Lesson 12 - Operator Overloading
Customising Operators

Summary

In this lesson we explore the subject of Operator Overloading.

New Concepts

Operators, overloading, assignment, friend functions.

Operator Overloading

In C++ it is possible to redefine many operators (e.g. `+`, `-`, `*`) that have so far provided arithmetic operations on primitive types like `int`, `double` etc. For instance, take a statement like $z = x + y$. The C++ language knows how to compute such a statement when the operands (x, y and z) are integer types. However, what if the operands consist of classes we've defined? Take a class like the Matrix class defined previously. We might like to be able to state `x = y * z;`, where `x`, `y` and `z` are Matrix objects, because these are valid operations in matrix algebra. With C++ we can express such a statement using Operator Overloading. In this lesson we will take our Matrix class from lesson 11, and implement functions and methods so that our Matrix objects can be manipulated in this way.

First we need to add our operator overloading methods to our Matrix class declaration. Take a look at the modified declaration beginning on line 1. On line 9 we have added a new class method called `operator=` that returns a reference to a `Matrix<T>` and takes a constant reference to another `Matrix<T>`. The strange name of this method actually states that this method is overriding the equals (`=`) operator for Matrix objects. Shortly we will implement this method and be able to express statements such as `x = y` where `x` and `y` are Matrix objects.

Now take a look at lines 10-11 and 12-13. On line 11 we overload the stream operator (`<<`) so we can pass Matrix objects to an `iostream`, like `cout`. This will allow us to easily display Matrix contents to screen. On line 13 we overload the multiplication operator so we can multiply two Matrix objects together.

```
 1 template <typename T>
 2 class Matrix
 3 {
 4 public:
 5    Matrix(int x = default_x, int y = default_y);
 6    Matrix(const Matrix<T>& src);
 7    ~Matrix();
 8    // overloaded operators on class
 9    Matrix<T>& operator=(const Matrix<T>& rhs);
10    template <typename E>
11    friend ostream& operator<<(ostream& ostr, const Matrix<E>& mtx);
12    template <typename E>
13    friend Matrix<E> operator*(const Matrix<E>& a, const Matrix<E>& b);
14
15    int get_x_size() const { return x_size;}
16    int get_y_size() const { return y_size;}
17    T get_element(int x, int y) const;
18    void set_element(int x, int y, T elem);
19    // constant elements
20    static const int default_x = 3;
21    static const int default_y = 3;
22 protected:
23    T** cells;
24    int x_size;
25    int y_size;
26 };
```

operator_overloading.cpp

There are a few new features being used in lines 10-13, so let's step through them one by one. First note that these are actually (global) functions rather than class methods. They do not belong specifically to the Matrix class but can access its private/protected data members because they are declared `friend` functions. Although this may seem counter-intuitive to allow access to such data-members when we took the trouble to protect them, in limited cases it can be useful. One of those cases is here, when the return type is not the original object of that class. In our Matrix example, the `operator=` method (line 9) will return the original object of the class and is therefore declared as a class method. However the `operator<<` function (line 11) will return an `ostream` reference and the `operator*` function (line 13) will return another Matrix object. As a good rule of thumb therefore, it makes sense to declare these as global `friend` functions.

Note the syntax on lines 10 and 12 denoting that these are template functions, beginning with the declaration `template <typename E>`. As the Matrix class is a template, we have to provide template functions to adapt to difference specialisations of Matrix objects. Note however we use `<typename E>` for our functions to differentiate from `<typename T>` for our class (see line 1).

Now let's move on to the actual definitions of our overloaded operators. First let's examine our class method for overloading the assignment operator with Matrix objects (line 27). We begin with an `if` statement on line 29. Notice the keyword `this` here. Within a class method you can use the keyword `this` when you want to refer to the actual object whose method "this is"; `this` returns a pointer to the object. The reason for this `if` statement therefore is to verify whether the parameter `rhs` ("right-hand-side" of the expression), is actually the same object we will be assigning to. It's legal syntax in C++ to say `x = x`, but in such cases we simply want to return the original `x`. We verify this by comparing the memory addresses of the the pointer `this` and `rhs`. If they're the same, we simply return `this`.

