Operators

• A simple mechanism for improving program readability...
• …or for making programs completely unreadable!
• Also provides more significant advantages when combined with templates (coming soon)
• “Syntactic sugar” replacing function calls
• Provide “operator” syntax for objects:

    Point p1, p2;
    ...
    if ( p1 == p2 ) {
        ...

Kinds of operators

• Numerical: +, -, *, ++, %, etc.
• Comparison: >, >=, ==, !=, etc.
• Structural: [ ], *, &, etc.
• One operator is more important than the others:
• The assignment operator is provided (at no cost) if you don’t provide one yourself...
• …it might not be what you want.
• Default assignment is by assignment of all member objects, and bitwise copy of all native types.
• Objects containing addresses will be assigned “shallowly”
Arguments of operators

• Operators are functions.
• Operators are (almost) always unary or binary.
• Operators can be members, or regular functions.
• How many arguments does an operator take?

<table>
<thead>
<tr>
<th></th>
<th>Unary</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Regular</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

• It’s an error to declare an operator with the wrong number of arguments.

Member operator syntax

class Rational {
private:
    int m_n, m_d;
public:
    // Member binary operator
    bool operator==( const rational& other) const;
};

// rational.cpp
bool Rational::operator==(const Rational& other)const {
    // This way, 1/2 == 2/4 (should it?)
    return ((m_n * other.m_d) == (m_d * other.m_n));
}
Calling binary member operators

• For a binary member operator, the LHS becomes *this and the RHS becomes the argument:

Point p1, p2;
...
if ( p1 == p2 ) {
  ...
  // Equivalently:
  if ( p1.operator==(p2) ) {
    ...

Unary member operator syntax

class Rational {
  ...
public:
  // Unary negation, as in: x = -y
  Rational operator-() const;
};

  // rational.cpp

Rational Rational::operator-() const
{
  // assumes 2-argument constructor
  return Rational(-m_n, m_d);
}
Non-member operator syntax

class Rational {
  friend ostream& operator<<( ostream& os,
                  const Rational & p);
  ...
};

// rational.cpp
ostream& operator<<( ostream& os,
                  const Rational & r)
{
  os << "(" << r.m_n << "/" << r.m_d << ")";
  return os;
}
...
Rational r1;
cout << r1 << endl;

Calling non-member operators

Rational r1(1, 2);
cout << r1 << endl;

// Equivalently:
operator<<(cout, r1);
cout.operator<<(endl);  // Call to member operator

// FOR ILLUSTRATIVE PURPOSES ONLY! DON’T CODE LIKE THIS

operator<<(cout, r1) << endl;  //Yuck, but this works
operator<<(cout, r1).operator<<(endl) //Also works
Available operators

- Binary arithmetic: +, - , * , / , %
- Unary arithmetic: + , -
- Binary bitwise: & , ^ , |
- Unary bitwise: ~
- Binary logical: && , ||
- Unary logical: !
- Shift (and stream): << , >>
- Comparison (all binary): == , != , > , < , >= , <=
- Assignment: = , += , -= , *= , /= , %= , &= , ^= , |= , >>= , <<=
- Increment (two of each): ++ , --
- Binary structure: []
- Unary structure: * , ->
- Miscellaneous: , (), new, delete, new[], delete[]

Special operators

- These operators must be members:
  =, [], () and ->.
  This ensures the first arguments are lvalues.
- These operators are predefined (i.e. you get them for free): operator=, operator&, operator, (comma)
- The following operators cannot be overloaded:
  :: (scope), . (member selection), .* (member selection through pointer to member)
- Non-member operators must take at least one user-defined type.
Assignment operators

• Should return *this (by ref.), to allow for chaining.
• Almost always needs to check for one condition.

