UNIVERSITY of LOUISVILLE
dare to be great

CPP Chapter 8:
Inheritance

CECS130
Introduction to Programming Languages
Dr. Roman V. Yampolskiy

Introduction to Inheritance

- Inheritance is a relationship between two or more classes where derived class inherites behaviour and attributes of pre-existing (base) classes
- Intended to help **reuse** existing code with little or no modification

Inheritance

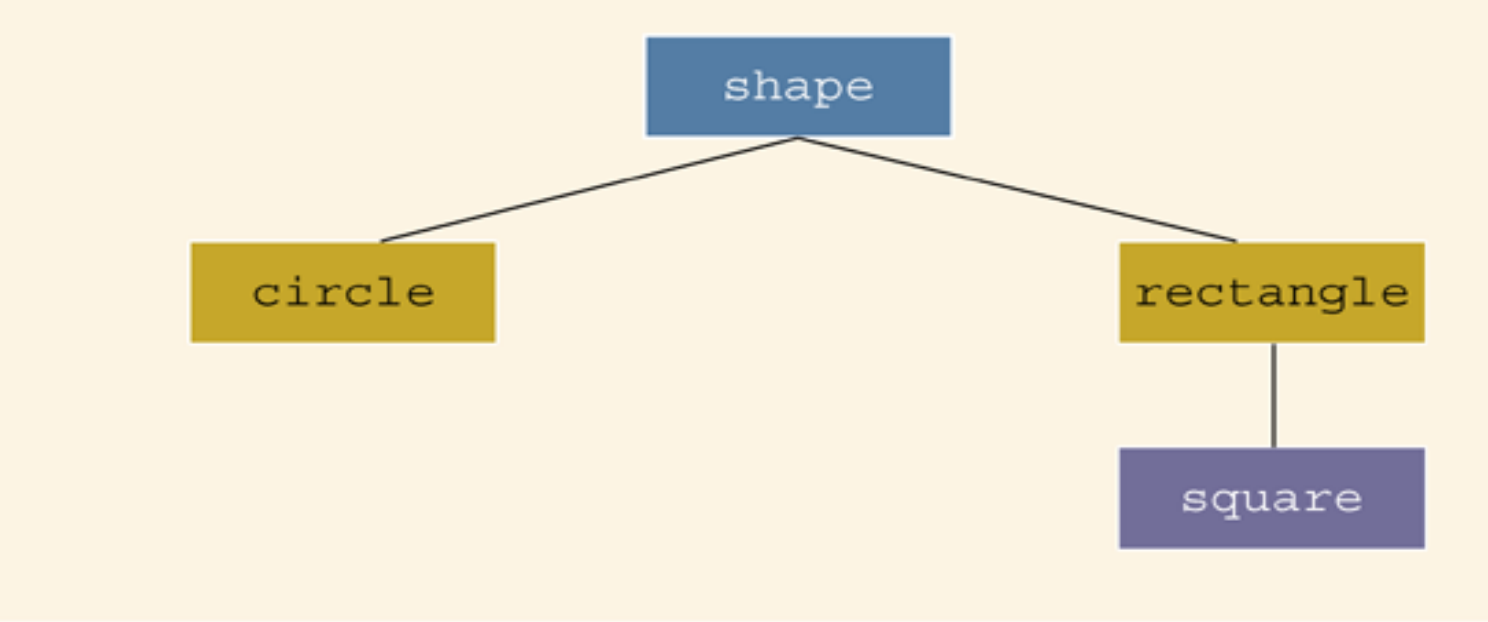
- Inheritance is an “is-a” relationship
- For instance: “every employee is a person”
- Inheritance lets us create new classes from existing classes
- New classes are called the **derived** classes
- Existing classes are called the **base** classes
- Derived classes inherit the properties of the base classes

Inheritance (continued)

- Single inheritance: derived class has a single base class
- Multiple inheritance: derived class has more than one base class
- Can be viewed as a tree (hierarchy) where a base class is shown with its derived classes

Inheritance Hierarchy

- Inheritance can be viewed as a tree-like, or hierarchical, structure wherein a base class is shown with its derived classes.



