Today

- Wrap up exercise
- Inheritance basics

Assignments

- HW1 (out, due Tues) ... see piazza for submission instructions, etc.

Quick note on HW1 and `assert(...)` statements

- Your program should not have any output (running with no output is success!)

- The `hw1.cpp` file performs a sequence of small tests (via `assert`)

  ```
  ...
  Security s1("GOOG");
  s1.set_share_value(1245);
  s1.set_holdings(120);  
  assert(s1.get_share_value() == 1245);
  ...
  ```

- An `assert(...)` statement:
  - takes a boolean expression
  - if the expression is true, moves to next statement
  - if the expression is false, halts your program:

  ```
  $ ./hw1
  ...: Assertion `should_sell(s3) == false' failed.
  Aborted (core dumped)
  ```
Anatomy of a Class (Review)

⇒ Exercise Question 4

A simple `BankAccount` class definition (interface):

```cpp
class BankAccount
{
    public:
        BankAccount(double min_bal);
        void deposit(double amount);
        double get_minimum_balance() const;
        double get_available_balance() const;
    private:
        double min_balance;
        double cur_balance;
};
```

A class has member functions (“behavior”) and member variables (“state”)

- members can be more or less hidden from clients

Public access

- member is fully visible to clients (e.g., `deposit`)
- member is fully visible to other members

Private access

- member is hidden from clients (e.g., `cur_balance`)
- but is fully visible to the member functions of the class
Anatomy of a Class (cont)

A simple class implementation:

```cpp
BankAccount::BankAccount(double min_bal)
    : min_balance(min_bal), curr_balance(min_bal)
{}

void BankAccount::deposit(double amount)
{
    if (amount > 0)
        cur_balance += amount;
}

double BankAccount::get_minimum_balance() const
{
    return min_balance;
}

double BankAccount::get_available_balance() const
{
    return cur_balance - min_balance;
}
```

**const variables vs const member functions**
- `const` can ensure variables are “constant” (immutable)
- `const` can also ensure functions don’t change member variable values (state)

Q: What is wrong with the following?

```cpp
BankAccount b1;
while (b1.cur_balance < 2 * b1.min_balance)
    b1.deposit(10);
```

Both `cur_balance` and `min_balance` are private (hidden)
Class Inheritance

A simple inheritance example ...

class SavingsAccount : public BankAccount
{
    public:
        SavingsAccount(double min_bal, double rate);
        void apply_interest();
    private:
        double interest_rate;
};

Here SavingsAccount is declared as a subclass of BankAccount

A child class directly “inherits” from a parent class

A derived class (directly or indirectly) “inherits” from a base class

Derived classes inherit member variables and functions

- single inheritance: a child class directly inherits from one parent
- multiple inheritance: we won’t discuss this ... it’s tricky

Public vs Private vs Protected access

- subclasses can access public and protected base-class members
- subclasses cannot access private members base-class members
- protected section only gives subclasses (derived classes) access to members
Inheritance forms a **class hierarchy** ... *in single inheritance*:

- the hierarchy induces a tree structure
- there is always one **root** class
- a **leaf** class has no child classes

A simple example of an account class hierarchy ...

- Which is the root class?
- Which classes are derived from the root class?
- What are the base classes?
- What is **StudentCheckingAccount** derived from?
- What are the parent–child class relationships?
Returning to our simple example ...

```cpp
class SavingsAccount : public BankAccount
{
    public:
        SavingsAccount(double min_bal, double rate);
        void apply_interest();
    private:
        double interest_rate;
};
```

The "public" keyword in "public BankAccount" means that for clients:

- public members in BankAccount are public in SavingsAccount
- protected members in BankAccount are protected in SavingsAccount
- private members in BankAccount are private in SavingsAccount

If protected (private) was used instead ...

- public and protected BankAccount members are
- ... protected (private) in SavingsAccount

public inheritance is the more common case ...

Q: What BankAccount members can be accessed from SavingsAccount?
Every child-class object has a parent-class object “part”

- this means child-class objects are also parent-class objects
- i.e., every SavingsAccount object is also a BankAccount object

BankAccount b1(50);
b1.deposit(100);
SavingsAccount a1(50, 0.05);
a1.deposit(75);

Inheritance can be used for “subtype polymorphism” in C++

- whenever we use a child object when a parent object is expected

```cpp
void print_balance(const BankAccount& acct) {
    cout << "$" << acct.get_available_balance() << endl;
}

print_balance(b1);   // prints $50
print_balance(a1);   // prints $25
```

Later we’ll discuss an issue when “overriding” member functions in derived classes
Access Examples

Assume this is our class ...

```cpp
class Sphere
{
    public:
        int radius;
};
```

And this is our client program ...

```cpp
int main()
{
    Sphere s;
    s.radius = 10; // Q: Is this statement legal? ... YES!
}
```

Now assume we have a child class ...

```cpp
class Ball : public Sphere
{
    public:
        int getSize() const
        {
            if (radius < 4) // Q: Is this statement legal? ... YES!
                return 3;
        }
};
```
Now assume this is our class ...

```cpp
class Sphere
{
    private:
        int radius;
};
```

And this is still our client program ...

```cpp
int main()
{
    Sphere s;
    s.radius = 1/zero.alt1; // Q: Is this statement legal? ... NO!
}
```

And assume we have this child class still ...

```cpp
class Ball : public Sphere
{
    public:
        int getSize() const
        {
            if (radius < 4) // Q: Is this statement legal? ... NO!
                return 3;
        }
};
```

Q: What are ways to allow the child class to access parent class data?

