" << endl;
    return 0;
}
```
 Protected Members

• Private members of a base class can not be accessed directly by a derived class member function
  – Code for print_grade_report() would not compile since ‘name_’ is private to class Person

• Base class can declare variables with protected storage class
  – Private to anyone not inheriting from the base
  – Derived classes can access directly
Public/Private/Protected Access

- Derived class sees base class members using the base class' specification
  - If Base class said it was public or protected, the derived class can access it directly
  - If Base class said it was private, the derived class cannot access it directly
- public/private identifier before base class indicates HOW the public base class members are viewed by clients (those outside) of the derived class
  - public => public base class members are public to clients (others can access)
  - private => public & protected base class members are private to clients (not accessible to the outside world)

```cpp
class Person {
  public:
    Person(string n, int ident);
    string get_name();
    int get_id();
  private: // INACCESSIBLE TO DERIVED
    string name_; int id_;}

class Student : public Person {
  public:
    Student(string n, int ident, int mjr);
    int get_major();
    double get_gpa();
    void set_gpa(double new_gpa);
  private:
    int major_; double gpa_;
};

class Faculty : private Person {
  public:
    Faculty(string n, int ident, bool tnr);
    bool get_tenure();
  private:
    bool tenure_;}
```
Inheritance Access Summary

• Base class
  – Declare as protected if you want to allow a member to be directly accessed/modified by derived classes

• Derive as public if...
  – You want users of your derived class to be able to call base class functions/methods

• Derive as private if...
  – You only want your internal workings to call base class functions/methods

<table>
<thead>
<tr>
<th>Inherited Base</th>
<th>Public</th>
<th>Protected</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Public</td>
<td>Protected</td>
<td>Private</td>
</tr>
<tr>
<td>Protected</td>
<td>Protected</td>
<td>Protected</td>
<td>Private</td>
</tr>
<tr>
<td>Private</td>
<td>Private</td>
<td>Private</td>
<td>Private</td>
</tr>
</tbody>
</table>

External client access to Base class members is always the more restrictive of either the base declaration or inheritance level

class Person {
  public:
    Person(string n, int ident);
    string get_name();
    int get_id();
  private: // INACCESSIBLE TO DERIVED
    string name_; int id_;  
};

class Student : public Person {
  public:
    Student(string n, int ident, int mjr);
    int get_major();
    double get_gpa();
    void set_gpa(double new_gpa);
  private:
    int major_; double gpa_; 
};

class Faculty : private Person {
  public:
    Faculty(string n, int ident, bool tnr);
    bool get_tenure();
  private:
    bool tenure_; 
};

int main(){
  Student s1("Tommy", 73412, 1);
  Faculty f1("Mark", 53201, 2);
  cout << s1.get_name() << endl; // works
  cout << f1.get_name() << endl; // fails
}
When to Inherit Privately

- Suppose I want to create a FIFO (First-in, First-Out) data structure where you can only
  - Push in the back
  - Pop from the front
- FIFO is a special List
- Do I want to inherit publicly from List
- NO!!! Because now the outside user can call the base List functions and break my FIFO order
- Inherit privately to hide the base class public function and make users go through the derived class' interface
  - Private inheritance defines an "as-a" relationship

```cpp
class List{
  public:
    List();
    void insert(int loc, const int& val);
    int size();
    int& get(int loc);
    void pop(int loc);
  private:
    IntItem* _head;
};

Base Class

class FIFO : public List // or private List
{ public:
  FIFO();
  push_back(const int& val)
    { insert(size(), val); } // or private
  int& front();
    { return get(0); } // or private
  void pop_front();
    { pop(0); } // or private
};

Derived Class

FIFO f1;
  f1.push_back(7); f1.push_back(8);
  f1.insert(0,9)
```
Overloading Base Functions

• A derived class may want to redefine the behavior of a member function of the base class
• A base member function can be overloaded in the derived class
• When derived objects call that function the derived version will be executed
• When a base objects call that function the base version will be executed

```cpp
class Car{
public:
    double compute_mpg();
private:
    string make; string model;
};

double Car::compute_mpg()
{
    if(speed > 55) return 30.0;
    else return 20.0;
}

class Hybrid : public Car {
public:
    void drive_w_battery();
    double compute_mpg();
private:
    string batteryType;
};
double Hybrid::compute_mpg()
{
    if(speed <= 15) return 45; // hybrid mode
    else if(speed > 55) return 30.0;
    else return 20.0;
}
```
Scoping Base Functions

- We can still call the base function version by using the scope operator (::)
  - `base_class_name::function_name()`

```cpp
class Car{
   public:
      double compute_mpg();
   private:
      string make; string model;
};

class Hybrid : public Car {
   public:
      double compute_mpg();
   private:
      string batteryType;
};

double Car::compute_mpg()
{
   if(speed > 55) return 30.0;
   else return 20.0;
}

double Hybrid::compute_mpg()
{
   if(speed <= 15) return 45; // hybrid mode
   else return Car::compute_mpg();
}
```
Inheritance vs. Composition

• Software engineers debate about using *inheritance (is-a)* vs. *composition (has-a)*

• Rather than a Hybrid “is-a” Car we might say Hybrid “has-a” car in it, plus other stuff
  – Better example when we get to Lists, Queues and Stacks

• While it might not make complete sense verbally, we could re-factor our code the following ways...

• Interesting article I’d recommend you read at least once:

```cpp
class Car {
public:
    double compute_mpg();
public:
    string make; string model;
};

double Car::compute_mpg() {
    if(speed > 55) return 30.0;
    else return 20.0;
}

class Hybrid {
public:
    double compute_mpg();
private:
    Car c_; // has-a relationship
    string batteryType;
};
double Hybrid::compute_mpg() {
    if(speed <= 15) return 45; // hybrid mode
    else return c_.compute_mpg();
}
```
Another Composition

- We can create a FIFO that "has-a" a List as the underlying structure
- Summary:
  - Public Inheritance => "is-a" relationship
  - Composition => "has-a" relationship
  - Private Inheritance => "as-a" relationship "implemented-as"

```cpp
class List{
public:
    List();
    void insert(int loc, const int& val);
    int size();
    int& get(int loc);
    void pop(int loc);
private:
    IntItem* _head;
};
```

**Base Class**

```cpp
class FIFO
{
private:
    List mylist;
public:
    FIFO();
    push_back(const int& val)
    { mylist.insert(size(), val); }
    int& front();
    { return mylist.get(0); }
    void pop_front();
    { mylist.pop(0); }
    int size() // need to create wrapper
    { return mylist.size(); }
};
```