Inheritance hierarchy

The general syntax of a derived class is:

```
class className: memberAccessSpecifier baseClassName
{
    member list
};
```

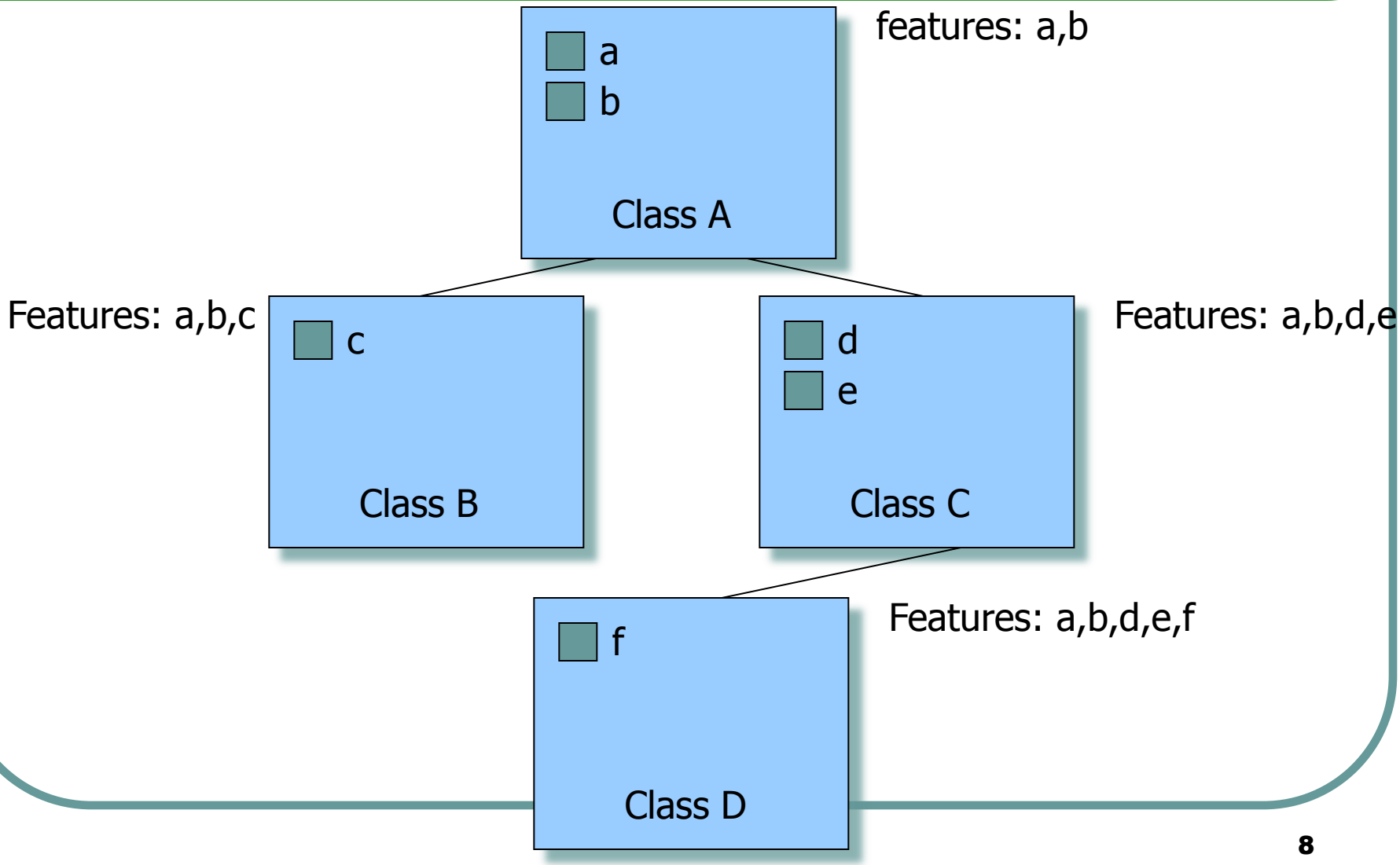
- Where `memberAccessSpecifier` is `public`, `protected`, or `private`.
- When no `memberAccessSpecifier` is specified, it is assumed to be a `private` inheritance.

Inheritance

- Inheritance can be chained
  - Derived class can inherit another class, which inherits another class and so on
  - When changing the base class all the derived classes will also change
  - Example:

