Chapter 7

The C++ Programming Language

Operator Overloading
Operator Overloading

Having defined the class `Complex`, we want to be able to add two complex numbers.

Defining a standard function would make the code harder to read.

If only we had `+` as a function we could have overloaded it.

The compiler knows that it does not know how to add complex numbers.
Operator Overloading

Almost all standard C++ operators can be overloaded for classes (and structures) by defining functions of the form `operator OP`, where `OP` is one of the predefined C++ operators.

Here is the first version of `operator+( )`, implemented as a member function:

class Complex {
   public:
      Complex(double r = 0, double i = 0)
         : real(r), imag(i) {}
      Complex operator +(Complex const&) const;
   private:
      double real, imag;
};

Complex Complex::operator +(Complex const & z) const
{ return Complex(real + z.real, imag + z.imag); }

This implementation works – but it is wrong

operator+ does not modify its arguments

operator+ always returns a temporary object
Using Overridden Operators

Let \( z_1, z_2 \) and \( z_3 \) be Complex:

\[
\text{Complex } z_1(1, 2), \ z_2(-1, 4), \ z_3;
\]

then the C++ code:

\[
z_3 = z_1 + z_2;
\]

actually calls two operator functions:

\[
z_3.\text{operator } = (z_1.\text{operator } + (z_2));
\]

where the function \text{operator }=() is predefined by the compiler.
Pre-Defined Operators

- The only operators defined by default for all classes are:
  - **Address of**: `operator&` *(unary)*
  - **Assignment**: `operator=` *(binary)*
    - Allows assignment of one object of the class to another.
    - Memberwise (recursive) copy assignment
  - **Sequencing**: `operator,` *(binary)*

- Even these are just a result of a "Historical Accident" (compatibility with C).
Basic Rules of Operator Overloading

- The overloaded instance of an operator must contain at least one argument of a class type. 
  - Otherwise other versions of the operator may be invoked.
- Only the predefined operators may be overloaded.
- The predefined precedence and associativity direction of any operator cannot be changed.
- The unary/binary nature of the operator cannot be changed.
- These cannot be overloaded:
  - `sizeof`, `::`, `.`*, `?:`

Click

Click

Click

Click

Pointer to Member
The Rational Number Class

class Rational {
    public:
        Rational(int top = 0, int bottom = 1): t(top), b(bottom) { normalize();}
    // ...

    private:
        int t, b;
        void normalize() // A private member function.
        {
            if (b < 0) {
                b = -b;
                t = -t;
            }

            int divisor = gcd(abs(t), b);
            t /= divisor;
            b /= divisor;
        }
};

Where is the bug?
gcd() is defined elsewhere
A Unary Operator Member

class Rational {
    //...

public:
    //...
        Rational operator -() const
        {
            return Rational(-t,b);
        }
    //...
};

Rational r(-1,3);
Rational s = -r; // Activates r.operator-()
class Rational {

    // ....

public:
    // ....
    Rational operator +(Rational const& r) const
    {
        return Rational(t*r.b + r.t*b, b*r.b);
    }
    Rational operator -(Rational const& r) const
    {
        return Rational(t*r.b - r.t*b, b*r.b);
    }
    Rational operator *(Rational const& r) const
    {
        return Rational(t*r.t, b*r.b);
    }
    Rational operator /(Rational const& r) const
    {
        return Rational(t*r.b, b*r.t);
    }

    // Note that Rational() calls Rational::normalize()
Type Conversion via Constructors

In our example, the Rational constructor provides conversion from int to Rational, thus we can write:

Rational r, r2(1,2),r3(1,3);
...
r = r2 + r3; // Same as r2.operator+(r3)
...
r = r2 - 73; // Same as r2.operator-(Rational(73))
...
r = r2 * 'a'; // Same as r2.operator*(Rational(int('a')))
...
// And even:
r = r3 * 13.1 // Same as r3.operator*(Rational(int(13.1)))

A Ctor with a single argument acts as a conversion operator

How can we prevent this conversion to int
Symmetric Operators

Rational $r$, $r_2(1,2), r_3(1,3)$;
...

We are not interested in monsters like:

3 += $r$;

Even an int on the left is unreasonable

i += $r$; // operator+= should not be symmetric
...

