Operator Overloading

CS2704: Object-Oriented Software Design and Construction

Constantinos Phanouriou
Department of Computer Science
Virginia Tech

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
Operator Overloading

- **Operator overloading** is the ability to define a new meaning for an existing operator
- The list of operators includes:
  - mathematical: +, -, *, /, etc.
  - relational: <, >, ==, etc.
  - logical: &&, ++, !, etc.
  - access operators: [ ], ->
  - assignment operator: =
  - stream I/O operators: <<, >>
  - type conversion operators, and several others

Overloading

- Each operator has predefined and unchangeable meaning for the built-in types (int, float, char, etc.)
- Each operator can be given a specific interpretation for individual user-defined classes or combination of user-defined classes
- Compiler recognizes which function to use by signature (types of arguments)
- Overloading can dangerous and few object-oriented languages allow it
Reasons for Overloading (1)

• There are a number of reasons why a class designer may decide to provide extensions to one or more of the built-in operators:
  – Support natural, suggestive usage
  – Semantic integrity
  – Uniformity with built-in types

Reasons for Overloading (2)

• Natural, suggestive usage
  – The most natural way to convey the intended meaning of an operation may be through the predefined operators.

• Example:
  – In defining a class that represents complex numbers, the best way to represent adding two complex numbers is by giving a new meaning to the plus operator (+) rather than invent a member function with a suggestive name (“addTo”)
Reasons for Overloading (3)

- Semantic integrity
  - Assignment for objects with pointers necessary to ensure copied properly
  - Failure to properly handle assignments can lead to either memory leaks or run-time errors

Reasons for Overloading (4)

- Uniformity with built-in types
  - Able to use objects in situations expecting primitive values
  - Template may require the overloading of certain operators
Operators that can be Overloaded

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

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

Operators that cannot be Overloaded

- Scope resolution:  ::
- Member selection:  .
- Member selection through pointer to member:  .*
- The ternary conditional operator:  ?:
- The named operators:  sizeof and  typeid
- New operator tokens (e.g.,  **)
Syntax for Overloading Operators

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

Example: Safe array of integers

- Safe because subscripting operator will ensure that the subscript is within bounds

```cpp
class Array {
  private:
    int array[20];
  public:
    Array(int init=0);
    int& operator[](int I);
    ~Array();
  }
```
Example cont.

```cpp
int& Array::operator[](int i) {
    assert(i >= 0 && i < 20);
    return array[i];
}

Array array;
array[0] = 1;
array[1] = 1;
// Compute first twenty Fibonacci numbers
for (int i = 2; i < 20; i++)
    array[i] = array[i-1] + array[i-2];
```

Example cont.

```cpp
Array array;
array.operator[](0) = 1;
array.operator[](1) = 1;
// Compute first twenty Fibonacci numbers
for (int i = 2; i < 20; i++)
    array.operator[](i) =
        array.operator[](i-1) +
        array.operator[](i-2);
```
Basic Operator

- Notice that aside from its declaration, the Array object looks and feels like a built-in array type
- New and built-in types can be intermixed:

```cpp
Array safe;
int regular[20];

regular[10] = safe[10];
safe[0] = safe[0] + regular[0];
```

Overloading more operators

- Can can also add or subtract safe Arrays by overloading the + and - operators:

```cpp
class Array {
private:
    int array[20];
public:
    Array(int init=0);
    int& operator[](int I);
    Array operator+(Array& other);
    Array operator-(Array& other);
    ~Array();
};
```
Overloading more operators

Array a, b;       // Initialized to 0
Array one(1);    // Initialized to 1

Array c = a + b;
// Overloaded operators can be used in
// complicated expressions
Array d = a - b + one;

Overload operator+

- It is possible to overload the function that defines an operator
- For example:

  ```cpp
class Array {
    private:
      int array[20];
    public:
      ...
      Array operator+(Array& other);
      Array operator+(int increment);
  }
  ```
Overload operator+

Array Array::operator+ (Array& other) {
    Array result;
    for(int i=0; i<20; i++)
        result[i] = array[i] + other[i];
    return result;
}

Array Array::operator+ (int increment) {
    Array result;
    for(int i=0; i<20; i++)
        result[i] = array[i] + increment;
    return result;
}

Array one(1); // Initialized to 1

// add 1 to each element
Array two = one + 1;
// Array two = one.operator+(1);

