C++ Inheritance and Encapsulation

- Protected members
- Inheritance Type
- Public Inheritance
- Private Inheritance
- Protected Inheritance
- Special method inheritance
Private vs. Protected

- **Private:** *private* members can be accessed only by member functions and friends.
  - Members cannot be accessed by derived classes.
  - Main Rational: otherwise, inheritance could be used to break encapsulation.
    - By inheriting from a class, a user may gain access to its implementation details.
  - Counter Argument: language enforced encapsulation is used to protect against accidents, not against malice!
    - In many cases it would be impossible to implement new functionality in a derived class without access to encapsulated members.

- **Protected:** *protected* members can be accessed by member functions, friends and derived classes.
  - Benefit: true extendibility
    - A class cannot be extended if it discloses vital details from its extender.
    - private members allow the designer to reserve the right to modify a class at the cost of putting restrictions on extendibility.
## Visibility Levels in C++

<table>
<thead>
<tr>
<th></th>
<th>private</th>
<th>protected</th>
<th>public</th>
</tr>
</thead>
<tbody>
<tr>
<td>visibility</td>
<td>• members</td>
<td>• members</td>
<td>any</td>
</tr>
<tr>
<td></td>
<td>• friends</td>
<td>• friends</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• derived classes</td>
<td></td>
</tr>
<tr>
<td>encapsulation</td>
<td>maximal</td>
<td>moderate</td>
<td>none</td>
</tr>
<tr>
<td>extendibility</td>
<td>restricted</td>
<td>maximal</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>subclass cannot even do the same as superclass</td>
<td>subclass can do whatever the superclass can</td>
<td></td>
</tr>
<tr>
<td>modifiability</td>
<td>maximal</td>
<td>restricted</td>
<td>minimal</td>
</tr>
<tr>
<td></td>
<td>changes hidden from subclass</td>
<td>changes affect subclass</td>
<td>changes affect all clients</td>
</tr>
</tbody>
</table>
Inheritance Type

• For many purposes: subobject is just like a **data member**.

• **What is the visibility of a subobject?**
  – **Public inheritance:**
    • Most commonly used
    • Specified by a `public` keyword:
      ```
      class Derived: public Base {...};
      ```
    • Inheritance/existence of a sub-object: visible outside the class
  – **Private inheritance:**
    • Rarely used
    • Specified by the `private` keyword:
      ```
      class Derived: private Base {...};
      ```
    • Inheritance/existence of a sub-object: visible only inside the class
  – **Protected inheritance:** similar to private inheritance, only visible in derived as well.

• **Defaults:**
  – `struct`: Public inheritance
  – `class`: Private inheritance
  – Many compilers (Cfront, BC, ...) warn if the default is used
Example of Private Inheritance

• Default for class inheritance (unfortunate?)

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

• The fact that Car inherits from Engine is private:
  – The existence of an Engine subobject in Car is private
  – Only from inside Car it is possible to:
    • convert Car to Engine
    • convert Car* to Engine*
    • convert Car& to Engine&
    • call public or protected function members of Engine
    • access public or protected data members of Engine
Rules of Inheritance Type

• Base class is just like a member; it can be declared:
  – public (default if subclass is struct)
  – protected
  – private (default if subclass is class)

• The inheritance type controls rights to
  – type cast to super-class,
  – access to public members of superclass,
  – access to protected members of superclass, and
  – access to private members of superclass.
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(void)
{
    p = p1; // OK

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

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

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

void PrivateInherit::f(void)
{
    p = p1;  // OK

    p = p3;  // OK
    // private fact: “type of *p3
    // is a subtype of *p”
Inheritance Types: Members’ Visibility

class Base {
    public: int pub;
} base;

class PublicInherit: public Base {} pubHeir;
class ProtectedInherit: protected Base {} protHeir;
class PrivateInherit: private Base {} privHeir;

void OutsiderFunc(void)
{
    int i = pubHeir.pub;  // OK

    i = protHeir.pub;     // Error!  
                           // protHeir.pub is protected

    i = privHeir.pub;     // Error! privHeir.pub is private
### More on Members’ Visibility

```c++
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(void) {
    int i = pub;  // OK
    i = prot;    // OK
}
void Y::f(void){
    int i = pub;  // Error! pub is private
    i = prot;    // Error! prot is private
```

*oop*
• Relies on a deprecated feature of the language.
• Works in most current compilers.
Technique relies on the using keyword.
## 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>Public members of</td>
<td>Protected members of</td>
<td>Private members of</td>
</tr>
<tr>
<td><strong>Base Class</strong></td>
<td>derived class</td>
<td>derived class</td>
<td>derived class</td>
</tr>
<tr>
<td><strong>Protected Members</strong></td>
<td>Protected members</td>
<td>Protected members of</td>
<td>Private members of</td>
</tr>
<tr>
<td><strong>of Base Class</strong></td>
<td>of derived class</td>
<td>derived class</td>
<td>derived class</td>
</tr>
<tr>
<td><strong>Private Members of</strong></td>
<td><strong>Not accessible</strong></td>
<td><strong>Not accessible</strong></td>
<td><strong>Not accessible</strong></td>
</tr>
<tr>
<td><strong>Base Class</strong></td>
<td>in derived class</td>
<td>in derived class</td>
<td>in derived class</td>
</tr>
</tbody>
</table>
Food for Thought

- Does it make sense for a virtual method to be
  - Public?
  - Protected?
  - Private?

- Can a pure virtual method have a body?

- Does it make sense for destructors to be
  - Public?
  - Protected?
  - Private?

- Can a constructor be
  - Virtual?
  - Pure virtual?

- Can a destructor be
  - Virtual?
  - Pure virtual?