But it is really required to have expressions like:

... $r = 5 + 3 \times r_2$; // +, *, etc. should be symmetric
...
Symmetric Operators

Unfortunately, the above will not work, since the object which is the receiver of the message, must appear first.

\[ 3 \times r \quad \text{// Is equivalent to} \quad 3 \cdot \text{operator}(r) \]

Since \textit{this} is never converted from one type to another, we need a function that has no implicit parameter.

The \textbf{simplest solution} is to define a function that is \textbf{not a member function} (of any class).

Alas, this non-member function may \textbf{need access} to \textbf{private} members of the objects.
Friends

- A function $f$ or class $A$ may be defined to be a friend of class $B$.
- This allows access by $f$ or by all methods of $A$ to private fields of $B$.
- $B$ declares its friends.
- Useful for efficiency.

friendship is neither transitive nor symmetric

Do not compile :-)

```c++
int f(double);
class A { . . . };
class C {
    public: void g(int);
    . . .
};
class B {
    friend class A;
    friend int f(double);
    friend void C::g(int);
    private: int x;
    public: . . .
};
int f(double) { B b;
    printf("%d\n",b.x); }
```
Arithmetic Operators as friends

Operators can be defined as member or non-member functions. Let’s try the non-member version:

class Rational {
public:
    // ....
friend Rational operator* (Rational const & r1, Rational const & r2)
    { return Rational(r1.t*r2.t,r1.b*r2.b); }
friend Rational operator/ (Rational const & r1, Rational const & r2)
    { return Rational(r1.t*r2.b,r1.b*r2.t); }
    // ....
private:
    // ....
};

This implementation is a bit better: but should be improved!
Using Arithmetical Operators

Now we can write:

```c
int i;
...
r = i / (3 * r2);
...
```

With the compiler doing the necessary conversions. For a usage such as the above the parameters must be `const&` or pass by value. Why?
A Friend Unary Operator

```cpp
class Rational {
  public:
    //...
    friend Rational operator -(Rational const & v) {
      return Rational(-v.t, v.b);
    }
    //...
};

Rational r(-1,3);
Rational s = -r;
```

We will meet an exception to this rule

A member is preferred
**A reminder:** The hidden (implicit) parameter within any non-static member function, which is associated with the current object that the function operates with, is called `this`. It is a pointer to that object. `this` is the only way to explicitly call that object.

```cpp
class Complex {
    double real, imag;

public:
    double real_part() const
    { return this->real; }

    // ...
};
```

There is no real reason to use `this` in order to access object’s member. However, it is partially a matter of style: in Java, much code is written with explicit usage of `this`. 
Overloading Assignment Operators

- **Overloading** operators of the type `OP=` should be **done with care**:
  - Always use **member** functions
    » *Friends do not guarantee that left operand is an lvalue.*
  - The return type should be a reference to the class.
  - The operator should return reference to `*this`.
    » **C++ even allows constructs of the form:**
      
      $$(X += Y) *= Z;$$
      
    - The above is true even for simple assignment: `a = b = c`
  - The compiler may not enforce the rules above.
A Binary Operator Member  
(First version)

class Rational {
  public:
    // ....
    Rational& operator +=(Rational const& val)
    {
      t = t * val.b + val.t * b;
      b *= val.b;
      normalize();
      return *this;  // a reference to the object
    }
    // ....
};

Rational r1(1,2), r2(1,3), r3(1,4);
r1 += 5;
(r1 += r2) += r3;
Binary Operator Members
(Second version)

class Rational {
  public:
  // Using previously-defined operators:
  Rational& operator +=(Rational const& val) {
    return *this = *this + val; }
  Rational& operator -= (Rational const& val) {
    return *this = *this - val; }
  Rational& operator *= (Rational const& val) {
    return *this = *this * val; }
  Rational& operator /= (Rational const& val) {
    return *this = *this / val;
  }
};

In contrast to the above, experienced C++ programmers will define operator OP using operator OP=
Binary Operators Combination
(Best Version)

class Rational {
  public:   // ....
    Rational& operator*=(Rational const& val)
    { t *= val.t;   // Note: Back to 1st version
      b *= val.b;
      normalize();
      return *this;
    }
};

