COMP2012H

Generic Programming:
Overloading Operator Functions

From Math Notation to Operators in Programming Languages

- Depending on what programming language you’re using, to program the mathematical equation

  $$c = 2(a - 3) + 5b$$

  you might have to write out each function calls, as in

  ```
  c = add(mult(2, sub(a, 3)), mult(5, b))
  ```

- But most programming languages have operators which allow us to mimic the mathematical notation by writing:

  ```
  c = 2*(a-3) + 5*b;
  ```

- However, most languages (like C) only have operators defined for the *built-in* types.
- C++ is an exception: it allows you to redefine most of its operators for *user-defined* types. e.g. you may redefine `+`, `-`, etc. for types `Complex`, `Matrix`, `Array`, `String`, etc.

Example: Additions of Vectors

```
class Vector
{
    double _x, _y;
public:
    Vector(double x, double y) : _x(x), _y(y) { }
    double x() const { return _x; }
    double y() const { return _y; }
};
```

- To add 2 vectors, traditionally we would do it like this:

```
Vector add (const Vector& a, const Vector& b)
{
    return Vector( a.x() + b.x(), a.y() + b.y() );
}

Vector a(1, 3), b(-5, 7), c(22, 2), d;
d = add(a, add(b, c));
```

Non-Member Operator Function

- It would be nicer if we could write the last expression

```
d = add(a, add(b, c));
```

  instead as `d = a + b + c`.
- We can achieve that in C++ by simply replacing the name of the function `add()` by `operator+()`.

```
Vector operator+ (const Vector& a, const Vector& b)
{
    return Vector( a.x() + b.x(), a.y() + b.y() );
}

Vector a(1, 3), b(-5, 7), c(22, 2), d;
d = a + b + c;
```

Operator Syntax

- `operator+` is a formal function name that can be used like any other function name.
  - (It’s just like `add` in the example from the first slide.)
- Here we have used the “nickname”-syntax to call `operator+`. Technically, we could instead have used the “formal address” `operator+` as follows:

  ```
  d = operator+(operator+(a, b), c);
  ```

  (But nobody would really write code like this.)
- Operators in C++ are just like ordinary functions, except that they *also* have a nicer syntax for calling them similar to the usual mathematical notations.
- The operator `+` has a *formal name*, namely `operator+` (consisting of 2 keywords), and a “nickname" namely `+`.

Operator Syntax

- The nickname can only be used when calling the function.
- The formal name can be used in any context, when declaring the function, defining it, calling it, or taking its address.
- There is nothing that you can do with operators that cannot be done with ordinary functions. In other words, *operators are just syntactic sugar*.
- Be careful when defining operators. There is nothing that inhibits you from defining `+` to denote subtraction. There is nothing that inhibits you from defining `a = a + b` and `a += b` to have two different meanings. However, this would be extremely bad style – your code will become unreadable.

Don't shock the user!

C++ Operators

- Almost all operators in C++ can be overloaded except:

  `.`   `::`   `?:`   `sizeof`

- The C++ parser is fixed. That means that you can only *redefine existing operators*, but you CANNOT define *new* operators.
- Nor can you change the following properties of an operator:
  - Arity: the number of arguments an operator takes.

    e.g. `!x`   `x+y`   `a%b`   `s[j]`

    (So you are not allowed to re-define the plus operator to take 3 arguments instead of 2.)
  - Associativity: e.g. `a+b+c` is always identical to `(a+b)+c`.
  - Precedence: which operator is done first?

    e.g. `a+b*c` is treated as `a+(b*c)`.

C++ Operators

- All C++ operators already have predefined meaning for the built-in types. It is impossible to change this meaning; you can only *overload* the operator to have a meaning for your *own* (user-defined) classes (such as `Vector` in the example above).
- Therefore, every operator you define must have *at least one* argument of a user-defined class type.
- As a global function, `operator+` has two arguments. When it is called in an expression such as `a + b`, this is equivalent to writing `operator+(a, b)`.

