Operator Overloading

For : COP 3330.
Object oriented Programming (Using C++)
http://www.compgeom.com/~piyush/teach/3330

Piyush Kumar

Operator overloading

C++ allows overloading of most of the standard operators. We are allowed to define standard operators for new types.

The name of an operator, as a function, is "**operator**" followed by the operator itself.
- The operator in "**a - b**" is named "**operator-**".

Suppose we have a class **MyNum**, a new numerical type.
- The syntax for defining operators is
- ReturnType operator@(argument list…)
- For example:

```
MyNum operator-(const MyNum & a, const MyNum &b);
```

Example

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


// Without operator overloading:
MyNum add(const MyNum& x, const MyNum& y);
MyNum mul(const MyNum& x, const MyNum& y);

MyNum f(const MyNum& a, const MyNum& b, const MyNum& c)
{
  return add(add(mul(a,b), mul(b,c)), mul(c,a));    // Yuk...
}
```


Example

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

// With operator overloading:
  MyNum operator+ (const MyNum& x, const MyNum& y);
  MyNum operator* (const MyNum& x, const MyNum& y);

  MyNum f(const MyNum& a, const MyNum& b, const MyNum& c)
  {
   return a*b + b*c + c*a;
  }
```


More examples?

- Here are a few of the many examples of operator overloading:
  - `myString + yourString` might concatenate two std::string objects
  - `myDate++` might increment a Date object
  - `a * b` might multiply two Number objects
  - `a[i]` might access an element of an Array object
  - `x = *p` might dereference a "smart pointer" that "points" to a disk record — it could seek to the location on disk where p "points" and return the appropriate record into x

Operator Overloading

- Operator overloading makes life easier for the *users* of a class, not for the developer of the class!
- Supports more natural usage.
- Uniformity with built in types.

Operator Overloading: Another example

```
class Array {
public:
  int& elem(unsigned i)
           { if (i > 99) error(); return data[i]; }
private:
  int data[100];
};

int main()
{
  Array a;
  a.elem(10) = 42;
  a.elem(12) += a.elem(13);
  ...
}
```

```
class Array {
 public:
  int& operator[] (unsigned i)
           { if (i > 99) error(); return data[i]; }
 private:
  int data[100];
 };

 int main()
 {
  Array a;
  a[10] = 42;
  a[12] += a[13];
  ...
 }
```

Precedence of Operators

- `w = x = y = z`
  - Right associative
  - `(w = (x = (y = z)))`
- `w * x / y * z`
  - Left associative
  - `(((w * x) / y) * z)`

Operator Precedence in C++

::	scope resolution
a++ a--	postfix increment/decrement
a()	function call
a[]	subscript
a. a->	member access
++a --a	prefix increment/decrement
+a -a	unary plus/minus
! ~	logical/bitwise not
`(type)`	C-style cast
*a	dereference
&a	address-of
…	…

- Full List (cppreference.com) http://en.cppreference.com/w/cpp/language/operators

Operator precedence

- `x == y + z;`
- Precedence and associativity are fixed.
- We cannot create new operators. Nor can we change the precedence, associativity (grouping), or number of parameters.

Overloading operators

- The meaning of an operator for the built in types may not be changed.
  - `int operator+(int , int)`
  - not allowed
- Operators that can not be overloaded
  - `::, .*, ., ?:`

Operator Overloading

- **Can I overload `operator==` so it lets me compare two char[] using a string comparison?**
- No: at least one operand of any overloaded operator must be of some user-defined type (most of the time that means a class).
- And you shouldn't be using char[] anyways.

Operator Overloading

- Which ones should you override?
  - The ones your user wants you to. Do not confuse your users.
- Use common sense. If your overloaded operator makes life easier and safer for your users, do it; otherwise don't

Class member Vs non-member

Operators can be global or member. As a member function, the first operand becomes its object, that is, "`*this`".