Rational operator*(Rational const& a, Rational const& b)
{ Rational tmp(a);
  return    tmp *= b;
}

Rational operator*(Rational a, Rational const& b)
{return    a *= b;  }
Relational Friend Operators

We can go on and define many more friend operators, such as all the relational operators.

class Rational {
  public:
    // ....

    friend bool operator ==(Rational const&, Rational const&);
    friend bool operator !=(Rational const&, Rational const&);
    friend bool operator <=(Rational const&, Rational const&);
    friend bool operator >=(Rational const&, Rational const&);
    friend bool operator < (Rational const&, Rational const&);
    friend bool operator > (Rational const&, Rational const&);
    // ....
};
Member Operators vs. Friends

- Prefer members, if possible.
- Use non-members (sometimes friends) when you must (e.g., for symmetry).
- Use members to return a reference.
Making Copies of Existing Objects

The Copy Constructor
and Overloading Operator=
The Default Assignment Operator

- The default assignment operator, recursively assigns (using \texttt{operator\textasciitilde()}, a.k.a. \textit{copy-assignment}) each data member from the object on the right to the object on the left.

- This is \textbf{exactly} what we want for objects like \texttt{Complex}, or \texttt{Rectangle}.

```
Complex c0(2), c1(-1,2.3);
Point p0(3,2), p1(5,0.1), p2(0,0);
Rectangle r0(p1,p2), r1(p1,p0);
.
.
c0 = c1;
r0 = r1;
.
```
However, consider what will happen if we use the default assignment operator for the last version we had for String.

```plaintext
String s0("abc"), s1("wxyz");

s0 = s1;
```

The diagram shows the assignment operation between two strings, `s0` and `s1`, and the effect on their contents.
Non-default Assignment Operator (Cont.)

Here is how to naïvely comply with basic requirements for an assignment operator for the class `String`.

- Basically, we apply copy-assignment on each data member.

```cpp
String&
String::operator=(String const& rhs)
{
    if (this == &rhs) return *this;
    len = rhs.len;
    delete[] s;
    s = new char[len + 1];
    strcpy(s, rhs.s);
    return *this;
}
```

- Return a reference
- A member function
- No self assignment
- Free old memory
- Allocate new memory
- Work with new memory
- Return the lhs object

A safer, more efficient implementation is suggested in C++ FAQs
The Copy Constructor

• The copy-assignment operator copies an existing object into an existing object.
• But in many cases we have to create a new object that is an exact copy of an already existing object.
• The most visible case is when defining a variable that is initialized by an object of the same type:

```plaintext
int       i  = 3, j(5);
double    x  = i, y = x;
Complex   c0 = x, c1 = c0;
String    s0(“abc”), s1 = s0, s2(s0);
```
• Less visible cases are the other two:
  – Passing an argument, by value, into a function.
  – A function returns a value (and not a reference).

All three were used on the previous “Best Version” slide of `operator*`
Using Copy Constructor

class Rational {
    public: // ....
        Rational& operator*=(Rational const& val) {
            t *= val.t; // Note: Back to 1st version
            b *= val.b;
            normalize();
            return *this;
        }
    }

Rational operator*(Rational const& a, Rational const& b) {
    Rational tmp(a);
    return tmp *= b;
}

Rational operator*(Rational a, Rational const& b) {
    return a *= b;
}

Non-member, non-friend function

Here it is again

A better idiom: No double efforts
In all those three cases a constructor must construct the new object.
It is named the Copy Constructor.
It has a single parameter, of the class type.

```cpp
class String {
  public:
    . . .
    String(const String& src): len(src.len),
                      s(strcpy(new char[len+1], src.s)){} 
    . . .
};
```

How many copy-Ctors we may have in a class?
The Copy Constructor (Cont.)

• How could we manage without a copy Ctor for classes like Complex or Rational?
  » Because the compiler creates a default copy Ctor, whenever a class is defined without one.

• How does the default copy Ctor work?
  » It initializes each data field of the new object by the corresponding value from the initializing object using the appropriate copy Ctor !!!

• The same will be for user defined copy Ctor!

