601.220 Intermediate Programming

Spring 2023, Day 30 (April 7th)

Today's agenda

- Review exercise 29
- Day 30 recap questions
- Exercise 30

Reminders/Announcements

- HW6 is due **this evening** by 11 pm
  - Written assignment, no late submissions
- HW7 is due Friday, April 14th by 11 pm
- Final project team formation (2–3 members per team)
  - Submit the Google form in Piazza post 571 (pinned) by **11 am on Tuesday, April 11th**
  - If you aren't registered on a team by the deadline, you will be assigned to a team

Exercise 29 review

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;
  }
```

Exercise 29 review

Copy constructor and assignment operator

```
// in complex.cpp

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

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

Exercise 29 review

Overloaded operators for arithmetic, in `complex.h` (in the class definition for the `Complex` class):

```
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;
```

Since these are defined as member functions, they only need one parameter, which is the right-hand-side operand. (The left hand `Complex` object in the expression will be the receiver object.)

Exercise 29 review

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 review

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);
}
```

Exercise 29 review

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;
  }
```

delegate

This operator can't be a member function because the value on the left-hand-side is not an object. Also, this function technically doesn't need to be a `friend` because it invokes the public `Complex` times `float` operator.

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 ctor
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`)?

```
Foo f2(f1);
```

Initialization. This is a variable declaration of `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?

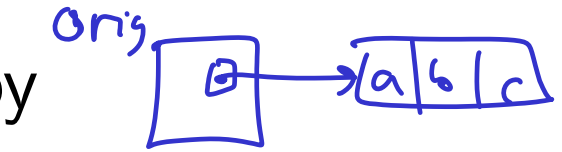
Deep copy: replicate dynamically-allocated objects/arrays.

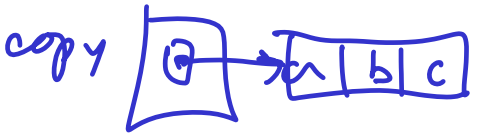
Shallow copy: just copy pointers to dynamically-allocated objects/arrays.

Example class:

```
cass CBuf {
private:
  char *buf; int capacity;
pubic:
  CBuf(int capacity)
    : buf(new char[capacity]), capacity(capacity) { }
  CBuf(const CBuf &other); // copy ctor
  // ...other member functions...
};
```

Copy constructor using deep copy



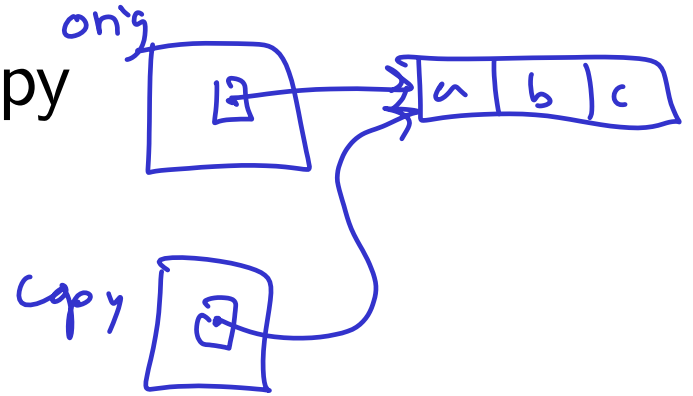



```
CBuf::CBuf(const CBuf &other)
  : buf(new char[other.capacity])
  , capacity(other.capacity) {
  for (int i = 0; i < capacity; i++) {
    buf[i] = other.buf[i];
  }
}
```

copy contents

The new object will have its own dynamically-allocated array, distinct from the original object.

Copy constructor using shallow copy




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

Shallow copy means two objects have pointers to the same dynamically-allocated array. If one object modifies the array, the changes are visible in the other object (because they are "sharing" the array.) Also: which object's constructor 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*.

Disabling value semantics

Alternately: a class with a nontrivial destructor could *prohibit* the copy constructor and assignment operator from being used by defining them in the class definition as `private`, and then not defining them.

The disadvantage of this approach is that the class will not have *value semantics*. So you can't copy, assign, pass by value, return by value, etc.

A subtle issue with assignment

lhs = rhs;

Consider the following assignment operator: ✗ lhs = lhs;

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

Can you spot the problem? (It is very subtle.)

Guarding against self-assignment

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

While this (probably) 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?
  }
}
```

Buggy version of assignment operator

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

Think about what happens when `rhs` and `*this` are the same object. The character array is deleted, but then we try to copy data from the uninitialized newly-allocated character array.

Fixing the bug

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

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

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

Exercise 30

- Linked list impementation in C++
- Implementation of copy constructor and assignment operator using deep copy
- Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes