CPSC 223
Algorithms & Data Abstract Structures

Lecture 6:
Loose Ends

Today …

• Homework
  – Homework 2 due today
  – Homework 3 assigned

• Wrap up overloading [Sect. 8.6]
• Static vs. Dynamic Binding [Sect. 8.2]
Operator Overloading

• Operator return values
  – the residual value of the expression containing that operator and its operands
  – pay close attention to the type and value returned
  – returned value allows operator to be used in larger expressions
  – … to be used as an operand in another operator (e.g., a + b + c)

C/C++ lvalues

An “lvalue”

– An addressable variable (a variable that can be assigned to)
– What goes on the “left side” of an equation, for example:
  ```
  int m = 5; // m is an lvalue
  3 = 2;     // not allowed, 3 is not an lvalue
  ```
– An lvalue can usually be assigned to …
– Except when const is used (a non-modifiable lvalue)
  ```
  const int m = 5;
  ```
C/C++ rvalues

An "rvalue"

- The opposite of an lvalue
- These values are not assignable (e.g., $5 + 2 = 8$)

Return values can make implementing operators tricky!

Operator Overloading

- All arithmetic, bitwise, relational, equality, logical, and compound assignment operators can be overloaded
- The address-of, dereference, increment, decrement, and comma operators can also be overloaded
- Many operators cannot be overloaded, including scope resolution (::), direct member access (.), direct pointer to member access (.*), conditional operator (?:), size of object operator (sizeof)
- Operators that must be overloaded as members include assignment (=), subscript ([]), indirect member access operator (->), and indirect pointer to member access operator (->*)
- It is easy to get carried away with overloading
  - can lead to code that is very hard to understand and maintain!!!
  - best to stick with basic operators (assignment, I/O, comparison) if any …
Operator Overloading Guidelines

• Determine if any of the class operations should be implemented as overloaded operators
  – Does an operator exist that performs behavior similar to our operations?
  – If so, consider overloading
  – If not, use member functions

*It is easy to get carried away overloading operators*
  – Some languages don’t support it at all (e.g., Java)

Operator Overloading Guidelines

• What to consider:
  – What *data types* are allowed as *operands*
  – Are the *operands* modified by the operation (e.g., `>>`)
  – What *data type* is returned as the residual value

*Is the residual value:*
  – an *rvalue* (an object returned by value)
  – or a modifiable *lvalue* (a reference to an object)
### Operator Overloading Guidelines

*Additional considerations … member vs. non-member*

- If the first operand is *not* always an object of the class
  - Sometimes with +, -, etc.
  - Always with >> and <<
  - Then overload the operator as a *non-member*

- If the first operand is *always* an object of the class
  - e.g., +=
  - Overload the operator function as a *member*

---

### Operator Overloading Guidelines

*Additional considerations … const*

- If the operands are *not modified* by the operator
  - the arguments should be *const* references

- As a method, if the operator does not modify the current object
  - overload it as a *const* member
Operator Overloading Guidelines

Additional considerations … lvalue vs. rvalue

- If the operator results in an *lvalue expression*
  - the return type should be *returned by reference*
  - for example = results in an lvalue expression

- If the operator results in an *rvalue expression*
  - Return by value (causing the copy constructor to be invoked)
  - For example “+” results in an rvalue expression

Operator Overloading

A note on operator=

```c
int x = 1, y = 2, z = 3;
x = y = z;
```

What values are in x, y, and z?

```
x = 3, y = 3, z = 3
```

```
int x = 1, y = 2, z = 3;
(x = y) = z;
```

What values are in x, y, and z now?

```
x = 3, y = 2, z = 3
```
The Assignment Operator

Overriding operator=

What are the 3 steps?

Key& operator=(const Key& rhs)
{

1). Check if rhs is the current object

2). Perform the copy (default is member-to-member)

What happens if we use the default for our List class?

3). Return the current object

}

Allows us to chain =’s together (a = b = c)

… and assign the result of equals (an lvalue!)

Stream Insertion (<<) and Extraction (>>)

• We want to support statements like these:

  Key k;
  cout << "Please enter a word: ";
  cin >> k;
  cout << "You entered: " << k << endl;

• For this we overload << and >> for Key

  ostream& operator<<(ostream& out, const Key& key);
  istream& operator>>(istream& in, Key& key);

Why these arguments?
Stream Insertion (<<) and Extraction (>>)

```cpp
ostream& operator<<(ostream& out, const Key& key);
istream& operator>>(istream& in, Key& key);
```

Can these be implemented as member functions?

No!

Why not?

– The first argument has to be of the class (here Key) …

• They must be implemented as non-member functions
  – We usually do this using the “friend” keyword

• How we declare these …

```cpp
class Key {
    public:
    friend ostream& operator<<(...)
    friend istream& operator>>(...)
    ...
    private:
    ... These are then implemented in Key.cpp
    ... Friend means the method has access to
    private data of Key
};
```
More on Inheritance …

Destructors and Inheritance

• Destructors called at the end of the object’s lifetime

• Destructors are called in the opposite order of their constructors
  – this means that the derived class destructor is called before its base class destructor
  – this sequences continues until the furthest base class destructor is invoked
  – a derived class destructor is guaranteed to have its base class members available for use
Member Functions and Inheritance

What happens when member functions in a hierarchy have the same name? Does overloading occur?

NO!

– overloading means that we have unique signatures for the same named function within the same scope

– in an inheritance hierarchy, each class has its own separate class scope

– overloading doesn’t apply between classes

Member Functions and Inheritance

Instead, inheritance allows methods in a base class to be hidden in a derived class

– The behavior of the functions is redefined in the derived class

– Without changing the base class or requiring changes to existing client applications
Member Hiding

Methods are hidden when we specify a method that has
the same name as a method in a base class

– A method in a derived class hides a method in a base class,
even if the signatures are different

– When the method is accessed the derived member is used
(usually)

– What if the argument list of the client does not match any
of the functions defined within the derived class?

A compile error occurs