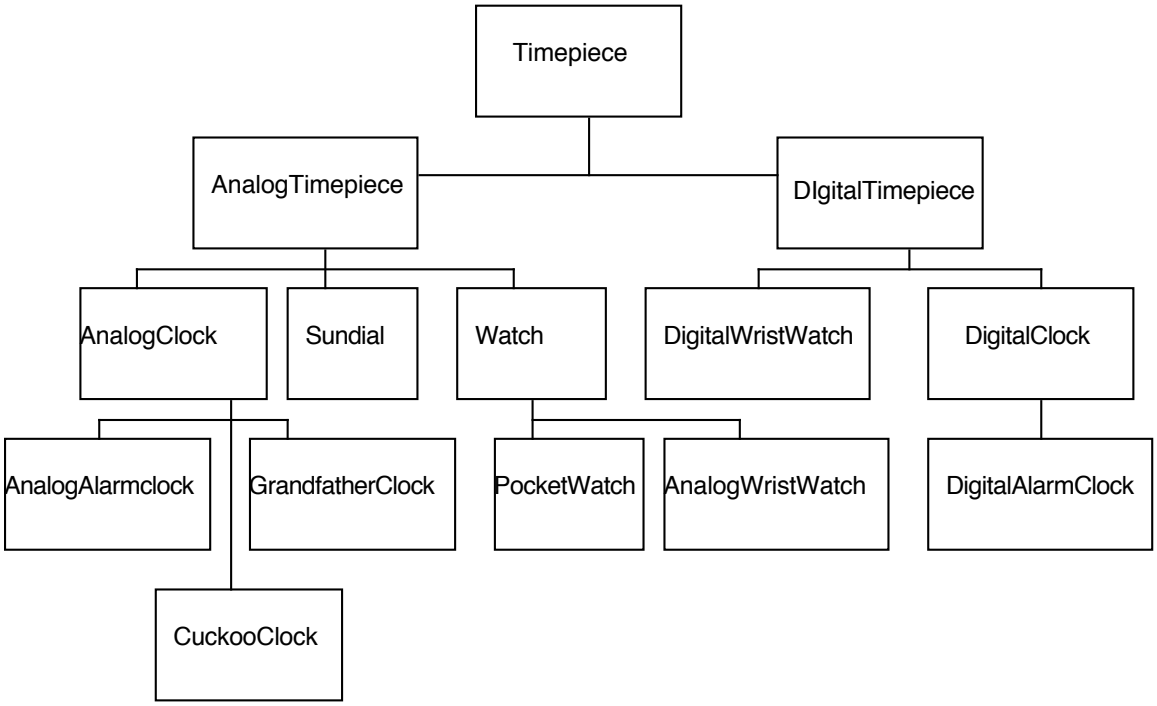
Chapter 8-1, Lecture notes

Advanced C++ Topics

Inheritance with objects help enforce the "wall" of ADT. You may think of inheritance as a rich uncle or aunt giving you a million dollars. But in the object-oriented world, inheritance would be a class being able to derive properties from a previously defined class. Like inheriting from your parents, some traits are the same, some different.

Figure 8-1 on page 388 shows relationships among various timepieces.



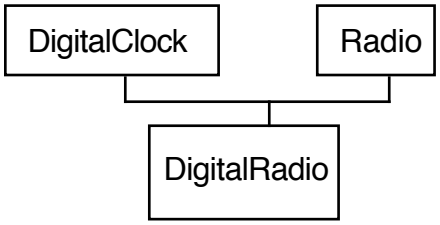
All digital clocks have the same operations such as:
Set the time
Advance the time
Display the time

The digital alarm clock is a digital clock with alarm methods like so:
Set the alarm
Enable the alarm
Sound the alarm
Silence the alarm

If you think of a group of digital clocks and a group of alarm clocks as classes then the digital alarm clock is a derived class of the digital clock. In C++, a derived class has the

data members and methods of the base class along with the members it defines. For instance, a cuckoo clock is derived from analog clock but would add a cuckoo.

A derived class can have more than one base class. For instance, digital radio would inherit the digital clock methods and then the radio methods like in figure 8-2 on page 389. That is called “multiple inheritance”.

