Overloading and Dynamic Arrays

Joel Willoughby

COP 3503

Outline

Agenda

- Overloading in general
- Operator overloading
- Dynamic Arrays

Outline

Outline

Overloading

- In programming, overloading is a concept applied to methods or functions
- The idea is, you have two functions that have the same name (you overload the name)
- Usually these functions accomplish similar things, but they go about it in different ways

```
1 int add_numbers(int a, int b);
2 int add_numbers(int a, int b, int c);
3 int add_numbers(int * nums, int n);
4 int add_numbers(vector<int> & nums);
```

Interpreting Overloading

- The compiler tells these functions apart by their signature, which is made up of return type and parameters
- So, from the way you use the functions in your code, the compiler knows which one to use:

```
1 int main(){
2   int a = 5, b = 6;
3   vector<int> vec;
4   // Insert 1, 2, and 3 into vec
5   int t1 = add_numbers(a, b);
6   int t2 = add_numbers(a, b, 5);
7   int t3 = add_numbers(vec);
8 }
```

Operators are functions

- Operators are nothing more than functions

```
1 int a = 2 + 3 * 5;
2 // This is the same thing as:
3 int a = add(2, multiply(3,5));
```

- How are the following different?

```
1 float a = 2.0 + 3.0 * 5.0;
2 float b = 2 + 3.0 * 5.0;
```

```
1 //First case:
2 int = int + int; // int add(int, int)
3 //Second case:
4 float = float + float // float add(float, float)
5 //Third case:
6 float = int + float // float add(int, float)
```

Operator overloading

- We are in no way limited to just floats and ints and the '+' operator
- What if you wanted to do matrix addition/multiplication easily?
- Multiply a vector by a scalar?
- Concatenate two lists?
- Compare two strings? (== works for string, but not for cstring)
- Be able to have cout work for some class you are making?

Common Overloadable Operators

- Arithmetic:

  `+ - * / % ^`

- Assignment:

  `= += -= *= /= %= ^= ++ --`

- Comparison:

  `== != < > <= >=`

- Member Access:

  `[]`

Some Words of Caution

- When the meaning of the operator is not immediately clear or could be disputed, do not overload it
- Make sure the operator you are overloading follows the accepted semantics
- You should provide all similar operations. This means if you overload +, you should also overload -. If you overload < or >, you should overload all comparators

Some Syntax

```
1 MyClass a, b;
2 a += b;
3 a.operator+=(b);
```

- There is no difference in the last two lines above, they accomplish the same thing
- This is because the function assigned to any operator @, is called operator@.
- Note that when we have a binary operator, the operator acts on one of the operands (a.operator+=(something)), and the second operand is just passed along.
- This means you overload in the following way:

```
1  class MyClass{
2    MyClass & operator+=(MyClass & rhs){
3      // Code to do += stuff goes here
4    }
5  };
```

Assignment

```
1 MyClass & MyClass::operator=(const MyClass & rhs){
2   // Assigment goes here.
3   // You will probably need a lot of things
4   // like this->something = rhs.something
5   return *this;
6 }
7
8 //Example usage
9 int main(){
10  MyClass a, b;
11  //Do some stuff with a
12  b = a; // same as b.operator=(a);
13 }
```

Notice that = returns a MyClass &, this is so operations like a = b = c; are possible. Usually, the return value for = is *this

Addition

```
1 const MyClass MyClass::operator+(const MyClass & rhs) const{
2   MyClass temp;
3   // populate temp with the result of this + rhs
4   return temp;
5 }
6
7 //Example Usage
8 int main(){
9   MyClass a, b;
10  // Initialize a and b
11  MyClass c;
12  c = a + b; // Same as c = a.operator+(b);
13 }
```

Notice it returns a const MyClass. This is to stop things like (a+b) = c. Also, notice we used the assignment operator in the above...

Subscript

This is used for lists and array type structures (linked lists, dynamic arrays, BSTs, etc)

```
1 int & ListOfInts::operator[](const int index){
2   // Return the element at index index
3 }
4
5 //Example usage
6 int main(){
7   ListOfInts nums;
8   //Add some things in
9   nums[2] = 5;
10 }
```

Notice we are returning an int &. This is so we can actually change the value in the above assignment. If we did not have a reference, we have what's called an immutable operation.

cout

This syntax is a bit stranger than others:

```
1  class MyClass{
2    // MyClass stuff
3    friend ostream & operator<<(ostream & os, const MyClass me);
4  }
5
6  ostream & operator<<(ostream & os, const MyClass me){
7    // Ideally something like os << me.toString()
8    // Just output the class as you would want it
9    return os;
10 }
11
12 //Example Usage
13 int main(){
14   MyClass a;
15   // Do some stuff to a
16   cout << a << endl;
17   // same thing as cout.operator<<(a).operator<<(endl);
18 }
```

A few things about cout overloading

- Notice that we return an ostream &. In fact, we return the exact one we are passed in. This is so we can chain cout statements

```
1 MyClass a;
2 cout << a << endl;
```

(cout « a) returns cout, so we can do cout « endl;

- Also, we have something called a friend. A friend function is a function that is outside of our class, but still has access to the members of our class.

- Notice the friend keyword is only in a declaration inside of our class. You must declare any functions friends in your class (only you can pick your friends)

- Why do we need a friend? Notice that operator« is called on cout, not our class, so it cannot be a member function. Thus, we need to write it outside of the class

Dynamic Arrays

- A dynamic array is an array-like data structure that grows dynamically
- This means that you can keep inserting elements into the array and it never gets full
- Instead, when it gets full, usually more memory is allocated, this is why they are called dynamic
- Examples you have probably used are C++'s vector and Java's ArrayList.

Some Terminology

- The capacity of the array is the amount that it currently can store. It is the physical size
- The size of the array is the number of elements that are in it. This is the logical size
- Two operations typically associated are grow and shrink
- Grow will grow the array when it becomes too full (to allow the user to add more)
- Shrink will shrink the array when it becomes too empty (so your array is not sucking up memory)

Growing

- To grow the array, we typically allocate a new array with double the capacity of the old one
- You then copy each element of the old array to the new one
- You then delete the old one
- Why double? Think about how often you will be copying items if you use a constant value

Outline

Questions

???