Today …

• Homework 1 topics
  – Inheritance
  – Various uses of const
  – Basic operator overloading
    • operator==
    • operator<
    • etc.
  – The “this” keyword
  – Assignment operator and copy constructor

• Readings
  – Ch. 8: 8.1–8.4, 8.6

We won’t be able to cover everything today
So, the due date for HW 1 is postponed to next Thursday (9/9/10)
What do we mean by “Inheritance”?

**Inheritance** is a relationship where one class *derives* the behavior and/or structure of another class

- A major feature of *object-oriented programming languages*
- Useful for organizing and modularizing code
- Can enable reuse of existing code
  - New classes subclass (inherit) from existing classes

Basic Terminology

- A **child class** *directly* inherits from a **parent class**
- A **derived class** inherits from a **base class**
- Derived classes **inherit** data and functions (**members**)
  - single inheritance (a class directly inherits from one class)
  - multiple inheritance (we won’t talk about this … its tricky)
- Inheritance forms a **class hierarchy**
  - with single inheritance …
  - the hierarchy induces a tree structure
  - there is always one **root class**
  - a **leaf class** has no child classes
An Example: C++ Single Inheritance

Inheritance is specified \textit{in the derived class}  
\begin{itemize}
  \item Child classes “inherit from” parent classes
\end{itemize}

\begin{itemize}
  \item The \textit{public} keyword says:
  \begin{itemize}
    \item public members in the parent class are public in the child
  \end{itemize}
\end{itemize}

\begin{itemize}
  \item Inheritance is often called an “is-a” relationship
  \begin{itemize}
    \item Here: Savings is-a Account
  \end{itemize}
\end{itemize}

```c++
class Savings : public Account
{
    public:
    ...
};
```
An Example: C++ Single Inheritance

// base class
class Account {
    public:
        Account();
        double getBalance();
        private:
            double balance;
};

// derived class
class Checking : public Account {
    public:
        Checking();
        double getMonthlyFees();
        private:
            double monthlyFees;
};

class Savings : public Account {
    public:
        Savings();
        double getMonthlyInterest();
        private:
            double monthlyInterest;
};

Subclass objects contain all members of their superclasses … and can be used wherever the superclass objects can
Single Inheritance

Not all inherited members may be accessible

– Members functions and variables have their own visibility

Members are either public, protected, or private:

– A public member is always visible and accessible
– A protected member is only visible in inherited classes
– A private member is only visible in their defining class

In the case of private variables and functions …

– Memory is still allocated in derived classes
  • E.g., balance is allocated for each SavingsAccount object
– Can only be accessed from the base class
  • Enables information hiding and modularity
  • A base class can change without affecting derived classes

Savings Account Object

Account::getBalance()
balance = 1000.00

Savings::getMonthlyInterest()
monthlyInterest = 1.01
Examples ...

class Sphere1
{
    public:
        int radius;
}

int main()
{
    Sphere1 s;
    s.radius = 10;
}

Examples ...

class Sphere1
{
    public:
        int radius;
}

int main()
{
    Sphere1 s;
    s.radius = 10;
}
Examples ...

class Sphere1
{
    public:
        int radius;
}

int main()
{
    Sphere1 s;
    s.radius = 10;
}

Is this statement legal???
YES!
The radius member is declared public
This means clients can access it

Examples ...

class Sphere1
{
    public:
        int radius;
}

class Ball1 : public Sphere1
{
    public:
        int getSize()
        {
            if(radius < 4)
                return 3;
            ...
        }
}

A child class of Sphere1
Uses public inheritance
(also a “client“)
Examples ...

class Sphere1
{
    public:
        int radius;
}

class Ball1 : public Sphere1
{
    public:
        int getSize()
        {
            if(radius < 4)
                return 3;
            ...
        }
}

Is this statement legal???

Do you see any design issues here?

Examples ...

class Sphere1
{
    public:
        int radius;
}

class Ball1 : public Sphere1
{
    public:
        int getSize()
        {
            if(radius < 4)
                return 3;
            ...
        }
}

Is this statement legal???