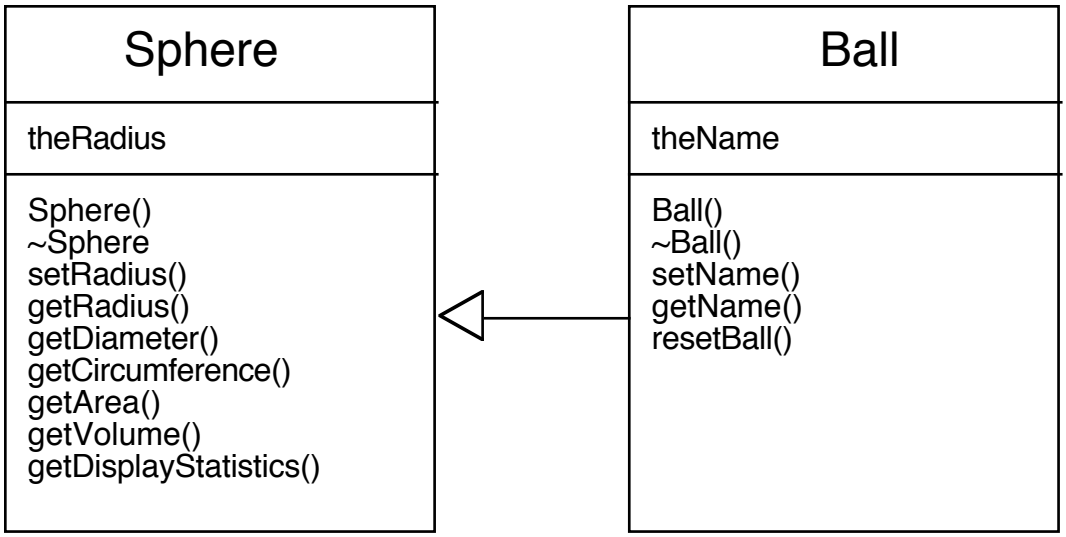


This allows reuse of software components. Let’s use a simpler example of inheritance using the sphere class being inherited by the ball class. The sphere class is:

```
class Sphere
{
public:
// constructors
        Sphere();
        Sphere(double initialRadius);
        // copy constructor and destructor supplied by the compiler
// Sphere operations:
        void setRadius(double newRadius);
        double getRadius() const;
        double getDiameter() const;
        double getCircumference() const;
        double getArea() const;
        double getVolume() const;
        void displayStatistics() const;

private:
        double theRadius;  // the sphere’s radius
}; // end Sphere
```

A ball would have all of the sphere methods but would be add ball methods. The UML would look like:



The ball class is:

```
class Ball : public Sphere
{
public:
// constructor
        Ball();

        Ball(double initialRadius, const string& initalName);
        // Creates a ball with radius initialRadius and name initialName.

        // copy constructor and destructor supplied by the compiler

// additional or revised operations:
        void getName(string& currentName) const;
        // Determines the name of a ball

        void setName(const string& newName);
        // Sets (alters) the name of an existing ball

        void resetBall(double newRadius, const string& newName);
        // Sets (alters) the radius and name of an existing ball to newRadius,
        //    and newName, respectively

        void displayStatistics() const;
        // Displays the statistics of a ball

private:
        string theName;  // the ball's name
};  // end Ball
```

The Ball class has two data members: theRadius is inherited and theName which is new. Ball has all the Sphere methods and the new Ball methods of getName, setName, resetBall, and displayStatistics. A derived class cannot access the private data members of a base class directly. Inheritance does not imply access. You get a locked vault to inherit but can't open it. However, Ball can use the Sphere access methods to get at theRadius.

You implement the Ball methods like so:

```
Ball::Ball() : Sphere()
{
        setName("");
} // end default constructor

Ball::Ball(double initialRadius, const string& initalName) : Sphere(initialRadius)
{
        setName(initialName);
} // end constructor

void Ball::getName(string& currentName) const
{
        currentName = theName;
} // endgetName

void Ball::setName(const string& newName)
{
        theName = newName;
}

void Ball::resetBall(double newRadius, const string& newName)
{
        setRadius(newRadius);
        setName(newName);
} // end resetBall

void Ball::displayStatistics() const
{
        cout << "Statistics for a " << theName << " : ";
        Sphere::displayStatistics();
} // end displayStatistics
```

Note that the Ball constructors also call the Sphere constructors then call setName to set the Ball theName. The resetBall calls the Sphere's setRadius and the Ball's setName. The displayStatistics function of Ball has the same name as the displayStatistics function of Sphere. To have the Ball displayStatistics call the Sphere displayStatistics you need to specify the Sphere class name such as "Sphere::displayStatistics();".

When the compiler can determine which class to call at compile time (like the Sphere::displayStatistics()"), it is called "early binding" or "static binding". The constructors for a derived class, as well as the derived class destructor, execute before the base class. So Ball's constructor would execute before Sphere's constructor.

We know about public and private. There is a third category: protected. Protected section would hide members from the class's client but not from the derived class. So with the Sphere class, if you made theRadius into a protected member and not private, then the Ball class could access it. Typically, however, data members are private and the derived class must use accessor and mutator methods to get at the data members.

