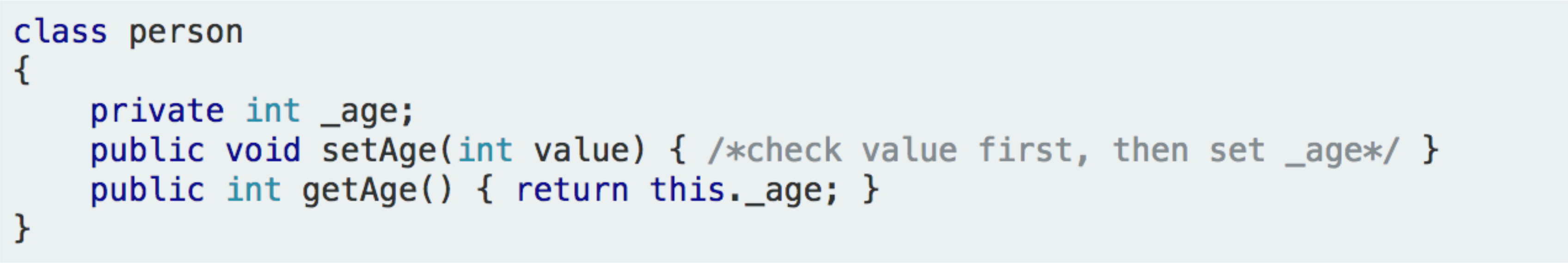
CS 206

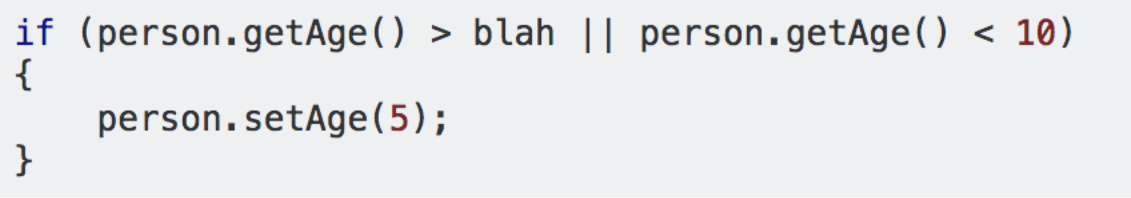
Lecture 20 – Object Orientation Wrap

Accessor Functions

- Many times, one has code like this



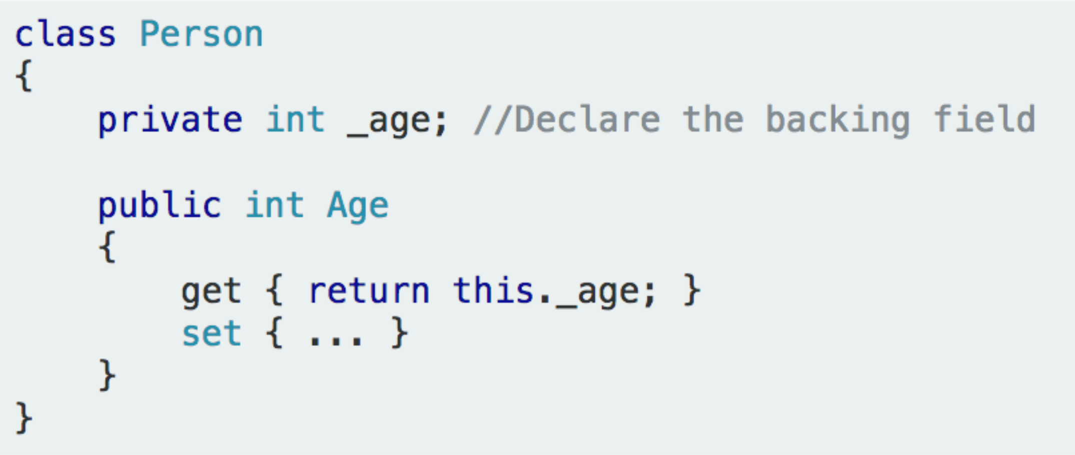
```
class person
{
    private int _age;
    public void setAge(int value) { /*check value first, then set _age*/ }
    public int getAge() { return this._age; }
}
```



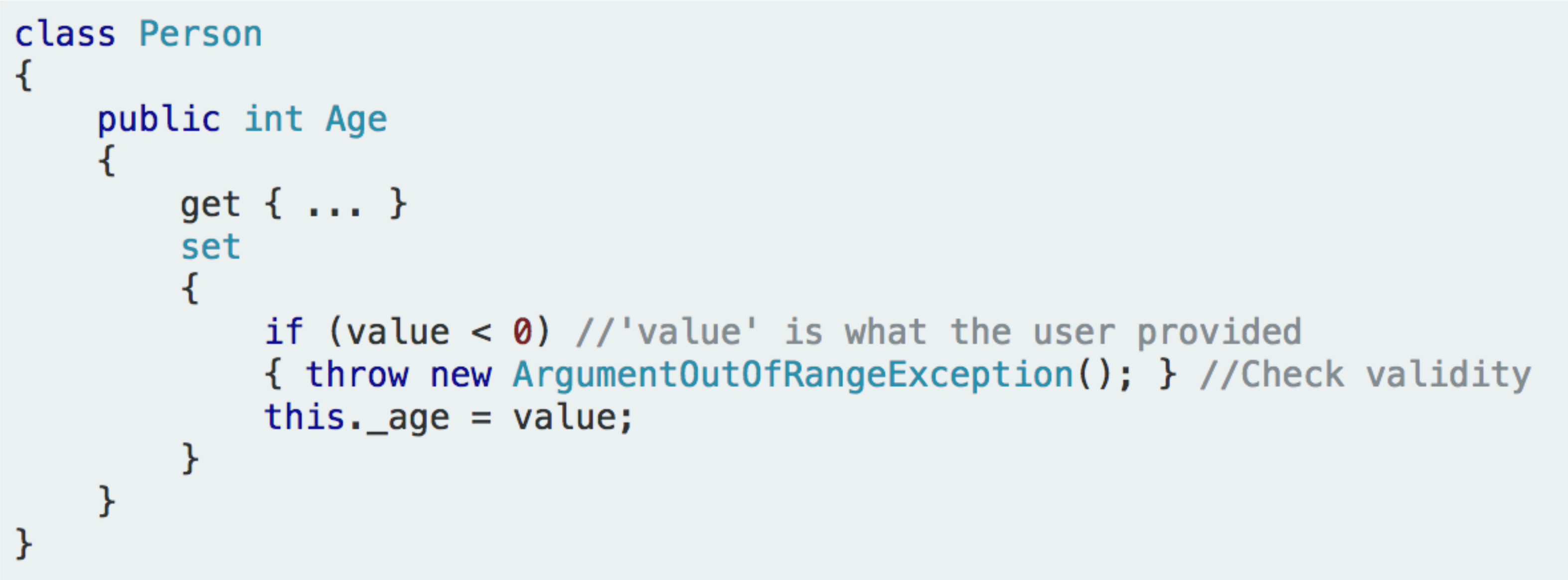
```
if (person.getAge() > blah || person.getAge() < 10)
{
    person.setAge(5);
}
```