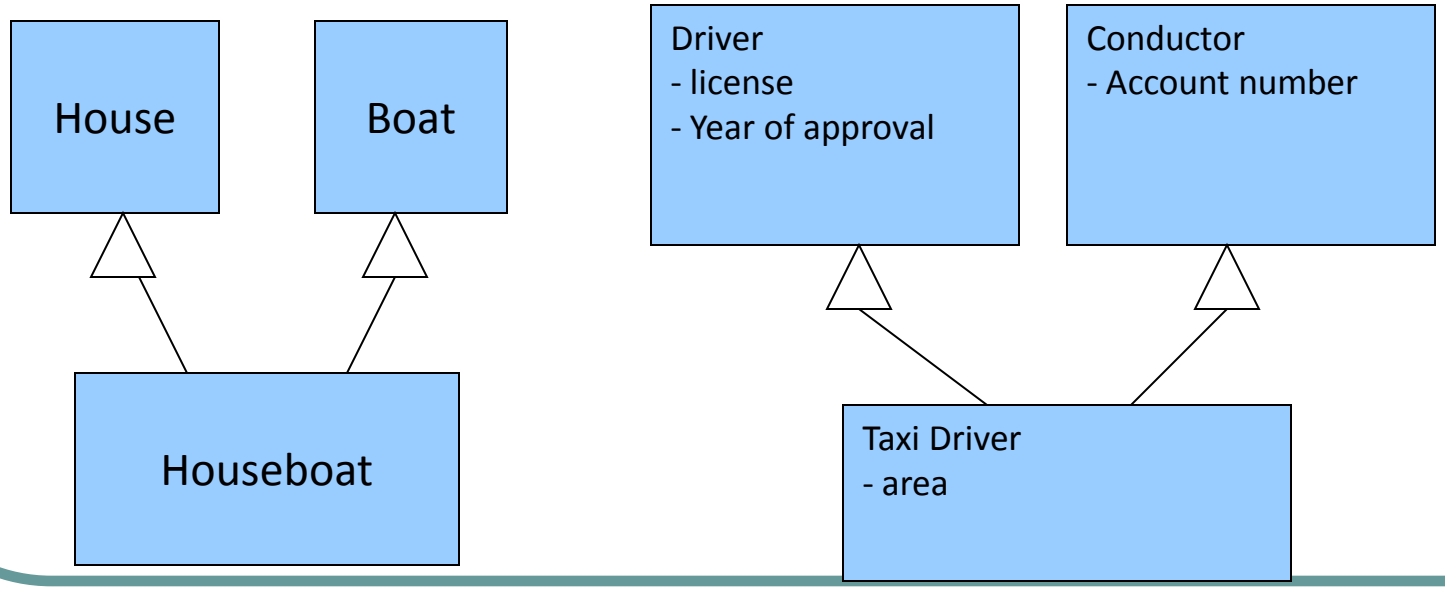
    Mammal <– Human <– Worker <- Programmer
- Could mammal be a derived class?
- If so, what would be the base class?

Picture about Inheritance



Multiple Inheritance

- In multiple inheritance a derived class has multiple base classes
- C++ supports multiple base classes



Inheritance Example: Syntax

```
class Circle : public Figure
{


}
```

Inheritance: Example

```
class Figure {
    public:
        int x, y;
};

class Circle : public Figure {
    public:
        int radius;
};

int main() {
    Circle a;
    a.x = 0;
    a.y = 0;
    a.radius = 10;
}
```

Multiple Inheritance: Example

– i.e., a class can be simultaneously derived from two or more base classes, e.g.,

```
class X { int a; };
class Y : public X { int b; };
class Z : public X { int c; };
class YZ : public Y, public Z { /* . . . */ };
```

– Derived classes **Y**, **Z**, & **YZ** inherit the data members & methods from their respective base classes

- Use scope resolution operator to resolve ambiguity

Inheritance (continued)

All member variables of the base `class` are also member variables of the derived `class`.

Similarly, the member functions of the base `class` are also member functions of the derived `class`.

Accessing Class Members through Inheritance

- private
  - Is accessible only via the base class
- public
  - Is accessible everywhere (base class, derived class, other classes)
- protected
  - Is accessible by the base class and derived classes

private Members of a Base Class

The `private` members of a base class are private to the base class; hence, the members of the derived `class` cannot directly access them.

In other words, when you write the definitions of the member functions of the derived `class`, you cannot directly access the `private` members of the base class.

Protected Members of a Class

- Private members of a class cannot be directly accessed outside the class
- For a base class to give derived class access to a `private` member
  - Declare that member as protected
- The accessibility of a `protected` member of a class is in between `public` and `private`
- A derived class can directly access the `protected` member of the base class

Private Inheritance

If `memberAccessSpecifier` is `private`, then

- `public` members of `A` are `private` members of `B` and can be accessed by member functions (and friend functions) of `B`
- `protected` members of `A` are `private` members of `B` and can be accessed by member functions (and friend functions) of `B`
- `private` members of `A` are hidden in `B` and can be accessed by member functions (and friend functions) of `B` through the `public` or `protected` members of `A`