YES!
radius is declared public
Here we inherit public access to it
Examples ...

class Sphere2
{
    private:
        int radius;
}

int main()
{
    Sphere2 s;
    s.radius = 10;
}

Examples ...

class Sphere2
{
    private:
        int radius;
}

int main()
{
    Sphere2 s;
    s.radius = 10;
}
Examples ...

class Sphere2
{
    private:
    int radius;
}

int main()
{
    Sphere2 s;
    s.radius = 10;
}

Is this statement legal???
NO!
The radius member is declared private
This means clients cannot access it

A child class of Sphere2

class Ball2 : public Sphere2
{
    public:
    int getSize()
    {
        if(radius < 4)
            return 3;
        ...
    }
}
Examples …

class Sphere2
{
    private:
        int radius;
}

class Ball2 : public Sphere2
{
    public:
        int getSize()
        {
            if(radius < 4)
                return 3;
            ...
        }
}

Is this statement legal???

Examples …

class Sphere2
{
    private:
        int radius;
}

class Ball2 : public Sphere2
{
    public:
        int getSize()
        {
            if(radius < 4)
                return 3;
            ...
        }
}

What should we do?

Is this statement legal???

NO!
radius is declared private
It is inherited, but not visible
Examples …

```cpp
class Sphere3 {
    public:
        int getRadius();
        void setRadius(int r);
    private:
        int radius;
};

class Ball1 : public Sphere3 {
    public:
        int getSize() {
            if(getRadius() < 4)
                return 3;
            
        }
    }
}

int main() {
    Sphere3 s;
    s.setRadius(10);
}
```

**Opt. 1: Create accessors and mutators (also called “getters” and “setters”)**

Is this legal?

```cpp
class Sphere3 {
    protected:
        int radius;
};

class Ball1 : public Sphere3 {
    public:
        int getSize() {
            if(radius < 4)
                return 3;
            
        }
    }
}

int main() {
    Sphere3 s;
    s.radius = 10;
}
```

**Opt. 2: Use protected visibility**
Examples ...

class Sphere3
{
    protected:
        int radius;
}

opt. 2: Use protected visibility

Is this legal?

class Ball1 : public Sphere1
{
    public:
        int getSize()
        {
            if(radius < 4)
                return 3;
            ...
        }
}

int main()
{
    Sphere3 s;
    s.radius = 10;
}

NO!
radius is declared protected only visible in subclasses

Single Inheritance

Using getters and setters for private data

– can be tricky to hide internal data sufficiently

– … and we must not expose internal structures via public methods (see: “good” software design)

– Overall though the preferred approach

Protected members allow derived classes to access base class members, while still restricting client access

– this could mean making data members protected (simpler)

– or providing only protected member functions for getting and setting private data (sometimes safer, but more work)
Constructors and Inheritance

The constructor of a base class

• is *always invoked before* a derived class constructor
  – we can assume that in the constructor of the derived class
    the base class members have been initialized
  – the constructor of the derived class is called last, after all
    inherited classes in the hierarchy (in order)

By default

– the base class constructor is *implicitly* invoked by the
  derived class constructor
– we can also explicitly call a base class constructor

Constructors and Inheritance

```cpp
// base class
class Account {
    public:
        Account();
        float getBalance();
    private:
        float balance;
};

Account::Account() : balance(0) {
    cout << "account constructor\n";
}

// derived class
class Checking : public Account {
    public:
        Checking();
        float getMonthlyFees();
    private:
        float monthlyFees;
};

Checking::Checking() : monthlyFees(5) {
    cout << "checking constructor\n";
}
```

*After the client says:*

Account a;

*What is printed?*

account constructor

*After the client says:*

Checking c;

*What is printed?*

account constructor checking constructor
Constructors and Inheritance

We can explicitly initialize the base class in initialization lists

```
Checking::Checking(int bal) : Account(bal) {...}
```