Member Operator Function

- Member functions are called using the “dot syntax" by specifying an object of, for example, type `Vector`.
  - The expression `a + b` is equivalent to `a.operator+(b)`.
  - Thus, when we define `operator+` as a member function of Vector, it has only one argument – the first argument is *implicitly* the object on which the member function is invoked.

```
class Vector {
    double _x, _y;
public:
    Vector(double x, double y) : _x(x), _y(y) { }
    double x() const { return _x; }
    double y() const { return _y; }
    Vector operator+ (const Vector& b) const
        { return Vector( _x + b._x, _y + b._y ); }
};
```

Member and Non-Member Operator Function

- Whenever the compiler sees an expression of the form `a+b`, it converts this to the two possible representations

```
operator+(a, b)
a.operator+(b)
```

  and verifies whether one of those two operator functions are defined.
- Note: It is an <u>error</u> to define both.

Example: Member or Non-Member Function?

- Let's define a multiplication operator to multiply a vector with a scalar. This should all work:

```
Vector a(1,0), b(2, 3);
Vector c = 2 * a;          // c == (2, 0)
a = c + b * 3;             // a == (8, 9)
```

- Can we define the multiplication operator as a member function of Vector?
- Remember that the compiler converts the expression `a*b` to `a.operator*(b)`. So the expression `2*a` is converted to `2.operator*(a)`!

Example: Member or Non-Member Function?

- This doesn't work! 2 is an object of type `int`, and we cannot define a new member function for this type.
- So our only choice is to define the multiplication operator as a global non-member function:

```
Vector operator* (double s, const Vector& a)
{
    return Vector(s * a.x(), s * a.y());
}
```

Example: Operator Function for Printing

- Very often you would like to provide a printing service for your user-defined classes, and the most natural way of doing that is to define the `<<` operator for your class.

```
ostream& operator<<(ostream& os, const Vector& a)
{
    os <<  ')' <<   a.x() <<   ',' <<   a.y() <<   ')';
    return os;
}
```

- `ostream` is the base class for all possible output streams.
- In particular, the standard output stream `cout` and the error output stream `cerr` are objects of classes derived from `ostream`.

Example: Operator Function for Printing

- Why does the operator return an output stream?
- Because we like to write expressions such as:

```
Vector a(1, 0);
cout << " a = " << a << "\n";
```

- The second line is equivalent to:

```
operator<<( operator<<( operator<<(cout, " a = "), a), "\n");
```

- This can only work if `operator<<` returns the output stream itself.
- Quiz: Could we have defined `operator<<` as a member function?

Operator: Member or Non-Member Functions?

- The operators: “=" (assignment), “[ ]" (indexing), “( )" (call) are required by C++ to be defined as class member functions.
- A member operator function has an implicit first argument of the class. => if the left operand of an operator must be an object of the class, it can be a member function.
- If the left operand of an operator must be an object of other classes, it must be a non-member function. e.g. `operator<<`
- To allow automatic conversion of types using the conversion constructor, for commutative operators like “+", “-", “*", it is usually preferred to be defined as non-member functions. e.g.

```
String x("dot"), y("com"), z;
z = x + y;
z = x + "com";
z = "dog" + y;
```

How to Differentiate Prefix and Postfix Operators?

```
class Vector {
    // …
public:
    Vector() : _x(0.0), _y(0.0) { }
    Vector(double x, double y) : _x(x), _y(y) { }
    Vector operator++() { ++ _x; ++ _y; return *this; }
    Vector operator++(int)
        { Vector temp( _x, _y); _x++; _y++; return temp; }
};

int main() {
    Vector a(1.2, 3.4), c, d;
    c = ++a;                    // a = (2.2, 4.4) and c = (2.2, 4.4)
    d = a++;                    // a = (3.2, 5.4) and d = (2.2, 4.4)
}
```