You can even have public, private, and protected inheritance. The syntax is:

class DerivedClass : *kind* BaseClass

The rules for kinds of inheritance would be:

1. Public inheritance: Public and protected members of the base class remain public and protected members of the derived class
2. Protected inheritance: Public and protected members of the base class are protected members of the derived class
3. Private inheritance: Public and protected members of the base class are private members of the derived class.

In all cases, private members of a base class remain private to the base class and cannot be accessed by a derived class. The public inheritance is the most used. Private is used to implement one class in terms of another class. Protected inheritance is not often used.

Inheritance deals with ancestor/descendant relationships among the classes. There are other relationships: is-a, has-a, and as-a.

The is-a relationship: the ball is-a sphere – it was derived from the sphere class. Wherever you use an object of Sphere, you can also use an object of Ball. That is "object type compatibility".

For example:

```
void displayDiameter(Sphere thing)
{
        cout << "The diameter is " << thing.getDiameter() << ".\n";
} // end displayDiameter
```

Define mySphere and myBall as:

```
Sphere      mySphere(2.0);
Ball        myBall(5.0, "Volleyball");
```

You can do the following calls to displayDiameter:

```
displayDiameter(mySphere);  // mySphere's diameter
displayDiameter(myBall);    // myBall's diameter
```

The second call has a Ball object being passed to a Sphere parameter – works.

The has-a relationship would have a class be part of another class. For instance, a ball-point pen "has a" ball as its point. You would not use public inheritance in this case but use it as a data member, like so:

```
class Pen
{
        …
private:
        Ball point;
}; // end Pen
```

So the ball point pen has-a ball. The has-a relationship is "containment".

The as-a relationship would define one class in terms of another. For instance, you can implement the ADT stack "as a" list by using private inheritance. For instance:

```
class Stack : private List
```

All the methods and members of List would be private in the Stack class and the Stack class would be able to use the List methods to implement the stack. The underlying List operations would be hidden from the client.

Chapter 8-2, Virtual Methods and Late Binding

Early binding is where the compiler would determine which method to call – base class or derived class. However, early binding can lead to problems. For instance:

```
Sphere *spherePtr = &mySphere;
```

The spherePtr would point to mySphere and the following:

```
spherePtr->displayStatistics();
```

would point to the displayStatistics in the Sphere class. However, if you do the following:

```
spherePtr = &myBall;
```

then the statement of:

spherePtr->displayStatistics();

would still invoke the Sphere's version of displayStatistics() instead of the Ball's version. The compiler had did the early binding and it remains with the Sphere and not the Ball. You can change the function to become a "virtual method" to allow "late binding" (also called "dynamic binding" or binding at execution time) by just putting "virtual" in front of the definition, like so:

```
class Sphere
{
public:
        // everything as before, except displayStatistics
        …
        virtual void displayStatistics() const;
        …
}; // end Sphere
```

Now the "spherePtr->displayStatistics()" will point to the Ball displayStatistics and not the one in Sphere. By the way, any overriding method like with displayStatistics in Ball would be virtual even if the "virtual" keyword was not included (but put the "virtual" in to give any programmer a clue what is happening).

Every class that defines a virtual method would have a "virtual method table" or VMT. The VMT would be invisible to the programmer but would have pointers as to the correct instructions. It allows the late binding. By the way, constructors can NOT be virtual while destructors can be. You cannot override a virtual method's return type.

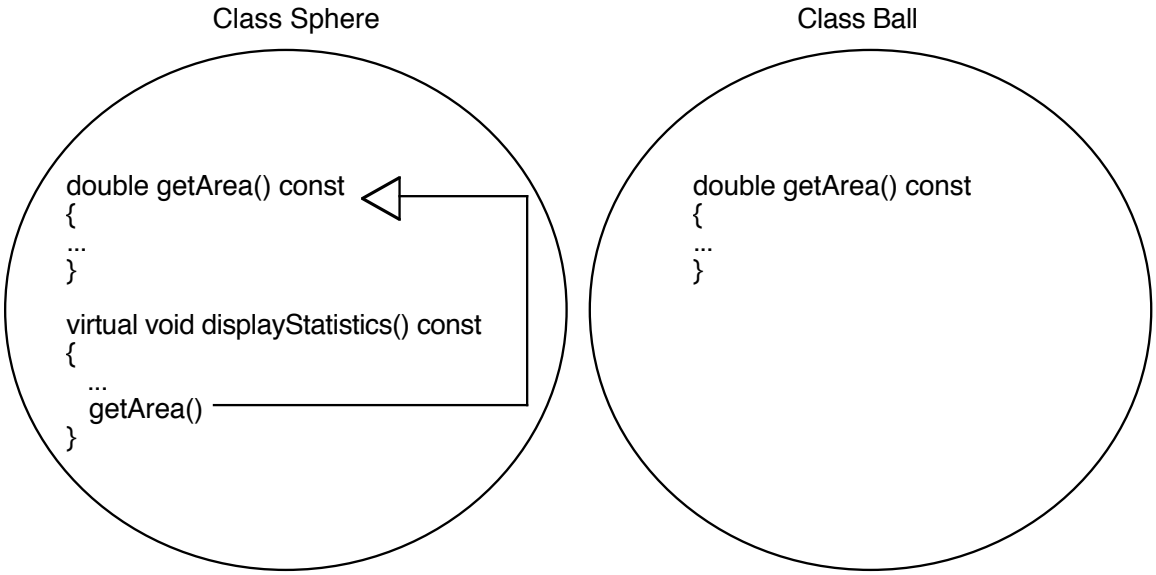
Now you can run into a subtle problem with late binding. For instance what if you decided NOT to have displayStatistics in the Ball derived class and just use the Sphere version. However, Ball overrode the getArea() function and created a function like so:

```
double getArea() const;
```

