Operator Overloading

Topics Covered

- Why overload operators?
- Syntax
- Classification and specifications.
- Overload resolution
- Examples

Why overload operators?

Think of operators as shorthand notions for (common) operations. For example, when we say

*x - y * z*

we mean

multiply y by z and subtract the result from x

or in a pure procedural programming language without built-in operators

subtract(x, multiply(y, z))

Operators make the presentation of commonly-used concepts and operations terse, readable, and concise. C++ has a set of built-in operators that work on built-in types. These operators have very well-defined behavior and relationships among each other (such as precedence).

C++ has this philosophy of having user-defined types behave just like built-in types. Therefore, if we can say

```
int x(2), y(3), z;
z = x + y;
```

for `int`, then we ought to be able to say

```
complex x(2, 3), y(1, 2), z;
z = x + y;
```

for the user-defined type `complex`. That is to say, we need to define the + operator that works with `complex` objects. Analogous to function overloading, in which we allow functions with the same name but different signatures to coexist in a program, we now

need to overload operators. (In fact, you can think of operator overloading as a special form of function overloading.)

To conclude, we need operator overloading for a) syntactic and semantic orthogonality among built-in and user-defined objects, and b) natural syntax.

Syntax

To overload a built-in operator **@**, we need to define a function with the name *operator***@**. Operator functions can be either member functions or nonmember functions. For example, to overload the + operator for complex objects, we can either define a nonmember function

```
complex operator+(complex, complex);
```

or a member function

```
complex complex::operator+(complex);
```

A use of the operator **@** is only a shorthand for an explicit call of the operator function. For example, a statement

```
complex a = b + c; // b and c are complex objects
```

will be translated to either

```
complex a = operator+(b, c)
```

or

```
complex a = b.operator+(c)
```

depending on which one is defined, and if both are defined, overload resolution determines which. (In this case, it will be ambiguous since both are exact matches.)

Both unary and binary operators can be overloaded (but not the ternary operator `?:`).

- For any binary operator *@*, *a@b* can be interpreted as *`a.operator@(b)`* or *`operator@(a, b)`*.
- For any prefix unary operator *@*, *@a* can be interpreted as either *`a.operator@()`* or *`operator@(a)`*.
- For any postfix unary operator *@*, *a@* can be interpreted as either *`a.operator@(int)`* or *`operator@(a, int)`*.

The `int` parameter in the postfix cases is used to differentiate it from its prefix counterpart. The value is not used by the function. To summarize,

Operator Type	**Nonmember**	**Member**

	Syntax	Example	Syntax	Example
Binary	`op@(a,b)`	`op+(a,b);` `c = a + b;`	`a.op@(b)`	`cplx::op+(a);` `c = a + b;`
Prefix Unary	`op@(a)`	`op++(a);` `++a;`	`a.op@()`	`cplx::op++();` `++a;`
Postfix Unary	`op@(a,int)`	`op++(a, int);` `a++;`	`a.op@(a,int)`	`cplx::op++(int);` `a++;`

Classifications and Specifications

Operators that can be overloaded can be classified into several categories, depending on their functionality and common implementation structures:

Category	Example
Arithmetic, bitwise, relational and logical operators	+, ^, <=, &&
Increment and decrement operators	++, --
The assignment operator	=
Other assignment operators	+=, ^=, <<=
Dereferencing and subscripting operators	->, *, ->*, []
"Function" operator	()
I/O operators	>>, <<
Memory management operators	new, new[], delete, delete[]
Others	The comma operator

The following three operators cannot be overloaded:
:: (scope resolution),
. (member selection), and
.* (member selection through pointer to member).

We will discuss a couple of the categories listed, which will be summarized later using a table. Students are encouraged to figure out the details of those commonly-used categories.

Increment and decrement operators

To simplify discussion, we will only consider the increment operator.
It turns out that the prefix increment operator is the easiest to implement. Usually, it is implemented as a member function with the following structure:

```
class T {
    const T& operator++() {
      // self-increment
      return *this;
    }
};
```

The postfix increment operator usually operate in three steps:
1. Make a temporary copy of this object.
2. Increment this object.
3. Return the temporary object by value.

