65. Operator Overloading

Objectives

- Overloading Operators

Review

The name of some methods are awkward to use.

For certain data (objects) we are more used to operators rather than method names.

Consider the following:

```
// Fraction.h

class Fraction
{
public:
    Fraction(int n=0, int d=1)
        : n_(n), d_(d)
    {}
    Fraction plus(const Fraction &) const;
    void print() const;
private:
    int n_, d_; // numerator,denominator
};
```

```
// Fraction.cpp

#include "Fraction.h"

Fraction Fraction::plus(const Fraction & a) const
{
    return Fraction(n_ * a.d_ + d_ * a.n_, d_ * a.d_);
}
```

```
#include "Fraction.h"

int main()
{
    Fraction a(1,3), b(1,4); // a=1/3 and b=1/4
    Fraction c = a.plus(b); // basically, c = a + b

    return 0;
}
```

So, if we want `c = a + b * c - d`, it would have to look like

```
    c = (a.plus(b.mult(c))).subtract(d)
```

YUCK!!!

Operator as a Method

```
// Fraction.h

class Fraction
{
public:
    Fraction(int n=0, int d=1)
        : n_(n), d_(d)
    {}
    Fraction operator+(const Fraction &) const;
private:
    int n_, d_; // numerator,denominator
};
```

```
// Fraction.cpp

#include "Fraction.h"
Fraction Fraction::operator+(const Fraction & b) const
{
    return Fraction(n_ * b.d_ + d_ * b.n_, d_ * b.d_);
}
```

Re-defining `+`, so that we have `+` for `int`, `double`, and `Fraction`.

```
#include "Fraction.h"

int main()
{
    Fraction a(1,3), b(1,4);
    Fraction c = a + b;
    Fraction d = a.operator+(b);

    return 0;
}
```

Same as **Fraction c = a.operator+(b)**

Most programming languages allow you to define operators (with their own syntax). There are languages that do not have operator overloading. Example: Java.

```
// Fraction.cpp

#include "Fraction.h"

Fraction Fraction::operator*(const Fraction & b) const
{
    return Fraction(n_ * b.n_, d_ * b.d_);
}
```

Test it:

```
#include "Fraction.h"

int main()
```

```
{
    Fraction a(1, 3), b(1, 4);
    Fraction c = a * b;

    return 0;
}
```

Operator in/outside the class

In general suppose `a,b` are objects of class `C`.

C++ will interpret `a + b` as either

    a + b is the same as a.operator+(b)

or

    a + b is the same as operator+(a, b)

For `a.operator+(b)`, C++ will look for `operator+` in the class `C` (i.e., the class of `a`) that accepts `b` (object of class `C`)

For `operator+(a, b)`, C++ will look for an `operator+` not in any class that accepts `a,b` of class `C`

For instance, for `Fraction` class

```
class Fraction
{
public:
    ...
    Fraction operator+(const Fraction & b) const {...}
};
```

will let you execute `a + b` where `a,b` are `Fraction` objects as `a.operator+(b)`

... or ...

... if you prefer an operator **outside** the class:

```
class Fraction
{
public:
    ...
};

Fraction operator+(const Fraction & a, const Fraction & b)
{
    ...
}
```

will let you execute `a + b`, where `a,b` are `Fraction` objects, as `operator+(a, b)`.

It is preferable to have an operator in a class if possible. You **cannot** have both:

```
class Fraction
{
public:
    Fraction operator+(const Fraction & b) const
    {
      ...
    }
```

```
};

// DANGER! can't have both above and the following together
Fraction operator+(const Fraction & a, const Fraction & b)
{
    ...
}
```

Why? Because C++ does not know which `operator+` to use if you give C++ both options. So you should only have either:

```
  Fraction Fraction::operator+(const Fraction &) const
```

or

```
  Fraction operator+(const Fraction &,
                     const Fraction &)
```

but not both.

You can overload the following:

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

Warning: You **cannot** define your own functions for

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

Precedence rules

Usual precedence rules apply automatically. So if you have `operator+` and `operator*` for class C, then

```
a = b + c * d;
```

is the same as

```
a = b + (c * d);
```

i.e.,

```
a = b.operator+(c.operator*(d));
```

Reminder: Review precedence rules from CISS240.

