Today

- Operator Overloading
- Vector class

Assignments

- HW2 out (due thurs)
Operator Overloading

In C++ we can overload operators for our own classes

- this can improve code readability/writability
- unless the operators aren’t “natural” for the class

```cpp
Rectangle r1(10, 10);
Rectangle r2(10, 20);
...
if (r1 < r2)
  // do something
else
  // do something different
```

Our plan:

- relational comparators (\(<\), \(==\), etc.)
- basic arithmetic operators (binary and unary)
- increment and decrement (both versions)
- friend functions

⇒ You will implement a variety of operators for HW2
In general, to overload an operator, you write a function like:

\[
\text{return-type operator } \text{op} (\text{param-list}) \nonumber
\]

- where \( \text{op} \) is \(<\), +, ==, etc.
- you might have already done this in CPSC 122 with \text{operator}=(...) 
- number and type of parameters depends on \text{operator}
- and similarly with the return type

Most operators are either **unary** (one arg) or **binary** (two args)

- as a member function, current object (\text{this}) always implicitly the 1st arg
- in a binary operator, often refer to first arg as \text{lhs} (left-hand side)
- ... and second as \text{rhs} (right-hand side)

Can call the operator directly or using the \text{operator} function

- e.g., using the rectangle classes \text{operator}(<(...)) function

\[
\text{cout} \ll r1 < r2 \ll \text{endl;} \quad \quad \quad \quad \quad \quad \quad \text{// prints: true (1)}
\]
\[
\text{cout} \ll r1.\text{operator}(r2) \ll \text{endl;} \quad \quad \quad \quad \quad \quad \quad \text{// prints: true (1)}
\]

- you would never actually do this ... but highlights operators are just functions
Relational Comparison Operators

Adding `operator<` to our Rectangle class ...

- in this case, it is similar to `less_than()`

```cpp
class Rectangle
{
    public:
        ...
        bool operator<(const Rectangle& rhs) const;
    ...
}
```

And then in the implementation file ...

```cpp
bool Rectangle::operator<(const Rectangle& rhs) const
{
    return area() < rhs.area();
}
```

**Exercise:** Implement `==` and `>` reusing `<` ...

```cpp
bool Rectangle::operator==(const Rectangle& rhs) const
{
    return !(this < rhs) && !(rhs < *this);
}
```

```cpp
bool Rectangle::operator>(const Rectangle& rhs) const
{
    return rhs < *this;
}
```

You can implement all the relational comparators with only `<` !!!!
Arithmetic Operators

Adding operator+ to Rectangle

What should the behavior of operator+ be? (eg, Rectangle r = r1 + r2;)

- ... takes a rhs rectangle
- ... adds lengths and widths of *this and rhs
- ... creates a new rectangle object (to hold the result of the addition)
- ... returns the new rectangle (e.g., to store or print result)

Q: What is the return type of operator+?

class Rectangle
{
  public:
  ...
  Rectangle operator+(const Rectangle& rhs) const;
  ...
};

Q: How should operator+ implemented?

Rectangle Rectangle::operator+(const Rectangle& rhs) const
{
  Rectangle result;
  result.length = length + rhs.length;
  result.width = width + rhs.width;
  return result;
}
Unary Operators

Adding (unary minus) operator- to “rotate” a Rectangle

Similar to addition, doesn’t modify the object, gives a new object ... e.g.:

```cpp
int x = 4;
int y = -x;
cout << x << endl;  // prints 4
cout << y << endl;  // prints -4
```

Defining the overloaded operator

```cpp
class Rectangle
{
public:

...  // ... incomplete
Rectangle operator-() const;

...  // ... incomplete
};
```

And the implementation ...

```cpp
Rectangle Rectangle::operator-() const
{
Rectangle result;
result.width = length;
result.length = width;
return result;
}
```
Increment (++) and Decrement (--) operators

