C++ Programming/Operators/Operator Overloading

< C++ Programming

Operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism(part of the OO nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments. Operator overloading is usually only syntactic sugar. It can easily be emulated using function calls.

Consider this operation:

```
add (a, multiply (b,c))
```

Using operator overloading permits a more concise way of writing it, like this:

```
a + b * c
```

(Assuming the * operator has higher precedence than +.)

Operator overloading can provide more than an aesthetic benefit, since the language allows operators to be invoked implicitly in some circumstances. Problems, and critics, to the use of operator overloading arise because it allows programmers to give operators completely free functionality, without an imposition of coherency that permits to consistently satisfy user/reader expectations. Usage of the `<<` operator is an example of this problem.

```
// The expression
a << 1;
```

Will return twice the value of `a` if `a` is an integer variable, but if `a` is an output stream instead this will write "1" to it. Because operator overloading allows the programmer to change the usual semantics of an operator, it is usually considered good practice to use operator overloading with care.

To overload an operator is to provide it with a new meaning for user-defined types. This is done in the same fashion as defining a function. The basic syntax follows (where @ represents a valid operator):

```
return_type operator@(argument_list)
{
    // ... definition
}
```

Contents

Not all operators may be overloaded, new operators cannot be created, and the precedence, associativity or arity of operators cannot be changed (for example ! cannot be overloaded as a binary operator). Most operators may be overloaded as either a member function or non-member function, some, however, must be defined as member functions. Operators should only be overloaded where their use would be natural and unambiguous, and they should perform as expected. For example, overloading + to add two complex numbers is a good use, whereas overloading * to push an object onto a vector would not be considered good style.

> **Note:**
> Operator overloading should only be utilized when the meaning of the overloaded operator's operation is unambiguous and practical for the underlying type and where it would offer a significant notational brevity over appropriately named function calls.

A simple Message Header

```
// sample of Operator Overloading

#include <string>

class PlMessageHeader
{
    std::string m_ThreadSender;
    std::string m_ThreadReceiver;

    //return true if the messages are equal, false otherwise
    inline bool operator == (const PlMessageHeader &b) const
    {
        return ( (b.m_ThreadSender==m_ThreadSender) &&
                 (b.m_ThreadReceiver==m_ThreadReceiver) );
    }

    //return true if the message is for name
    inline bool isFor (const std::string &name) const
    {
        return (m_ThreadReceiver==name);
    }

    //return true if the message is for name
    inline bool isFor (const char *name) const
    {
        return (m_ThreadReceiver==name);// since name type is std::string, it
becomes unsafe if name == NULL
    }
};
```

> **Note:**
> The use of the `inline` keyword in the example above is technically redundant, as

functions defined within a class definition like this are implicitly inline.

Operators as member functions [edit]

Aside from the operators which must be members, operators may be overloaded as member or non-member functions. The choice of whether or not to overload as a member is up to the programmer. Operators are generally overloaded as members when they:

1. change the left-hand operand, or
2. require direct access to the non-public parts of an object.

When an operator is defined as a member, the number of explicit parameters is reduced by one, as the calling object is implicitly supplied as an operand. Thus, binary operators take one explicit parameter and unary operators none. In the case of binary operators, the left hand operand is the calling object, and no type coercionwill be done upon it. This is in contrast to non-member operators, where the left hand operand may be coerced.

```
    // binary operator as member function
    //Vector2D Vector2D::operator+(const Vector2D& right)const [...]

    // binary operator as non-member function
    //Vector2D operator+(const Vector2D& left, const Vector2D& right)[...]

    // binary operator as non-member function with 2 arguments
    //friend Vector2D operator+(const Vector2D& left, const Vector2D& right) [...]

    // unary operator as member function
    //Vector2D Vector2D::operator-()const {...}

    // unary operator as non-member function[...]
    //Vector2D operator-(const Vector2D& vec) [...]
```

Overloadable operators [edit]

Arithmetic operators [edit]

- **+** (addition)
- **-** (subtraction)
- ***** (multiplication)
- **/** (division)
- **%** (modulus)

As binary operators, these involve two arguments which do not have to be the same type. These operators may be defined as member or non-member functions. An example illustrating overloading for the addition of a 2D mathematical vector type follows.

```
Vector2D Vector2D::operator+(const Vector2D& right)
{
    Vector2D result;
    result.set_x(x() + right.x());
```

```
    result.set_y(y() + right.y());
    return result;
}
```

It is good style to only overload these operators to perform their customary arithmetic operation. Because operator has been overloaded as member function, it can access private fields.

Bitwise operators [edit]

- **^** (XOR)
- **|** (OR)
- **&** (AND)
- **~** (complement)
- **<<** (shift left, insertion to stream)
- **>>** (shift right, extraction from stream)

All of the bitwise operators are binary, except complement, which is unary. It should be noted that these operators have a lower precedence than the arithmetic operators, so if ^ were to be overloaded for exponentiation, x ^ y + z may not work as intended. Of special mention are the shift operators, << and >>. These have been overloaded in the standard library for interaction with streams. When overloading these operators to work with streams the rules below should be followed:

1. overload << and >> as friends (so that it can access the private variables with the stream be passed in by references
2. (input/output modifies the stream, and copying is not allowed)
3. the operator should return a reference to the stream it receives (to allow chaining, cout << 3 << 4 << 5)

An example using a 2D vector

```
friend ostream& operator<<(ostream& out, const Vector2D& vec) // output
{
    out << "(" << vec.x() << ", " << vec.y() << ")";
    return out;
}

friend istream& operator>>(istream& in, Vector2D& vec) // input
{
    double x, y;
    // skip opening paranthesis
    in.ignore(1);

    // read x
    in >> x;
    vec.set_x(x);

    // skip delimiter
    in.ignore(2);

    // read y
    in >> y;
    vec.set_y(y);
```

```
    // skip closing paranthesis
    in.ignore(1);

    return in;
}
```

Assignment operator [edit]

The assignment operator, **=**, **must be a member function**, and is given default behavior for user-defined classes by the compiler, performing an assignment of every member using its assignment operator. This behavior is generally acceptable for simple classes which only contain variables. However, where a class contains references or pointers to outside resources, the assignment operator should be overloaded (as general rule, whenever a destructor and copy constructor are needed so is the assignment operator), otherwise, for example, two strings would share the same buffer and changing one would change the other.

In this case, an assignment operator should perform two duties:

1. clean up the old contents of the object
2. copy the resources of the other object

For classes which contain raw pointers, before doing the assignment, the assignment operator should check for self-assignment, which generally will not work (as when the old contents of the object are erased, they cannot be copied to refill the object). Self assignment is generally a sign of a coding error, and thus for classes without raw pointers, this check is often omitted, as while the action is wasteful of cpu cycles, it has no other effect on the code.

Example

```
class BuggyRawPointer { // example of super-common mistake
    T *m_ptr;
    public:
    BuggyRawPointer(T *ptr) : m_ptr(ptr) {}
    BuggyRawPointer& operator=(BuggyRawPointer const &rhs) {
        delete m_ptr; // free resource; // Problem here!
        m_ptr = 0;
        m_ptr = rhs.m_ptr;
        return *this;
    };
};

BuggyRawPointer x(new T);
x = x; // We might expect this to keep x the same. This sets x.m_ptr == 0. Oops!

// The above problem can be fixed like so:
class WithRawPointer2 {
    T *m_ptr;
    public:
    WithRawPointer2(T *ptr) : m_ptr(ptr) {}
    WithRawPointer2& operator=(WithRawPointer2 const &rhs) {
        if (this != &rhs) {
            delete m_ptr; // free resource;
            m_ptr = 0;
```

```
            m_ptr = rhs.m_ptr;
        }
        return *this;
    };
};

WithRawPointer2 x2(new T);
x2 = x2; // x2.m_ptr unchanged.
```

Another common use of overloading the assignment operator is to declare the overload in the private part of the class and not define it. Thus any code which attempts to do an assignment will fail on two accounts, first by referencing a private member function and second fail to link by not having a valid definition. This is done for classes where copying is to be prevented, and generally done with the addition of a privately declared copy constructor

Example

```
class DoNotCopyOrAssign {
    public:
        DoNotCopyOrAssign() {};
    private:
        DoNotCopyOrAssign(DoNotCopyOrAssign const&);
        DoNotCopyOrAssign &operator=(DoNotCopyOrAssign const &);
};

class MyClass : public DoNotCopyOrAssign {
    public:
        MyClass();
};

MyClass x, y;
x = y; // Fails to compile due to private assignment operator;
MyClass z(x); // Fails to compile due to private copy constructor.
```

Relational operators [edit]

- **==** (equality)
- **!=** (inequality)
- **>** (greater-than)
- **<** (less-than)
- **>=** (greater-than-or-equal-to)
- **<=** (less-than-or-equal-to)

All relational operators are binary, and should return either true or false. Generally, all six operators can be based off a comparison function, or each other, although this is never done automatically (e.g. overloading > will not automatically overload < to give the opposite). There are, however, some templates defined in the header <utility>; if this header is included, then it suffices to just overload operator== and operator<, and the other operators will be provided by the STL.

