CS 246 Spring 2018 — Tutorial 5

June 6, 2018

Summary

1 Destructor

- The destructor is a method which is called when a object is destroyed. This is either when it is heap allocated and `delete` is called on it, or when it goes out of scope.
- A default destructor is provided for us by the compiler. This destructor will call the destructors for all fields that are objects. Note that it will not call `delete` on fields which are pointers, because pointers are not objects.
- Consider the following structure

```
struct BigInteger {
    // Digits are listed in *reverse order*
    // So a BigInteger representing 123 has
    // {'3', '2', '1'} as the value of digits
    // Also, we are using char arrays to save space.
    char *digits;

    // As long as (size <= capacity) we are good
    int size;
    int capacity;
```

[1]Note: this concept, comparing to C++ standard, is greatly simplified.

```
    BigInteger(int value = 0);
}
```

- In our structure above, the array which `digits` points at will not be freed when the current `BigInteger` is destroyed.
- We need to write our own destructor in this case. For the `BigInteger` struct:

```
~BigInteger() {
    delete [] digits;
}
```

- **Question:**
  - Why don't we set `digits` to `nullptr`?
- Destructors do not have a return type.

2 Copy Constructors

- What happens when we run the following code?

```
// In some function...
BigInteger n = 10;
BigInteger *ptr_to_copy = new BigInteger{n};

delete ptr_to_copy;
```

- If we run this code, we might see that `n` now has garbage values in some of its digits. Why?
- The pointer `n.digits` is now a dangling pointer because the pointers were deleted when we deleted `n`.
- By default, the copy constructor copies all fields **by value**. This means that when pointers are copied, the value (address) of the original pointer is copied to the new pointer and **they both point to the same memory**. This is known as a **shallow copy**.
- We do not want these objects to be sharing memory. We want the copy constructor to create a copy of the whole array, not just the address (This is known as a **deep copy**).
- To do this, we need to implement our own copy constructor.

```
BigInteger::BigInteger(const BigInteger &in)
    : size{in.size},
      capacity{in.capacity},
      digits {new char[capacity]} {
    for (int i = 0; i < size; ++i) {
        digits[i] = in.digits[i];
```

```
    }
}
```

- The copy constructor is called in three situations:
  - When initializing an object from another object.
  - When passing an object as an argument to a function by value.
  - When returning an object by value from a function.

3 Copy Assignment Operator

- What happens when we run the following code?

```
BigInteger n1{22};
BigInteger *n2 = new BigInteger{42};
n1 = *n2;
delete n2;
```

- Similar to the copy constructor example, we might get garbage values printed.
- The reason is similar to the case with the copy constructor: The compiler gives a default version of the copy assignment operator, but it only performs a shallow copy, not a deep copy.

3.1 Copy-and-Swap Idiom

- There's a straight-forward way to implement copy assignment operator:

```
BigInteger &BigInteger::operator=(const BigInteger &in){
    if (this != &in) {
        BigInteger copy = in;

        swap(digits, copy.digits);
        size = copy.size;
        capacity = copy.capacity;
    }

    return *this;
}
```

- The way this assignment operator is written is referred to as the **copy-and-swap** idiom.
- The copy-and-swap idiom is a method to copy over data by swapping our data with the data we want to copy.
- In a copy assignment operator, we are first creating a local **deep copy** of the `BigInteger` we are copying. This calls the copy constructor (which we have written to do a **deep copy**).

- We then **swap** our `digits` pointer with the `digits` pointer of the `BigInteger copy`. This means that our `BigInteger` gets the data we want, and the `BigInteger copy` gets our old data.
- Since the `BigInteger copy` is on the stack, it will go out of scope when we exit the function. This means that when that happens, our old data will be automatically deleted by the destructor.