- Providing direct access to the actual variable breaks encapsulation
    - Not recommended
- Instead, can use accessor functions

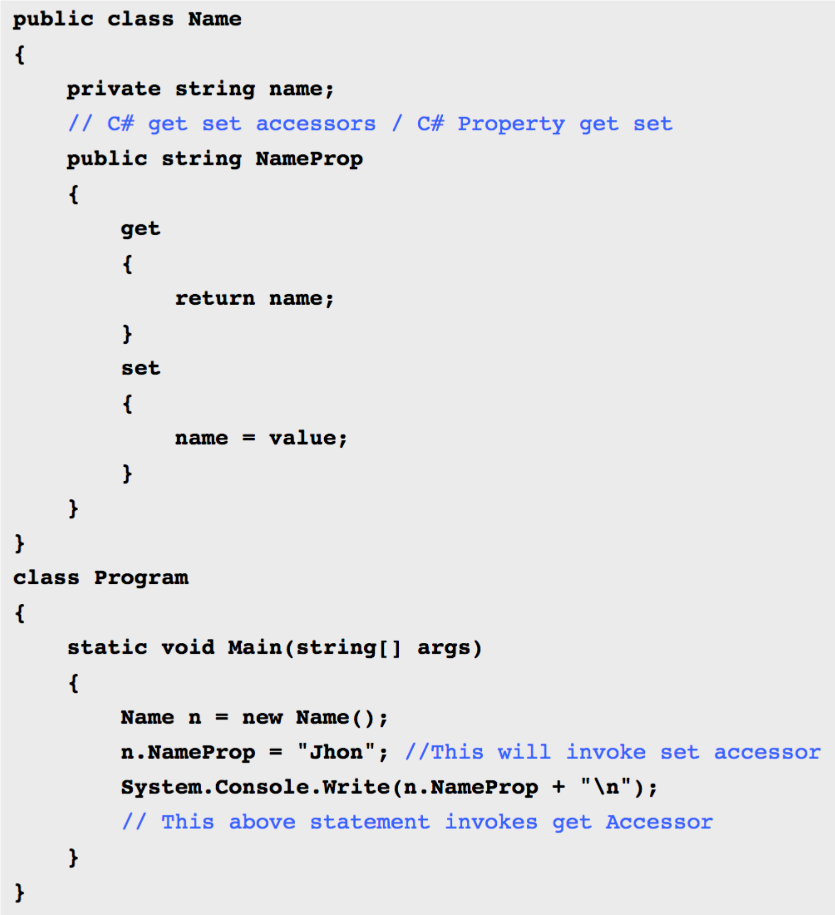
Accessor Functions



```
class Person
{
    private int _age; //Declare the backing field

    public int Age
    {
        get { return this._age; }
        set { ... }
    }
}
```



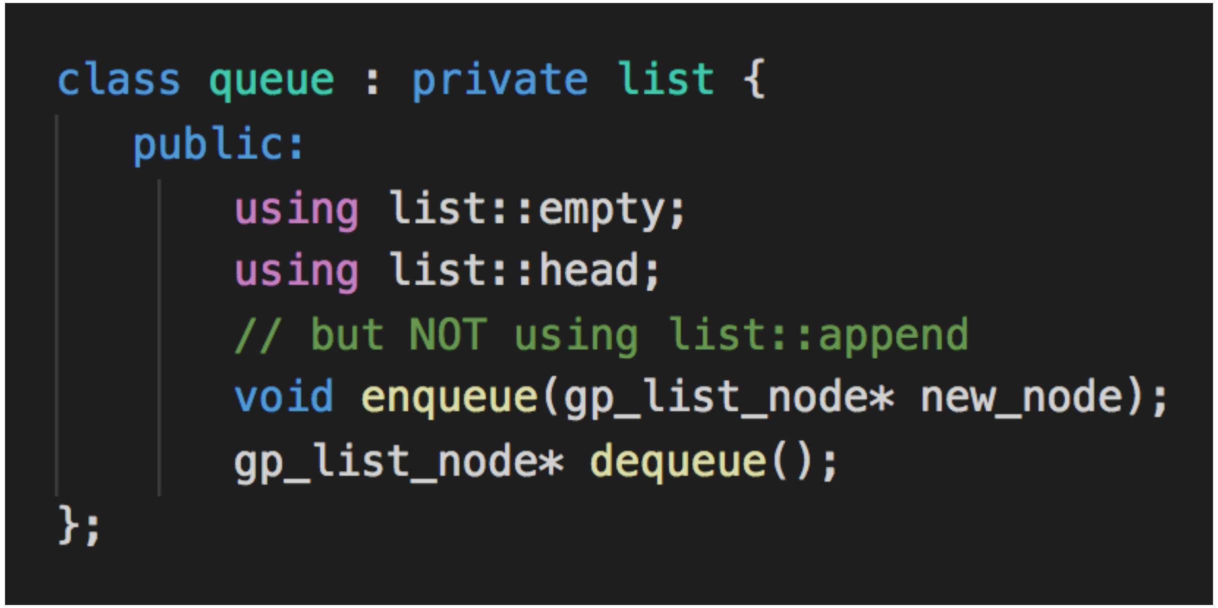
```
class Person
{
    public int Age
    {
        get { ... }
        set
        {
            if (value < 0) //'value' is what the user provided
            { throw new ArgumentOutOfRangeException(); } //Check validity
            this._age = value;
        }
    }
}
```



```
public class Name
{
    private string name;
    // C# get set accessors / C# Property get set
    public string NameProp
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Name n = new Name();
        n.NameProp = "Jhon"; //This will invoke set accessor
        System.Console.Write(n.NameProp + "\n");
        // This above statement invokes get Accessor
    }
}
```