Logical operators [edit]

- **!** (NOT)
- **&&** (AND)
- **||** (OR)

The logical operators AND are used when evaluating two expressions to obtain a single relational result.The operator corresponds to the boolean logical operation AND,which yields true if operands are true,and false otherwise.The following panel shows the result of operator evaluating the expression.

The ! operator is unary, && and || are binary. It should be noted that in normal use, && and || have "short-circuit" behavior, where the right operand may not be evaluated, depending on the left operand. When overloaded, these operators get function call precedence, and this short circuit behavior is lost. It is best to leave these operators alone.

Example

```
bool Function1();
bool Function2();

Function1() && Function2();
```

If the result of Function1() is false, then Function2() is not called.

```
MyBool Function3();
MyBool Function4();

bool operator&&(MyBool const &, MyBool const &);

Function3() && Function4()
```

Both Function3() and Function4() will be called no matter what the result of the call is to Function3() This is a waste of CPU processing, and worse, it could have surprising unintended consequences compared to the expected "short-circuit" behavior of the default operators. Consider:

```
extern MyObject * ObjectPointer;
bool Function1() { return ObjectPointer != null; }
bool Function2() { return ObjectPointer->MyMethod(); }
MyBool Function3() { return ObjectPointer != null; }
MyBool Function4() { return ObjectPointer->MyMethod(); }

bool operator&&(MyBool const &, MyBool const &);

Function1() && Function2(); // Does not execute Function2() when pointer is null
Function3() && Function4(); // Executes Function4() when pointer is null
```

Compound assignment operators [edit]

- **+=** (addition-assignment)
- **-=** (subtraction-assignment)
- ***=** (multiplication-assignment)

- **/=** (division-assignment)
- **%=** (modulus-assignment)
- **&=** (AND-assignment)
- **|=** (OR-assignment)
- **^=** (XOR-assignment)
- **<<=** (shift-left-assignment)
- **>>=** (shift-right-assignment)

Compound assignment operators should be overloaded as member functions, as they change the left-hand operand. Like all other operators (except basic assignment), compound assignment operators must be explicitly defined, they will not be automatically (e.g. overloading = and + will not automatically overload +=). A compound assignment operator should work as expected: A @= B should be equivalent to A = A @ B. An example of += for a two-dimensional mathematical vector type follows.

```
Vector2D& Vector2D::operator+=(const Vector2D& right)
{
    this->x += right.x;
    this->y += right.y;
    return *this;
}
```

Increment and decrement operators [edit]

- **++** (increment)
- **--** (decrement)

Increment and decrement have two forms, prefix (++i) and postfix (i++). To differentiate, the postfix version takes a dummy integer. Increment and decrement operators are most often member functions, as they generally need access to the private member data in the class. The prefix version in general should return a reference to the changed object. The postfix version should just return a copy of the original value. In a perfect world, A += 1, A = A + 1, A++, ++A should all leave A with the same value.

Example

```
SomeValue& SomeValue::operator++() // prefix
{
    ++data;
    return *this;
}

SomeValue SomeValue::operator++(int unused) // postfix
{
    SomeValue result = *this;
    ++data;
    return result;
}
```

Often one operator is defined in terms of the other for ease in maintenance, especially if the function call is complex.

```
SomeValue SomeValue::operator++(int unused) // postfix
{
    SomeValue result = *this;
    ++(*this); // call SomeValue::operator++()
    return result;
}
```

Subscript operator [edit]

The subscript operator, **[ ]**, is a binary operator which **must be a member function** (hence it takes only one explicit parameter, the index). The subscript operator is not limited to taking an integral index. For instance, the index for the subscript operator for the std::map template is the same as the type of the key, so it may be a string etc. The subscript operator is generally overloaded twice; as a non-constant function (for when elements are altered), and as a constant function (for when elements are only accessed).

Function call operator [edit]

The function call operator, **( )**, is generally overloaded to create objects which behave like functions, or for classes that have a primary operation. The function call operator must be a member function, but has no other restrictions - it may be overloaded with any number of parameters of any type, and may return any type. A class may also have several definitions for the function call operator.

Address of, Reference, and Pointer operators [edit]

These three operators, operator&(), operator*() and operator->() can be overloaded. In general these operators are only overloaded for smart pointers, or classes which attempt to mimic the behavior of a raw pointer. The pointer operator, operator->() has the additional requirement that the result of the call to that operator, must return a pointer, or a class with an overloaded operator->(). In general A == *&A should be true.

