Overloaded operators

Object-Oriented Programming with C++

Overloading operators

- Allows user-defined types to act like built-in types
- Another way to make a function call.

Overloaded operators

Unary and binary operators can be overloaded:

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

operator new   operator delete

operator new[]   operator delete[]

Operators you can't overload

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

`sizeof` `typeid`

`static_cast` `dynamic_cast` `const_cast`

`reinterpret_cast`

Restrictions

- Only existing operators can be overloaded (you can't create a ** operator for exponentiation)
- Operators must be overloaded on a class or enumeration type
- Overloaded operators must
  - Preserve number of operands
  - Preserve precedence

C++ overloaded operator

- Just a function with an operator name!
  - Use the `operator` keyword as a prefix to name

    `operator *` (...)

C++ overloaded operator

- Just a function with an operator name!
  - Use the `operator` keyword as a prefix to name

    `operator *`(...)
- Can be a member function
  - Implicit first argument

    String String::**operator+**(const String& that);

C++ overloaded operator

- Just a function with an operator name!
  - Use the `operator` keyword as a prefix to name

    `operator *(...)`

- Can be a member function
  - Implicit first argument

    `String String::operator+(const String& that);`

- Can be a global (free) function
  - Both arguments explicit

    `String operator+(const String& l, const String& r);`

How to overload

- As member function
  - Implicit first argument
  - No type conversion performed on receiver

Operators as member functions

```
class Integer {
public:
  Integer( int n = 0 ) : i(n) {}
  Integer operator+(const Integer& n) const {
      return Integer(i + n.i);
  }
  ...
private:
   int i;
};
```

Member functions

```
Integer x(1), y(5), z;
x + y;  ====>  x.operator+(y);
```

- Implicit first argument
- Developer must have access to class definition
- Members have full access to all data in class
- No type conversion performed on receiver

```
z = x + y;
z = x + 3;
z = 3 + y;
```

Member functions

```
Integer x(1), y(5), z;
x + y;  ====>  x.operator+(y);
```

- Implicit first argument
- Developer must have access to class definition
- Members have full access to all data in class
- No type conversion performed on receiver

```
z = x + y; √
z = x + 3;
z = 3 + y;
```

Member functions

```
Integer x(1), y(5), z;
x + y; ====> x.operator+(y);
```

- Implicit first argument
- Developer must have access to class definition
- Members have full access to all data in class
- No type conversion performed on receiver

```
z = x + y; √
z = x + 3; √
z = 3 + y;
```

Member functions

```
Integer x(1), y(5), z;
x + y;  ====>  x.operator+(y);
```

- Implicit first argument
- Developer must have access to class definition
- Members have full access to all data in class
- No type conversion performed on receiver

```
z = x + y; √
z = x + 3; √
```

~~`z = 3 + y;`~~

Member functions…

- For binary operators (+, -, *, etc) member functions require one argument.
- For unary operators (unary -, !, etc) member functions require no arguments:

```
Integer operator-() const {
    return Integer(-i);
}
...
z = -x;
```

Member functions…

- For binary operators (+, -, *, etc) member functions require one argument.
- For unary operators (unary -, !, etc) member functions require no arguments:

```
Integer operator-() const {
  return Integer(-i);
}
...
z = -x; // ???
```

Member functions…

- For binary operators (+, -, *, etc) member functions require one argument.
- For unary operators (unary -, !, etc) member functions require no arguments:

```
Integer operator-() const {
  return Integer(-i);
}
...
z = -x; // z.operator=(x.operator-());
```

How to overload

- As a global function
  - –Explicit first argument
  - –Type conversions performed on both arguments
  - –Can be made a friend

Operator as a global function

```
Integer operator+(
    const Integer& lhs,
    const Integer& rhs);
Integer x, y;
x + y     ====> operator+(x, y);
```

- Explicit first argument
- Developer does not need special access to classes
- May need to be a friend
- Type conversions performed on both arguments

Global operators (friend)

```
class Integer {
public:
 friend Integer operator+(const Integer&, const Integer&);
 ...
private:
 int i;
};
Integer operator+(const Integer& lhs, const Integer& rhs)
{
  return Integer( lhs.i + rhs.i );
}
```