https://stackoverflow.com/questions/6554210/what-is-the-purpose-of-accessors

Inheritance and Visibility



```
class queue : private list {
    public:
        using list::empty;
        using list::head;
        // but NOT using list::append
        void enqueue(gp_list_node* new_node);
        gp_list_node* dequeue();
};
```

What has changed here?

- Should private members of a base class be visible to methods of a derived class?
- Should public members of a base class always be public members of a derived class?
- Protected members
  - A protected member is visible only to methods of its own class or of classes derived from that class
  - Can also be used for base classes

C++ Rules

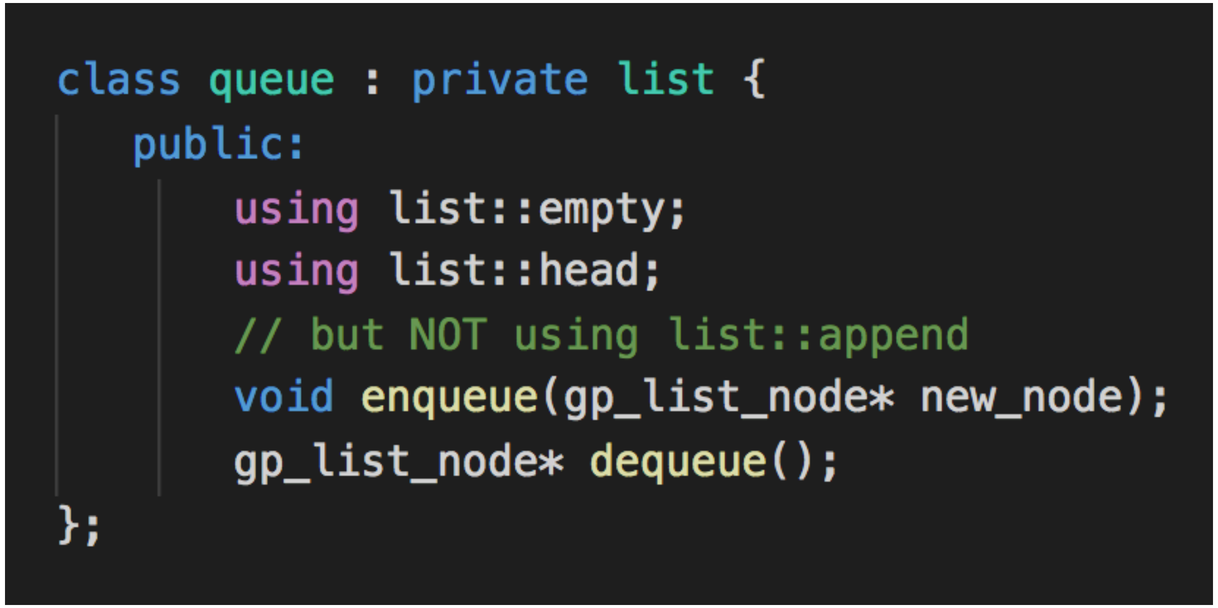
- Any class can limit the visibility of its members
  - Public members are visible anywhere the class declaration.
  - Private members are visible only inside the class’s methods.
  - Protected members are visible inside methods of the class or its descendants.
- A derived class can restrict the visibility of members of a base class, but can never increase it.
- Protected members
  - A protected member is visible only to methods of its own class or of classes derived from that class
  - Can also be used for base classes

Examples

- Private members are visible
    - Only inside a class' methods
- Protected members are visible
    - Methods of the class, or its descendants
- Private members of a base class are ____ visible in a derived class
    - Never
- class **derived** : public **base** {
    - What happens to **protected** and **public** members of **base** in **derived**?
    - Remain the same
- class **derived** : private **base** {
    - What happens to **protected** and **public** members of **base** in **derived**?
    - Become **private**
- class **derived** : protected **base** {
    - What happens to **protected** and **public** members of **base** in **derived**?
    - Become **protected**

Examples

- Can you change the visibility of members in derived classes?