Next we release any memory held in the object to be re-assigned (lines 32-35). If we omitted this step we would inflict a memory leak because there would be no way to refer to the original Matrix object after the assignment operation took place. Following our clean up operation, we allocate new memory for the data that will be held in our re-assigned object (lines 38-42). Finally we copy the values from the `rhs` parameter to the re-assigned object (lines 45-49).

```
27 template <typename T>
28 Matrix<T>& Matrix<T>::operator=(const Matrix<T>& rhs) {
29    if(this == &rhs) return (*this);
30 
31    // release old memory
32    for(int i = 0; i < x_size; ++i)  {
33          delete[] cells[i];
34    }
35    delete[] cells;
36 
37    // allocate new memory
38    cells = new T*[rhs.x_size];
39    for(int i = 0; i < rhs.x_size; ++i) {
40       cells[i] = new T[rhs.y_size];
41       memset(cells[i], 0, (rhs.y_size * sizeof(T)));
42    }
43 
44    // copy values
45    for(int i = 0; i < rhs.x_size; ++i) {
46       for(int j = 0; j < rhs.y_size; ++j) {
47          cells[i][j] = rhs.cells[i][j];
48       }
49    }
50    return *this;
51 }
```

operator_overloading.cpp

Now let's examine our definition for overloading the stream operator `<<`. Recall earlier we noted this is a global `friend` function, rather than a class method. Hence note that we do not use the scope operator (::) before the function name `operator<<`. The first parameter of this function you'll see is a reference to an `ostream`, or output stream. Streams crop up in a variety of tasks such as file I/O and networking code, but in this case we can use the `ostream` parameter to pass `cout` to this function and print our Matrix object to screen. Our Matrix object `mtx` is the second function parameter, in the form of a constant reference.

In the `operator<<` function we iterate through each element of the Matrix object and feed it to the `ostream`. Here we can stipulate the format to print our Matrix. Note that because this was declared as a `friend` function we can access all the private and protected data members of the `mtx` object. Finally we return the `ostream` as a reference.

```
52 template <typename E>
53 ostream& operator<<(ostream& ostr, const Matrix<E>& mtx) {
54    for(int i = 0; i < mtx.x_size; ++i) {
55       for(int j = 0; j < mtx.y_size; ++j) {
56          ostr << mtx.cells[j][i] << ", ";
57       }
58       ostr << "\n";
59    }
60    ostr << "\n";
61    return ostr;
62 }
```

operator_overloading.cpp

Next we define our function for overloading the multiplication operator. This function will accept two Matrix references, `a` and `b`, and compute and return their product matrix. In order to compute the product of two matrices we implement 3 'nested' `for` loops that allow us to index into our matrices. The result appears rather complicated but it allows us to pull a row x from matrix `a` and perform vector multiplication with a column y from matrix `b`, which we save in the matrix `result`. Finally we return `result`.

```
63 template <typename E>
64 Matrix<E> operator*(const Matrix<E>& a, const Matrix<E>& b) {
65    Matrix<E> result(a.x_size, b.y_size);
66    for(int i = 0; i < a.x_size; ++i) {
67       for(int j = 0; j < a.x_size; ++j) {
68          for(int k = 0; k < a.x_size; ++k) {
69             result.cells[i][j] += (a.cells[k][j] * b.cells[i][k]);
70          }
71       }
72    }
73    return result;
74 }
```

operator_overloading.cpp

Finally it's time to create some Matrix objects and try out our overridden operators. On line 76 we create `mtx1`. Notice on line 78 we pass our matrix to `cout` via the overridden stream operator `<<`. On lines 80 and 84 we create another two matrices and on line 85 we test our assignment operator. The effect of this is to release the memory held in `mtx3` before assigning its contents to the values of `mtx2`. Finally on line 88 we create a new Matrix object `product` by the multiplication of `mtx1` and `mtx2`, using our overridden multiplication operator.

```
75 int main() {
76    Matrix<int> mtx1;
77    mtx1.set_element(1,1,4);
78    cout << mtx1;
79
80    Matrix<int> mtx2;
81    mtx2.set_element(2,2,2);
82    cout << mtx2;
83
84    Matrix<int> mtx3;
85    mtx3 = mtx2;
86    cout << mtx3;
87
88    Matrix<int> product = (mtx1 * mtx3);
89    cout << product;
90
91    return 0;
92 }
```

operator_overloading.cpp

Algorithmic Complexity

There are numerous ways and means of optimising one's code to squeeze the best performance from our applications. Possibly the most effective, yet overlooked means of optimising code begins with the design of the algorithm used. When evaluating the design of our algorithms we tend to estimate the performance of an algorithm in terms of the input size. If we analyse a sorting or searching algorithm, for instance, the input size is the number of elements to be sorted or searched. Furthermore, we usually categorise algorithms in terms of their time and space requirements.