Recall the difference between prefix and postfix ++

```cpp
int x = 1;
int y = ++x;
int z = x++;
cout << "x = " << x << endl; // prints 3
cout << "y = " << y << endl; // prints 2
cout << "z = " << z << endl; // prints 2
```

- increment modifies value of `x` ... not true with unary minus and addition
- prefix version increments `x`, returns incremented value
- postfix version returns `x`, then increments `x`

postfix requires more work (more operations) ...

```cpp
class Rectangle
{
    public:
        ...
        Rectangle& operator++(); // prefix version
        Rectangle operator++(int); // postfix version
        ...
};
Rectangle& Rectangle::operator++()
{
    ++length;
    ++width;
    return *this;
}
Rectangle Rectangle::operator++(int)
{
    Rectangle tmp = *this; // create a temporary copy
    ++length;
    ++width;
    return tmp;
}
```
Overloaded operators as non-member function (friends)

A **friend** function of a class is:

- a non-member function
- that can access protected & private class members

Example:

```cpp
class Rectangle
{
    public:
    ...

    friend Rectangle operator+(const Rectangle& lhs, const Rectangle& rhs);

    ...
};

// the function implementation ...
Rectangle operator+(const Rectangle& lhs, const Rectangle& rhs)
{
    Rectangle result;
    result.length = lhs.length + rhs.length;
    result.width = lhs.width + rhs.width;
    return result;
}
```

Help provide greater control of the left-hand versus right-hand operand types
Stream insertion and extraction operators

Stream insertion (\(<<\)) and extraction (\(>>\)) must be non-member functions!

- e.g., for: `cout << r1; or cin >> r1;`

Example

```cpp
class Rectangle
{
    public:
    ...
    friend std::ostream& operator<<(std::ostream& out, const Rectangle& rhs);
    friend std::istream& operator>>(std::istream& in, Rectangle& rhs);
};
```

```cpp
std::ostream& operator<<(std::ostream& out, const Rectangle& rhs)
{
    out << rhs.length << " " << rhs.width;
    return out;
}

std::istream& operator>>(std::istream& in, Rectangle& rhs)
{
    in >> rhs.length;
    in >> rhs.width;
    return in;
}
```

Q: Explain the signatures (const, references, return types)

Q: Why do we return the streams?
C++ Vector Basics

The vector class encapsulates an implementation of a “resizable array”

Some of the basic member functions:

- `void push_back(const value_type& val);` — add val to end
- `size_t size() const;` — get length (int n = int(v.size()))
- `bool empty() const;` — true if size is zero
- `void clear();` — removes all elements
- `const_ref at(size_type n) const;` — get element at pos n
- `const_ref operator[](size_type n) const;` — eg: x = v[2];

The vector class is a “parametric” (or “generic”) type ...

```cpp
// create a vector of ints
vector<int> xs;
xv.push_back(10);
xv.push_back(20);
assert(xs[0] == 10);

// create a vector of strings
vector<string> ys;
ys.push_back("foo");
ys.push_back("bar");
ys.push_back("baz");
```

Note that need to use `std::vector` as opposed to just `vector`
Removing an element with the `erase` function

```cpp
// assuming v.size() >= 1
v.erase(v.begin()); // removes first element

// assuming v.size() >= 2
v.erase(v.begin() + 1); // removes second element

// for some int variable i
v.erase(v.begin() + i); // removes i-th element
```

Need to watch out for vector length when removing elems!

Inserting elements within a vector using the `insert` function

```cpp
vector<int> v; // creates {}

v.insert(v.begin(), 13); // result is {13}

v.insert(v.begin() + 1, 15); // result is {13, 15}

v.insert(v.begin() + 0, 11); // result is {11, 13, 15}

v.insert(v.begin() + 1, 12); // result is {11, 12, 13, 15}

v.insert(v.begin() + 3, 14); // result is {11, 12, 13, 14, 15}
```

Again, be careful of vector length

- if 2 element vector (e.g., \{13, 14\})
- inserting at index 3 or higher, will result in a segmentation fault