To ensure consistency between prefix and postfix increment, the second step above is usually implemented as a call to the prefix increment operator function:

```
const T operator++(int) {
   T temp = *this;
  ++(*this); // calls (*this).operator++()
  return temp;
}
```

Notice that the postfix increment operator function involves two copies, one for copy-constructing the temporary object and the other for the return value. Therefore, generally speaking, postfix increment is more expensive than prefix increment. That's why you see code like this:

```
for(vector::iterator vi = v.begin(); vi != v.end(); ++vi)
{ . . . }
```

Other assignment operators

These operators have similar implementation as the increment/decrement operators. For example,

```
const T& operator*=(const T& t) {
      // self-multiply
      return *this;
}
```

Arithmetic operators

It turns out that arithmetic operators can be implemented using the assignment operators we just discussed:

```
const T operator*(const T& t) {
      *this *= t; // calls (*this).operator*=(t)
      return *this; // have to return a copy
}
```

However, the arithmetic operators are usually implemented as nonmember functions and we leave their implementation as an exercise.

I/O operators

The input and output operators are implemented as nonmember functions. It cannot be implemented as a member function since the streams (cin, cout, and etc) are on the left hand side of the operator.

An alternative to defining the >> operator is to have a constructor that takes an istream.

An alternative to declaring the << operator a friend to the class is to implement a (public) print() member function and make operator<<() call it.

Category	Example	Specification	
		Member or Nonmember	**Common Prototype**
Arithmetic, bitwise, and relational operators	+, ^, <=	NM	`friend const T op@(const T&, const T&);`
Logical operators	&&, ||	M/NM	Don't overload them

Increment and decrement operators	++, --	M	`T::op@(); // prefix`
			`T::op@(int); // postfix`

The assignment operator	=	**M**	`const T& T::op=(const T&);`

Other assignment operators	+=, ^=, <<=	M	`const T& T::op@(const T&); // or`
			`const T& T::op@(const T2&);`

Dereferencing and subscripting operators	->, []	**M**	`T2 T::op->();`
			`T3& T::op[](int i);`
			`const T3& T::op[](int i) const;`

"Function" operator	()	**M**	`T2 T::op()(. . .);`

I/O operators	>>, <<	NM	`T(istream&); // or`
			`istream& op>>(istream&, T&);`
			`ostream& op<<(ostream&, const T&);`

Memory management operators	new, new[], delete, delete[]	M/NM	Don't overload them
Others	The comma operator	M/NM	Don't overload it.

M: usually implemented as member functions
NM: usually implemented as nonmember functions
M: have to be member functions (enforced by the language)
@: a wildcard for operators in a certain category

Overload resolution

Some important facts about overloading in C++:

1. Return types are not considered in overload resolution.
The reason is to keep resolution for an individual operator or function call context-independent.

2. Functions declared in different non-namespace scopes do not overload.
For example,

```
void f(int);

void g()
{
    void f(double);
    f(1);// calls f(double)
}
```

This also implies that overload resolution is not applied across different classes, since a class is a scope. For example,

```
class A {
public:
    int f(int a) { return a; }
};

class B : A {
public:
    int f() { return 1; }
    // B::f() shadows A::f(int)
};

int main()
{
    B b;
    b.f(1);   // ERROR: no B::f(int) defined;
                // only B::f() is in scope
}
```

3. Overloading works across namespaces.

4. Resolution for multiple arguments follows common sense.
If no single function is a ***better*** match, the call is ambiguous. The definition of a better match is described below.

5. Overload resolution happens before access control checks.
For example,

```
class A {
public:
    void f(double d) {}
};

class B : A {
```

```
private:
    void f(int i) {}     // still shadows A::f(double)
                         // though it's private
};

int main()
{
    B b;
    b.f(1.5); // ERROR: tried to access B::f(int),
              // which is private
}
```

Another way of putting it is that access control does not affect overload resolution. In fact, we can think of function invocation in C++ as being composed of four steps:

1. Identify scopes in which the functions to be searched for.
2. Apply overload resolution to find a best match (or avoid applying overload resolution where it is not allowed).
3. Access control check.
4. Invoke the function.

Now let us look at overload resolution criteria. When having a function call, these criteria are tried in order until a match is found. If two functions end up matching the same criterion, the call is rejected as ambiguous.

1. Exact match (array to pointer, T to const T)
2. Match using promotions (bool to int, char to int, short to int, float to double)
3. Match using standard conversions (int to double, float to int, T * to void *, int to unsigned int)
4. Match using user-defined conversions
5. Match using ellipsis

Operator Overloading Example: Time

class declaration

```
class istream;
class ostream;

class Time {

    friend istream &operator>>(istream&, Time&);
    friend ostream &operator<<(ostream&, const Time&);
    friend const Time operator+(const Time&, const Time&);
    friend const Time operator-(const Time&, const Time&);
    friend bool operator==(const Time&, const Time&);
    friend bool operator!=(const Time&, const Time&);
    friend bool operator>(const Time&, const Time&);
    friend bool operator>=(const Time&, const Time&);

public:

    // Constructors
    Time();
    Time(int, int);
    Time(const Time &);

    /* Increment operators */
    const Time& operator+=(const Time&);
    const Time& operator-=(const Time&);

    /* The assignment operator */
    const Time& operator=(const Time &);

private:
    enum {
      MAX_HOUR = 24, MAX_MINUTE = 60
    };

    int hour;
    int minute;
    void setFields(int, int);
};
```

class definition

```
#include <stream.h>
#include <ctype.h>
#include "time.h"

/* helper to convert a 2-char string of a number into the actual number */
inline int convert_time(int c1, int c2)
{
   if (isdigit(c1) && isdigit(c2)) {
      return 10 * (c1 - '0') + (c2 - '0');
   } else {
      return 0;
   }
}

/* reads in military-style time */
istream &operator>>(istream &is, Time &time)
{
   /* this is to prevent buffer overflow */
   const static int buflen = 256;
   char buffer[buflen];
   int oldlen = is.width(buflen);

   /* the actual reading and decoding */
   is >> buffer;
   time.hour = convert_time(buffer[0], buffer[1]);
   time.minute = convert_time(buffer[2], buffer[3]);

   return is;
}

/* writes out in military-style time */
ostream &operator<<(ostream &os, const Time &time)
{
   /* width() and fill() are used to output something like "02" */
   os.width(2);
   os.fill('0');
   os << time.hour;
   os.width(2);
   os.fill('0');
   os << time.minute;

   return os;
}

const Time operator+(const Time &t1, const Time &t2)
{
   Time t(t1);
   t += t2;
   return t;
}
```

```
const Time operator-(const Time &t1, const Time &t2)
{
   Time t(t1);
   t -= t2;
   return t;
}

bool operator==(const Time &t1, const Time &t2)
{
   return ((t1.hour == t2.hour) && (t1.minute == t2.minute));
}

bool operator!=(const Time &t1, const Time &t2)
{
   return !(t1 == t2);
}

bool operator>(const Time &t1, const Time &t2)
{
   if (t1.hour > t2.hour) {
      return true;
   } else if (t1.hour == t2.hour && t1.minute > t2.minute) {
      return true;
   }
   return false;
}

bool operator>=(const Time &t1, const Time &t2)
{
   return (t1 > t2) || (t1 == t2);
}

Time::Time()
{
   setFields(0, 0);
}

Time::Time(int hr, int min)
{
   setFields(hr, min);
}

Time::Time(const Time &t)
{
   setFields(t.hour, t.minute);
}

inline void add_time(int &val, int incr, int base, int &carry)
{
   val += incr;
   if (val > base) {
      carry++; val -= base;
   }
}
```

```
const Time& Time::operator+=(const Time &t)
{
   int dummy;

   add_time(minute, t.minute, MAX_MINUTE, hour);
   add_time(hour, t.hour, MAX_HOUR, dummy);

   return *this;
}

inline void subtract_time(int &val, int decr, int base, int &carry)
{
   val -= decr;
   if (val < 0) {
      carry--;
      val += base;
   }
}

const Time& Time::operator-=(const Time &t)
{
   int dummy;

   subtract_time(minute, t.minute, MAX_MINUTE, hour);
   subtract_time(hour, t.hour, MAX_HOUR, dummy);

   return *this;
}

const Time& Time::operator=(const Time &t)
{
   setFields(t.hour, t.minute);
   return *this;
}

void Time::setFields(int h, int m)
{
   hour = h;
   minute = m;
}
```