- Substraction operator as a friend/global :

```
MyNum operator-(const MyNum & a, const MyNum & b);
```

- As a **member** of class `MyNum`:

```
class MyNum {
public:
   MyNum operator-(const MyNum & b) const;
   …
};

MyNum MyNum::operator-(const MyNum & b) const
{  // Continue as usual
```

Member Vs Non-Member

- If MyNum::operator- defined as member function
  - Num1 – Num2
  - Num1.operator-(Num2)
- As a non-member function
  - Num1 – Num2
  - operator-(Num1,Num2)

Non-Member

- When operators are defined as nonmember functions, they might need friendship:

```
class Sales_item {
   friend bool operator==(const Sales_item&, const Sales_item&);
   friend std::istream& operator>>(std::istream&, Sales_item&);
   friend std::ostream& operator<<(std::ostream&, const Sales_item&);
// other members as before
…
}
ostream& operator<<(ostream& out, const Sales_item& s)
{
   out << s.isbn << "\t" << s.units_sold << "\t"
      << s.revenue << "\t" <<  s.avg_price();
   return out;
}
```

Member Vs Non-Member

- ' = ', ' [] ', ' () ', ' -> ' must be defined as members (else -> compile time error)
- Like assignment, compound-assignment operators ordinarily ought to be members. (but not required)

Member Vs Non-Member

- Operators that change the state of the object such as increment, decrement and dereference -- usually should be member functions.
- Symmetric operators like arithmetic, equality, relational, and bitwise operators, are best left as non-members.

Guideline:

- Not a good idea to overload
  - "," , "&" , " && " , "||"
  - These operators have built in meanings that become inaccessible if we define our own.
- An operation to test whether the object is empty could be represented by "operator!" (or in a good state…)

Guideline

vector ,deque ,list ,slist

- A class that has "+", "="
  - The provide "+="
- Classes that will be used as the key type of an associative container should define "<"
  - "An Associative Container is a variable-sized Container that supports efficient retrieval of elements (values) based on keys. It supports insertion and removal of elements, but differs from a Sequence in that it does not provide a mechanism for inserting an element at a specific position. [1] "

Associative Containers:
set , multiset , hash_set , hash_multiset , map , multimap , hash_map , hash_multimap

Types in containers

- Define "==" and "<" for types that will be stored in even sequential containers.
- Many algorithms (e.g. find()) assume that these operators exist

Find() example

```
list<MyNum> nums;
list<MyNum>::iterator nums_iter;

nums.push_back ("3");
nums.push_back ("7");
nums.push_front ("10");

nums_iter = find(nums.begin(), nums.end(), "3"); // Search the list.
if (nums_iter != nums.end())
{
   cout << "Number " << (*nums_iter) << " found." << endl; // 3
}
else
{
   cout << "Number not found." << endl;
}

// If we found the element, erase it from the list.
if (nums_iter != nums.end()) nums.erase(nums_iter);
// List now contains: 10 7
```

Equality and relational operators

- If you define '==' then also define '!='
- If you define '<' then also define '<=, >=, >'

Output Operator

- To be consistent with the IO library, the syntax of output operator is:

```
ostream&
operator<<(ostream& out, const Your_Class_Type& s){
        // any special logic to prepare object

        // actual output of members
        out << // …

        // return ostream object
        return out;
}
```

Output Operators

- Generally, output operators print the contents of the object with minimal formatting. They should not print a newline.

IO operators: Always Non-Member

- // if operator<< is a member of Sales_item
  - Sales_item item; item << cout;
- The usage is the opposite of normal way we use output operators.

Input operator overloading

- Must deal with end of file/errors.

```
istream&
operator>>(istream& in, Sales_item& s)
{
    double price;
    in >> s.isbn >> s.units_sold >> price;
    // check that the inputs succeeded
    if (in)
        s.revenue = s.units_sold * price;
    else

        s = Sales_item();  // input failed: reset object to default state
    return in;
}
```

- When designing an input operator, decide what to do about error handling.

