Today

- Intro to Sequence ADT
- C++ Odds and Ends for HW-2

Assignments

- EX-3
- HW-1 due
- HW-2 out
**Basic Sequence ADT**

**Sequence:**
- Similar to an array, but size can “grow” and “shrink” (unlike an array)
- Ordered collection of items accessed/updated by index (duplicates allowed)
- Can store values of any type (e.g., a sequence of ints, of strings, ...)

**Basic operations supported by Sequence**
- `size()`: gives the number of elements currently in sequence
- `insert(v, i)`: insert value `v` at position `i` (shifts values)
- `clear()`: removes all values from sequence
- `erase(i)`: removes value at index `i` from sequence (shifts values)
- `contains(v)`: checks if value `v` is a member of the sequence
- `operator[i]`: access/update sequence elements via the `[i]` operator

We’ll do 2 implementations: linked list (HW2) and Resizable arrays (HW3)

To implement Sequence, need to understand some C++ specifics:
- operator overloading
- dynamic vs static binding and pure virtual functions (and abstract classes)
- templates
- exceptions
- essential operations
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); // width, height
Rectangle r2(10, 20);
... 
if (r1 < r2) // area comparison
  // do something
else
  // do something different
```

Our plan:

- relational comparators (<, ==, etc.)
- basic arithmetic operators (binary and unary)
- friend functions
In general, to overload an operator, you write a function like:

```cpp
return-type operator op(param-list)
```

- where `op` is `<`, `+`, `==`, etc.
- you did this in CPSC 122 with `operator=(...)`
- number and type of parameters depends on operator
- return type depends on operator

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

- as a member function, current object (`this`) always implicitly the 1st arg
- in a binary operator, I'll often refer to first operand as `lhs` (left-hand side)
- ... and second as `rhs` (right-hand side)

Operator can be called directly (preferred!) or with operator function

```cpp
cout << (r1 < r2) << endl;    // prints: true (1)
cout << r1.operator<(r2) << endl; // prints: true (1)
```

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

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

```cpp
class Rectangle {
private:
    int width, height;
public:
    ...
    bool operator<(const Rectangle& rhs) const;
    ...
}
```

- in a call `r1 < r2` ...
- the **lhs** is `r1` (which is the current object) and the **rhs** is `r2`

And then in the implementation file ...

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

**Exercise:** Implement `>` and `==` without using `area()` ... i.e., use `<` and `*this`

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

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

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

**Adding operator+ to Rectangle**

What should the behavior of `operator+` be? \( \text{eg: } r3 = r1 + r2; \)

- ... takes a `rhs` rectangle \((r2)\)
- ... adds lengths and widths of `*this` \((r1)\) and `rhs` \((r2)\)
- ... creates a new rectangle object (to hold the result of the addition)
- ... returns the new rectangle (copied into \(r3\))

The signature:

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

Q: How can we implement `operator+`?

```cpp
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:
...
    Rectangle operator-() const;
...
};
```

And the implementation ...

```cpp
Rectangle Rectangle::operator-() const
{
    Rectangle result;
    result.width = length;
    result.length = width;
    return result;
}
```
**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&, const Rectangle&);
...
};

// 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;
}
```

**Overloaded operators can always be implemented outside the class**

- provides greater control of lhs versus rhs operand types
- e.g., a < operator to compare rectangles and circles
- don’t need to be friend functions unless access protected/private members