Binary & unary operators

Some operators are binary (example `*`), some are unary (example `!`), and some are both (example `-`)

```
C a = b - c;
C d = -b;
```

Binary

Unary

The function/method called is based on signature:

```
class C
{
    ...
    C operator-();
    C operator-(C);
    ...
};
```

In general for operator `<op>`

`a <op> b`	is the same as	`a.operator<op>(b)` or `operator<op>(a,b)`
`<op>a`	is the same as	`a.operator<op>()` or `operator<op>(a)`
`a<op>`	is the same as	`a.operator<op>(int)` or `operator<op>(a,int)`

Here's how C++ looks for operators to execute for `+` (unary and binary), `+=`, `++` (pre and post)

`+a`	is the same as	`a.operator+()`	or	`operator+(a)`
`a += b`	is the same as	`a.operator+=(b)`	or	`operator+=(a,b)`
`a + b`	is the same as	`a.operator+(b)`	or	`operator+(a,b)`
`++a`	is the same as	`a.operator++()`	or	`operator++(a)`
`a++`	is the same as	`a.operator++(0)`	or	`operator++(a,0)`

Pre and post operators

```
class Int
{
public:
    Int(int x=0)
        : x_(x)
    {}

    void operator++()
    {
        std::cout << "pre\n"; ++x_;
    }

    void operator++(int a)
    {
        std::cout << "post " << a << "\n"; x_++;
    }

    int get() { return x_; }

private:
    int x_;
};
```

```
int main()
{
    C x(4);
    ++x;
    std::cout << x.get() << "\n";
    x++;
    std::cout << x.get() << "\n";

    return 0;
}
```

C++ translates `++x` and `x++` as:

`++x` is the same as `x.operator++()`
`x++` is the same as `x.operator++(0)`

i.e., for post operators, C++ calls operator with a value of 0.

Exercise: Overload the operators needed for the following code to run:

```
int main()
{
    Int i(5), j(9), k(0);
    j = i++; j = ++i;
    i = j--; i = --j;

    return 0;
}
```

Restriction on arguments

Whether the operator is unary or binary is fixed. Just try to recall their meanings from CISS240. For example, `%` (remainder operator) is a **binary** operator: `3 % 5` is meaningful; `5%` has no meaning. So you cannot overload `operator%` by declaring and defining a unary `%`:

```
C C::operator%()
```

`operator%` **must** be

```
C C::operator%(C)
```

or

```
C C::operator%(const C &)
```

etc.

Other correct declarations:

```
C* operator&();
```

See operators `[]`, `()`, `->`, etc. later.

Hiding pre-defined operators

Here are the pre-defined class operators:

```
=     &     ,
```

If you do not want these pre-defined operators to be callable, just declare them as private. You need **not** define them:

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

Operators <op> and <op>=

Defining `+` does not give you `+=` automatically.

Here's some advice: define + in terms of `+=`, `-` in terms of `-=`, etc. For instance:

```
class Int
{
public:
    ...
    Int & operator+=(const Int & c)
    {
        x_ += c.x_;

        return (*this);
    }

    Int operator+(Int c) const
    {
        return (Int(*this) += c);
    }
    ...

private:
    int x_;
};
```

A **nonmember function** is just a function not in a class (i.e., not a method). If the function works with objects from a particular class, then the function is included with the class. You should

1. Put the function prototype in the header file (`*.h`)
2. Define the function in the implementation file (`*.cpp`)
3. Note that the function prototype is outside the class declaration

Since the function prototype is out the class, it does **not** have access to the private members of the class. You can overwrite this for efficiency (later). Note that here `+=` is more efficient than `+`, same for integer `+` and `+=`. Also note that we want to do

```
(a += b) += b;
c = (a += b); // i.e., a+=b; c=a;
```

That's why `operator+=` returns a reference to `*this`.

```
class Int
{
public:
    ...
    Int & operator+=(const C & c)
    {
        x += c.x;
```

```
        return (*this);
    }
    ...
};
```

And `operator+` can be written like this:

```
class Int
{
public:
    ...
    Int operator+(const Int c) const
    {
        return (Int(*this) += c);
    }
    ...
};
```

Commutation

At this point for `Int` objects `a` and `b`, `a + b` and `b + a` make sense.

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

C++ will look for `a.operator+(b)` or `operator+(a,b)`. BUT ... what if you want to do `a + 1` or `1 + a`? We'll talk about `1 + a` first ...

If `a` is an `Int` object,

`1 + a`	is	`1.operator+(a)`
		or
		`operator+(1, a)`

`1` is not an object so `1.operator+(a)` does not make sense.