Usually, you want to share code between operator= and copy Ctor, using hidden functions
The Big Three

Marshal Cline, the author of the book *C++ FAQs*, has noted the following:

In almost all cases of class definitions, whenever you need to define one of

- A copy Ctor
- A Dtor
- A copy assignment

YOU NEED THEM ALL

The following “equation” is the “reason”: \( \text{operator=} = \text{Dtor} + \text{copyCtor} \)
C++11: Move Semantics

- Simple copy:

```cpp
String a("12"), b(""), c(a); // c is copy-constructed
...
...
b = a; // a is assigned to b
```

Caveat: Memory allocation may fail, with all due consequences
C++11: Move Semantics  (Cont.)

• Not so simple copy (a temporary is involved):

\[
a = b + c; \quad \text{// } b + c \quad \text{is a temporary (immediately destroyed)}
\]

\[
\begin{array}{c}
\text{String } f(String \ s) \ {\{\text{return } s;\}} \quad \text{// } s \quad \text{disappears}\\
c = f(b); \quad \text{// } b \quad \text{is copied, } s \quad \text{is copied out, then assigned to } c
\end{array}
\]

Caveats: The assignments may fail, even when the temporaries exist. Construction+deletion CPU is wasted
C++11: Move Semantics  

• Not so simple copy – improved:

\[ a = b+c; \] // \( b+c \) is a temporary (immediately destroyed)

\[ c = f(b); \] // \( b \) is copied, \( s \) is copied out, then assigned to \( c \)

**Problem:** We cannot catch temporaries!

**Benefits:** Assignments cannot fail, after the temporaries are created.
Construction+deletion CPU is minimal
C++11: Move Semantics (Cont.)

- C++11 added syntax that catches temporaries
  
  ```
  T&& t = a + b; // A non-const rvalue-reference!
  – There is much to be said about it, but I skip it (inquire!)
  ```

- The **basic way** to use it for move operations
  – There are many relevant issues to be considered

```c++
String::String(String&& other) // Move copyCtor
: len(other.len), s(other.s) // No allocation
{ other.len = 0; other.s = nullptr; }

String& & String::operator=(String&& other)
{
    delete[] s; // Move copyAssign
    len = other.len; s = other.s; // No allocation
    other.len = 0; other.s = nullptr;
    return *this;
}
```
Operator Overloading Test Case

Input/Output Operators

by

Ayal Itzkovitz
Input/Output with C++

```c
#include <stdio.h>
int i;
double f;
.
scanf("%d %lf", &i, &f);
printf("%d %f\n", i, f);
fprintf(stderr, "%f", f);
```

```cpp
#include <iostream.h>
int i;
double f;
.
cin >> i >> f;
cout << i << f << endl;
cerr << f;
```

- `cin`, `cout`, `cerr` are objects which represent the stdin, stdout, stderr respectively.
- Using operator `>>` and operator `<<` objects can perform input/output.
- Operations are performed according to the types of the objects - no formats.
Using I/O operators rather than function calls

• Easier to program.
  – No need to specify type (%d, %f, etc)

• Object read/write themselves. Information hiding!
  – We ask the object to write itself.
  – Object asks to get the appropriate data for its internal data structures.
  – Try printf(“??”, complex);

• Robust code.
  – Avoid bugs like scanf(“%f”, &i); (i is int)

• Future extensions.
  – Future classes should just implement operators >> and << for themselves.
**istream - member functions** (partial list)

**Construction/Destruction — Public Members**
- `istream` Constructs an istream object.
- `~istream` Destroys an istream object.

**Input Functions — Public Members**
- `get` Extracts characters from the stream.
- `peek` Returns a character without extracting it from the stream.
- `getline` Extracts characters from the stream (extracts and discards delimiters).

**Other Functions — Public Members**
- `putback` Puts characters back to the stream.

**Operators — Public Members**
- `operator >>` Extraction operator for various types.
Using istream member functions

**Reading a line from cin**

```cpp
const int max_line = 70;
char line[max_line];
cin.getline(line, max_line);
```

**Reading a single char from cin without removing it from queue**

```cpp
int c;
c = cin.peek();
if (c == EOF) return;
```

**Putting a read character back**

```cpp
int c = cin.get();
if (c == 'l') {
    load_db();
    cin.putback(c);
}
```
### ostream - member functions (partial list)

#### Construction/Destruction — Public Members

- **ostream** Constructs an ostream object.
- **~ostream** Destroys an ostream object.