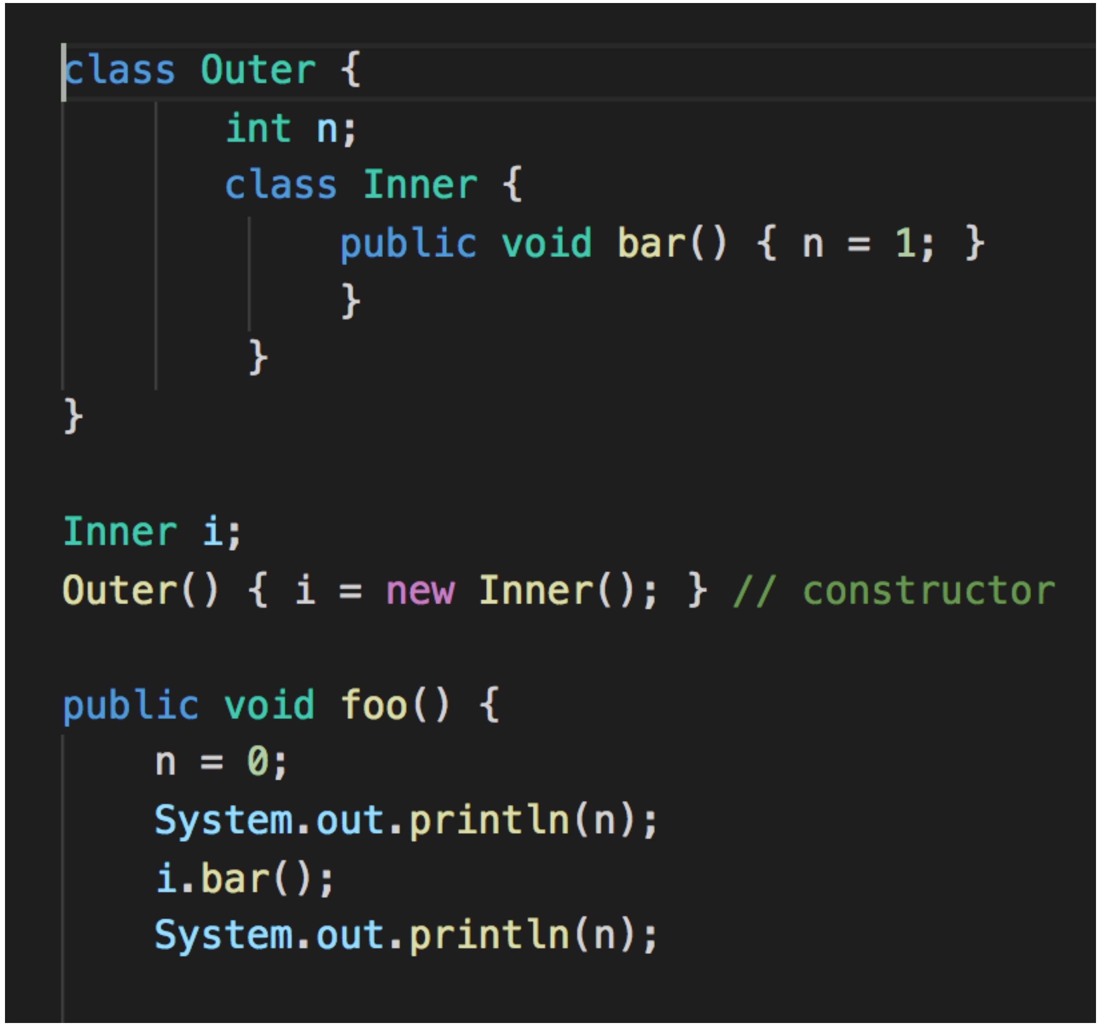
```
class queue : private list {
    public:
        using list::empty;
        using list::head;
        // but NOT using list::append
        void enqueue(gp_list_node* new_node);
        gp_list_node* dequeue();
};
```

Java and C#

- Have notions of public, **protected** and **private**
- However, do **<u>not</u>** allow the base classes to be **protected** or **private**
- **Implications?**
  - Derived class can neither increase nor restrict the visibility of base class members
- Protected in Java
  - Protected member is visible in derived class + entire package in which class is declared
- Member with no explicit modifier in Java is
  - Visible through the package in which class is declared
  - **<u>Not</u>** in derived classes that reside in other packages

Nesting Classes

- il **Inner** is a member of **Outer**, can **Inner's** methods see **Outer's** members, and if so, which instance do they see?
- C++
  - allow access to only the **static** members of the outer class
  - Serves as a means of information hiding
- Java
  - allows a nested (*inner*) class to access arbitrary members of its surrounding class

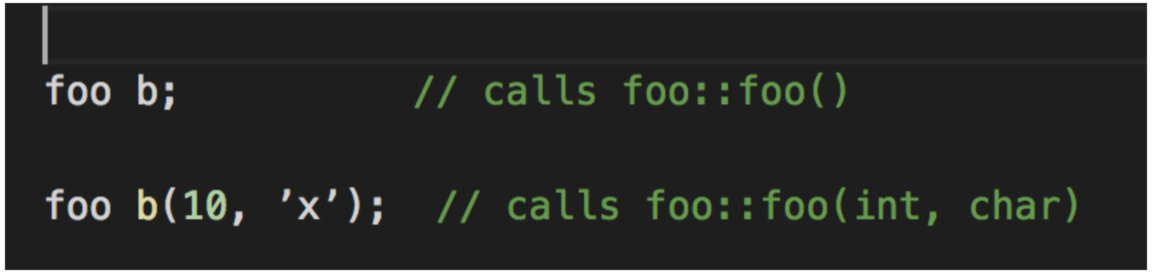


```
class Outer {
       int n;
       class Inner {
            public void bar() { n = 1; }
            }
        }
}

Inner i;
Outer() { i = new Inner(); } // constructor

public void foo() {
    n = 0;
    System.out.println(n);
    i.bar();
    System.out.println(n);
```

Choosing Constructors

- A class can have multiple constructor
  - Just like overloading methods
  - Some may take arguments also

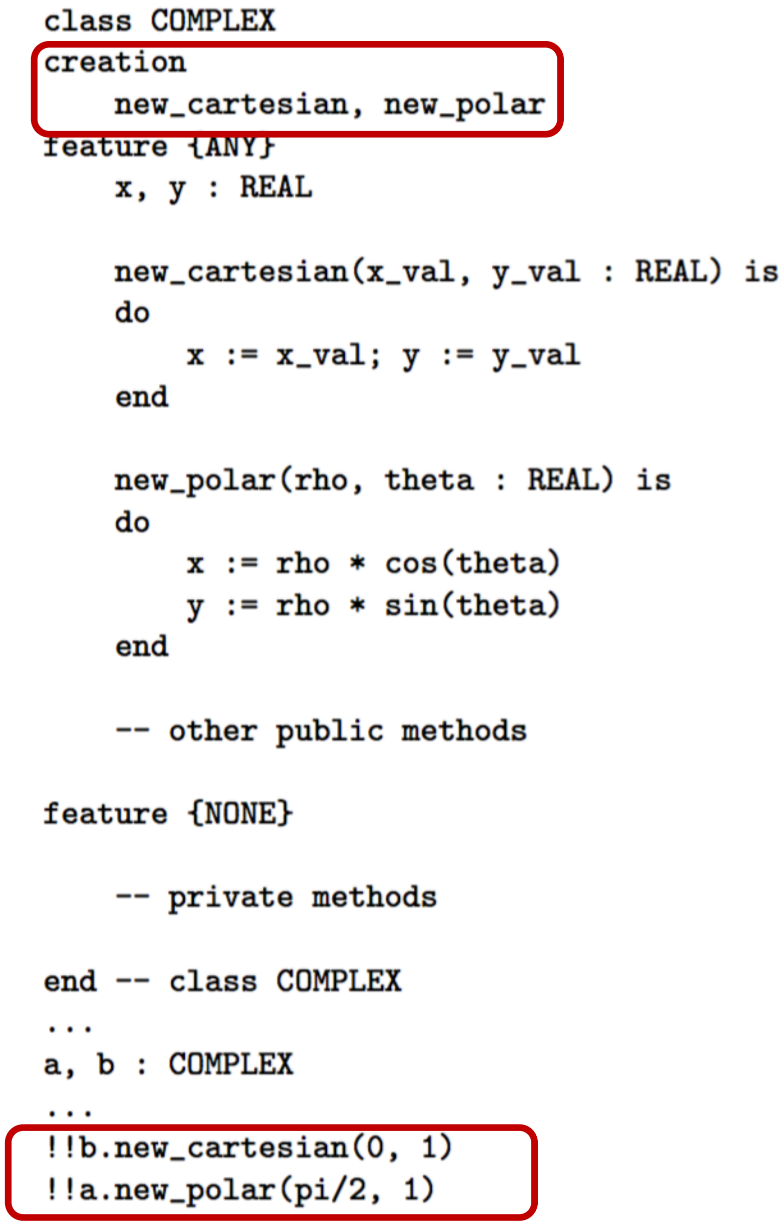
- C++ variable of class type **foo** is declared with no initial value, then the compiler will call foo’s zero-argument constructor



```
foo b;          // calls foo::foo()

foo b(10, ’x’);  // calls foo::foo(int, char)
```

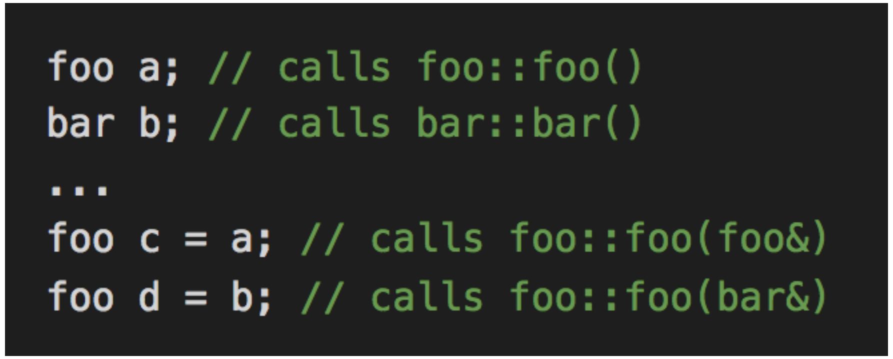
Constructors

- Some languages allow for constructors with different names
    - Eiffel example
- “creation” keyword specifies constructor methods



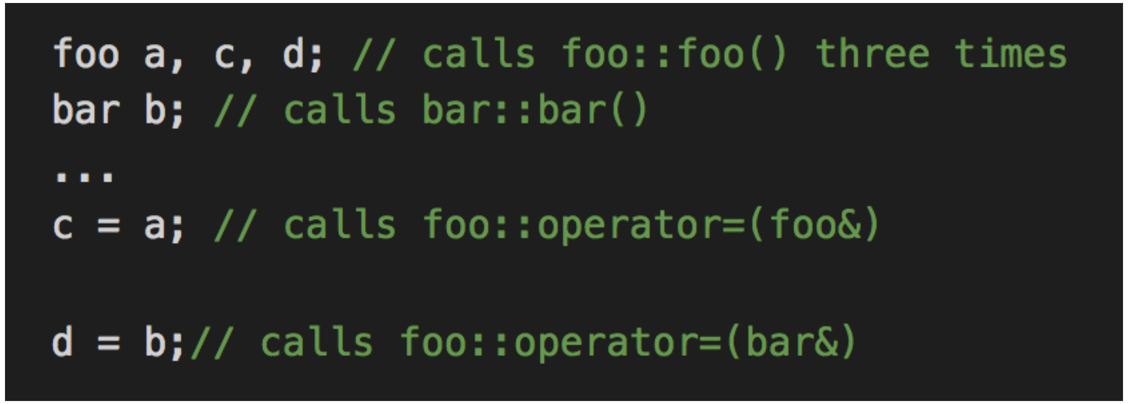
```
class COMPLEX
creation
    new_cartesian, new_polar
feature {ANY}
    x, y : REAL

    new_cartesian(x_val, y_val : REAL) is
    do
        x := x_val; y := y_val
    end

    new_polar(rho, theta : REAL) is
    do
        x := rho * cos(theta)
        y := rho * sin(theta)
    end

    -- other public methods

feature {NONE}

    -- private methods

end -- class COMPLEX
...
a, b : COMPLEX
...
!!b.new_cartesian(0, 1)
!!a.new_polar(pi/2, 1)
```

Assignment and Initialization



```
foo a; // calls foo::foo()
bar b; // calls bar::bar()
...
foo c = a; // calls foo::foo(foo&)
foo d = b; // calls foo::foo(bar&)
```

- programmer’s intent - declare a new object whose initial value is “the same” as that of the existing object
- C++ provides single-argument constructor called a *copy constructor*
- equals sign (=) in these declarations indicates **initialization**, not **assignment**.