Therefore, **you will need to define this function outside the `Int` class:**

```
Int operator+(int c, const Int & a);
```

```
// Int.h

class Int
{
public:
    ...
    Int & operator+=(const Int & c);
    Int operator+(Int c) const
    {
        return (Int(*this) += c);
    }
    ...
private:
    int x_;
};

Int operator+(int, const Int &);
```

```
// Int.cpp

...
Int operator+(int a, const int & b)
{
    return Int(a) + b;
}
...
```

You have the same issue if you want to do this:

```
Int a(6);
Int b = 5 * a;
```

is the same as

```
Int b = operator*(5,a);
```

You will need to define the function outside the `Int` class:

```
Int operator*(int c, const Int & a);
```

Now back to the other case: `a + 1`. If you have a class `C`, then

```
C c = 1;
```

is the same as

```
C c(1);
```

If `C(1)` is a valid constructor call. In this case, the constructor was called **implicitly**. That means that for the `Int` class,

```
Int i(42), j(0);
j = i + 1; // becomes j = i + Int(1);
```

So you do **not** need `operator+(const Int&, int);`

Exercise: How many constructor calls are there? Where?

```
#include <iostream>

class Int
{
public:
    Int(int x)
        : x_(x)
    {
        std::cout << "Int\n";
    }

    Int operator+(const Int & i)
    {
        return Int(x_ + i.x_);
    }

private:
    int x_;
};
```

```
#include "Int.h"

int main()
{
    Int i(42), j(0);
    j = i + 1;

    return 0;
}
```

operator[]

If `c` is a `C` object, C++ translates `c[n]` like this:

`c[n]` is the same as `c.operator[](n)`

`operator[]` must be a member function.

```
class IntDynArr
{
public:
    IntDynArr(int s)
        : size_(s), p_(new int[s])
    {}

    ~IntDynArr() { delete [] p_; }

    int operator[](int n) { return p_[n]; }

private:
    int size_;
    int * p_;
};
```

```
int main()
{
    IntDynArr c(100);
    std::cout << c[5] << std::endl;

    return 0;
}
```

But what about this?

```
int main()
{
    IntDynArr c(100);
    c[5] = 23; // ERROR!
    std::cout << c[5] << std::endl;

    return 0;
}
```

```
class IntDynArr
{
public:
    IntDynArr(int s)
        : size_(s), p_(new int[s])
    {}

    ~IntDynArr() { delete [] p_; }
    ...
```

```
    int & operator[](int n)
    {
        return p_[n];
    }
    ...

private:
    int size_;
    int * p_;
};
```

Why a reference?

Of course you can includes checks like `size >= 0` for constructor, and argument of `[]` is within bounds:

```
...
int & operator[](int n)
{
    if (n >= 0 && n < size)
        return p_[n];
}
...
```

But what if `n < 0` or `n >= size`?

Later, we will handle such errors with **exceptions**.

Now include

```
...
int & operator[](int n) const
{
    if (n >= 0 && n < size)
        return p_[n];
}
...
```

What is the point of this?

operator()

Suppose you have class `C` and `c` is an object of class `C`. C++ translates `c(a)` like this:

`c(a)` is the same as `c.operator()(a)`

The point: makes object look like a function

```
class Square
{
public:
    int operator()(int x)
    {
        return x * x;
    }
};

int main()
{
    Square square;
    std::cout << square(5);

    return 0;
}
```

Why not just use functions? Make object remember previous computation(s).

```
class Square
{
public:
    Square()
        : last_square_(-1)
    {}

    int operator()(int x)
    {
        if (last_square_ != -1 && x == last_x_)
        {
            return last_square_;
        }
        else
        {
            last_x_ = x;
            last_square_ = x * x;

            return last_square_;
        }
    }

private:
    int last_x_, last_square_;
};
```

This is helpful if you frequently compute the last square and the computation of `x * x` costs more than the computation of the boolean values in the code.

You can also remember not just the history of one computation, but use an array.

Some computations are so intensive that some form of “history” of computations is absolutely necessary.

Here's the Fibonacci sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

You start off with 0, 1. Subsequent terms are the sum of the previous two. Here's a simple Fibonacci function:

```
int fib(int n)
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

Try to execute `fib` for high values of `n`:

```
for (int n = 0; n < 100; n++)
{
     Std::cout << n << ',' << fib(n) << '\n';
}
```

Why is it so slow? `fib(90)` calls `fib(89)` and `fib(88)` and adds their return values. `fib(89)` calls `fib(88)` and `fib(87)` and adds their return values. Etc. There are many re-computations!!!

Let's remedy this by making a class:

```
class Fibonacci
{
public:
    Fibonacci()
    {
        lookup_[0] = 0;
        lookup_[1] = 1;
        for (int i = 2; i < 100; i++)
        {
            lookup_[i] = -1;
        }
    }

private:
    int lookup_[100];
}
```

In `lookup_`, if `lookup_[i]` is `-1`, it means that the `i`th value of Fibonacci has not been stored in `lookup_[i]` yet.

Now add `operator()`:

```
int Fibonacci::operator(int n)
{
    if (n < 100) // within lookup_'s range
    {
        // not stored in lookup_ yet
        if (lookup_[n] == -1)
            lookup_[n] = (*this)(n - 1) + (*this)(n - 2);

        return lookup_[n];
    }
    else // outside lookup_'s range
    {
        return (*this)(n - 1) + (*this)(n – 2);
    }
}
```

Now test this version of the Fibonacci computation:

```
Fibonacci fib;

for (int n = 0; n < 100; n++)
{
    std::cout << n << ',' << fib(n) << '\n';
}
```

Another reason to use objects that look like function: the computation involves two (or more) values but one of them does not change. Store that unchanging value in the object.

```
class AddBy
{
public:
    AddBy(int x) : x_(x) {}
    int operator()(int y) { return x_ + y; }
private:
    int x_;
};
```

Test it with:

```
AddBy f(5);
std::cout << f(3) << ',' << f(11) << '\n';
AddBy g(10);
std::cout << g(3) << ',' << g(11) << '\n';
```

You can also provide methods to modify the `x_` in the class.

operator=

The default = operator will perform member-wise copy. For example: If you have the following class

```
class vec2i
{
private:
    int x_, y_;
};
```

and you have objects `p, q` of `vec2i`. Then `p=q` will have the same effect as `p.x_ = q.x_`, `p.y_ = q.y_`.

WARNING: You might actually not want `operator=` to behave as above. Why? Can you think of some examples where the default `operator=` is bad?

You can overwrite the default = by creating your own `operator=` . You definitely want to do this if your object holds on to some form of system resource. Example: There is a pointer in your object.

The default `operator=` is called the **default copy operator** (not to be confused with the copy constructor which is a constructor that has an object as parameter). Note that you can do this:

```
SomeClass a, b, c, d;
a = b = c = d;
```

This is because the second statement is

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

i.e.,

```
a = (b = (c.operator=(d)));
```

This implies that `c.operator=(d)` has a return value. This will be mentioned later.

`operator=` Returning Value

What do we want to achieve with

```
a = b = c = d;
```

We want to set `c` to `d`, then `b` to `c`. Therefore in

```
a = (b = (c.operator=(d)));
```

we want (basically)

```
c = d;
a = b = c;
```

Therefore for a class `C` we want `operator=` to look like this:

```
C & C::operator=(const C & c)
{
    ... some code ...

    return (*this);
}
```

What if we have the following instead? Why is this version a bad idea?

```
C C::operator=(const C & c)
{
    ... some code ...

    return (*this);
}
```

Assignment of same object

Depending on what you want to do, you usually do not want to change the object `obj1` if you execute `obj1 = obj2` if `obj2` is actually the same as `obj1`.

```
C & C::operator=(const C & x)
{
    if (this == &x)
        return (*this);

    ... some code ...

    return (*this);
}
```

Cascading `operator=`

Note that if you have

```
class vec2iTemperature
{
private:
    vec2i p_;
    double temp_;
};
```

and objects `t1, t2` of `vec2iTemperature`, then `t1 = t2` is the same as `t1.p_ = t2.p_` and `t1.temp_ = t2.temp_`. `t1.p_ = t2.p_` will call the `=` of `vec2i`. The above puts object inside object. See later section on object composition.

Initialization

Reminder: The following does not call `operator=`

```
Date d = today;
```

The above is actually an initialization and not an assignment. Initialization always involves the constructor. In the above case, the copy constructor is called. In other words, the above is the same as

```
Date d(today);
```

Avoiding member-wise copy

If your object is going to use some resource that requires code to manually acquire and release, you **must** write your own

- – constructor (to acquire resource)
- – destructor (to release resource)
- – copy constructor (to prevent memberwise copy)
- – `operator=` (to prevent memberwise copy)

Here's a standard example where the “resource” is memory ...