Arithmetic operators

```
Sales_item& Sales_item::operator+=(const Sales_item& rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}

// assumes that both objects refer to the same isbn
Sales_item
operator+(const Sales_item& lhs, const Sales_item& rhs)
{
    Sales_item ret(lhs);  // copy lhs into a local object that we'll return
    ret += rhs;           // add in the contents of rhs
    return ret;           // return ret by value
}
```

Operator==

```
inline bool
operator==(const Sales_item &lhs, const Sales_item &rhs)
{
    return lhs.units_sold == rhs.units_sold &&
         lhs.revenue == rhs.revenue &&
         lhs.same_isbn(rhs);
}
```

== means objects contain same data.
Also define !=
Classes which define == are easier to use with STL. (e.g. find())

Operator< and Sorting

```
//
// sort.cpp
//
  #include <vector>
  #include <algorithm>
  #include <functional>
  #include <iostream>

 struct associate
 {
  int num;
  char chr;

  associate(int n, char c) : num(n), chr(c){};
  associate() : num(0), chr(' '){};
 };
```

Operator< and sorting

```
bool operator<(const associate &x, const associate &y)
 {
  return x.num < y.num;
 }

ostream& operator<<(ostream &s, const associate &x)
 {
  return s << "<" << x.num << ";" << x.chr << ">";
 }
```

Operator< and Sorting

```
int main ()
  {
   vector<associate>::iterator i, j, k;

   associate arr[20] =
        {associate(-4, ' '), associate(16, ' '),
        associate(17, ' '), associate(-3, 's'),
        associate(14, ' '), associate(-6, ' '),
        associate(-1, ' '), associate(-3, 't'),
        associate(23, ' '), associate(-3, 'a'),
        associate(-2, ' '), associate(-7, ' '),
        associate(-3, 'b'), associate(-8, ' '),
        associate(11, ' '), associate(-3, 'l'),
        associate(15, ' '), associate(-5, ' '),
        associate(-3, 'e'), associate(15, ' ')};
```

```
// Set up vectors
   vector<associate> v(arr, arr+20), v1((size_t)20),
             v2((size_t)20);

    // Copy original vector to vectors #1 and #2
   copy(v.begin(), v.end(), v1.begin());
   copy(v.begin(), v.end(), v2.begin());

    // Sort vector #1
    sort(v1.begin(), v1.end());

    // Stable sort vector #2
   stable_sort(v2.begin(), v2.end());

    // Display the results
   cout << "Original    sort      stable_sort" << endl;
   for(i = v.begin(), j = v1.begin(), k = v2.begin();
      i != v.end(); i++, j++, k++)
   cout << *i << "     " << *j << "     " << *k << endl;

   return 0;
  }
```

```
Output :
Original   sort     stable_sort
<-4; >    <-8; >    <-8; >
<16; >    <-7; >    <-7; >
<17; >    <-6; >    <-6; >
<-3;s>    <-5; >    <-5; >
<14; >    <-4; >    <-4; >
<-6; >    <-3;e>    <-3;s>
<-1; >    <-3;s>    <-3;t>
<-3;t>    <-3;l>    <-3;a>
<23; >    <-3;t>    <-3;b>
<-3;a>    <-3;b>    <-3;l>
<-2; >    <-3;a>    <-3;e>
<-7; >    <-2; >    <-2; >
<-3;b>    <-1; >    <-1; >
<-8; >    <11; >    <11; >
<11; >    <14; >    <14; >
<-3;l>    <15; >    <15; >
<15; >    <15; >    <15; >
<-5; >    <16; >    <16; >
<-3;e>    <17; >    <17; >
<15; >    <23; >    <23; >
```

Subscript operator

- Must be a member-function
- Ordinarily, a class that defines subscript, defines two versions.
  - A nonconst member returning a reference.
  - A const member returning a const reference.