    Rational& Rational::operator=(const Rational& other) {
        if(this == &other) // self assignment
            return *this;

        m_n = other.m_n;
        m_d = other.m_d;
        return *this;
    }

operator= and copy constructor

• If you have a copy c’tor, you should have an operator=.
• Once you have operator=, a copy c’tor is very simple:

    class Rational {
        public:
            Rational(const Rational& other);
            Rational& operator=(const Rational& other);
            ...;
    };

    Rational::Rational(const Rational& other) {
        *this = other;
    }
Common uses of operators

• Most classes need =, many need ==, some need >.
• Numerical classes get natural syntax.
• String classes add +, += for concatenation, [] for access.
• Lookup structures add [] taking key types:

```cpp
class PhoneBook {
    String operator[](const String& key);
    ...
};

PhoneBook pb1;
...
cout << "Bob's number is " << pb1["Bob"] << endl;
```

The dual operator[] idiom

• It’s common to define two operator[]s
• Allows use on left hand side, and on const objects

```cpp
class MyDoubleArray {
    double& operator[](unsigned int i); // return is lvalue
    double operator[](unsigned int i) const; // not lvalue
    ...
};

void myFunc(const MyDoubleArray& src) {
    MyDoubleArray dest;
    // Use non-const op[] on left, const op[] on right
    dest[0] = src[1];
```
++ and -- syntax

class Rational {
...
public:
    Rational& operator--(); // prefix (result is an lvalue)
    const Rational operator--(int); // postfix (not an lvalue)
};

Rational& Rational::operator--()
{
    m_n -= m_d;
    return *this;
}

const Rational Rational::operator--(int)
{
    Rational temp = *this;
    m_n -= m_d;
    return temp;
}

Conversion by constructor

• Defining a constructor which takes one argument of type T gives conversion from T to your type.

class Rational {
private:
    int m_n, m_d;
public:
    Rational(int x) : m_n(x), m_d(1) {}
    ...;
};

void printRational(const Rational & r);

Rational r = 3; // Same as r(3)
printRational(3); // Implicit conversion - 1 level
Digression: conversion by c’tor

• Sometimes, we don’t want conversions:

```cpp
class String {
    String(int i); // initial size??
};
int myfunc( const String& s);

String s1(4); // OK
String s1 = 4; // Confusing
myfunc(4); // Error??

String foo()
{
    ...
    return 0; // probably error?!
}
```

Solution: Explicit c’tors

```cpp
class String {
    explicit String(int i); // initial size
};
int myfunc( const String& s);

String s1 = 4; // Now illegal
String s2(4); // OK
Myfunc(4); // Now illegal

String foo()
{
    ...
    return 0; // Also illegal, but MSVC6 allows this
}
```
Conversion operators

• Defining a constructor which takes one argument of type T gives conversion from T to your type.
• How can we convert the other way?
  1) Ask writer of T to include a constructor
  2) Define (unary) operator T() on our class

class Rational{
  private:
    int m_n, m_d;
  public:
    operator double() const { return ((double) m_n) / m_d; }
    ...;
}
Rational r;
double d = sqrt(r); // Use conversion operator

Ambiguity

• Defining conversion operators and other operators can lead to ambiguities:

class Rational{
  private:
    int m_n, m_d;
  public:
    Rational(int x): m_n(x), m_d(1) {} // m_n(1)
    operator int() const { return (m_n / m_d); } // truncate
    Rational operator+(const Rational& r) const;
    ...
};
Rational r(4);
int x = r + 3; // r -> int or 3 -> Rational??
Functors

- `operator()` can only be defined as a member.
- Classes which define `operator()` are functors.
- `operator()` can take any number of arguments
- Use functors in place of function pointers.

```cpp
class Random {
    // Member binary operator
    int operator()(int lower, int upper);
};

Random rand;
int sample = rand(5, 10);
mak_samples(100, rand);
```

Caveat

- Don’t overload `||` and `&&`
- Clients will expect usual semantics...
- ...and you can’t provide them!

```cpp
class boolObject
{
    bool operator||(const boolObject& other);
};

boolObject b = boolObject::TRUE;
boolObject c();
...
if ( b || c() )...
```
Advice

• Use operators in ways which mimic their native use.
  1) Don’t take or return unexpected types.
  2) Make sure you provide a complete set.
• Use named functions at first, build operators on those later.
• Be careful with conversion to other types.
• Define assignment if you define a copy constructor.