C++ Inheritance & Encapsulation

Technion - Israel Institute of Technology

Updated: Winter 2016-2017
C++ Inheritance and Encapsulation

- Private and Protected members
- Inheritance Type
- Public Inheritance
- Private Inheritance
- Protected Inheritance
- Special method inheritance
Private Members

• Private members can only be accessed by other class members and class friends

• Cannot be accessed by derived classes
  – Benefit: inheritance does not break encapsulation
    • Limiting cross-class dependencies matters even when one class inherits the other
  – Downside: limits extensibility of subclasses
Protected Members

• protected members can be accessed by other class members, friends and derived classes
  – Benefit: fully extensible by inheritance
  – Downside: limits flexibility of class provider
    • Base class can no longer be changed without worrying about the external effect of the change
<table>
<thead>
<tr>
<th>Visibility</th>
<th>Public</th>
<th>Protected</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visibility</td>
<td>all</td>
<td>members</td>
<td>members</td>
</tr>
<tr>
<td></td>
<td></td>
<td>friends</td>
<td>friends</td>
</tr>
<tr>
<td></td>
<td></td>
<td>derived classes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encapsulation</th>
<th>Public</th>
<th>Protected</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>none</td>
<td>partial</td>
<td>full</td>
</tr>
<tr>
<td>Extensibility</td>
<td>Maximal</td>
<td>Maximal</td>
<td>Restricted</td>
</tr>
<tr>
<td></td>
<td>subclass has full access to base</td>
<td>subclass has full access to base</td>
<td>subclass has limited access to base</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifiability</th>
<th>Public</th>
<th>Protected</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modifiability</td>
<td>Minimal</td>
<td>Restricted</td>
<td>Maximal</td>
</tr>
<tr>
<td></td>
<td>changes affect all users</td>
<td>changes do affect subclass</td>
<td>changes are hidden from subclass</td>
</tr>
</tbody>
</table>
Inheritance Types

• In a way, a sub-object is like a member variable
  – when first field, memory layout is similar
• Public inheritance:
  – most commonly used
  – specified using the `public` keyword:
  – `class Derived: public Base {...};`
  – Inheritance relation is visible outside the class
Inheritance Types (2)

• **Private inheritance:**
  – Rarely used
  – specified using the `private` keyword:
  ```cpp
  class Derived: private Base {...};
  ```
  – Inheritance relation is invisible outside the class

• **Protected inheritance:**
  – Similar to private, visible to subclasses

• **Defaults:**
  – `struct`: public inheritance
  – `class`: private inheritance
  – Compilers might warn if default is used.

**DO NOT USE THE DEFAULTS!**
Example of use of Private Inheritance

```cpp
class Car: private Engine {
    //...
};
```

- The fact that `Car` inherits from `Engine` is `private`
  - So the existence of an `Engine` sub-object in `Car` is `private`
- Only from inside `Car` it is possible to:
  - Cast from `Car` to `Engine`
    - Unless using C-style casts: `(Engine*)carPtr` does work. **Avoid that.**
  - Call public or protected member functions of `Engine`
  - Access public or protected member variables of `Engine`
Outside View of Inheritance Types

class Super {} *p;

class PublicInherit: public Super {} *p1;
class ProtectedInherit: protected Super {} *p2;
class PrivateInherit: private Super {} *p3;

void OutsiderFunc() {
    p = p1;    // OK

    p = p2;    // Error! protected fact that type of *p2
               is a subtype of *p

    p = p3;    // Error! private fact that type of *p3
               is a subtype of *p
}
class Super {} *p;

class PublicInherit: public Super {} *p1;
class PrivateInherit: private Super {
  void f();
} *p3;

void PrivateInherit::f()
{
  p = p1;  // OK
  p = p3;  // OK private fact that type of *p3
          is a subtype of *p
}
class Base {
    public: int pub;
} base;

class PublicInherit: public Base {}
public: int pubSub;

class ProtectedInherit: protected Base {}
protected: int protSub;

class PrivateInherit: private Base {}
private: int privSub;

void OutsiderFunc(){
    int i = pubSub.pub;  // OK
    i = protSub.pub;/// Error! protSub.pub is **protected**
    i = privSub.pub;/// Error! privSub.pub is **private**
}
More on Members’ Visibility

```cpp
class Base {
    public: int pub;
    protected: int prot;
};

class XBase: protected Base {};  
class X: public XBase { ... };

class YBase: private Base {};    
class Y: public YBase { ... };

void X::f() {
    int i = pub; // OK
    i = prot;   // OK
}

void Y::f() {
    int i = pub; // Error! pub is private
    i = prot;   // Error! prot is private
}
```
To increase the access of a member `pub1` of `Base` inherited from class `Derived`, use a using declaration:

```cpp
class Base {
public:
    int pub1,
    pub2;
protected:
    int prot;
private:
    int priv;
};

class Derived: private Base {
public:
    using Base::pub1;
    protected:
    using Base::pub2;
protected:
    using Base::prot;
};
```

You may increase the access of the following members:
- A member inherited as private
  - Visibility of private members cannot be increased, because they are not visible to the derived class
- A member either inherited or declared as protected
### Summary: Visibility in Inheritance Types

<table>
<thead>
<tr>
<th></th>
<th>Public Base Class</th>
<th>Protected Base Class</th>
<th>Private Base Class</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Public Members of</strong></td>
<td><strong>Public</strong> members of derived class</td>
<td><strong>Protected</strong> members of derived class</td>
<td><strong>Private</strong> members of derived class</td>
</tr>
<tr>
<td><strong>Base Class</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Protected Members of</strong></td>
<td><strong>Protected</strong> members of derived class</td>
<td><strong>Protected</strong> members of derived class</td>
<td><strong>Private</strong> members of derived class</td>
</tr>
<tr>
<td><strong>Base Class</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Private Members of</strong></td>
<td><strong>Not</strong> accessible in derived class</td>
<td><strong>Not</strong> accessible in derived class</td>
<td><strong>Not</strong> accessible in derived class</td>
</tr>
<tr>
<td><strong>Base Class</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Does it make sense for a **virtual method** to be:

- Public?
  - Sure
- Protected?
  - Sure
- Private?
  - Sometimes. This allows subclasses to customize some behavior (by overriding virtual function), without actually exposing implementation.
Can a pure virtual function have a body?
   – Yes, and that body can provide the base implementation
   – The class is still considered abstract and you do have to override the function, though
Does it make sense for destructors to be:

– Public?
  • Sure

– Protected?
  • Yes. This means the object could only be deleted by its own method (or friends)
  • Useful when using pools and factories

– Private?
  • Yes, similarly to the above
Can a *constructor* be:

- Virtual?
  - No. You always construct a concrete type, thus constructors are statically bound
  - *This won’t compile*

- Pure virtual?
  - Same as above
Can a destructor be:

– Virtual?
  • Yes, and it should be for any polymorphic class
  • This makes it possible to delete an object of derived type via a pointer to the base type

– Pure virtual?
  • Yes. As above, and it also makes the class abstract.
  • It must still have a body, though
Does it make sense for constructors to be:

– Public?
  • Sure
– Protected?
  • Yes. In case of constructor overloading, or by using the factory pattern (with inheritance).
– Private?
  • Yes, in case of constructor overloading, or by using the factory pattern.