// Pairwise addition of Arrays one and two
Array three = one + two;
// Array three = one.operator+(two);
Exercise on Array

- Array Equality Operator
  - Extend the Array class to include a test for an equality operator that returns 1 if all the elements of the two Array objects are the same, and 0 otherwise

For example:

```cpp
Array a, b;
// Give some values to a and b
if (a == b) {
}
```

Exercise cont.

```cpp
class Array {
  private:
    int array[20];
  public:
    ...

    _________________

  _____ Array::operator_______________ {

  }

}
```
Overloading named operators

class Array {
    private:
        ...
    public:
        ...
        void* operator new[](size_t);
        void* operator delete[](void*, size_t);
        ...
}

Operator Overloading

• Function that modify the meaning of operators can be defined:
  – as function members of a class
  – as non-member function
Using Overloaded Operators

- If operator $==$ defined as member function
  \[ \text{nme1} == \text{nme2} \]
  is the same as
  \[ \text{nme1.operator}==(\text{nme2}) \]
- If operator $==$ defined as nonmember function
  \[ \text{nme1} == \text{nme2} \]
  is the same as
  \[ \text{operator}==(\text{nme1, nme2}) \]

Using non-member functions

- There are two situations under which operator overloading must be done by functions that are not members of a specific class
- When the class to which the member function should be added is not available for modification
  - Typical for standard library classes (e.g., I/O streams)
- When type conversion of the arguments involved in the operation is desired
Rules for overloading

- If an expression of the form “x op y” is encountered, the compiler will check:
  - is there a member function in the class of object x of the form “operator op(Y)” where Y is the class of object y, and if not,
  - is there a non-member function of the form “operator op(X, Y)” where X is the class of object x, and Y is the class of object y.

Overloading with non-member

- When overloading is achieved using non-member functions, there are two cases to be considered:
  - overloaded operator uses:
    - only methods in the public interface of the class(es) involved in the overloading
    - requires access to the private data of the class(es)
Non-member, without special access

• Suppose we want the content of an array.
• We print it as follow:

```cpp
Array a;
// give values to a
cout << "[ ";
for(int i=0; i<19; i++)
    cout << a[i] << ", ";
cout << a[19] << "]";
```

Non-member, without special access

• We can improve the printing of an array by overloading the stream I/O operators (<<,>>)
• Then we can write:

```cpp
Array a;
// give values to a
cout << a;
```
Non-member, without special access

- Modifying the I/O stream classes is not possible since we do not have access to the implementation
- We need to define a non-member function:

```cpp
ostream& operator<< (ostream& os, Array& a) {
    os << "[ ";
    for(int i=0; i<19; i++)
        os << a[i] << ", ";
    os << a[19] << " ]";
    return os;
}
```

Non-member, with special access

- Non-member functions may be given special access to the private or protected data of a class
- Special access can be granted by declaring the function as “friend” (we will talk about friend functions later)
- Granting special access to private data should be used sparingly.
Binary and Unary Operators

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

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

---

Binary Operator Example

- Member Function:
  
  ```
  Array Array::operator-(int increment);
  Array one;
  Array two = one - 1;
  // Array two = one.operator-(1);
  ```

- Non-member Function:
  
  ```
  Array operator-(const Array &a, int increment);
  Array one;
  Array two = one - 1;
  // Array two = operator-(one, 1);
  ```
Multiple Overloading

• Can have two addition operators in a class
  \[
  \text{Complex operator+}(\text{const int}&) ; \\
  \text{Complex operator+}(\text{const Complex}&) ;
  \]
• Signature of function used to resolve which is used:
  \[
  \text{Complex } a, b ; \\
  a + 1 \ // \text{add an int} \\
  a + b \ // \text{add a Complex}
  \]

Prefix and Postfix Operators

• A prefix operator
  \[
  \text{Day operator++}() ; \ // \text{member} \\
  \text{Day operator++}(\text{Day}&) ; \ // \text{nonmember}
  \]
• A postfix operator
  \[
  \text{Day operator++}(\text{int}) ; \ // \text{int is dummy} \\
  \text{Day operator++}(\text{Day}&, \text{int}) ;
  \]
• The \text{int} is a dummy type to show postfix
Unary Operator Example (prefix)

- Member Function:
  ```
  Array Array::operator--();
  Array one;
  Array two = --one;
  // Array two = one.operator--();
  ```