```
class IntPointer
{
public:
    IntPointer()
        : p_(new int)
    {}

    IntPointer(const IntPointer & intptr)
        : p_(new int)
    {
        *p_ = *(intptr.p_);
    }

    ~IntPointer() { delete p_; }

    IntPointer & operator=(const IntPointer & intptr)
    {
        if (this == &intptr)
        {
            return (*this);
        }

        *p_ = *(intptr.p_);

        return (*this);
    }

private:
    int * p_;
};
```

You should document your `IntPointer` to let the user know that `IntPointer` objects do not share memory. Why? Because if you use the regular pointers:

```
int * p = new int;
int * q;
q = p;
```

then `p` and `q` do point to the same integer value.

Exercise. What about `IntDynArr`? Write the copy constructor and `operator=`.

Exercise. What about `IntArr`?

`operator>>` and `operator<<`

You've been using `<<` and `>>` since day 1 of CISS240. From your previous notes you know that you can overload `operator>>` and `operator<<`. You also know from CISS240 that you can output `int`, `char[]`, `double`, etc., i.e., C++ has overloaded `operator>>` in `iostream.h` .You also know that you can do

```
std::cout << "A" << 3 << 5.5;
```

So the operators return an object.

```
#include <iostream>

#ifndef INT_H
#define INT_H

class Int
{
public:
    Int(int x0 = 0) : x_(x0) {}
    int get() const { return x_; }
    ...
    void operator<<(std::ostream &, const Int &);
    ...
private:
    int x_;
};

#endif
```

`std::cout` is a `std::ostream` object that's already created for you. `std::ostream` is a C++ class.

```
#include "Int.h"

void operator<<(std::ostream & cout, const Int & c)
{
    cout << c.get();
}
```

2. Same as `operator<<(cout, c.get())`

```
#include "Int.h"

int main()
{
    Int c(42);
    std::cout << c;

    return 0;
}
```

1. Same as `operator<<(std::cout,c)`

Cascading operator<<

If you want to do the following:

```
Int c(42);
std::cout << c << "\n";
```

you need to return an `ostream` object from your `operator<<` so that the above becomes

```
Int c(42);
operator<<(operator<<(std::cout, c), "\n");
```

```
#include <iostream>

#ifndef INT_H
#define INT_H

class Int
{
public:
    Int(int a)
        : x(a)
    {}

    int get() const { return x; }

private:
    int x;
};

std::ostream & operator<<(std::ostream &, const Int &);

#endif
```

```
#include "Int.h"

std::ostream & operator<<(std::ostream & cout, const Int & c)
{
    cout << c.get();

    return cout;
}
```

```
#include "Int.h"

int main()
{
    Int c(42), d(43);
    std::cout << c << "," << d << "\n";

    return 0;
}
```

Friends

You can let `operator<<` have access to the private members of the object by making your `operator<<` a **friend** in your class. We will re-visit friends later.

```
#include <iostream>

#ifndef INT_H
#define INT_H

class C
{
public:
    Int(int x0 = 0)
        : x_(x0)
    {}

    int get() const { return x_; }

    friend std::ostream & operator<<(std::ostream &, const Int &);

private:
    int x_;
};

#endif
```

```
#include "Int.h"

std::ostream & operator<<(std::ostream & cout, const Int & c)
{
    cout << c.x_;
    return cout;
}
```

```
#include "Int.h"

int main()
{
    Int c(42),d(43);
    std::cout << c << "," << d << "\n";

    return 0;
}
```

In general, if the header of a function `f` is a friend in a class `C`, then `f` has access to things in private section of class `C`:

```
class C
{
public:
```

```
    friend void f(C & c);

private:
    int x_;
    void m() { x++; }
}

void f(C & c)
{
    c.x_ = 0; // f can access c.x_ (private)
    c.m();    // f can access c.m  (private)
}
```

Do NOT declare too many friends in a class – this breaks the principle of information hiding and makes the class harder to maintain in case you need private members variables or methods in the future.

Of course friends can be completely avoided.

operator>>

For input, just change `ostream` to `istream`. Everything else is pretty similar. You should be able to figure the differences out yourself. Here's an example to guide you.

```
#include <iostream>

#ifndef FRACTION_H
#define FRACTION_H

class Fraction
{
public:
    Fraction(int a, int b)
        : n_(a), d_(b)
    {}

    friend std::istream & operator>>(std::istream &, Fraction &);
    friend std::ostream & operator<<(std::ostream &, const Fraction &);

private:
    int n_, d_;
};

#endif
```

```
#include "Fraction.h"

std::ostream & operator<<(std::ostream & cout, const Fraction & r)
{
    cout << r.n_ << "/" << r.d_;
    return cout;
}

std::istream & operator>>(std::istream & cin, Fraction & r)
{
    cin >> r.n_ >> r.d_;
    return cin;
}
```

```
#include "Fraction.h"

int main()
{
    Fraction c(0,1), d(0,1);

    std::cout << c << "," << d << "\n";
    std::cin >> c >> d;
    std::cout << c << "," << d << "\n";

    return 0;
}
```