Global operators

- Binary operators require two arguments
- Unary operators require one argument
- Conversion:

```
z = x + y;
z = x + 3;
z = 3 + y;
z = 3 + 7;
```

- If you don't have access to private data members, then the global function must use the public interface (e.g. accessors)

Global operators

- Binary operators require two arguments
- Unary operators require one argument
- Conversion:

```
z = x + y; // operator+(x, y)
z = x + 3;
z = 3 + y;
z = 3 + 7;
```

- If you don't have access to private data members, then the global function must use the public interface (e.g. accessors)

Global operators

- Binary operators require two arguments
- Unary operators require one argument
- Conversion:

```
z = x + y; // operator+(x, y)
z = x + 3; // operator+(x, Integer(3))
z = 3 + y;
z = 3 + 7;
```

- If you don't have access to private data members, then the global function must use the public interface (e.g. accessors)

Global operators

- Binary operators require two arguments
- Unary operators require one argument
- Conversion:

```
z = x + y; // operator+(x, y)
z = x + 3; // operator+(x, Integer(3))
z = 3 + y; // operator+(Integer(3), y)
z = 3 + 7;
```

- If you don't have access to private data members, then the global function must use the public interface (e.g. accessors)

Global operators

- Binary operators require two arguments
- Unary operators require one argument
- Conversion:

```
z = x + y; // operator+(x, y)
z = x + 3; // operator+(x, Integer(3))
z = 3 + y; // operator+(Integer(3), y)
z = 3 + 7; // Integer(10)
```

- If you don't have access to private data members, then the global function must use the public interface (e.g. accessors)

Tips: Members vs. Free functions

- Unary operators should be members
- = () [] -> ->* must be members
- All other binary operators as non-members

Argument passing

- If it is read-only pass it in as a const reference (except built-ins)
- Make member functions const that don't change the class (boolean operators, +, -, etc)
- For global functions, if the left-hand side changes pass as a reference (stream inserters)

Return values

- Select the return type depending on the expected meaning of the operator. For example,
  - For `operator+` you need to generate a new object. Return the created object.
  - Logical operators should return `bool` (or `int` for older compilers).

The prototypes of operators

- +-*/%^&|~

The prototypes of operators

- +-*/%^&|~
  - T operator X(const T& l, const T& r);

The prototypes of operators

- +-*/%^&|~
    - T operator X(const T& l, const T& r);
- ! && || < <= == >= >

The prototypes of operators

- +-*/%^&|~
  - T operator X(const T& l, const T& r);
- ! && || < <= == >= >
  - bool operator X(const T& l, const T& r);

The prototypes of operators

- +-*/%^&|~
  - T operator X(const T& l, const T& r);
- ! && || < <= == >= >
  - bool operator X(const T& l, const T& r);
- []

The prototypes of operators

- +-*/%^&|~
  - T operator X(const T& l, const T& r);
- ! && || < <= == >= >
  - bool operator X(const T& l, const T& r);
- []
  - E& T::operator [](int index);

Operators ++ and --

- How to distinguish postfix from prefix?
  - `i++` or `++i`

Operators ++ and --

- How to distinguish postfix from prefix?
  - `i++` or `++i`
- Postfix forms take an int argument -- compiler will pass in 0 as that int

```
class Integer {
public:
     ...
     Integer& operator++();   //prefix++
     Integer operator++(int); //postfix++
     Integer& operator--();   //prefix--
     Integer operator--(int); //postfix--
     ...
};
```

Operators ++ and --

```
Integer& Integer::operator++() {
    this->i += 1;        // increment
    return *this;        // fetch
}
// int argument not used so leave unnamed so
// won't get compiler warnings
Integer Integer::operator++( int ){
    Integer old( *this );   // fetch
    ++(*this);              // increment
    return old;             // return
}
```

Operators ++ and --

```
Integer& Integer::operator++() {
    this->i += 1;       // increment
    return *this;       // fetch
}
// int argument not used so leave unnamed so
// won't get compiler warnings
Integer Integer::operator++( int ){
    Integer old( *this );   // fetch
    ++(*this);              // increment
    return old;             // return
}
```