- Non-member Function:
  ```
  Array operator--(const Array &a);
  Array one;
  Array two = --one;
  // Array two = operator--(one);
  ```

Unary Operator Example (postfix)

- Member Function:
  ```
  Array Array::operator--(int x);
  Array one;
  Array two = --one;
  // Array two = one.operator--();
  ```

- Non-member Function:
  ```
  Array operator--(const Array &a, int x);
  Array one;
  Array two = --one;
  // Array two = operator--(one);
  ```
Array Example (1)

class Array {
private:
    int array[20];
public:
    Array(int init=0);
    int& operator[](int I);
    Array operator+(Array& other);
    Array operator-(Array& other);

    Array operator++(); // Prefix
    Array operator++(int); // Postfix

    ~Array();
};

Array Example (2)

Array Array::operator++ () {
    Array result;
    for(int i=0; i<20; i++)
        result[i] = ++array[i];
    return result;
} // Prefix

Array Array::operator++ (int dummy) {
    Array result;
    for(int i=0; i<20; i++)
        result[i] = array[i]++;
    return result;
} // Postfix
Array Example (3)

- Same functions, but as non-members

```c
Array operator++ (Array &array) {
    Array result;
    for(int i=0; i<20; i++)
        result[i] = ++array[i];
    return result;
} // Prefix

Array operator++ (Array &array, int dummy) {
    Array result;
    for(int i=0; i<20; i++)
        result[i] = array[i]++;
    return result;
} // Postfix
```

Prefix and Postfix Operators (1)

- Prefix and postfix simply refer to whether the operator appears before or after the object
- The implementation determines if the object will be returned before or after it is modified (or if modified at all)
Prefix and Postfix Operators (2)

- If postfix operator is missing, then prefix operator will be used and a warning will be generated
  - `Array b = a++;`

- If prefix operator is missing, then an error will be generated
  - `Array b = ++a;`

Predefine Meanings for Operators

- The meanings of some built-in operators are defined to be equivalent to some combination of other operators on the same argument
- Example: (assume `a` is an `int`)
  - `++a, a+=1;, a=a+1` (are equivalent)

- Such relations do not hold for user-defined operators unless the user happens to define them that way
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
  - `Complex operator+(int, const Complex&);`

- Class source not always available
  - E.g., `ostream`

- Type casting

Primitive Data Types (1)

- With member function we can have:
  
  ```
  Array one;
  Array two = one - 1;
  // Array Array::operator-(int increment);
  ```

- Suppose we want the following:
  
  ```
  Array one;
  Array two = 1 - one;
  // Array int::operator-(Array &array);
  // Not possible since int is a primitive type
  ```
Primitive Data Types (2)

Array one;
Array two = 1 - one;

• We need a non-member function:
  // Array two = operator-(1, one);

```
Array operator+ (int increment, Array &array) {
    Array result;
    for(int i=0; i<20; i++)
        result[i] = array[i] + increment;
    return result;
}
```

I/O Operators

• I/O operators ("<<", ">>") on streams
• Do not have access to class code
• Define operator << as nonmember function
  ostream& operator<<(ostream&, const Name&);
• Must have access to data in Name class
  – Provide accessors, or
  – Make operator friend of Name class
Friends (1)

- An ordinary member function declaration specified three logically distinct things:
  1. The function can access the private part of the class declaration, and
  2. the function is in the scope of the class, and
  3. the function must be invoked on an object (has a this pointer)

```cpp
void add(int x);
```

Friends (2)

- A static member function declaration specified two logically distinct things:
  1. The function can access the private part of the class declaration, and
  2. the function is in the scope of the class, and
  3. the function must be invoked on an object (has a this pointer)

```cpp
static void add(int x);
```
Friends (3)

• A friend member function declaration specified one logically distinct things:
  1. The function can access the private part of the class declaration, and
  2. the function is in the scope of the class, and—
  3. the function must be invoked on an object (has—a this pointer)
friend void add(int x);

When to use friend

• Avoid the use of friends unless the alternative is to:
  – use global data
  – use public data members
Friend Example (1)

- Suppose we want an operator that multiplies a Matrix by a Vector.
  - Do we make this operator part of the Vector or the Array, or both?
  - We can declare the `operator*` a friend of both

Friend Example (2)

```cpp
class Vector {
    float v[4];
    //...
    friend Vector operator*(
            const Matrix&, const Vector&);
};

class Matrix {
    Vector v[4];
    //...
    friend Vector operator*(
            const Matrix&, const Vector&);
};

Vector operator*(
    const Matrix& m, const Vector& v) {
```
}
Friends

- A friend declaration can be placed in either the private or public part of the class declaration (it does not matter where).
- A friend function behaves like any member function part of the class.

