Outline

1. Access Modes
2. Polymorphism
3. Abstract Classes
4. Assignment
Outline

1. Access Modes
2. Polymorphism
3. Abstract Classes
4. Assignment
Access to Inherited Members

- The three access modes.
  - Public
  - Protected
  - Private

- Base-class members and derived-class members have their own member-access modes (public, protected, private).

- The inheritance relation itself has a separate access mode (public, protected, private).
Access to Inherited Members

- We will normally use *public inheritance*.
- Public and protected members of the base class will be accessible from within the scope of the derived classes.
- Private members of the base class are accessible only from within the base class.
Public Inheritance Accessibility

Example (Public Inheritance)

class A
{
    public: int funcA;
    protected: int memA;
}
class B : public A
{
    public: int funcB;
    protected: int memB;
}
int main()
{
    A objA;
    B objB;
}
### Public Inheritance Accessibility

<table>
<thead>
<tr>
<th>Object</th>
<th>From Scope of A</th>
<th>From Scope of B</th>
<th>From Global Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcA()</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>funcB()</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objA.funcA()</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>objB.funcA()</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>objB.funcB()</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>memA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>memB</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objA.memA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objB.memA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objB.memB</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
We will not use private inheritance.

Private inheritance limits public access by way of class $B$ objects to the public members of class $B$.

However, from within the scope of class $B$, there is still access to the class $A$ public and protected members.
Private Inheritance Accessibility

Example (Private Inheritance)

```cpp
class A {
    public: int funcA();
    protected: int memA;
}

class B : private A {
    public: int funcB();
    protected: int memB;
}

int main()
{
    A objA;
    B objB;
}
```
## Public Inheritance Accessibility

<table>
<thead>
<tr>
<th>Object</th>
<th>From Scope of A</th>
<th>From Scope of B</th>
<th>From Global Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>funcA()</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>funcB()</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objA.funcA()</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>objB.funcA()</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objB.funcB()</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>memA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>memB</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objA.memA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objB.memA</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>objB.memB</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Outline

1. Access Modes
2. Polymorphism
3. Abstract Classes
4. Assignment
Polymorphism

**Definition (Polymorphism)**

*Polymorphism* is the ability of an object of one type to function as an object of a different type. The “actual” type of the object may not be determined until run time. This is called *late binding* or *dynamic binding*.

- A function that specifies a base-class object in its parameter list may accept a derived-class object in its place.
- Polymorphism works because the derived-class object *IS-A* base-class object.
- We have already seen this used in the constructors.
Polymorphism and Passing by Value

- If a function passes the base-class object \textit{by value}, then the derived-class object is considered to be an object of the base class.
- Why does this happen?
Polymorphism and Passing by Value

- If a function passes the base-class object *by value*, then the derived-class object is considered to be an object of the base class.

- Why does this happen?
- This happens because the base-class copy constructor was used to create the local object.
If a function passes the base-class object \textit{by value}, then the derived-class object is considered to be an object of the base class.

Why does this happen?

This happens because the base-class copy constructor was used to create the local object.

The local object loses its derived-class data members and functions.
Example (Polymorphism)

```cpp
int main()
{
    Man man("John");
    Woman woman("Jane");
    Describe(man);
    Describe(woman);
}

void Describe(Person p)
{
    cout << p << endl;  // Is p a Man, a Woman?
    return;
}
```
Demonstration

- Run the program `DescribePeople.cpp` to see what happens.
If the function passes the base-class object by reference, then the derived-class object is considered to be an object of the derived class.
Example

Example (Polymorphism)

```c++
int main()
{
    Man man("John");
    Woman woman("Jane");
    describe(man);
    describe(woman);
}

void describe(Person& p)
{
    cout << p << endl;  // Is p a Man, a Woman?
    return;
}
```
Demonstration

- In the program `DescribePeople.cpp`, make the `describe()` function parameter a reference parameter.
- Run the program to see what happens.
Virtual Functions

- When base class and a derived class have distinct functions of the same name, how does the compiler know which one to invoke?
- If the base-class function is `virtual`, then the computer will invoke the member function of that name that is closest to the class of the invoking object.
- Write the keyword `virtual` at the beginning of the function prototype.
Example (Virtual Functions)

class Person
{
    virtual void output(ostream& out) const
        out << name << ' ' << sex;
};
Example

Example (Virtual Functions)

```cpp
int main()
{
    Man man("John");
    Woman woman("Jane");
    describe(man);
    describe(woman);
}

void describe(Person& p)
{
    cout << p << endl;  // What will happen?
    return;
}
```
Demonstration

- In the file `person.h`, make `output()` a virtual function.
- Run the program `DescribePeople.cpp` to see what happens.
Virtual Functions and Value Parameters

What happens when the function is virtual and the parameter is a value parameter?
```cpp
int main()
{
    Man man("John");
    Woman woman("Jane");
    describe(man);
    describe(woman);
}

void describe(Person p)
{
    cout << p << endl;    // What will happen?
    return;
}
```
Demonstration

- In the program `DescribePeople.cpp`, make the function parameter a value parameter.
- Run the program to see what happens.
Pure Virtual Functions

- A function may be designated as a pure virtual function.
- Write
  
  ```
  virtual function(parameters) = 0;
  ```
- The function is *not* instantiated at this level in the class hierarchy.
- The function *must* be instantiated (sooner or later) in the derived classes.
Pure Virtual Functions

- This is done when
  - the function must be implemented at some level in the hierarchy,
  - but there is not enough information at the top (base) level to implement it there.

- Example?
Abstract Classes

Definition (Abstract Class)

An abstract class is a class that contains a pure virtual function. No object of an abstract class may be instantiated.

- Function parameters of an abstract class type must be passed by reference.
- Why?
Example

Example (Pure Virtual Functions)
class Person
{
    virtual void output(ostream& out) const = 0;
};
Example

Example (Pure Virtual Functions)

```cpp
int main()
{
    Man man("John");
    Woman woman("Jane");
    describe(man);
    describe(woman);
}

void describe(Person& p)
{
    cout << p << endl;   // What will happen?
    return;
}
```
Demonstration

- In the file `person.h`, make the `output()` function pure virtual.
- In `DescribePeople.cpp`, make the function parameter a reference parameter.
- Run the program to see what happens.
Demonstration

- In `DescribePeople.cpp`, make the function parameter a value parameter.
- Run the program to see what happens.
Example

Example (Abstract Class)

- Circles, squares, and triangles are shapes.
- **Create a Shape class as a base class.**
Example (Abstract Class)

- Shape
  - Circle
  - Rectangle
  - Triangle
Example (Abstract Class)

- Each shape has an area and a perimeter.
- However, we cannot find the area or perimeter until we know the particular shape.
- Therefore, Shape should be an abstract class.
Inheritance: Polymorphism and Virtual Functions

Robb T. Koether (Hampden-Sydney College) | Inheritance: Polymorphism and Virtual Functions | Mon, Mar 15, 2010 | 37 / 39

Example

Example (Abstract Class)

Shape
virtual area() = 0;
virtual peri() = 0;

Circle
area() = \pi r^2;
peri() = 2\pi r;

Rectangle
area() = bh;
peri() = 2(b + h);

Triangle
area() = (1/2)bh;
peri() = a + b + c;
Assignment

Homework

- Read Section 14.5, pages 810 - 822.
- Read Section 14.6, pages 823 - 827.