Encapsulation: Incorrectly Accessing Protected-Example

```
class Figure {
    protected:
        int x, y;
};

class Circle : public Figure {
    public:
        int radius;
};

int main() {
    Circle a;
    a.x = 0; //direct access
    a.y = 0; //illegal
    a.radius = 10;
}
```

```
example.cpp: In function ‘int main()’:
example.cpp:5: error: ‘int Figure::x’ is protected
example.cpp:17: error: within this context
example.cpp:5: error: ‘int Figure::y’ is protected
example.cpp:18: error: within this context
```

Encapsulation: Correctly Accessing Protected-Example

```
class Figure
{
   protected:
      int x_, y_;
};

class Circle : public Figure
{
   private:
      int radius_;
   public:
      Circle(int x, int y, int radius);
};
```

```
Circle::Circle(int x, int y, int
    radius)
{
   x_ = x;
   y_ = y;
   radius_ = radius;
}

int main()
{
   Circle a(0,0,10);
}
```

Encapsulation: private Can't be Accessed Directly -Example

```
class Figure
{
   private:
      int x_, y_;
};

class Circle : public Figure
{
   private:
      int radius_;
   public:
      Circle(int x, int y, int radius);
};
```

```
Circle::Circle(int x, int y, int radius)
{
   x_ = x;
   y_ = y;
   radius_ = radius;
}

int main()
{
   Circle a(0,0,10);
}
```

```
example.cpp: In constructor 'Circle::Circle(int, int, int)':
example.cpp:5: error: 'int Figure::x_' is private
example.cpp:18: error: within this context
example.cpp:5: error: 'int Figure::y_' is private
example.cpp:19: error: within this context
```

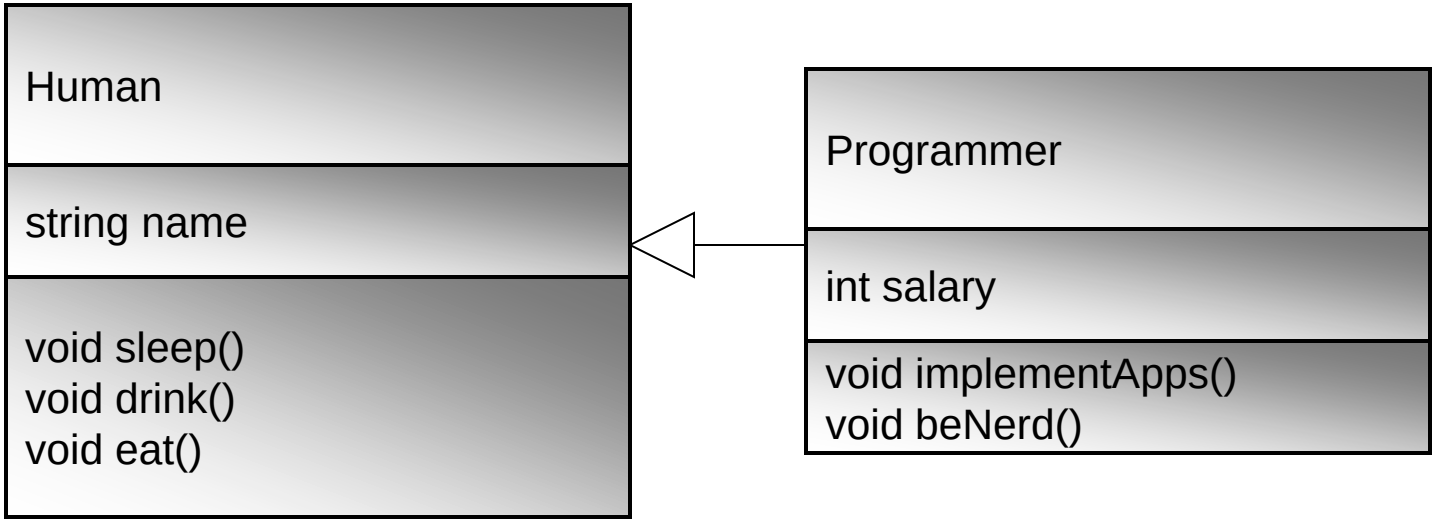
Encapsulation: private Can be Accessed Indirectly-Example

```
class Figure
{
    private:
        int x_, y_;
    public:
        void SetX(int x);
        void SetY(int y);
};
void Figure::SetX(int x) {
    x_ = x;
}
void Figure::SetY(int y) {
    y_ = y;
}
```