More friend Usage

- A member function of one class can be the friend of another. For example:

```cpp
class List_iterator {
    //...
    int* next();
};
class List {
    friend int* List_iterator::next();
    //...
}
More friend Usage

- Also, if all the member function of one class are friends of another class, then there is a shorthand for this:

```cpp
class List {
    friend class List_iterator;
    //...
};
```

Finding Friends

- Like a member declaration, a `friend` declaration does not introduce a name into an enclosing scope

```cpp
class Matrix {
    friend class Xform;
};

Xform x; // Error: no Xform in scope
```
Automatic Type Conversions (1)

- Declaring:
  ```cpp
class Date {
    public:
      Date(string);
      long operator-(const Date&) const;  
      //...
  };
```

- Allows the following (constructor can convert string)

  ```cpp
today = "November 9, 1999";
long left = deadline - "November 9, 1999";
```

Automatic Type Conversions (2)

```cpp
today = "November 9, 1999";
long left = deadline - "November 9, 1999";
```

- This works because the compiler will translate to:

  ```cpp
long left =
    deadline.operator-(Date("November 9, 1999"));
```
Automatic Type Conversions (3)

• However, this doesn’t work

  long total = “November 1, 1999” - deadline;

• Reason: no function string::operator-(Date)

• Solution: define nonmember function

  long operator-(const Date&, const Date&);

Type of conversions (1)

  int i;
  float f;

  f = i; // Implicit conversion

  // Explicit conversion
  f = (float)i;
  f = float(i);
Type of conversions (2)

• Implicit conversion
  – compiler is responsible for determining that a conversion is needed and how to perform it

• Explicit conversion
  – programmer assume full responsibility
  – two different, but equivalent, syntaxes for explicit conversion

Conversion Uses

• Type conversion is needed for:
  – resolving mismatched types in assignments and expressions
  – when passing parameters to functions

• The existence of type conversions makes it possible to use one type when a different type may be expected
Type Conversion Operators (1)

- Constructors can play a role in type conversion
- A constructor can be viewed as a way of converting one data type to another
- Example:

  ```cpp
class Date {
  public:
    Date(string);
    // ...
  }
```

Type Conversion Operators (2)

- Constructors cannot specify:
  - an implicit conversion from a user-defined type to a basic type (because the basic type are not classes), or
  - a conversion from a new class to a previously defined class (without modifying the declaration for the old class)
- These problems can be solved by defining a conversion operator for the source type
Conversion Operator

• A conversion operator is a member function:
  – X::operator T(), where T is a type name, defines a
    conversion from X to T. No return value.

• Example:

```cpp
class Date {
public:
    Date(string);
    operator string();
    // ...
};

Date::operator string() {
    string s;
    // ...
    return s;
}
```

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);
  ```
• Can declare constructor as `explicit` - modifier before constructor in class declaration
Explicit Example

class Queue {
public:
    explicit Queue(int n) { /* Init */ }
};
a = 5;       // Illegal
a = Queue(5);  // Legal

- "explicit" prevents the calling of the constructor implicitly

Overloading Guidelines (1)

- Define operators primarily to mimic conventional usage
- For large operands, use const reference argument types
- For large results, consider optimizing the return
- Prefer the default copy operations if appropriate for a class
Overloading Guidelines (2)

- Redefine or prohibit copying if the default is not appropriate for a type
- Prefer member functions over nonmembers for operations that need access to the representation
- Prefer nonmember functions over functions for operations that do not need access to the representation

Overloading Guidelines (3)

- Use namespaces to associate helper functions with “their” class
- Use nonmember functions for symmetric operators
- Use () for subscripting multidimensional arrays
- Make constructors that take a single “size argument” explicit
Overloading Guidelines (4)

- For non-specialized uses, prefer the standard string to the result of your own exercises
- Be cautious about introducing implicit conversions
- Use member functions to express operators that require an lvalue as its left-hand operand