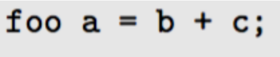


```
foo a, c, d; // calls foo::foo() three times
bar b; // calls bar::bar()
...
c = a; // calls foo::operator=(foo&)

d = b;// calls foo::operator=(bar&)
```

- c and d are initialized with the zero-argument constructor
- use of the equals sign indicates **assignment**, not **initialization**

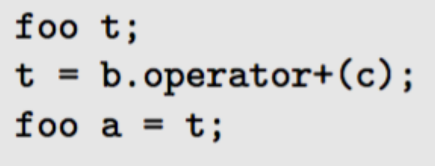
Assignment and Initialization

- For statement –



```
foo a = b + c;
```

- The following happens



```
foo t;
t = b.operator+(c);
foo a = t;
```

  - compiler creates a hidden, temporary object to be the target of the + operation
  - generated code will include calls to zero-argument constructor and the destructor for t
  - Will call a copy constructor to move t into a

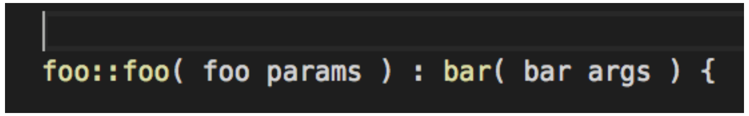
- Programmers who create explicit temporary objects to break up complex expressions may see similar unexpected costs

Execution Order of Constructors

- every object has to be initialized before it can be used
- If **B** is derived from A
    - An A constructor needs to be called before B constructor
- Why?
    - So that derived class is guaranteed never to see its inherited fields in an inconsistent state
- You create an object of type B
    - creation operation specifies arguments for a *B* constructor
    - Where does the compiler obtain arguments for the *A* constructor?

Constructor Execution Order

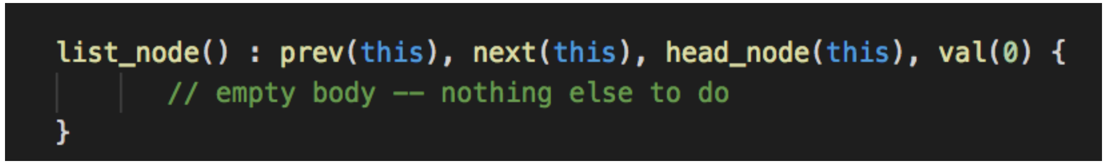
- Allow the header of the constructor of a derived class to specify base class constructor arguments:



```
foo::foo( foo params ) : bar( bar args ) {
```

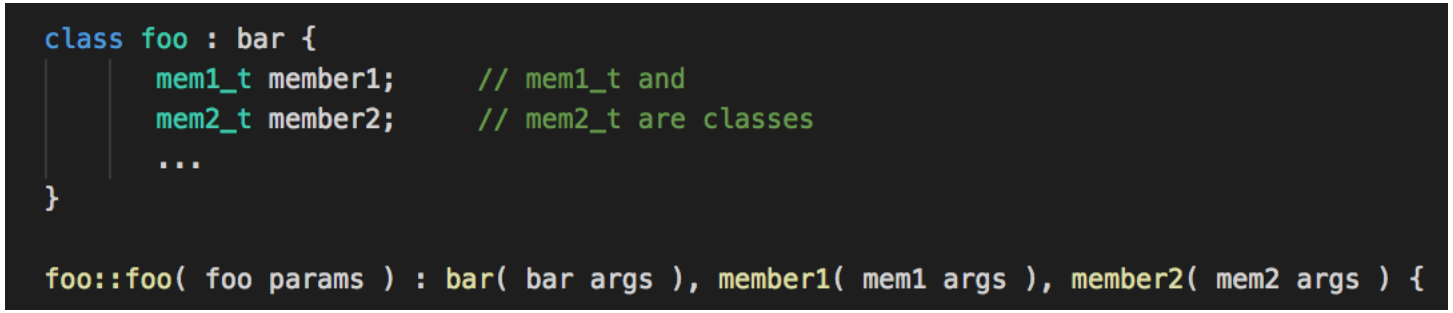
The list *foo params* consists of formal parameters for this particular foo constructor.

- C++ programmer can specify constructor arguments or initial values for members of the class



```
list_node() : prev(this), next(this), head_node(this), val(0) {
        // empty body -- nothing else to do
}
```

- If members are themselves objects of some nontrivial class



```
class foo : bar {
        mem1_t member1;     // mem1_t and
        mem2_t member2;     // mem2_t are classes
        ...
}

foo::foo( foo params ) : bar( bar args ), member1( mem1 args ), member2( mem2 args ) {
```

Multiple Inheritance

- In C++
  ```
  class professor : public teacher, public researcher {
          ...
      }
  ```
- We get all the members of teacher ***and*** all the members of researcher
  - If there's anything that's in both (same name and argument types), then calls to the member are ambiguous; the compiler disallows them

Multiple Inheritance

- Can create own member in the merged class

```
professor::print () {
    teacher::print ();
    researcher::print (); ...
}
```

- Or could get both:

```
professor::tprint () {
    teacher::print ();
}
professor::rprint () {
    researcher::print ();
}
```

