# Day 30 (Fri 04/08)

- exercise 29 review
- day 30 recap questions
- exercise 30

## Announcements/reminders

✳ - "Midterm Project Contributions" survey on Gradescope
  - **must** be submitted this evening by 11pm

→ - HW6 due Wednesday 4/13 by 11pm
  - written homework, late submissions not allowed

# Exercise 29

Overloading the output stream insertion operator for the Complex class

```
// in complex.h (in the Complex class definition)
friend std::ostream &operator<<(std::ostream &out, const Complex &c);

// in complex.cpp
std::ostream &operator<<(std::ostream &out, const Complex &c) {
  out << c.rel << " + " << c.img << "i";
  return out;
}
```

```
(((cout << a) << b) << c);
```

# Exercise 29

## Copy constructor and assignment operator

```
// in complex.cpp

Complex::Complex(const Complex& rhs)
  : rel(rhs.rel), img(rhs.img) {
}

Complex &Complex::operator=(const Complex& rhs) {
  if (this != &rhs) {  ←
    rel = rhs.rel;
    img = rhs.img;
  }
  return *this;
}
```

# Exercise 29

Overloaded operators for arithmetic, in complex.h. Note that these really should be defined as const member functions, since they don't modify the left hand object.

```
    Complex operator+(const Complex& rhs) const;
    Complex operator-(const Complex& rhs) const;
    Complex operator*(const Complex& rhs) const;
    Complex operator*(const float& rhs) const;
    Complex operator/(const Complex& rhs) const;
```

# Exercise 29

Implementations of arithmetic operators in complex.cpp

```
Complex Complex::operator+(const Complex& rhs) const {
  Complex sum(rel+rhs.rel, img+rhs.img);
  return sum;
}

Complex Complex::operator-(const Complex& rhs) const {
  return *this + (rhs * -1.0f);
}
```

# Exercise 29

## Implementation of multiplication

```
Complex Complex::operator*(const Complex& rhs) const {
  float a = rel;
  float b = img;
  float c = rhs.rel;
  float d = rhs.img;
  // (a+bi) * (c+di) = a*c + (a*d)i + (b*c)i + (b*d)i^2
  //                 = (a*c - b*d) + (a*d + b*c)
  return Complex(a*c - b*d, a*d + b*c);
}
```

-1

# Exercise 29

Non-member * operator for float times Complex

```
// in complex.h
friend Complex operator*(float lhs, const Complex &rhs);

// in complex.cpp
Complex operator*(float lhs, const Complex &rhs) {
  return rhs * lhs;
}
```

# Day 30 recap questions

1. What is difference between initialization and assignment?
2. Does the line f2 = f1; use initialization or assignment (assume Foo is a class and f1 and f2 are both of type Foo)?
3. Does the line Foo f2 = f1; use initialization or assignment (assume Foo is a class and f1 is of type Foo)?
4. What is a shallow copy and what is a deep copy?
5. What is the rule of 3?

1. What is difference between initialization and assignment?

Initialization: a constructor is called when an object's lifetime begins.

Assignment: the = operator (assignment) is used to assign new data to an existing object, replacing its current contents.

Examples:

```
std::string s("hello");               // initialization of s
std::string s2 = "hello again"; // initialization of s2
std::string s3;                         // initialization of s3 using default constructor
s3 = s;                                  // assignment to s3
```

2. Does the line f2 = f1; use initialization or assignment (assume Foo is a class and f1 and f2 are both of type Foo)?

Assignment. It is not a variable declaration of f2, so f2 has already been initialized.

3. Does the line Foo f2 = f1; use initialization or assignment (assume Foo is a class and f1 is of type Foo)?

Initialization.  This is a variable declaration for f2, and f1 is being provided as the initial value, so the copy constructor is called to initialize f2 with f1's contents.

4. What is a shallow copy and what is a deep copy?

Deep copy: replicate dynamically allocated objects/arrays. no sharing

Shallow copy: just copy pointers to dynamically allocated objects/arrays. sharing of representation

```
class CBuf {
private:
  char *buf; int capacity;
public:
  CBuf(int capacity)
    : buf(new char[capacity])
    , capacity(capacity) { }
  // ...other member functions...
};
```

Copy constructor doing deep copy:

```
CBuf(const CBuf &other)
  : buf(new char[other.capacity]))
  , capacity(other.capacity) {
  for (int i = 0; i < capacity; i++)
    { buf[i] = other.buf[i]; }
}
```

alloc own representation object

copy data

Copy constructor doing shallow copy:

```
CBuf(const CBuf &other)
  : buf(other.buf), capacity(other.capacity)
{ }
```

copy pointer value → sharing rep.

Shallow copy means two objects have pointers to the same dynamically allocated array? (Which object should delete it?)

# Shallow copy vs. deep copy

Shallow copy tends to be problematic because either

- multiple objects try to deallocate the dynamic memory (double free, a serious memory error)

- no object tries to deallocate the dynamic memory (memory leak, also a fairly serious bug)

If you are implementing a class that manages dynamic memory, the copy constructor and assignment operator should do deep copy.

5. What is the rule of 3?

If a class has a non-trivial destructor (e.g., the destructor deletes a dynamically allocated object or array), then it also needs*

- a copy constructor

- an assignment operator

Both of these should do a deep copy.

"disable value semantics"

* or, should prohibit copy ctor & assignment op from being used

make them private

```
class CBuf {
private:
  char *buf; int capacity;
public:
  CBuf(int capacity) : buf(new char[capacity]) , capacity(capacity) { }

  CBuf(const CBuf &other)
    : buf(new char[other.capacity]), capacity(other.capacity) {
    for (int i = 0; i < capacity; i++) { buf[i] = other.buf[i]; }
  }

  CBuf &operator=(const CBuf &rhs) {  // Note: subtle bug here!
    delete[] buf;
    buf = new char[rhs.capacity]; capacity = other.capacity;
    for (int i = 0; i < capacity; i++) { buf[i] = other.buf[i]; }
    return *this;
  }
  // ...other member functions...
};
```

deep copy

deep copy

We can't rule out the possibility that an object might be assigned to itself!

While this wouldn't happen explicitly, it could happen fairly easily because of references:

```
void foo(CBuf &a, CBuf &b) {
  if ( /* some condition */ ) {
    a = b;     // do we really know that a and b refer to different objects?
  }
}
```

x = x ;

// buggy version of CBuf assignment operator

```
  CBuf &operator=(const CBuf &rhs) {  // Note: subtle bug here!
    delete[] buf;
    buf = new char[rhs.capacity];
    capacity = other.capacity;
    for (int i = 0; i < capacity; i++) { buf[i] = other.buf[i]; }
    return *this;
  }
```

Think about what happens if rhs and *this are the same object!

The character array is deleted, but then we try to copy data from the deleted character array.

// fixing the bug

```
 CBuf &operator=(const CBuf &rhs) {  // Note: subtle bug here!
   if (this != &rhs) {
     delete[] buf;
     buf = new char[rhs.capacity];
     capacity = other.capacity;
     for (int i = 0; i < capacity; i++) { buf[i] = other.buf[i]; }
   }
   return *this;
 }
```

Now the assignment properly does nothing if rhs and *this are the same object.

✓ You should get in the habit of using this idiom when you implement assignment operators.