- The arguments to the base constructor can consist of
  - values given as arguments to the derived class constructor
  - constants
  - global variables (is this a good idea?)
  - or expressions made up of these values

- We cannot use
  - data members of the derived class
  - return values of member functions
  - the derived class has not been initialized yet

```
// base class
class Account {
    public:
        Account(float b = 0);
        float getBalance();
    private:
        float balance;
};

Account::Account(float b) : balance(b) {
    cout << "account constructor\n";
}

// derived class
class Checking : public Account {
    public:
        Checking(float b=0, float f=5);
        float getMonthlyFees();
    private:
        float monthlyFees;
};
```

Checking::Checking(float b, float f) : Account(b), monthlyFees(f) {
    cout << "checking constructor\n";
}

What are the values of data members for:

- Account a1;
  - balance = 0
- Account a2(5000);
  - balance = 5000
- Account a3(500, 0);
  - compile error!
- Checking c1;
  - balance = 0, monthlyFees = 5
- Checking c2(500);
  - balance = 500, monthlyFees = 5
- Checking c3(500, 0);
  - balance = 500, monthlyFees = 0
Constructors and Inheritance

*If a base class constructor is not in the initialization list*

- The *default constructor* of the base class is called
- If there is no default constructor for the base class, then we get a *compile error*

*Recall:* a default constructor has no arguments … or else every argument has a default value

Subtype Polymorphism

- Subclass objects can be used when superclass objects are expected …
  - This is referred to as “*subtype polymorphism*”
  - This doesn’t work in the other direction!
- This is an example of code reuse …
  - Subclassing is one form of reuse
  - Polymorphism is another
Subtype Polymorphism

- A simple example:

```cpp
class Bank {
    public:
        void addAccount(Account a);
        ...
    }
    ...
Account a;
Bank b;
b.addAccount(a);  // Is this OK?
Savings s;
b.addAccount(s);  // Yes!
```

- A simple example:

```cpp
class Bank {
    public:
        void addSavings(Savings s);
        ...
    }
    ...
Savings s;
Bank b;
b.addSavings(s);  // Is this OK?
Account a;
b.addSavings(a);  // No!
```

s has an Account “part”
a doesn’t have a Savings “part”
Subtype Polymorphism

• The same holds true for assignment

Account a;
Savings s;
a = s; ← This works: We’re “casting” s to an Account
s = a; ← This doesn’t work: Why not?

What about these statements?

Using the const keyword ...
The various uses of `const`

A simple class for managing stock shares ...

```cpp
class Stock {
public:
    Stock();
    Stock(string aCompany, int theShares);
    int getShares();
    string getCompany();
    void setShares(int newShares);
    Stock worthMore(Stock s);
private:
    string company;
    int shares;
};
```

The compiler is our friend!
It is saving us from introducing bugs later

The various uses of `const`

Let's say we want to do the following

```cpp
const Stock s1("GOOG", 1000);
s1.setShares(900); // sell shares
```

- What will happen when we compile?

  Compile Error!

  "error: passing 'const Stock' as 'this'
  argument of 'void Stock::setShares(int)'
  discards qualifiers"

  - Why is the compiler complaining?

  ... Because `s1` is a constant, and we are modifying it!
The various uses of `const`

Let's say now we want to do the following

```cpp
cost Stock s1("GOOG", 1000);
cout << s1.getShares() << endl; // check shares
```

- What will happen when we compile?

  **Compile Error!**

  "error: passing 'const Stock' as 'this'
  argument of 'void Stock::getShares()'
  discards qualifiers"

  – Why is the compiler complaining now?

  … The compiler can’t tell we aren’t modifying s1

The various uses of `const`

We need to tell the compiler `getShares()` won’t change the invoking object

```cpp
int getShares() const;
```

- Now when we compile

  ```cpp
cost Stock s1("GOOG", 1000);
cout << s1.getShares() << endl; // check shares
```

  – The compiler is happy

- All “getters” should be `const` functions