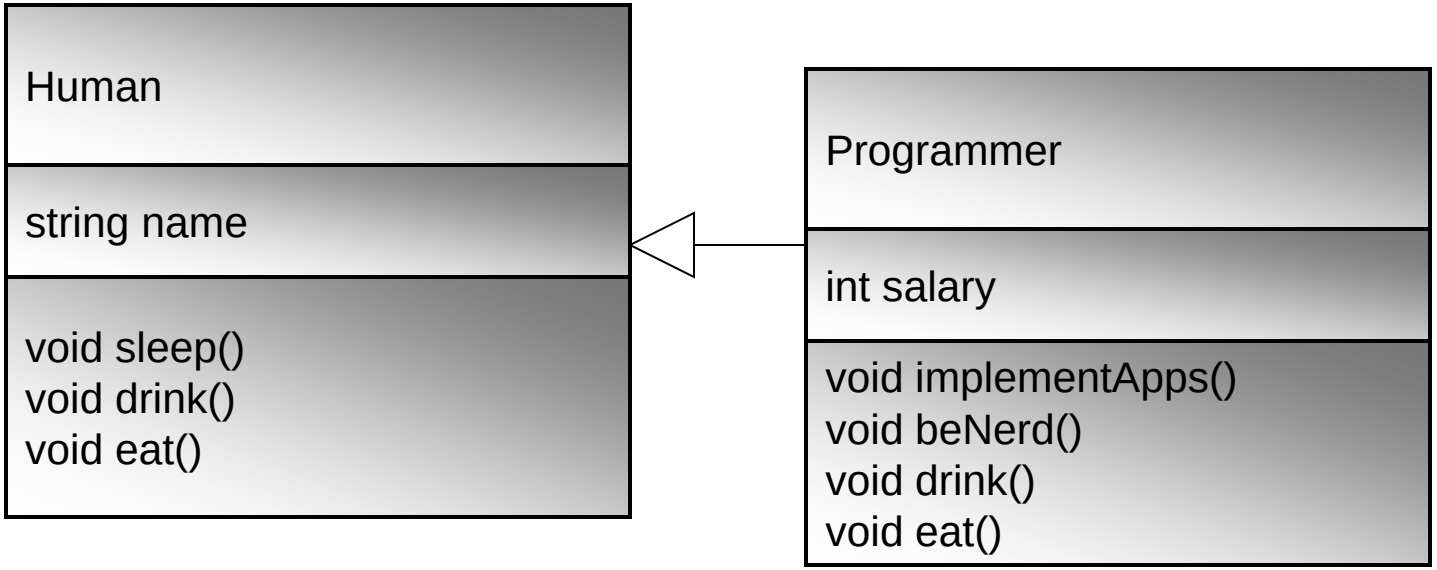
```
class Circle : public Figure
{
    private:
        int radius_;
    public:
        Circle(int x, int y, int radius);
};
Circle::Circle(int x, int y, int radius) {
    SetX(x);
    SetY(y);
    this->radius_ = radius;
}
int main() {
    Circle a(0,0,10);
}
```

Overriding Functions

- What are Programmer's attributes and methods?



Overriding Functions



- Since programmer eats and drinks differently than humans (only Coke and Pizza)
- The eat() and drink() methods are overridden in Programmer.

redefining vs. overloading

Overloading a function is a way to provide more than one function with the same name but with different signatures to distinguish them.

To redefine a function, the function must be defined in the derived class using the same signature and same return type as in its base class.

Redefining (Overriding) Member Functions of the Base Class

- To redefine a `public` member function of a base class
  - Corresponding function in the derived class must have the same name, number, and types of parameters
    - Same signature

Redefining (Overriding) Member Functions of the Base Class (continued)

- If derived class overrides a `public` member function of the base `class`, then to call the base class function, specify:
  - Name of the base class
  - Scope resolution operator (`::`)
  - Function name with the appropriate parameter list