Example

```
#include <vector>
using std::vector;
#include <iostream>
using std::cout; using std::endl;

class Foo {
public:
Foo(): data(100) { for (int i = 0; i != 100; ++i) data[i] = i; }
   int &operator[](const size_t);
   const int &operator[](const size_t) const;
   // other interface members
private:
   vector<int> data;
   // other member data and private utility functions
};
```

Example

```
int& Foo::operator[](const size_t index)
{
      return data[index];   // no range checking on index
}

const int& Foo::operator[](const size_t index) const
{
      return data[index];   // no range checking on index
}

int main() {
   Foo f;

   cout << f[50] << endl;
   return 0;
}
```

Increment and Decrement operators

```
// increment and decrement

  CheckedPtr operator++(int);    // postfix operators
  CheckedPtr operator--(int);

  CheckedPtr& operator++();      // prefix operators
  CheckedPtr& operator--();
```

Increment and Decrement operator

```
// postfix: increment/decrement object but return unchanged value
CheckedPtr CheckedPtr::operator++(int)
{
   // no check needed here, the call to prefix increment will do the check
   CheckedPtr ret(*this);   // save current value
   ++*this;                 // advance one element, checking the increment
   return ret;              // return saved state
}

// prefix: return reference to incremented/decremented object
CheckedPtr& CheckedPtr::operator++()
{
   if (curr == end)
      throw out_of_range
          ("increment past the end of CheckedPtr");

   ++curr;              // advance current state
   return *this;
}
```

++ / --

- Advice: Make them member functions.
- For consistency,
  - Prefix operations should return a reference to the incremented/decremented object
  - Postfix operations should return the “old” value (and not the reference)

Calling prefix/postfix operators explicitly

Mytype.operator++(0); // postfix
Mytype.operator++(); // prefix

Call operator and Function objects

```
#include <iostream>
using std::cout; using std::endl;

struct absInt {
   int operator()(int val) {
      return val < 0 ? -val : val;
   }
};

int main() {
   int i = -42;
   absInt absObj;  // object that defines function call operator
   unsigned int ui = absObj(i);   // calls absInt::operator(int)
   cout << i << " " << ui << endl;
   return 0;
}
```

Function call operator

- Must be a member function
- Can be overloaded for different parameters.
- Objects that have call operator overloaded are often referred to as “Function Objects”

Function Objects

- A **Function object**, often called a **functor** or **functionoid**, is a computer programming construct allowing an object to be invoked or called as if it were an ordinary function, usually with the same syntax.
- An advantage of function objects in C++ is performance because unlike a function pointer, a function object can be inlined.

Template Function Objects

```
class GenericNegate
{
public:
 template <class T> T operator() (T t) const {return -t;}
};

int main()
{
 GenericNegate negate;
 __int64 val = 10000000000i64;
 cout<< negate(5.3333); // double
 cout<< negate(val); // __int64
}
```

Using count_if

```
// determine whether a length of a given word is 6 or more
bool GT6(const string &s){
    return s.size() >= 6;
}

vertor<string>::size_type wc =
            count_if(words.begin(),
                     words.end(), GT6);
```

Using FOs with STL

```
class GT_cls {
public:
          GT_cls(size_t val =0) : bound(val) {}
          bool operator() (const string &s){
                    return s.size() >= bound; }
private:
          std::string::size_type bound;
}

cout << count_if  ( words.begin() , words.end(), GT_cls(6))
       << " words 6 characters or longer" << endl;

for(size_t I = 0; I != 11; ++I){
          cout << count_if  ( words.begin() , words.end(), GT_cls(i))
                 << " words " << I << " characters or longer" << endl;
}
```

Using FOs with STL

```
struct adder {
    double sum;
    adder() : sum(0) {};
    void operator()(double x) { sum += x; }
  };

  vector<double> V;
  ...
  adder result = for_each(V.begin(), V.end(), adder());
  cout << "The sum is " << result.sum << endl;
```

Using FOs with STL

```
vector<int> V(100);
generate(V.begin(), V.end(), rand);
```

A slight deviation

```
// print.hpp
#include <iostream>

template <class T>
inline void PRINT_ELEMENTS(
        const T& collection, const char * optcstr = "" ) {

        typename T::const_iterator pos;
        std::cout << optcstr;
        for ( pos = collection.begin(); pos != collection.end(); ++pos)
                std::cout << *pos << ' ' ;
        std::cout << std::endl;
}
```

Usage: PRINT_ELEMENTS(myvec, "initialized to");

How do we do the same thing using function objects?