Mix-in Inheritance

- Classes can inherit from only one “real” parent
- Can “mix in” any number of interfaces, simulating multiple inheritance

- Interfaces appear in Java, C#, Go, Ruby, etc.
  - contain only abstract methods, no method bodies or fields

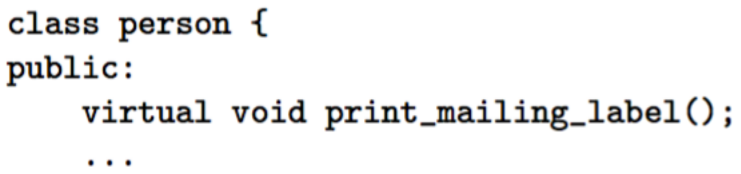
- In short, the key difference from an inheritance is that mix-ins does NOT need to have a "is-a" relationship like in inheritance.

Dynamic Binding

- Question:
- if child is derived from parent, have a `parent* p` that points to an object that's actually a child, what member function do I get when I call `p->f`?
  - Normally I get p's f, because p's type is parent*.
  - But if f is a virtual function, I get c's f.

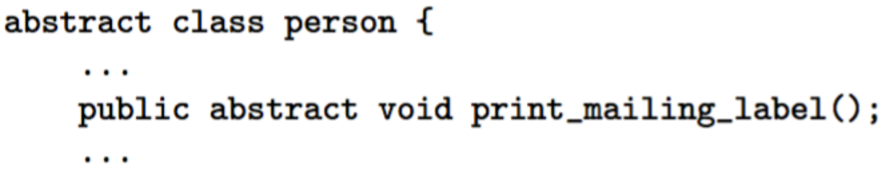
Virtual Functions and Their Use

- Using *virtual* functions the programmer can specify that particular methods should use dynamic binding
- Calls to virtual methods are *dispatched* to the appropriate implementation at run time, based on the class of the object



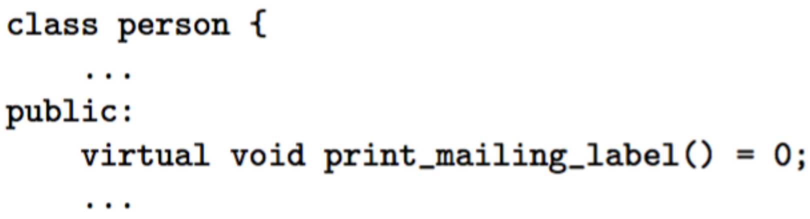
```
class person {
public:
    virtual void print_mailing_label();
    ...
```

- Sometimes, it is possible to omit the body of a virtual method in a base class : call the method *abstract*.



```
abstract class person {
    ...
    public abstract void print_mailing_label();
    ...
```

C#

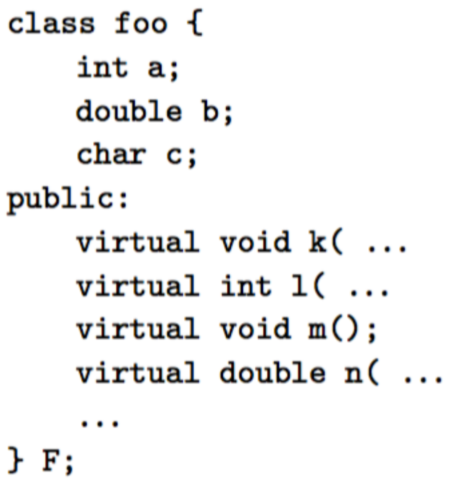


```
class person {
    ...
public:
    virtual void print_mailing_label() = 0;
    ...
```

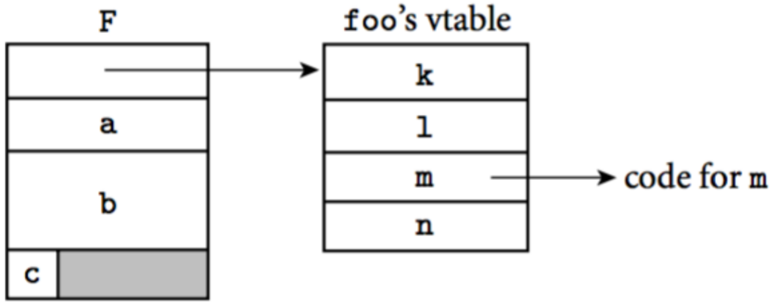
C++

Looking Up Members/Implementing Virtual Methods

- *vtable* : *virtual method table*
    - each object with a record whose first field contains the address of vtable