#### Flag and Format Access Functions — Public Members

- **precision** Sets or reads the stream's floating-point format display precision.
- **width** Sets or reads the stream's output field width.

#### Operators — Public Members

- **operator <<** Insertion operator for various types.

#### Manipulators

- **endl** Inserts a newline sequence and flushes the buffer.
- **ends** Inserts a null character to terminate a string.
Using ostream member functions

cout.width(8);
cout.precision(2);
cout.operator<<(12.3456);  // or  cout << 12.3456;
  > will print:  12.35

cout << '[' << 12.3456 << ']' << endl;
  > will print: [ 12.35]
  >
Adding I/O to the Rational Number Class

class Rational {
private:
    int t,b;
void normalize(); // A private member function.
public:
    // constructor
    Rational(int top = 0, int bottom = 1):
    t(top), b(bottom) { normalize();}

    // i/o operators
friend istream&
    operator>>(istream& in, Rational& r);
friend ostream&
    operator<<(ostream& out, Rational const& r);

    // other operators
    ...
};
Adding I/O to the Rational Number Class (cont.)

```cpp
// istream & operator>>(istream & in, Rational & r)
operator>>(istream & in, Rational & r)
{  in >> r.t >> r.b;
   return in;
}

// ostream & operator<<(ostream & out, Rational const & r)
operator<<(ostream & out, Rational const & r)
{ out << r.t << '/' << r.b;
   return out;
}
```

Rational r1(1,6), r2(1,3);

cout << r1 << '+' << r2 << '=' << r1+r2 << endl;
> will print: 1/6 + 1/3 = 1/2
A different way of overloading `operator<<`

```cpp
class Rational {
    ...
    ostream& print(ostream& os) const
    { return os << t << '/' << b; }
    ...
};
```

The same idea may be used to make relational operators (`==, >, ...`) non-friend

```cpp
// Non-friend, if Rational::print() is public
inline ostream&
operator<<(ostream& out, Rational const& r)
{ return r.print(out); }
```

The advantages of this will be clear in the future
Special Operators
class Rational {
    Rational& operator++() // Prefix (efficient)
    {
        return *this += 1;
    }
    Rational operator++(int) // Postfix, a dummy int
    {
        Rational tmp(*this); operator++(); return tmp;
    }
};

A quirk in C++ type system makes nonmember implementation safer.
Rational& operator++(Rational& r) // Prefix
{
    return r += 1;
}
Rational operator++(Rational& r, int) // Postfix
{
    Rational tmp(r); ++r; return tmp;
}

Implementing operator-- is done the same way

No need to declare friend in the class
The Concept of an Iterator

• An iterator’s job is to traverse a collection
  » A collection of elements is usually called a container

• The interface for iterators follows pointers
  » It allows using pointers as iterators
  » Iterators require overloading of at least `==`, `++`, `*`, `->`
    • For practical reasons, `!=` is considered more basic than `==`
  » Bidirectional iterators can go backward using `--`

Some iterators overload `+= n`, `-= n` (hence `+` and `–`)
They are called random-access iterators
They fully simulate array access (in that respect)
Iterator Types

• An **iterator** provides access to the elements
  » Accessing an element allows modifying it

• Const **containers** prevent element modifications
  » Therefore, a container should provide a **const_iterator**

• Some **containers** allow reverse traversal
  » Therefore, they provide two **reverse_iterators**
  » Usually, their four iterators are bidirectional

**Iterators** should be associated
with a specific type (class) of **containers**
They are usually defined as **nested classes**
Overloading Unary `operator*`

- Operator overloading should be used when intuitive
  - Overloading unary `*` is for pointer-like objects only
- Semantics of overloaded operator must be intuitive
  - Returned value is a reference to the pointed-to object

```cpp
class MemManager {
    public:
        MemManager(T *p): p_(p) {} // Take responsibility
    ~MemManager() {delete p_;} // Release
    T& operator*() {return *p_;}
    T const& operator*() const {return *p_;}
    T* release() {T *t = p_; p_ = NULL; return t;}
    private: T *p_; // Don’t allow copy Ctor/Assign
        MemManager(MemManager const&) = delete; // C++11
        MemManager& operator=(MemManager const&) = delete;
};
```
Overloading `operator->`