1. Supply getter (and setter) functions for state ... (aka accessors and mutators)

2. Make state protected instead of private
   - Recall: allows child class to access data, but other clients cannot
OPTION 1: getter and setter functions

Our new class ...

```cpp
class Sphere
{
public:
    int get_radius() const;
    void set_radius(int r);
private:
    int radius;
};
```

Our client program ...

```cpp
int main()
{
    Sphere s;
    s.set_radius(10);
}
```

Our child class ...

```cpp
class Ball : public Sphere
{
public:
    int getSize() const
    {
        if (get_radius() < 4)
            return 3;
    }
};
```

Q: How is this any different than just making the data public?

- Note: don’t have to always provide both getters and setters
- Note: not all getters and setters expose state (e.g., `get_available_balance()`)
OPTION 2: Make data protected

Our new class ...

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

Our client program ...

```cpp
int main ()
{
    Sphere s;
    s.radius = 10; // Q: Is this legal? ... NO!
}
```

Our child class ...

```cpp
class Ball : public Sphere {
public:
    int getSize () const
    {
        if (radius < 4) // Q: Is this legal? ... YES!
            return 3;
    }
};
```

Q: What are some potential issues with protected?
- With public, you completely “expose” implementation details
- With protected, you partially “expose” implementation details
- Both can break modularity (encapsulation and information hiding)
- Advocates for only private, and for private data with some protected getters
Constructors and Inheritance

Consider this implementation of SavingsAccount

```cpp
// note: we're using `initializater list' syntax below
SavingsAccount::SavingsAccount(double min_bal, double rate) :
    BankAccount(min_bal), interest_rate(rate)
{};

void SavingsAccount::apply_interest()
{
    if (get_available_balance() > 0) {
        tot_bal = get_available_balance() + get_minimum_balance();
        deposit(tot_bal * interest_rate);
    }
}
```

Base class constructor is invoked before derived class constructor

- constructor of all inherited classes are called in order
- from root class (first) to parent class (last)

In our example ...

- we explicitly call the parent class constructor (in the initialization list)
- since we need to pass in the minimum balance
- otherwise, default constructor would be called for us
Consider these classes:

```cpp
class Sphere
{
    public:
        Sphere () { cout << "constructing a sphere" << endl; }
        ...
};

class Ball : public Sphere
{
    public:
        Ball () { cout << "constructing a ball" << endl; }
};
```

Q: What is printed after the following?
    Sphere s;

Q: What is printed after the following?
    Ball b;

Q: What would happen if we removed the **Sphere** constructor?
    - the automatic default constructor would be used
    - which means “constructing a sphere” wouldn’t be printed
Member Function Overriding

Overriding occurs when child class redefines parent class function

```cpp
class BankAccount
{
    public:
        BankAccount(double min_bal);
        ...
        void withdraw(double amt); // subtract amt from cur_balance
    private:
        double min_balance;
        double cur_balance;
};

class SavingsAccount : BankAccount
{
    public:
        SavingsAccount(double min_bal, double rate, double fee);
        ...
        void withdraw(double amt); // subtract amt plus a fee
    private:
        double interest_rate;
        double withdraw_fee;
};
```

Default static binding associates (“binds”) function at compile-time

```cpp
BankAccount b(50);
b.deposit(100);
b.withdraw(20);
cout << b.get_available_balance() << endl; // prints: 30

SavingsAccount s(50, 0.05, 3);
s.deposit(100);
s.withdraw(20);
cout << s.get_available_balance() << endl; // prints: 27
```
Static vs Dynamic Member Function Binding

But, static binding can get in the way of subtype polymorphism ...

    SavingsAccount s(50, 0.05, 3);
    s.deposit(100);

    BankAccount& b = s;     // Similar: BankAccount* b = &s;
    b.withdraw(20);          // And: b->withdraw(20);

    cout << b.get_available_balance() << endl;    // prints: 30!!!

Q: Why doesn’t the SavingsAccount version get called?
   • the compiler uses the declared type of b (BankAccount)
   • and so the version of withdraw in BankAccount

Instead we want the compiler to use **Dynamic Binding**
   • where the compiler uses the actual type of the b object at runtime (SavingsAccount)

```cpp
class BankAccount
{
    private:
        ...
        virtual void withdraw(double amt); // forces dynamic binding
        ...
};
```

   • dynamic binding of function recursively applies to all derived classes
   • can, but don’t need to use virtual on function in derived classes