C++ Operators

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.

```
Complex z0(1,2), z1(3.5), z2;
z2 = z0 + z1;  // Syntax error
```

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

```
x + y     // Add two doubles
add(z0,z1) // Add two Complexes
```

```
// But operator+ IS a function!
operator+()  // The name of the function
```
Operator Overloading

Almost all standard C++ operators can be overloaded for classes (and structures) by defining functions of the form \texttt{operator OP}, where \texttt{OP} is one of the \texttt{predefined} C++ operators. Here is the first version of \texttt{operator+()}, implemented as a member function:

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

This implementation works, but it is incomplete!

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

\texttt{operator+} does not modify its arguments

\texttt{operator+} always returns a temporary object
Let $z_1, z_2, z_3$ be Complex:

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 (unless we implement our own version of it).
Pre-Defined Operators

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

- However, the existence of the above operators in C++ is just a “Historical Accident” (compatibility with C).
Basic Rules of Operator Overloading

- Only the predefined operators may be overloaded.
- The predefined *precedence* and *associativity* direction of any operator *cannot be changed*.
- The following operators cannot be overloaded: `sizeof` `::` `.` `.*` `?:`
- The *unary/binary* nature of the operator *cannot be changed*.
- 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.
- When a binary operator invokes a member function the left argument becomes `this` and the right argument is passed as a parameter; when a unary operator invokes a member function, the argument becomes `this`.

**Introduction to Systems Programming**
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;
    }
};

There is a bug here

gcd() is defined elsewhere
class Rational {
    // ...
public:
    // ...
    Rational operator -() const {
        return Rational(-t,b);
    }
    // ...
};

Rational r(-1,3);
Rational s = -r;  // Activates the operator 
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()  
};

This implementation also works, but is still incomplete!
Type Conversion via Constructors

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

```cpp
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 can act as a conversion operator.

Note this undesired conversion to `int`. (Implementing another constructor accepting a `double` parameter is possible but not optimal.)
Symmetric Operators

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

We are not interested in constructions like
$3 += r$;

And an int on the left-hand side is unreasonable:
$i += r; // operator+= should not be symmetric$

But it is really desired to have expressions like
$r = 5 + 3 \times r_2; // +, *, etc., should be symmetric$
Unfortunately, the above will not work, since the object which is the receiver of the message, must appear **first**.

```c
3 * r  // Is equivalent to 3.operator*(r)
```

This will try to activate a member function of “class” **int**

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

The **simplest solution** is to define a function that is not a member function (of any class).

Alas, this non-member function may **need access to private members of the objects**. For this, we have a “backdoor” via the **friend directive** (next slide).
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: friendship is given, not taken!

Useful for efficiency, but not for quality code.

*friend*ship is neither transitive nor symmetric
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:
    // ....
};

These functions are also inline since they are implemented in the class.

This implementation is a bit better, but it should still be improved!
Now we can write

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

Letting the compiler do the necessary conversions.

For this to happen, the parameters must be `const&`, or be passed by value.
A Friend Unary Operator

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

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

Whenever possible, a member function is preferred over a friend function. However, we will meet an exception to this rule.
Reminder: “this” is a pointer to the hidden (implicit) parameter of a non-static member function, that is, to the object on which the function operates. Using this is the only way to explicitly refer to this object.

```
class Complex {
  double real, imag;
public:
  double real_part() const {
    return this->real;
  }
  // . . .
};
```

Except for returning the entire object, there is no reason to use this in order to access its members (see exception below). However, it is a matter of style: In Java, much code is written with explicit usage of this.)

In this example, if the function real_part had also defined a local automatic variable whose name was real, then the only way the function could refer to the data member real was via this. However, this is a very poor naming style.
Overloading operators of the type \texttt{OP=} should be done with care:

- **Always** use member functions (since friends do not always guarantee that left operands are l-values).
- The return type should be a **reference** to the class, and the operator should return a reference to \texttt{*this}.
- Note that C++ even allows constructs of the forms

  \[(X +\text{=} Y) *\text{=} Z; \quad \text{and} \quad X +\text{=} (Y *\text{=} Z);\]

- The above is true even for simple assignment: \texttt{a=b=c};
- The compiler may not enforce the rules above, thus, it’s the programmer’s responsibility to follow them!
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;
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; }
// ....
};

C++ programmers, who would like to avoid friend functions, will do the opposite: Define operator OP using operator OP=.
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;
}
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&);
        // ....
};

Not all of them need to be friend!

Can none of them be friend?
What is the minimum possible?
Member Operators vs. Friends Functions

- 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 Operator=
The default assignment operator, assigns recursively (using `operator=()`, a.k.a. the copy-assignment operator) each data member from the object on the right to the object on the left.

This is exactly what we want for objects like Complex or Rectangle (that is, objects with no dynamic memory).

```cpp
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;
...```

Introduction to Systems Programming
However, we certainly do not want to invoke the default assignment operator for the version of String that uses dynamic memory!

String s0(“abc”), s1(“wxyz”);

BANG

s0 = s1;
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
The Copy Constructor

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

```c
int i = 3, j(5);
double x = i, y = x;
Complex c0 = x, c1 = c0;
String s0("abc"),
    s1 = s0, s2(s0);
```

- Two other cases:
  - Passing by value an argument to a function
  - Returning a (non-reference) value from function

All three cases were used in the “best version” slide of operator* (see next slide)
Using the Copy Constructor (repeated)

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);
}

Here it is again

The combination (explicit dependency)
Non-member, non-friend function

A better idiom: No double efforts
In all those three cases, the new object is constructed by a **Copy Constructor**

- It has a **single** parameter, which must be of the same 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? Can we pass the argument “src” by value?
Q: How could we manage without a copy Ctor for classes like Complex or Rational?
A: The compiler creates a default copy Ctor, whenever a class is defined without a user-implemented one.
Q: How does the default copy Ctor work?
A: It initializes each data field of the new object by the corresponding value, taken from the initializing object, using the appropriate copy Ctor!
The default copy Ctor fits situations where no dynamic memory is involved.