4 Lvalues and Rvalues[2]

- An **lvalue** is any entity which has an address accessible from code. They get their name because an lvalue is originally defined to be a value which can occur on the left side of an assignment expression[3].
  - An lvalue reference is denoted by `&`.
- An **rvalue** is any entity which is not an lvalue. They get their name because an rvalue can only occur on the right side of an assignment expression.
  - An rvalue reference is denoted by `&&`.
- An rvalue reference can used for extending the lifetime of temporary objects, while allowing the user to modify the value.
- However, in this course, you only need to know one use case of rvalue references: as the only parameter to move constructors and move assignment operators.
- The overload resolution mechanism identifies if a value is an rvalue or not. If it is, the function (or method) accepting a rvalue reference is called. For example, a value returned from a function is an rvalue:

```
BigInteger giveMeBigInteger() {
    BigInteger n;
    return n;
}

// assuming that no optimization is enabled, calls BigInteger(BigInteger &&)
BigInteger n{giveMeBigInteger()};
```

5 Move Constructor

- Suppose we have the following operator overload:

[2]Note: this concept, comparing to C++ standard, is greatly simplified.

[3]That's not entirely accurate. Const values cannot appear on the left hand side of an assignment expression, but they are still considered lvalues

```
BigInteger operator+(BigInteger lhs, int rhs){
    BigInteger retVal;
    // Exercise: Complete the implementation
    return retVal;
}
```

- When we run this function, the copy contructor will be run to make a copy of the `BigInteger` which we pass as a parameter. It then returns a temporary object.
- Now, if we have something like `BigInteger n2 = n1 + 2`, what happens?
  - The copy constructor will run to create `n2` from the object returned from `operator+`.
  - However, this is a temporary object (rvalue) which will be destroyed as soon as we are done copying it.
- **Idea:** we should transfer the ownership of the data from one object to the newly created object instead of creating an actual (deep) copy of the data.
- How can we do that? Since we know we have an rvalue, we should write a constructor which takes an rvalue reference.

```
BigInteger::BigInteger(BigInteger &&in)
    : digits{in.digits}, size{in.size}, capacity{in.capacity}{
  in.digits = nullptr;
}
```

- If a copy constructor (i.e. with argument of const lvalue reference) and a move constructor are both present in a struct definition, passing a rvalue reference of an instance of the struct into the constructor call will invoke the move constructor.
- If a move constructor is not available, the copy constructor will always be called regardless what kind of value you pass into the constructor call. What happens if there is no copy constructor for the struct?
- **Important note:** When defining a move constructor, we must set all pointers which will be deleted by the destructor to be `nullptr`, or the destructor will delete the data we transfered while the object goes out of scope.

6 Move Assignment Operator

- Similar to the move constructor, we may want to have the following:

```
BigInteger n1{3};
BigInteger n2{1};

n2 = n1 + 2;
```

- We want to make an assignment operator which take an rvalue as well.

```
BigInteger &BigInteger::operator=(BigInteger &&rhs) {
    // Typically, the self assignment check is not required.
    // but added here to prevent someone from actually performing
    // a self-assignment
    if (this != &rhs){
        size = rhs.size;
        capacity = rhs.capacity;
        delete [] digits;
        digits = rhs.digits;
        rhs.digits = nullptr;
    }
    return *this;
}
```

- **Exercise:** Implement the move assignment operator using `std::swap`

7 The Rule of Five

- If you have to write one of the following:
  - copy constructor
  - move constructor
  - copy assignment operator
  - move assignment operator
  - destructor

  Most of the time you should probably write all five. Why?
- Think about when it's not necessary to write all five.

8 Vim Tips of the Week: Operator and Motion

- We have seen commands like `cw`, but in fact they are a combination of an **operator** (`c`) and a **motion** (`w`), and they are much more powerful than single commands.
- Some operators that we have seen: `y, d, c`
- Some motions that we have seen: `w, b, ^, $, /`
- You can combine any operator and any motion: for example `d$` deletes from the current character to the end of line.
- Here are a few more motions to make this really useful:

`f<char>` goes to the next occurrence of `<char>` after the cursor in the current line

`t<char>` like `f`, but goes to the character before `<char>`

And there are some motions that can only be used after an operator, and in visual mode (but not normal mode):

`i<object>` inside the closest pair of `<object>`

`a<object>` around the closest pair of `<object>`

where `<object>` can be left/right parenthesis/bracket/brace.

- For example:

  `ci(` deletes everything inside (but doesn't include) the closest pair of parentheses, and enters insert mode (really useful if you want to change all parameters or arguments).

  `dt;` deletes everything up until the `;` character on the current line.