Overriding Functions: Example

```
class rectangleType
{
public:
     void setDimension(double l, double w);
        //Function to set the length and width of the rectangle.
        //Postcondition: length = l; width = w;

     double getLength() const;
        //Function to return the length of the rectangle.
        //Postcondition: The value of length is returned.

     double getWidth() const;
        //Function to return the width of the rectangle.
        //Postcondition: The value of width is returned.

     double area() const;
        //Function to return the area of the rectangle.
        //Postcondition: The area of the rectangle is
        //               calculated and returned.

     double perimeter() const;
        //Function to return the perimeter of the rectangle.
        //Postcondition: The perimeter of the rectangle is
        //               calculated and returned.

      void print() const;
        //Function to output the length and width of
        //the rectangle.

      rectangleType();
        //Default constructor
        //Postcondition: length = 0; width = 0;

      rectangleType(double l, double w);
        //Constructor with parameters
        //Postcondition: length = l; width = w;

private:
      double length;
      double width;
};
```

```
void rectangleType::setDimension(double l, double w)
{
    if (l >= 0)
        length = l;
    else
        length = 0;

    if (w >= 0)
        width = w;
    else
        width = 0;
}

double rectangleType::getLength() const
{
    return length;
}

double rectangleType::getWidth()const
{
    return width;
}

double rectangleType::area() const
{
    return length * width;
}
```

```
double rectangleType::perimeter()  const
{
     return 2 *  (length + width);
}

void rectangleType::print() const
{
    cout << "Length = "  << length
         << "; Width = " << width;
}

rectangleType::rectangleType(double l, double w)
{
    setDimension(l, w);
}

rectangleType::rectangleType()
{
    length = 0;
    width = 0;
}
```

```
class boxType: public rectangleType
{
public:
    void setDimension(double l, double w, double h);
      //Function to set the length, width, and height
      //of the box.
      //Postcondition: length = l; width = w; height = h;

    double getHeight() const;
      //Function to return the height of the box.
      //Postcondition: The value of height is returned.

    double area() const;
      //Function to return the surface area of the box.
      //Postcondition: The surface area of the box is
      //                calculated and returned.

    double volume() const;
      //Function to return the volume of the box.
      //Postcondition: The volume of the box is
      //               calculated and returned.

    void print() const;
      //Function to output the length and width of a rectangle.

    boxType();
      //Default constructor
      //Postcondition: length = 0; width = 0; height = 0;

      boxType(double l, double w, double h);
        //Constructor with parameters
        //Postcondition: length = l; width = w; height = h;

  private:
      double height;
  };
```

The definition of the member function print of the class boxType is:

```
void boxType::print() const
{
    rectangleType::print();
    cout << "; Height = " << height;
}
```

The definition of the function setDimension is:

```
void boxType::setDimension(double l, double w, double h)
{
    rectangleType::setDimension(l, w);

    if (h >= 0)
        height = h;
    else
        height = 0;
}

double boxType::area() const
{
    return  2 *  (getLength() * getWidth()
                + getLength() * height
                + getWidth() * height);
}

double boxType::getHeight() const
{
    return height;
}

double boxType::volume() const
{
    return rectangleType::area() * height;
}
```

boxType

From the definition of the `class boxType`, it is clear that the `class boxType` is derived from the `class rectangleType`, and it is a `public` inheritance.

Therefore, all `public` members of the `class rectangleType` are `public` members of the `class boxType`.

The `class boxType` also overrides (redefines) the functions `print` and `area`.

Extending the Scope Resolution Operator

In general, while writing the definitions of the member functions of a derived `class` to specify a call to a `public` member function of the base `class`, we do the following:

- If the derived `class` overrides a `public` member function of the base `class`, then to specify a call to that `public` member function of the base `class` you use the name of the base `class`, followed by the scope resolution operator, `::`, followed by the function name with the appropriate parameter list.
- If the derived `class` does not override a `public` member function of the base `class`, you may specify a call to that `public` member function by using the name of the function and the appropriate parameter list.

Calling Base Class Constructors

- A constructor is used to construct an instance of a class.
- Unlike data fields and functions, the constructors of a base class are not inherited in the derived class.
- They can only be invoked from the constructors of the derived classes to initialize the data fields in the base class.
- The syntax to invoke it is as follows:

```
DerivedClass(parameterList): BaseClass() {
  // Perform initialization
}
```

or

```
DerivedClass(parameterList): BaseClass(argumentList) {
  // Perform initialization
}
```

No-Args Constructor in Base Class

A constructor in a derived class must always invoke a constructor in its base class.

If a base constructor is not invoked explicitly, the base class's no-arg constructor is invoked by default.

For example,

```
public Circle()
{
  radius = 1;
}
```

is equivalent to

```
public Circle(): GeometricObject()
{
  radius = 1;
}
```

```
public Circle(double radius)
{
  this->radius = radius;
}
```

is equivalent to

```
public Circle(double radius)
  : GeometricObject()
{
  this->radius = radius;
}
```

Constructor and Destructor Chaining

Constructing an instance of a class invokes the constructors of all the base classes along the inheritance chain.

A base class's constructor is called before the derived class's constructor.

Conversely, the destructors are automatically invoked in reverse order, with the derived class's destructor invoked first.

This is called *constructor and destructor chaining*.

Pointers to a Base Class

- A pointer to a base class can store the address of a derived object

```
Base * b;
Derived d;
b = &d;
```

- Compiler assumes the stored object is of the base class
- To call a function you've overridden in a derived class need to use **virtual** functions

Virtual Functions

To enable dynamic binding for a function the function must be declared virtual in the base class

If a function is defined virtual in a base class, it is automatically virtual in all its derived classes.