Most algorithms perform differently depending on the input elements provided, but usually we are most interested in the upper bound performance a.k.a the 'big O' complexity, partly because the worst case behaviour is often the dominant behaviour in an algorithm. Consider, for example, searching a data-set for an element. It may be that most of the time the element is not present which may cause the worst case of the searching algorithm to execute. To find the upper bound we estimate the costs of each expression in the algorithm and ignore constant modifications as these become insignificant as the input size grows.

For the purposes of demonstration, let's analyse the performance of `Insertion Sort`. `Insertion Sort` comprises of essentially an outer loop and an inner loop. The outer loop iterates from 1 to $n-1$ where n is the number of elements to be sorted. The inner loop, meanwhile, iterates until the correct position in the final array of numbers has been found. If we begin with the behaviour of the inner loop then in the worst case, it will have to iterate through the whole array before locating a position to insert a new element. This means the inner loop iterates once for the 1st element, twice for the second and so forth until the outer loop exits giving us a performance of $T(n) = (n(n+1)/2) - n$. We ignore modifications made by constant elements leaving us with a running time of $O(n^2)$.

```cpp
for(int i = 1; i < arr.size; ++i) { // outer loop iterates over n - 1 elements
   int j = i;
   while(j > 0 && arr[j - 1] > arr[j] { // inner loop locates correct position in array
      swap(arr, j, j - 1); // perform a constant time swap operation
      j--;
   }
}
```

insertion_sort.cpp

We can now compare the performance of `Insertion Sort` with other sorting algorithms. The `Quick Sort` algorithm, for instance, has a upper bound of $O(n\ log(n))$ in the average case and $O(n^2)$ in the worst case. We can assume (with some important caveats) that as the input size n grows, `Quick Sort` will always be faster at sorting n numbers compared to `Insertion Sort`. Furthermore, the difference in performance time between `Quick Sort` and `Insertion Sort` will grow as n increases. It's important to realise, however, when comparing the behaviour of algorithms we are primarily interested in the rate of change in behaviour over time with ever greater values of n. Consequently, we tend to be concerned with cases where n is large.

On a final note, keep in mind that although one algorithm may have superior performance over another, it may still be entirely sensible to favour an inferior algorithm in certain circumstances. You might want to employ a brute force search algorithm for small values of n, for example, which will be far quicker than an algorithm like `Binary Search` which has excellent performance in cases where n is large and requires a sorted data set. Similarly, when deciding on a sorting algorithm, in cases where the items to be sorted are already 'mostly sorted', `Insertion Sort` will typically provide superior performance to `Quick Sort`.

Average Case Behaviour (Time complexity)

$O(n^2)$	$O(n\ log(n))$	$O(n)$	$O(log(n))$
Insertion Sort	Quick Sort	Linear Search	Binary Search
Bubble Sort	Heap Sort		
	Merge Sort		

Worst Case Behaviour (Time complexity)

$O(n^2)$	$O(n\ log(n))$	$O(n)$	$O(log(n))$
Quick Sort	Heap Sort	Linear Search	Binary Search
Insertion Sort	Merge Sort		
Bubble Sort			

Complexity of some generic tasks

Complexity	*Example*
$O(1)$	Fetching the first element from a set of data
$O(log\ n)$	Splitting a set of data in half and then repeatedly splitting those halves
$O(n)$	Traversing a set of data
$O(n\ log(n))$	Repeatedly splitting a set of data in half and traversing each half
$O(n^2)$	Traversing a set of data for each member of another equally sized set
$O(2^n)$	Generating all possible subsets of a set of data
$O(n!)$	Generating all possible permutations of a set of data