Operators

Sanjeev Qazi

What are operators?

- Operators is just another name for functions that have a funny name :=)

- See the integer expressions:
  - 4 + 5 operator is "+"
  - 4 - 5 operator is "-"
  - 4 * 5 operator is "*"
  - 4 / 5 operator is "/"

- In C++, operators on built in primitive types are defined.
- But if we want to have similar operators on classes (or enums), we need to define these ourselves.

- C++ gives us the ability to define operators on classes or enums.
- (operators on classes are the ones most commonly used)

- These operators have to be one of the allowed operators by the language.

Why do we need to define operators?

- Which is easier to understand ?
  - Complex a, b, c, d; // Complex is a class representing the complex numbers
  - Add(Multiply(Add(a, b), c), d);
- OR
  - Complex a, b, c, d;
  - d + c * (a + b)

Why do we need to define operators?

- For many classes, it makes sense to define operators for things like addition, subtraction, etc.
- Although we can do without operators in such cases (by defining appropriately named member function like Add, Subtract,…), it is more intuitive to have operators.
- Example:
  - Complex numbers (See code example) can have operators for:
    - Addition (operator +)
    - Subtraction (operator -)
    - Multiplication (operator *)
- For syntax of declaring operators, see the Complex class example.

Member and non-member operators

- Operators can be defined as members of a class (as we have seen), or outside of the class (i.e., non-member operators).
- The reason for defining members outside of a class is for symmetric operation.
- When member operator is used, the **object** on whom the operator function is called, needs to be the **left operand**.
- Example:
  - Complex c1 (4, 5);
  - Complex c2 = c1 + 5; ← This would invoke *Complex::operator +*
  - Complex c3 = c1 + c2; ← This would invoke *Complex::operator +*
  - Complex c4 = 5 + c1; ← This would invoke non member *operator + (double, const Complex &)*

With respect to the 4th bullet above:

If non-member operator were **not** to be used, then it is like saying that 5 would need to have the operator + defined that takes a Complex object as an argument.

Non-member operators

- So, if we were to define only the member operators, then expressions like 5.0 + c1 would not compile (Assume c1 is an instance of class Complex).
- Nor would expressions like 5 * c1
- Another example is from our MyString class:
  - Lets say MyString class has an == operator declared as:
    - *bool MyString::operator == (const MyString & other) const*
    - Using above, we can compile the following statements:
      - MyString s1, s2;
      - bool b1 = s1 == s2;
      - bool b2 = s1 == "some string literal";
    - But, we cannot compile the following :
      - bool b3 = "some string literal" == s1;

Non-member operators

- So, for compiling
  - bool b3 = "some string literal" == s1;
  - We need to add a non member operator == as follows:
    - *bool operator == (const char *, const MyString &)*
- **Question**:
  - Consider the statement below:
    - **"some string literal1" == "some string literal2";**
    - Will this compile, given the == operators we have declared so far?
    - What operator would this use?

Assignment operator

- This operator has a predefined meaning for operands of class types (compiler generated "=" operator), however we can overload this operator.

- You should overload the "=" operator if you need to do something more than a compiler generated "=" operator would do.
- Note that if you think you need to provide a copy constructor, then you need to overload the "=" operator also, and provide the destructor also.

- Assignment operators should return a reference to the object, so that assignments can be chained, as in:
  - a = b = c;
  - What this really looks like is :
    - a = (b = c) ← Right to left association.

  - If you don't return a reference, it is still OK, but you cannot do chain assignment as shown above.

Assignment operator

- You can have more than one assignment operator, essentially one for every type that you want to be assignable to the object of your class.

- For our MyString class, we may have:
  - MyString & operator = (const MyString & other);
  - MyString & operator = (const char * other);
  - MyString & operator = (int length); ← *Not recommending this type of assignment, this is just as an example.*
- Assignment operator is essentially the same as a copy constructor, but with the following differences:
  - Returns a reference to self (usually).
  - Usually guards against assignment to self.
  - May release any memory or other resource and acquire new memory or resource.
  - In case of assignment, the object on left already exists, but in case of copy constructor, the object is being created.

MyString assignment operator

```
MyString &  MyString::operator  =  ( const  MyString &  other )
{
    if ( &other != this )                    ← guard against self assignment
    {
        char * buffer = NULL;
        if ( other.m_Data  !=  NULL )
        {
            buffer = new char [ other.Length() + 1 ];
            CopyString( buffer, other.m_Data );
        }

        delete [] m_Data;                    ← Very important: delete m_Data after
                                                allocation and copying

        m_Data = buffer;
    }

    return *this;                            ← return  reference to self.
}
```

Subscript operator []

- You need to provide the subscript operator when you have something like a collection and you need to provide random access to the elements in the collection.

- Example:
  - char & MyString::operator[] (unsigned int index);

  - Above operator is used as:
    - MyString s1("Hello");
    - char ch = s1[1]; OR
    - s1[1] = 'E';

  - Note that the access operator returns a reference to a character, that's why the subscript operator can appear on the left side of the assignment statement.

Subscript operator []

- Usually, subscript operators are provided in pairs as follows:
  - char & MyString::operator[] (unsigned int index);
  - const char & MyString::operator[] (unsigned int index) const;

- This is so that we can use the subscript operator on const objects also.

- Example:

  - const MyString s1("Hello");
  - char ch = s1[1]; ← Will use the second [] operator above.