Note that overloading operator& invokes undefined behavior:

> ISO/IEC 14882:2003, Section 5.3.1
> The address of an object of incomplete type can be taken, but if the complete type of that object is a class type that declares operator&() as a member function, then the behavior is undefined (and no diagnostic is required).

Example

```
class T {
    public:
        const memberFunction() const;
};

// forward declaration
class DullSmartReference;

class DullSmartPointer {
    private:
        T *m_ptr;
    public:
        DullSmartPointer(T *rhs) : m_ptr(rhs) {};
        DullSmartReference operator*() const {
```

```
            return DullSmartReference(*m_ptr);
        }
        T *operator->() const {
            return m_ptr;
        }
};

class DullSmartReference {
    private:
        T *m_ptr;
    public:
        DullSmartReference (T &rhs) : m_ptr(&rhs) {}
        DullSmartPointer operator&() const {
            return DullSmartPointer(m_ptr);
        }
        // conversion operator
        operator T() { return *m_ptr; }
};

DullSmartPointer dsp(new T);
dsp->memberFunction(); // calls T::memberFunction

T t;
DullSmartReference dsr(t);
dsp = &dsr;
t = dsr; // calls the conversion operator
```

These are extremely simplified examples designed to show how the operators can be overloaded and not the full details of a SmartPointer or SmartReference class. In general you won't want to overload all three of these operators in the same class.

Comma operator [edit]

The comma operator,() **,** can be overloaded. The language comma operator has left to right precedence, the operator,() has function call precedence, so be aware that overloading the comma operator has many pitfalls.

Example

```
MyClass operator,(MyClass const &, MyClass const &);

MyClass Function1();
MyClass Function2();

MyClass x = Function1(), Function2();
```

For non overloaded comma operator, the order of execution will be Function1(), Function2(); With the overloaded comma operator, the compiler can call either Function1(), or Function2() first.

Member Reference operators [edit]

The two member access operators, operator->() and operator->*() can be overloaded. The most common use of overloading these operators is with defining expression template classes, which is not a common programming

technique. Clearly by overloading these operators you can create some very unmaintainable code so overload these operators only with great care.

When the -> operator is applied to a pointer value of type (T *), the language dereferences the pointer and applies the . member access operator (so x->m is equivalent to (*x).m). However, when the -> operator is applied to a class instance, it is called as a unary postfix operator; it is expected to return a value to which the -> operator can again be applied. Typically, this will be a value of type (T *), as in the example under Address of, Reference, and Pointer operators above, but can also be a class instance with operator->() defined; the language will call operator->() as many times as necessary until it arrives at a value of type (T *).

Memory management operators [edit]

- **new** (allocate memory for object)
- **new[ ]** (allocate memory for array)
- **delete** (deallocate memory for object)
- **delete[ ]** (deallocate memory for array)

The memory management operators can be overloaded to customize allocation and deallocation (e.g. to insert pertinent memory headers). They should behave as expected, **new** should return a pointer to a newly allocated object on the heap, **delete** should deallocate memory, ignoring a NULL argument. To overload **new**, several rules must be followed:

- **new** must be a member function
- the return type must be void*
- the first explicit parameter must be a size_t value

To overload **delete** there are also conditions:

- **delete** must be a member function (and cannot be virtual)
- the return type must be void
- there are only two forms available for the parameter list, and only one of the forms may appear in a class:
  - void*
  - void*, size_t

Conversion operators [edit]

Conversion operators enable objects of a class to be either implicitly (coercion) or explicitly (casting) converted to another type. Conversion operators must be member functions, and should not change the object which is being converted, so should be flagged as constant functions. The basic syntax of a conversion operator declaration, and declaration for an int-conversion operator follows.

```
operator ''type''() const; // const is not necessary, but is good style
operator int() const;
```

Notice that the function is declared without a return-type, which can easily be inferred from the type of conversion. Including the return type in the function header for a conversion operator is a syntax error.

```
double operator double() const; // error - return type included
```

Operators which cannot be overloaded [edit]

- **?:** (conditional)
- **.** (member selection)
- **.*** (member selection with pointer-to-member)
- **::** (scope resolution)
- **`sizeof`** (object size information)
- **typeid** (object type information)

To understand the reasons why the language doesn't permit these operators to be overloaded, read "Why can't I overload dot, ::, **`sizeof`**, etc.?" at the Bjarne Stroustrup's C++ Style and Technique FAQ (http://www.stroustrup.com/bs_faq2.html#overload-dot).