It is not necessary to add the keyword virtual in the function declaration in the derived class.

Virtual Functions: Syntax Example

```
class Base {
    public:
    virtual void f();
}

class Derived {
    public:
    void f() {cout<<“This is f”; };
}
```

static matching vs. dynamic binding

Matching a function signature and binding a function implementation are two separate issues.

The *declared type* of the variable decides which function to match at compile time.

The compiler finds a matching function according to parameter type, number of parameters, and order of the parameters at compile time.

A virtual function may be implemented in several derived classes.

C++ dynamically binds the implementation of the function at runtime, decided by the *actual class* of the object referenced by the variable.

Use Virtual functions?

If a function defined in a base class needs to be redefined in its derived classes, you should declare it virtual to avoid confusions and mistakes.

On the other hand, if a function will not be redefined, it is more efficient without declaring it virtual, because it takes more time and system resource to bind virtual functions dynamically at runtime.

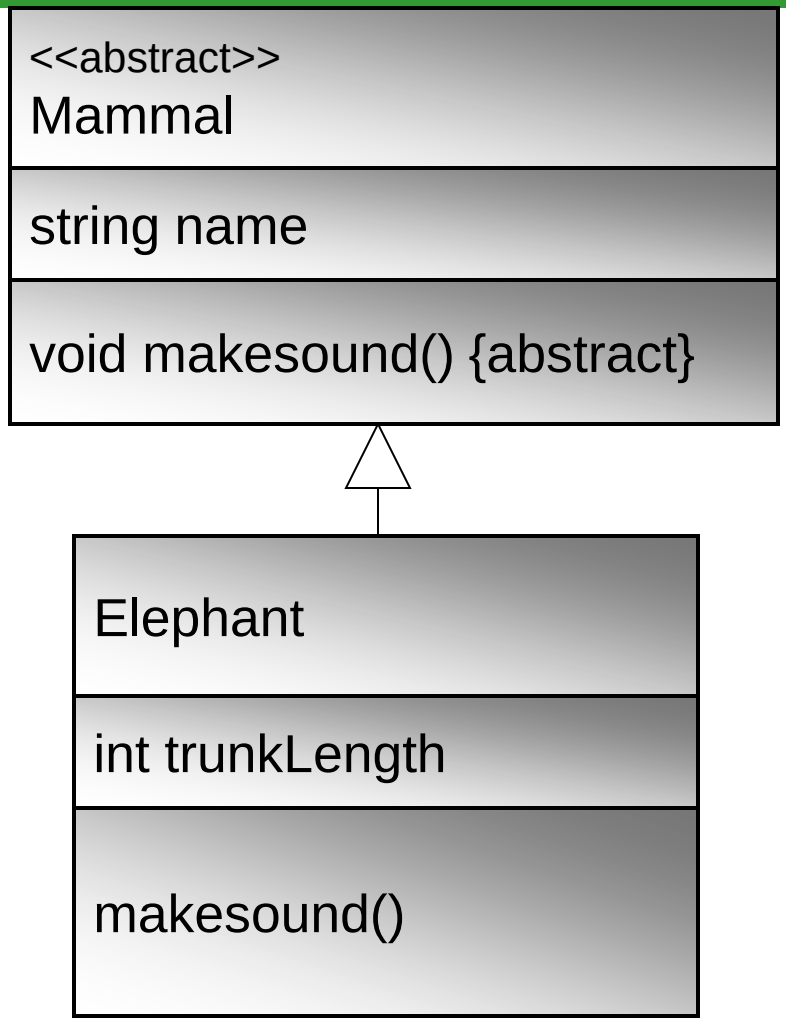
Abstract Classes

In the inheritance hierarchy, classes become more specific and concrete *with each new derived class*.
If you move from a derived class back up to its parent and ancestor classes, the classes become more general and less specific.
Class design should ensure that a base class contains common features of its derived classes.
Sometimes a base class is so abstract that it cannot have any specific instances.
Such a class is referred to as an *abstract class*.

