CS2704

Topic 17
Operator Overloading

Outline

• Operators that can be overloaded
• Syntax for overloading
  – Member vs nonmember operators
  – Binary and unary operators
  – Prefix and Postfix operators
• Overloading stream operators
• Type conversion

Overloading & Polymorphism

• Overloading is considered “ad-hoc” polymorphism
• Can define new meanings (functions) of operators for specific types
• Compiler recognizes which function to use by signature (types of arguments)

Reasons for Overloading

• Support natural, suggestive usage
  Ex. Addition means “addition”
• Semantic integrity
  Assignment for objects with pointers necessary to ensure copied properly
• Uniformity with built-in types
  Able to use objects in situations expecting primitive values

Operators that can be Overloaded

+ - * / % ^ &
| ~ ! = < > +=
-= *= /= %= ^= &= |=
<< >> >>= <<= == != <=
>= && || ++ - ->* ,
-> [] () new new[] delete delete[]

Rule: if you don’t know what it means, don’t overload it!

Syntax for Overloading Operators

• Declared like other methods
• Ex. as method of the Name class
  bool operator==(const Name&)
• Ex. As nonmember function
  bool operator==(const Name&, const Name&);
• Subsequent examples show member and nonmember forms
Using Overloaded Operators

- If `operator==` defined as *member* function
  
  `nme1 == nme2`

  is the same as
  
  `nme1.operator===nme2`

- If `operator==` defined as *nonmember* function
  
  `nme1 == nme2`

  is the same as
  
  `operator==(nme1, nme2)`

Binary and Unary Operators

- A subtract operator
  
  `Complex operator-(const Complex&);`

- A negate operator
  
  `Complex operator-();`

Prefix and Postfix Operators

- A prefix operator
  
  `Day operator++(); //member`

- A postfix operator
  
  `Day operator++(int); // int is dummy`

- The `int` is a dummy type to show postfix

Multiple Overloading

- Can have two addition operators in a class
  
  `Complex operator+(const int&);`

- Signature of function used to resolve which is used:
  
  `Complex a, b;`

  `a + 1 // add an int`

  `a + b // add a Complex`

Overloading Resolution

- Suppose have code
  
  `X x;`

  `Y y;`

  `x + y`

- Overloading resolution is what compiler uses to determine what definition of “+” to use.

Overloading Resolution (2)

- Compiler looks for ‘+’ in this order
  
  1. Member function in `X` of form
     
     `- X::operator +(Y)`

  2. Nonmember function of form
     
     `- ::operator+(X,Y)`

  3. Return type is not part of lookup
When Nonmember Functions

- Operators on primitive data types
  - Ex., Complex operator+(int, const Complex&);
- Class source not always available
  - E.g., ostream
- Type casting

I/O Operators

- I/O operators ("<<", ">">") on streams
- Do not have access to class code
- Define operator<< as nonmember function
  
- Must have access to data in Name class
  - Provide accessors, or
  - Make operator friend of Name class

Example Name Class

class Name {
public:
  Name(string first, string last):
    first_name(first), last_name(last) {} 
private:
  string first_name;
  string last_name;
  friend ostream& operator<<(ostream&, const Name&); 
};

Example Name operator<<

- Function has access to data of Name class
- Also, easy to provide accessors for Name class

Automatic Type Conversions

- Declaring:
  
- Allows the following (constructor can convert string)
  
- However, this doesn’t work
  
- Reason: no function string::operator-(Date)
- Solution: define nonmember function

Type Conversion Operators

- Constructors sufficient unless target type of conversion not a class, or can’t be changed
- Can define type conversion operators
- Ex: `Date::operator string() const;`
  - Convert Date object to string object
  - No return type

Type Conversion Caution

- Single argument constructors are used for type conversion
- Ex: `Queue(int n)` constructor for size `n`
  
  ```cpp
  a = 5;  // same as writing
  a = Queue(5);  // don’t want this
  ```
- Can declare constructor as `explicit` - modifier before constructor in class declaration

Overloading Guidelines

- Avoid violating expectations about operator
- Provide complete set of properly related operators: `a = a + b` and `a += b` do same
- Avoid type conversions
- Define operator as class member unless necessary