**Usually, if you implement operator= and the copy Ctor by yourself, you would want to share code between them using private functions**
The reason is shared functionality: \( \text{operator=} = Dtor + \text{copyCtor} \)
Objects `cin`, `cout`, `cerr` replace C’s `stdin`, `stdout`, `stderr`, respectively.

Objects perform input/output by using `operator>>` and `operator<<`, which are C++ operator and not external functions.

There are **no format specifiers**; operations are performed according to the types of the objects.

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

```cpp
#include <iostream.h>
int i;
double f;
...
 cin  >> i >> f;
 cout << i << f << endl;
cerr << f;
```
I/O Operators instead of Function Calls

• **Easier to program**
  – No need to specify type (%d, %f, etc.)
• **Object read/write themselves**
  – Information hiding!
  – The user asks the object to read/write itself, and only the object accesses its internal data structures
• **Robust code**
  – Avoid bugs like forgetting the “&” or mixing types:
    ```c
    scanf ("%f", &i);  // (i is int)
    ```
• **Future extensions**
  – In C we cannot add format specifiers, like in
    ```c
    printf ("%?", complex);
    ```
  – In C++, we can add I/O ops `>>` and `<<` to new classes.
• **Caveat**
  – If many specifiers are used, the calling sequence in C++ looks much more cumbersome than in C
**istream: Member Functions (partial list)**

### Construction/Destruction
- **istream** Constructs an object
- **~istream** Destroys an object

### Input Functions
- **get** Extracts characters from the stream
- **peek** Returns a stream character *without extracting it*
- **getline** Extracts characters from the stream until **EOL**

### Other Functions
- **putback** Puts characters back to the stream

### Operators
- **operator>>** Extraction operator for **various types**
Using \textit{istream}

\begin{verbatim}
Reading a line from \textit{cin}:
const int max_line = 70;
char line[max_line];
\textit{cin}.getline(line, max_line);

Reading one char from \textit{cin}:
int c;
c = \textit{cin}.get();
\end{verbatim}

\begin{verbatim}
Reading a single char from \textit{cin}
without removing it from queue:
int c;
c = \textit{cin}.peek();
if (c == EOF) return;

Putting back a read character:
int c = \textit{cin}.get();
if (c == 'l') {
    load_db();
    \textit{cin}.putback(c);
}
\end{verbatim}
**ostream: Member Functions (partial list)**

**Construction/Destruction**
- `ostream` Constructs an object
- `~ostream` Destroys an object

**Flag and Format Access Functions**
- `precision` Gets or sets the stream's floating-point format display precision
- `width` Gets or sets the stream's output field width.

**Operators**
- `operator<<` Insertion (write) operator for various types

**Manipulators**
- `endl` Inserts a newline code and flushes the buffer
- `ends` Inserts a null character to terminate a string
Using ostream

int i = cout.width(); // say i equals 7
cout.width(i+1); // now width is 8
cout.precision(2);
cout.operator<<(12.3456);
    // or  cout << 12.3456;
> will print:  12.35 (3 leading spaces)
cout << '[' << 12.3456 << ']' << endl;
> will print: [ 12.35]
>
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
    ... 
};

Note in both operators, which operand is on the left and which is on the right!
**I/O Operators in the Rational Class (cont.)**

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

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

// Non-friend function
inline ostream& operator<<(ostream& out, Rational const& r)
{ return r.print(out); }

Using the same idea as for the symmetric operators!
class Rational {
    Rational& operator++() // Prefix (efficient)
    {
        return *this += 1;
    }
    Rational operator++(int) // Postfix, a dummy int
    {
        Rational tmp(*this); operator++(); return tmp;
    }
};

An alternative, using external but non-friend functions ():
Rational& operator++(Rational& r) // Prefix
    {
        return r += 1;
    }
Rational operator++(Rational& r, int) // Postfix
    {
        Rational tmp(r); ++r; return tmp;
    }

operator-- is implemented in the same way
The Concept of an Iterator

- An iterator’s job is to traverse a collection of elements (a container)
- The interface for iterators follows pointers
  - It allows using pointers as iterators
  - Iterators require overloading of at least $\text{==}$, $\text{++}$, $\text{*}$, $\rightarrow$
  - For practical reasons, operator $\text{!=}$ is considered more basic than $\text{==}$
  - Bidirectional iterators can go backward using $\leftarrow$

Some iterators overload $\text{+= } n$, $\text{-= } n$ (hence $\text{+}$ and $\text{-}$)

They are called random-access iterators

They fully simulate array access

Like pointers, the “+” operation on two iterators would not be allowed in general (but “−” is OK!)
**Iterator Types**

- An *iterator* provides access to the elements, and, thus, allowing to 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* (each one with a different default direction). 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. Therefore, overloading unary operator* is for pointer-like objects only.
- The semantics of overloaded operator must be intuitive. The 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;
};
```

Function cannot be called!
Overloading operator->

- operator-> is binary, but it takes no arguments.
- When applying overloaded operator-> as in 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 is a C++ class which has iterators.

```cpp
class Set {   // const 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 is a C++ class which has iterators, and that set \( s \) 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; // operator-> has precedence
    // over prefix operator++ !
  cout << *i;
}

Using an Iterator Example

Assume that Set is a C++ class which has iterators, and that set \( s \) 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; // operator-> has precedence
    // over prefix operator++ !
  cout << *i;
}