STL FOs

- #include <functional>
- Types of function objects.
  - Arithmetic FOs:
    - plus<Type>, minus<type>, multiplies<type>, divides<type>, modulus<type>, negate<type>
  - Relational FOs:
    - equal_to<Type>, not_equal_to<type>, greater<type>, greater_equal<type>, less<type>, less_equal<type>
  - Logical FOs:
    - logical_and<type>, logical_or<type>, logical_not<type>

STL FOs

- Another classification
  - Unary FOs:
    - negate<type> , logical_not<type>
  - Binary FOs:
    - The rest.

Example

```
// plus example
#include <iostream>    // std::cout
#include <functional>  // std::plus
#include <algorithm>   // std::transform

int main () {
  int first[]={1,2,3,4,5};
  int second[]={10,20,30,40,50};
  int results[5];
  std::transform (first, first+5, second, results, std::plus<int>());
  for (int i=0; i<5; i++)
          std::cout << results[i] << ' ';
  std::cout << '\n';
  return 0;
}
```

Using library FOs with Algorithms

```
sort(svec.begin(), svec.end(), greater<string>());
```

- Sorts the container in ascending order.
- The third argument is used to pass a **predicate function** to use to compare elements.

Function Adaptors and FOs

- FAs specialize and extend FOs.
- FA categories
  - Binders: Converts a binary FO to a unary FO by binding one of the operands to a given value
  - Negators: Reverses the truth value of a predicate function.

Function Adaptors (FAs)

- Binder adaptors: bind1st, bind2nd
  - bind1st binds the given value to the first argument of the binary function object. (bind2nd binds to the 2nd)
  - count_if(vec.begin(), vec.end(), bind2nd(less_equal<int>(), 10));

Function Adaptors

- Negators:
  - not1 : Reverses the truth value of a **unary** predicate.
  - not2 : Reverses the truth value of a **binary** predicate.
- Example:
  - count_if(vec.begin(), vec.end(), not1(bind2nd(less_equal<int>(), 10)));

Example (FAs n FOs)

```
#include <functional>
using std::plus; using std::negate;

#include <iostream>
using std::cout; using std::endl;

#include <vector>
#include <algorithm>
using std::count_if; using std::bind2nd; using std::not1; using std::ptr_fun;
using std::less_equal; using std::vector;

#include <iostream>
using std::cin;

#include <string>
using std::string;
```

```
bool size_compare(string s, string::size_type sz)
{
    return s.size() >= sz;
}

int main() {

    cout << plus<int>()(3,4) << endl; // prints 7

    plus<int> intAdd;       // function object that can add two int values
    negate<int>  intNegate;  // function object that can negate an int value

    // uses intAdd::operator(int, int) to add 10 and 20
    int sum = intAdd(10, 20);       // sum = 30

    cout << sum << endl;

    // uses intNegate::operator(int) to generate -10 as second parameter
    // to intAdd::operator(int, int)
    sum = intAdd(10, intNegate(10));  // sum = 0

    cout << sum << endl;
```

```
int arry[] = {0,1,2,3,4,5,16,17,18,19};

  vector<int> vec(arry, arry + 10);

  cout <<
  count_if(vec.begin(), vec.end(),
        bind2nd(less_equal<int>(), 10));
  cout << endl;

  cout <<
  count_if(vec.begin(), vec.end(),
        not1(bind2nd(less_equal<int>(), 10)));
  cout << endl;

  return 0;
}
```

Predicate functions

- Returns bool.
- Should be a pure function: Only depend on its input parameters.
  - F(x,y) should only depend on x,y.
- A predicate class is a FO whose operator() is a predicate.
  - Example : less_equal<int> ()

