CPSC 223
Algorithms & Data Abstract Structures

Lecture 4: Operator overloading (cont.) and pointer refresher

Today …

• Homework
  – Homework 1 due today
  – Homework 2 assigned
    • Also: finish operators for Key and Entry

• Continue with operator overloading

• Pointer refresher

• Readings
  – Ch. 4: 4.1 - 4.2
Stock example again …

Let's add a "<" operator to our Stock class …

```cpp
class Stock {
    public:
        int getShares() const;
        double getShareValue() const;
        ...
};

// in Stock.cpp:
bool operator<(const Stock& s1, const Stock& s2)
{
    ... what should go here???
}
```

Note similar to "worthMore" function

```cpp
// in Stock.cpp:
bool operator<(const Stock& s1, const Stock& s2)
{
    double v1 = s1.getShares() * s1.getShareValue();
    double v2 = s2.getShares() * s2.getShareValues();
    return v1 < v2;
}
```
Operator Overloading for Classes

```cpp
ClassName operator-(const ClassName& arg1, const ClassName& arg2);
```

We can also overload operators as member functions

```cpp
class ClassName {
    public:
        ClassName operator-(const ClassName& arg2) const;

    ...  
};
```

Stock example again ...

Let's add a "<" operator to our Stock class ...

```cpp
class Stock {
    public:
        int getShares() const;
        double getShareValue() const;
        bool operator<(const Stock& rhs) const; // as member

    ...  
};
```

```cpp
// in Stock.cpp:
bool Stock::operator<(const Stock& rhs) const
{
    double v1 = getShares() * getShareValue();
    double v2 = rhs.getShares() * rhs.getShareValues();
    return v1 < v2;
}
```

Why does this work?
The (implicit) this pointer

• Every function has an implicit argument named this
  class Stock {
    public:
      int getShares(const Stock* this) const;
      double getShareValue(const Stock* this) const;
    ...
  };

• The this variable is a pointer to an object of the class
  – In this case, a pointer to a Stock object

• Because it is a pointer, we call it the “this pointer”

The (implicit) this pointer

• Every function has an implicit argument named this
  class Stock {
    public:
      int getShares(const Stock* this) const;
      double getShareValue(const Stock* this) const;
    ...
  };

• Conceptually how the this pointer works:
  Stock s1;
  int x = s1.getShares();  // “rewritten”
  s1 is the “invoking object”
  int x = getShares(s1);  // The method is invoked on s1
Using the **this** pointer

We can use the this pointer within member functions

- E.g., to return the “current” (invoking) object

```cpp
const Stock& Stock::worthMore(const Stock& s) const
{
    double v1 = getShares() * getShareValue();
    double v2 = s.getShares() * s.getShareValue();
    if(v1 < v2)
        return s;
    else
        return *this;  // want to return current object
}
```

De-references the pointer, giving the current object!
Operator Overloading and this

As a member function:

```cpp
bool Stock::operator<(const Stock& rhs) const
{
    double v1 = getShares() * getShareValue();
    double v2 = rhs.getShares() * rhs.getShareValues();
    return v1 < v2;
}
```

As a non-member function:

```cpp
bool operator<(const Stock& s1, const Stock& s2)
{
    double v1 = s1.getShares() * s1.getShareValue();
    double v2 = s2.getShares() * s2.getShareValues();
    return v1 < v2;
}
```

Testing equality

We can similarly override the `==` operator

```cpp
bool Stock::operator==(const Stock& rhs) const
{
    // check that *this and rhs are “equal”
}
```

- When testing for equality, we generally want to:
  - Check that the two objects have the same values for member variables
  - This can sometimes be tricky (e.g., linked lists)
- In general, use member functions to override ops
  - Sometimes not possible (more later)
Overloading the Assignment operator

• What does the implicit assignment operator (=) do?
  – Generally assigns a value to a storage location
    ```
    Stock s1, s2;
    ...
    s1 = s2;   // s1 now contains a “copy” of s2
    ```
  – For objects, performs a member-to-member copy (using =)

• Problems occur for objects with pointers as members
  – After assignment, both objects will point to the same values

\[
\begin{array}{c}
\text{Object1} \\
\text{int val1 = 10} \\
\text{string* val2;}
\end{array}
\quad
\rightarrow
\quad
\begin{array}{c}
\text{Object2} \\
\text{int val1 = 10} \\
\text{string* val2;}
\end{array}
\]

After object2 assigned to object1

Overloading the Assignment operator

Overloading (=) gives us control of the copy process

• The basic signature:
  ```
  ClassName& operator=(const ClassName& rhs)
  ```
  – E.g.: Stock& operator=(const Stock& rhs)

• The general pattern:
  ```
  ClassName& ClassName::operator=(const ClassName& rhs)
  {
    if(this == &rhs)
      return *this; // rhs is invoking object
    ... do the copy ...
    return *this; // return current object
  }
  ```
Overloading the Assignment operator

When do we override assignment?

- When class members are pointer types
  - E.g., linked lists, dynamic arrays, and other dynamic structures
- However, standard to always override
  - Instead of relying on implicit copy

Copy Constructor

The copy constructor is very similar to assignment

```
Stock s2(s1);
```
- Here, we copy s1’s values when constructing s2

- Copy constructor signature:

```
ClassName(const ClassName& c);
```
  - E.g., Stock(const Stock& s);

- The main difference is that we do not need to:
  - Check if s is the current object
  - Return the current object
Access to private data

A simple copy constructor:

```cpp
Stock::Stock(const Stock& s) {
    shares = s.shares;
    shareValue = s.shareValue;
    ...
}
```

- Note that we have access to private members of `s`!
  - True for all operators overloaded as member functions
  - Also true for any member function with an argument of the same type as the current object
  - Good idea to always use getters/setters for these though …

Why?

C++ Pointer Refresher …
Pointer Refresher

• For ordinary variables, the C++ compiler allocates memory to hold the value

```
int x;
```

memory location @111 to hold an int

• Assigning a value to the variable places the value in the allocated memory space

```
x = 5;
```

value placed in the memory location @111

---

Pointer Refresher

• A **pointer** variable contains the location (or **address**) of a memory location

```
int x;
int* xptr;
```

memory location @222 to hold an int address

memory location @111 to hold an int

• Assigning a value to the variable places the value in the allocated memory space

```
x = 5;
xptr = &x;
```

x’s address placed in the memory location @222

value placed in the memory location @111
Pointer Refresher

• We can **dereference** a pointer variable to access the value stored in the memory location it points to.

```c
int x = 5;
int* xptr = &x;
*xptr = 6;
```

![](image1.png)

• We can also **dynamically allocate** memory at runtime using the `new` keyword, which returns a pointer variable.

```c
int* xptr = new int;
*xptr = 5;
```

![](image2.png)