Operators ++ and --

```
Integer& Integer::operator++() {
    *this += 1;          // increment
    return *this;        // fetch
}
// int argument not used so leave unnamed so
// won't get compiler warnings
Integer Integer::operator++( int ){
    Integer old( *this );   // fetch
    ++(*this);              // increment
    return old;             // return
}
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
    x++;
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
    x++;
       // calls x.operator++(0);
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
    x++;
       // calls x.operator++(0);
    --x;
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
    x++;
       // calls x.operator++(0);
    --x;
       // calls x.operator--();
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
    x++;
       // calls x.operator++(0);
    --x;
       // calls x.operator--();
    x--;
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
    x++;
       // calls x.operator++(0);
    --x;
       // calls x.operator--();
    x--;
       // calls x.operator--(0);
```

Using the overloaded ++ and --

```
// decrement operators similar to increment

    Integer x(5);
    ++x;
       // calls x.operator++();
    x++;
       // calls x.operator++(0);
    --x;
       // calls x.operator--();
    x--;
       // calls x.operator--(0);
```

- User-defined prefix is more efficient than postfix.

Relational operators

- implement != in terms of ==
- implement `>`, `>=`, `<=` in terms of `<`

```
class Integer {
public:
    ...
    bool operator==( const Integer& rhs ) const;
    bool operator!=( const Integer& rhs ) const;

    bool operator<( const Integer& rhs ) const;
    bool operator>( const Integer& rhs ) const;
    bool operator<=( const Integer& rhs ) const;
    bool operator>=( const Integer& rhs ) const;
}
```

Relational operators

```
bool Integer::operator==( const Integer& rhs ) const {
   return i == rhs.i;
}
// implement lhs != rhs in terms of !(lhs == rhs)
bool Integer::operator!=( const Integer& rhs ) const {
   return !(*this == rhs);
}

bool Integer::operator<( const Integer& rhs ) const {
   return i < rhs.i;
}
```

Relational operators...

```
// implement lhs > rhs in terms of lhs < rhs
bool Integer::operator>( const Integer& rhs ) const {
   return rhs < *this;
}
// implement lhs <= rhs in terms of !(rhs < lhs)
bool Integer::operator<=( const Integer& rhs ) const {
   return !(rhs < *this);
}
// implement lhs >= rhs in terms of !(lhs < rhs)
bool Integer::operator>=( const Integer& rhs ) const {
   return !(*this < rhs);
}
```

Operator []

- Must be a member function
- Single argument
- Implies that the object acts like an array, so it should return a reference

```
Vector v(100);   // create a vector of size 100
v[10] = 45;
```

Note: if returned a pointer you would need to do:

```
*v[10] =45;
```

See: vector.h, vector.cpp

Copying vs. Initialization

```
MyType b;
MyType a = b;
a = b;
```

Example: CopyingVsInitialization.cpp

Automatic operator= creation

- The compiler will automatically create one if it’s not explicitly provided.
- *memberwise assignment*

- Example: AutomaticOperatorEquals.cpp

Assignment operator

- Must be a member function
- Return a reference to *this

```
A = B = C;
// executed as A = (B = C);
```

Assignment operator

- Must be a member function
- Return a reference to *this

```
A = B = C;
// executed as A = (B = C);
```

- Be sure to assign to all data members: pointers…

Assignment operator

- Must be a member function
- Return a reference to *this

```
A = B = C;
// executed as A = (B = C);
```

- Be sure to assign to all data members: pointers…
- Check for self-assignment

Assignment operator skeleton

```
T& T::operator=( const T& rhs ) {
    // check for self assignment
    if ( this != &rhs ) {
        // perform assignment
    }
    return *this;
}
```

//This checks address, not value (*this != rhs)

Assignment operator

- For classes with *dynamically* allocated memory declare an assignment operator (and a copy constructor)
- To prevent assignment, explicitly declare `operator=` as private, or use `=delete;`

Value classes

- Appear to be primitive data types

Value classes

- Appear to be primitive data types
- Passed to and returned from functions

Value classes

- Appear to be primitive data types
- Passed to and returned from functions
- Have overloaded operators (often)

Value classes

- Appear to be primitive data types
- Passed to and returned from functions
- Have overloaded operators (often)
- Can be converted to and from other types