Make FOs adaptable.

```
list<Widget *> widgetPtrs;
class isOfInterest {
public:
        bool operator()(  const Widget * pw) const;
}

isOfInterest sInteresting;

list<Widget *>::iterator foundit = find_if ( widgetPtrs.begin(),
                                             widgetPtrs.end(),
                                             isInteresting);

If ( foundit != widgetPtrs.end() ) { // found what I was looking for.

}
```

Make FOs adaptable.

```
list<Widget *> widgetPtrs;
class isOfInterest {
public:
        bool operator()(  const Widget * pw) const;
}

isOfInterest sInteresting;

list<Widget *>::iterator foundit = find_if ( widgetPtrs.begin(),
                                             widgetPtrs.end(),
                                             not1(isInteresting)); // ERROR

If ( foundit != widgetPtrs.end() ) { // found what I was looking for.

}
```

Make FOs adaptable.

```
list<Widget *> widgetPtrs;
class isOfInterest : public unary_predicate<Widget *,bool> {
public:
        bool operator()(  const Widget * pw) const;
}

isOfInterest sInteresting;

list<Widget *>::iterator foundit = find_if ( widgetPtrs.begin(),
                                             widgetPtrs.end(),
                                             not1(isInteresting));

If ( foundit != widgetPtrs.end() ) { // found what I was looking for.

}
```

Make FOs adaptable.

```
list<Widget *> widgetPtrs;
class isActuallyBetter :
        public binary_predicate<Widget *, Widget *, bool> {
public:
        bool operator()(const Widget * pw1, const Widget * pw2) const;
}

isActuallyBetter isBetter;

list<Widget *>::iterator foundit = find_if ( widgetPtrs.begin(),
                                             widgetPtrs.end(),
                                             bind2nd(isBetter(), mywidget));

If ( foundit != widgetPtrs.end() ) { // found what I was looking for.

}
```

Conversion Operators

- In addition to defining conversions “to” a class type (using constructors), we can also define conversions “from” a class type.
- Why might we need conversion operators?
  - Our own SmallInt ?

Conversion Operators

- Syntax:
  - operator type();
- Always a member function

Usually should be const

```
class Mystring
{
public:
 Mystring();
 //convert Mystring to a C-string
 operator const char * () { return s; } //...
};

int n = strcmp(mystr, “C-String”); //OK, automatic conversion of str
                                    // to const char *
```

Do not use conversion operators (unless necessary)

```
class Rational {
Public:
        …
        operator double() const;
};

Rational r(1,2);
double d = 0.5 *r;

cout << r << "\t" << d << endl
```

```
#include <iostream>
using namespace std;

class Rational {
  int x, y;

public:
  explicit operator double() const {
    return double(x) / double(y);
  };
  Rational(int x, int y) : x(x), y(y){};
};

int main(void) {
  Rational r(1, 2);
  double d = 0.5 * double(r);
  cout << double(r) << "\t" << d << endl;
}
```

Conversion operators

```
template<class T>
class Array {
public:
        Array (int lb, int hb);
        Array (int size);
        T& operator[] (int index);
…
}

bool operator==(const Array<int>& lhs, const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

for (int i = 0; i < 10; ++i)
        if ( a == b[i] ) …
```

Conversion operators

```
template<class T>
class Array {
public:
        Array (int lb, int hb);
        Array (int size);
        T& operator[] (int index);
…
}

bool operator==(const Array<int>& lhs, const Array<int>& rhs);

Array<int> a(10);
Array<int> b(10);

for (int i = 0; i < 10; ++i)
        if ( a == b[i] ) …
Converted to : if (a == static_cast< Array<int> > (b(i)) …!!
```

Function objects revisited

- Design function objects for pass-by-value

```
template< class InputIterator, class Function >
Function for_each(InputIterator first, InputIterator last,
                Function f);
```

- Keep them small, and don't inherit them.

Conversion operators

- Only one class-type conversion is applied by the compiler. (else, compile time error)
- Used carefully, class-type conversions can greatly simplify code. Used too freely, they can lead to mysterious errors.

Spring Break Reading

- Reading: Sections 9.1, 9.2, 10.1, 10.2,
- Reading assignment: Chapter 14.