  - s1[1] = 'E'; ← Will be a compilation error, since the return type of the second [] operator above is a const char &

Subscript operator [ ]

- This operator cannot be a non-member.
- Note that the subscript operator takes only one argument.
- If you need to have more than one argument then you should overload the function call operator ().
- Example:
  - A Matrix class where you want to provide two arguments for accessing an element, a row and a column index.
- Operator () can take any number of arguments.

Operator ( )

- This can take any number of arguments (including none).
- Arguments can have default values (unlike other operators).
- Example:
  - Complex class has one of these. See the code for an example.
- Objects that have this operator defined (i.e., classes that have this operator) are also sometimes called *functors*.

Operator ++

- ++ is the increment operator.
- In C++ (like in C), increment can be *post* or *pre*.

- ```
  class MyClass
  {
  public:
      MyClass &  operator ++ ();          ← prefix operator (pre increment)
      MyClass    operator ++ ( int );     ← postfix operator (post increment)
  };
  ```

  The dummy int argument is used on the postfix operator to distinguish it from the prefix operator, since the operator itself ("++") is the same in the case of pre and post increment.

  The pre increment operator usually returns a reference to the object.
  The post increment operator usually returns the object by value.

  NOTE: Post increment is slightly more expensive than pre increment (Why?), hence pre-increment should be preferred over post increment if you don't really need post increment.
  ```

Operator --

- -- is the decrement operator.
- In C++ (like in C), decrement can be *post* or *pre*.

- You can refer to the previous slide on ++ and substitute ++ by – (i.e., declaration and usage considerations are the same as for ++)

Some restrictions...

- Following operators cannot be defined as non-members:
  - Assignment ("=")
  - Subscript ("[]")
  - Function ("()")
  - Member access arrow ("->")

- Only the predefined set of operators can be overloaded. You can refer to any book for this set.

- As an example, there is no exponentiation operator in this set. So you cannot use something like ** as the exponentiation operator. You will have to have a function like pow() or something similar.

- Note that x * * y is already a valid statement in C++.

Some more...

- You cannot change the meaning of operators for built in types (Aren't you glad !!!)
- This means you cannot have something like:
  - bool operator == (int, int) OR
  - bool operator == (const char *, const char *)
- You cannot define additional operators for built in types.
- Above restrictions essentially mean that any non-member operators have to have at least one parameter of class type (or enumeration).
- You cannot change the *arity* of operators.
- Operators cannot take default arguments (except the function call operator ())

- Following slides for your reference (may not be covered in class)

More on assignment...

- In some cases, you do not want assignment operator to be used on objects.
- In such cases, you will most likely not want the copy constructor to be used as well.

- However, the solution for this is NOT to not define the assignment operator and copy constructor. (Remember, if you do not provide the copy constructor, the compiler will generate one which will simply do a bitwise copy of the data. Likewise for the assignment operator).

- In such cases, you should declare the assignment operator and the copy constructor as private, and not provide any definition for them.
- Declaring them private ensures that no one outside the class can use it (compile time error if someone attempts to use).
- Not providing the definition ensures that if you mistakenly use it inside the class methods, then you will get a link error.

Slightly "potentially" incorrect assignment operator

Continuation of assignment operator discussion earlier in this handout

```
MyString & MyString::operator = (const MyString & other )
{
    if (&other != this)
    {
        delete [] m_Data;                          ← m_Data is deleted here.

        if (other.m_Data != NULL)
        {
            m_Data = new char [ other.Length() + 1 ];      ← Stmt 1
            CopyString( m_Data , other.m_Data );           ← Stmt 2
        }
        else
            m_Data = NULL;
    }
    return *this;
}
```

Continued from previous slide

- If Stmt **1** or Stmt **2** throw an exception, then control transfers out of the "=" operator, and the string you are trying to assign to is left in an inconsistent state (because m_Data is already deleted).

- As a result, when the string is deleted, its destructor will try to delete m_Data, and as we saw, that has already been deleted in the assignment operator.

- ```
  int foo()
  {
      MyString  s1 ("Hello"),   s2("SomeOtherString");
      s1 = s2;
  }
  ```
  ← *Lets say an exception is thrown by new or CopyString called from inside the assignment operator.*

  ← At this point, string s1 and s2 will be deleted as they go out of scope. s1.*m_Data* will be deleted again in the destructor, causing undefined results (most likely a program crash).
  ```

Operator new

- The *new* operator can be overloaded on a class.
- This means that when new is called on this class to create an object on the heap, then the global new is not called, the one on this class is called.

- Note that *new* is a static operator because when we call new on a class, we don't have an object at that point (we are actually trying to create one).
- What is interesting is that you don't have to use the static keyword to mark it as static when you declare it in the class.

- Example:
  - void * operator new (size_t); ← single object allocator
  - *void * operator new [] (size_t);* ← *array allocator*
  - *void * operator new [] (size_t, Complex *);* ← *placement new for arrays*

  - MyClass * mc1 = new MyClass(); ← Calls *new* defined in MyClass (for single object).

  - MyClass * mc2 = ::new MyClass(); ← Uses global scope resolution operator to call the global *new* operator (for single object).

Operator delete

- The *delete* operator can be overloaded on a class.
- This means that when delete is called on an object of this class, then the global delete is not called, the one on this class is called.
- Note that *delete* is a static operator also, like operator *new.*

- Example:
  - void operator delete (void *); ← single object delete
  - *void operator delete [] (void *);* ← array delete
  - *void operator delete [] (void *, Complex *);* ← placement delete
  - NOTE: *Don't worry about the placement delete, its here just FYI,*

  - delete mc1; ← Calls *delete* defined in MyClass.
  - ::delete mc2; ← Uses global scope resolution operator to call the global *delete* operator.