But even if a Ball object is allocated, like with:

```
spherePtr = &myBall;
```

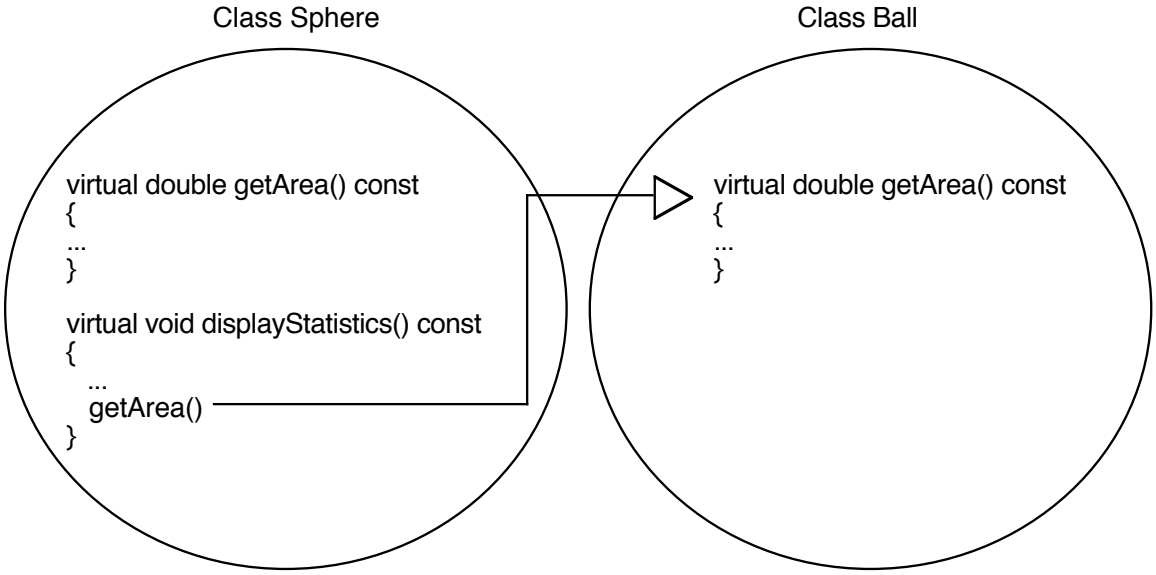
The compiler will do early binding and in the displayStatistics with the Sphere will call the Sphere getArea and NOT the Ball getArea, like shown in Figure 8-8 on page 401:



However, if you make the getArea in Sphere virtual like so:

virtual double getArea() const;

Then it will call the correct getArea function like in Figure 8-9 on page 403 like so:



Chapter 8-3, Friends

A C++ class can allow additional access to its private and protected members by declaring other functions and classes as “friends”. For example:

```
class Sphere
{
public:
// constructors and operations as before
        …
        friend void readSphereData(Sphere& s);
        friend void writeSphereData(Sphere& s);
private:
        double theRadius;       // the sphere's radius
}; // end Sphere
```

The functions of readSphereData and writeSphereData are not part of the Sphere class but defined outside of it. For example:

```
void readSphereData(Sphere& s)
{
        cout << "Enter sphere radius: ";
        cin >> s.theRadius;
} /// end readSphereData

void writeSphereData(Sphere& s)
{
        cout    << "The radius of the sphere is ";
                << s.theRadius << endl;
} // end writeSphereData
```

Now this violates the principle of information hiding but in a controlled fashion. The class Sphere must explicitly grant this access by declaring the function as a friend. Now the readSphereData and writeSphereData could be declared as members of Sphere. However, making them friends instead can give you more flexibility. For instance with the Ball class, you can call the functions via the sphere parameter without having to redo the Ball class.