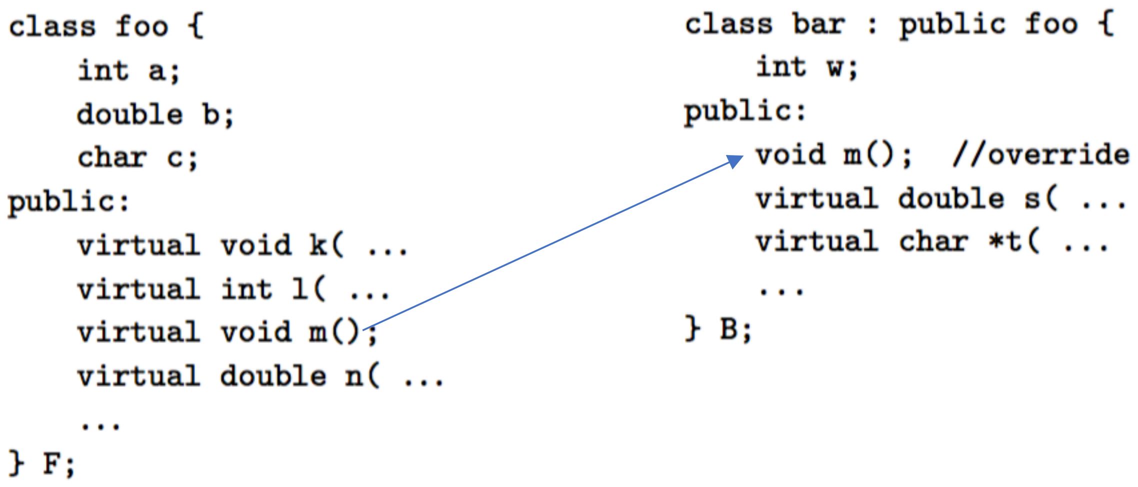


```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```



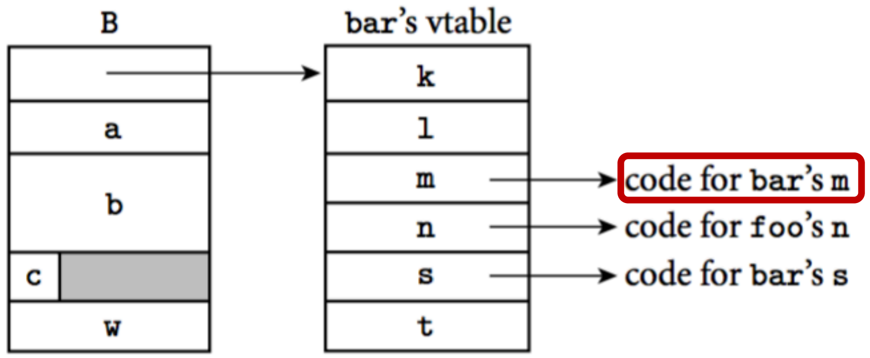
Looking Up Members/Implementing Virtual Methods

- For a derived object B
  - first 4 entries in the table represent the same members as they do for foo
  - m—has been overridden ; contains the address of the code for a different subroutine



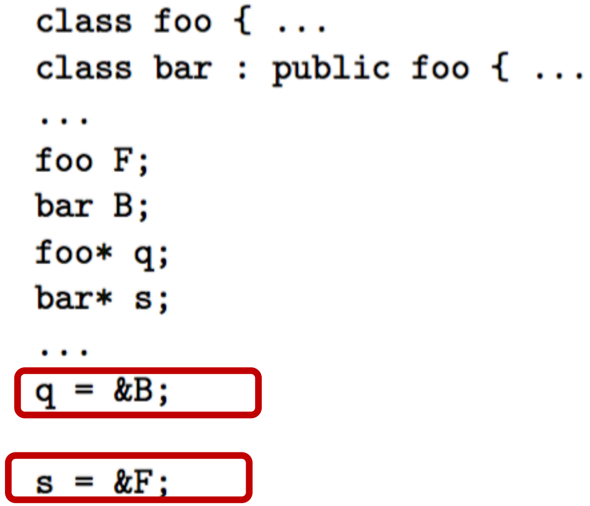
```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;
```

```
class bar : public foo {
    int w;
public:
    void m();  //override
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```



Example

- Is this OK?

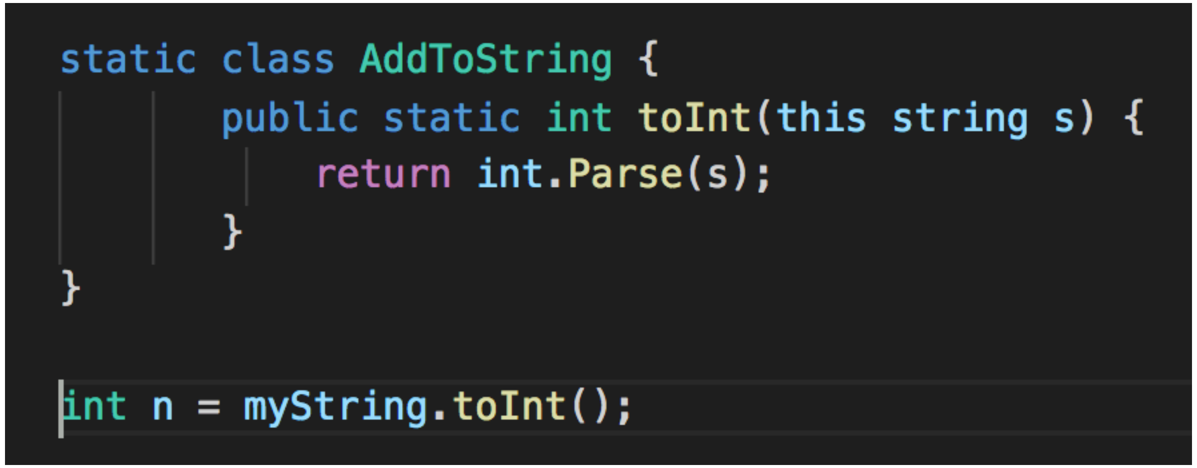


```
class foo { ...
class bar : public foo { ...
...
foo F;
bar B;
foo* q;
bar* s;
...
q = &B;

s = &F;
```

Extending without Inheritance

- Some inheritance is not an option -- dealing with preexisting code
- What if the class you want to extend doesn’t allow it? (**final** in Java)

- An extension method
  - must be **static**
  - must be declared in a **static** class.
  - first parameter must be prefixed with the keyword **this**.
  - The method can then be invoked as if it were a member of the class of which this is an instance:



```
static class AddToString {
        public static int toInt(this string s) {
            return int.Parse(s);
        }
}

int n = myString.toInt();
```

Encapsulation and Inheritance

- Example:

  ```
  class circle : public shape { ...
  ```

  anybody can convert (assign) a circle* into a shape*

  ```
  class circle : protected shape { ...
  ```

  only members and friends of circle or its derived classes can convert (assign) a circle* into a shape*

  ```
  class circle : private shape { ...
  ```

  only members and friends of circle can convert (assign) a circle* into a shape*