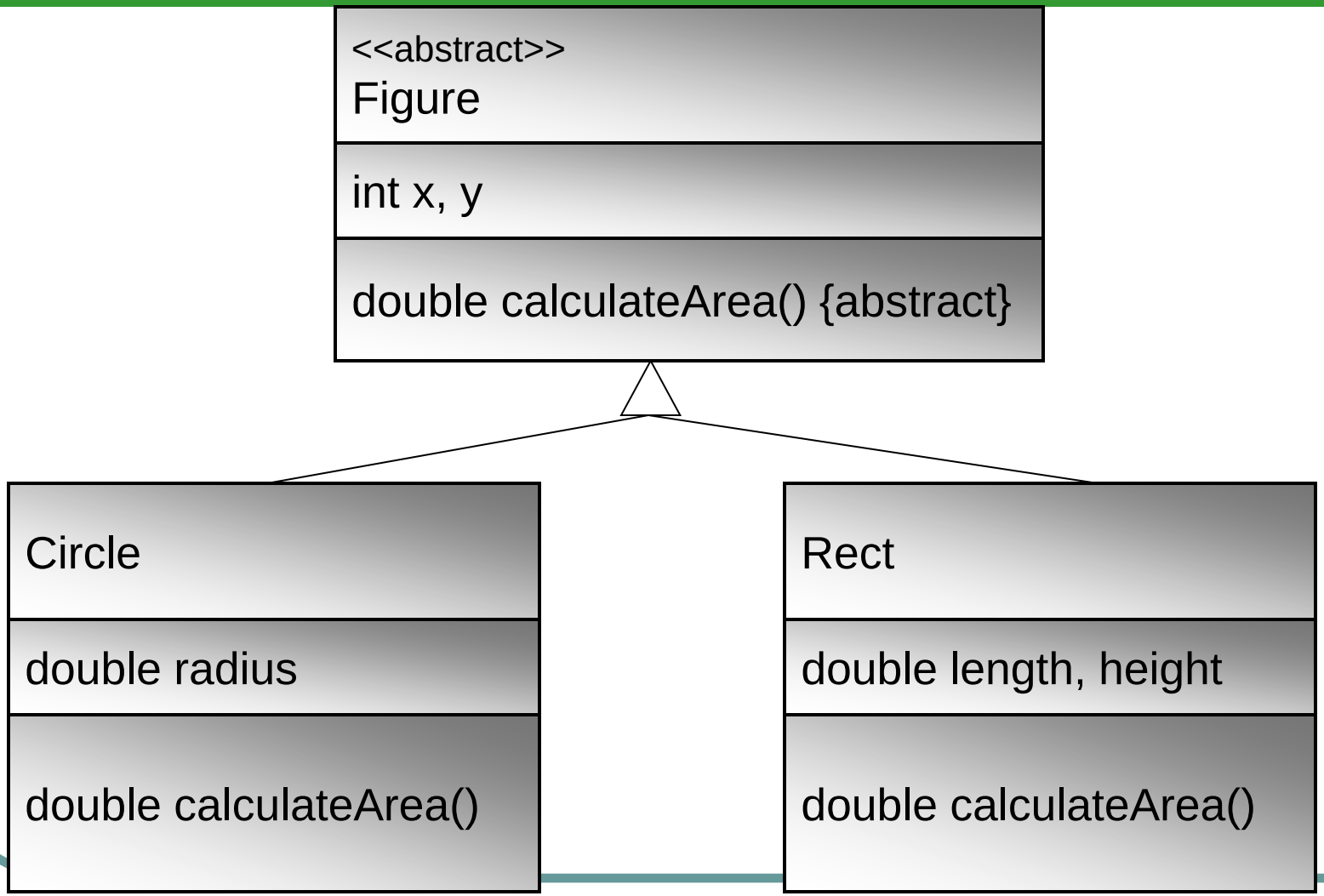
Abstract Class

- Abstract class is a class which you cannot instantiate (create objects)
- Abstract class in C++ has abstract methods (pure virtual functions), that do not have implementations
- These methods force derived classes to implement those methods
- You can inherit from an abstract class and create objects from the derived class, if it is a concrete one
- Abstract method is created by reserved word "virtual“ and setting the function equal to zero as in: =0

Example: Mammal



Example: Figure



Example of an Abstract class

```
class Figure
{
    private:
        int x_, y_;
    public:
        Figure(int x, int y) : x_(x), y_(y) {
            cout << "Figure Constructor\n";
        }
        ~Figure() {
            cout << "Figure Destructor\n";
        }

        virtual double calculateArea() = 0;
};
```

Example of a class Derived From an Abstract class

```
class Circle : public Figure
{
    private:
        double radius_;
    public:
        Circle(int x, int y, int radius) : Figure(x, y), radius_(radius) {
            cout << "Circle Constructor\n";
        }
        ~Circle() {
            cout << "Circle Destructor\n";
        }
        double calculateArea() {
            return 3.14 * radius_ * radius_;
        }
};
```

Example of Abstract class: main()

```
int main()
{
    Circle a(0,0,5);
    cout << a.calculateArea() << endl;

     // Figure f(0,0); This does not work, since figure is abstract:
}
```

The End!

Based on slides by Liang and Malik and Pohjolainen