- `operator->` is binary, but it **takes no arguments**
- When applying overloaded `operator->` like `a->m` it is **interpreted** as `(a.operator->())->m`
- Therefore, it must return a **pointer (like)** object
  
  » If the returned object is **not a raw pointer** it must have an overloaded `operator->` and the call is further delegated

```cpp
class MemManager {
    public:
        MemManager(T *p): p_(p) {}
        // Take responsibility
        // . . .
    T* operator->() { return p_; }
    T const* operator->() const { return p_; }
};
```
An Iterator Example

Assume that Set was a C++ class that had iterators

class Set {   // const s missing in order to shorten lines
public:   // . . .
    friend class Iterator;
    class Iterator {
        private:
            Set *s; int curr;   // The straightjacket
        friend class Set;
        explicit Iterator(Set *s = NULL)   // trick 4 end()
            : s(s), curr(s ? -1 : MAX_SIZE-1) { operator++();}
        public:
            bool operator==(Iterator &i) {return curr==i.curr;}
            bool operator!=(Iterator &i) {return !operator==(i);}
            Iterator& operator++() {++curr;while(curr < MAX_SIZE
                && s->elem[curr] == NULL) ++curr; return *this;}
            T* operator->() {return s->elem[curr];}
            T& operator*() {return *operator->();}
    };
    Iterator begin() { return Iterator(this); }   //STL-like
    Iterator end() { return Iterator(); }   //STL-like
};
Using an Iterator Example

Assume that Set was a C++ class that had iterators
And that S contains pointers to struct Complex

Set s;
// Populate s
// And then (using C++11)
for (auto i = s.begin(); i != s.end(); ++i) {
    if (i->abs() > 10) ++i->real;
    cout << *i;
}

An Iterator as a Generator

When we iterate over a range we use a for loop

\[
\text{for (double } i = \text{ first; } i \leq \text{ last; } ++i) \{ \ldots \}
\]

What if the range (and type?) is dictated by another class? Should that class supply an array \{7, \ldots , 7M\} for the values?

class range {
  double frst, last, step;
  public:
    range(double f, double l, double s)
      : frst(f), last(l), step(s) { /* sanity test */ }
  class Iterator {
    double val, step;  // Assume step > 0
    Iterator(double v, double s): val(v), step(s) {}  // Initializer list
  public:
    double operator*() { return val; }
    Iterator& operator++() { val+=step; return *this; }
    bool operator!=(const Iterator& i) const
      { return i.val < val-step/2 || i.val > val+step/2; }
  };
  Iterator begin() { return Iterator(frst, step); }
  Iterator end() { return Iterator(last, step); }
};
Using a Generator

(look for `yield return` – not C++)

Range \( r(9.3, 22.5, 0.4); \)
// . . . Maybe \( r \) is from a different file
for (auto \( i = r\).begin(); \( i != r\).end(); ++i) {
    auto \( d = *i; \) // \( d \) is `double`
    // Using \( d \) (or \( *i \) directly)
}

**C++11 provides** range based **for** statement

for (auto \( d : r \)) {
    // Using \( d \), which is `double`
}

**It works for any container with iterator+begin+end**

for (auto \( d : \{1.3, 4.9, -2.1, 5.6\} \)) {
    // Using \( d \), which is `double`
}
Using a Generator (Cont.)

We are **not limited** to fixed step or anything – *sky is the limit*

```cpp
class Prime {   // Not the complete class
  int frst, last;
  public:
    class Iterator {   // Not the complete class
      int val;
      Iterator(int v)  // v > 2 and is a prime
        : val(v) {if (!IsPrime(val) error());}
      Iterator& operator++() { do val += 2;
                               while (!IsPrime(val));  return *this;}
      int operator*() { return val;}
    };
    Iterator begin()const {return Iterator(frst);}
    Iterator end()const {return Iterator(last);}
};

for (auto p: Prime(19, 997)) { /* Use p */  }
```