Value classes

- Appear to be primitive data types
- Passed to and returned from functions
- Have overloaded operators (often)
- Can be converted to and from other types
- Examples: Complex, Date, …

User-defined type conversions

- A conversion operator can be used to convert an object of one class into
  - an object of another class
  - a built-in type
- Compilers perform implicit conversions using:
  - Single-argument constructors
  - implicit type conversion operators

Single argument constructors

```
class PathName {
  string name;
public:
  // or could be multi-argument with defaults
  PathName(const string&);
  ~ PathName();
};
...
string abc("abc");
PathName xyz(abc); // OK!
xyz = abc;         // OK abc => PathName
```

Example: AutomaticTypeConversion.cpp

Prevent implicit conversions

- New keyword: *explicit*

```
class PathName {
  string name;
public:
  explicit PathName(const string&);
  ~ PathName();
};
...
string abc("abc");
PathName xyz(abc); // OK!
xyz = abc;         // error!
```

Example: ExplicitKeyword.cpp

Conversion operations

- Operator conversion
  - Function will be called automatically
  - Return type is same as function name

```
class Rational {
public:
   ...
   operator double() const;  // Rational to double
}
Rational::operator double() const {
  return numerator_/(double)denominator_;
}
Rational r(1,3); double d = 1.3 * r; // r=>double
```

General form of conversion ops

- X::operator *T*()
  - –Operator name is any type descriptor
  - –No explicit arguments
  - –No return type
  - –Compiler will use it as a type conversion from $X \Rightarrow T$

C++ type conversions

- Built-in conversions
  - Primitive

    `char` $\Rightarrow$ `short` $\Rightarrow$ `int` $\Rightarrow$ `float` $\Rightarrow$ `double`

    $\Rightarrow$ `int` $\Rightarrow$ `long`

  - Any type T

`T` $\Rightarrow$ `T&`	`T&` $\Rightarrow$ `T`	`T*` $\Rightarrow$ `void*`
`T[]` $\Rightarrow$ `T*`	`T*` $\Rightarrow$ `T[]`	`T` $\Rightarrow$ `const T`

C++ type conversions

- Built-in conversions
  - Primitive

    `char` $\Rightarrow$ `short` $\Rightarrow$ `int` $\Rightarrow$ `float` $\Rightarrow$ `double`

    $\Rightarrow$ `int` $\Rightarrow$ `long`

  - Any type T

`T`$\Rightarrow$ `T&`	`T&`$\Rightarrow$ `T`	`T*`$\Rightarrow$ `void*`
`T[]`$\Rightarrow$ `T*`	`T*`$\Rightarrow$ `T[]`	`T`$\Rightarrow$ `const T`

- User-defined T $\Rightarrow$ C
  - if `C(T)` is a valid constructor call for `C`
  - if `operator C()` is defined for `T`

C++ type conversions

- Built-in conversions
  - Primitive

    `char` $\Rightarrow$ `short` $\Rightarrow$ `int` $\Rightarrow$ `float` $\Rightarrow$ `double`

    $\Rightarrow$ `int` $\Rightarrow$ `long`

  - Any type T

`T` $\Rightarrow$ `T&`	`T&` $\Rightarrow$ `T`	`T*` $\Rightarrow$ `void*`
`T[]` $\Rightarrow$ `T*`	`T*` $\Rightarrow$ `T[]`	`T` $\Rightarrow$ `const T`

- User-defined T $\Rightarrow$ C
  - if `C(T)` is a valid constructor call for `C`
  - if `operator C()` is defined for `T`
- BUT
  - See: TypeConversionAmbiguity.cpp

Do you want to use them?

- In general, be careful!
  - Cause lots of problems when functions are called unexpectedly.
- Use explicit conversion functions. Instead of using the conversion operator, declare a member function in `class Rational`:

```
double to_double() const;
```

Overloading and type conversion

- C++ checks each argument for a "best match"
- Best match means cheapest
  - Exact match is cost-free
  - Matches involving built-in conversions
  - User-defined type conversions

Overloading

- Just because you can overload an operator doesn't mean you should.
- Overload operators when it makes the code easier to read and maintain.