

## 9.9 — Overloading the parenthesis operator

BY ALEX ON OCTOBER 25TH, 2007 | LAST MODIFIED BY ALEX ON JULY 9TH, 2018

All of the overloaded operators you have seen so far let you define the type of the operator's parameters, but not the number of parameters (which is fixed based on the type of the operator). For example, `operator==` always takes two parameters, whereas `operator!` always takes one. The parenthesis operator (`operator()`) is a particularly interesting operator in that it allows you to vary both the type AND number of parameters it takes.

There are two things to keep in mind: first, the parenthesis operator must be implemented as a member function. Second, in non-object-oriented C++, the `()` operator is used to call functions. In the case of classes, `operator()` is just a normal operator that calls a function (named `operator()`) like any other overloaded operator.

### An example

Let's take a look at an example that lends itself to overloading this operator:

```
1  class Matrix
2  {
3  private:
4      double data[4][4];
5  public:
6      Matrix()
7      {
8          // Set all elements of the matrix to 0.0
9          for (int row=0; row < 4; ++row)
10             for (int col=0; col < 4; ++col)
11                 data[row][col] = 0.0;
12     }
13 };
```

Matrices are a key component of linear algebra, and are often used to do geometric modeling and 3D computer graphics work. In this case, all you need to recognize is that the `Matrix` class is a 4 by 4 two-dimensional array of doubles.

In the lesson on [overloading the subscript operator](#), you learned that we could overload `operator[]` to provide direct access to a private one-dimensional array. However, in this case, we want access to a private two-dimensional array. Because `operator[]` is limited to a single parameter, it is not sufficient to let us index a two-dimensional array.

However, because the `()` operator can take as many parameters as we want it to have, we can declare a version of `operator()` that takes two integer index parameters, and use it to access our two-dimensional array. Here is an example of this:

```
1  #include <cassert> // for assert()
2  class Matrix
3  {
4  private:
5      double data[4][4];
6  public:
7      Matrix()
8      {
9          // Set all elements of the matrix to 0.0
10         for (int row=0; row < 4; ++row)
11             for (int col=0; col < 4; ++col)
12                 data[row][col] = 0.0;
13     }
14
15     double& operator()(int row, int col);
16     const double& operator()(int row, int col) const; // for const objects
17 };
18
19 double& Matrix::operator()(int row, int col)
20 {
21     assert(col >= 0 && col < 4);
22     assert(row >= 0 && row < 4);
23 }
```

```

24     return data[row][col];
25 }
26
27 const double& Matrix::operator()(int row, int col) const
28 {
29     assert(col >= 0 && col < 4);
30     assert(row >= 0 && row < 4);
31
32     return data[row][col];
33 }

```

Now we can declare a Matrix and access its elements like this:

```

1  #include <iostream>
2
3  int main()
4  {
5      Matrix matrix;
6      matrix(1, 2) = 4.5;
7      std::cout << matrix(1, 2);
8
9      return 0;
10 }

```

which produces the result:

4.5

Now, let's overload the () operator again, this time in a way that takes no parameters at all:

```

1  #include <cassert> // for assert()
2  class Matrix
3  {
4  private:
5      double data[4][4];
6  public:
7      Matrix()
8      {
9          // Set all elements of the matrix to 0.0
10         for (int row=0; row < 4; ++row)
11             for (int col=0; col < 4; ++col)
12                 data[row][col] = 0.0;
13     }
14
15     double& operator()(int row, int col);
16     const double& operator()(int row, int col) const;
17     void operator()();
18 };
19
20 double& Matrix::operator()(int row, int col)
21 {
22     assert(col >= 0 && col < 4);
23     assert(row >= 0 && row < 4);
24
25     return data[row][col];
26 }
27
28 const double& Matrix::operator()(int row, int col) const
29 {
30     assert(col >= 0 && col < 4);
31     assert(row >= 0 && row < 4);
32
33     return data[row][col];
34 }
35
36 void Matrix::operator>()()

```

```

37     {
38         // reset all elements of the matrix to 0.0
39         for (int row=0; row < 4; ++row)
40             for (int col=0; col < 4; ++col)
41                 data[row][col] = 0.0;
42     }

```

And here's our new example:

```

1  #include <iostream>
2  int main()
3  {
4      Matrix matrix;
5      matrix(1, 2) = 4.5;
6      matrix(); // erase matrix
7      std::cout << matrix(1, 2);
8
9      return 0;
10 }

```

which produces the result:

0

Because the () operator is so flexible, it can be tempting to use it for many different purposes. However, this is strongly discouraged, since the () symbol does not really give any indication of what the operator is doing. In our example above, it would be better to have written the erase functionality as a function called `clear()` or `erase()`, as `matrix.erase()` is easier to understand than `matrix()` (which could do anything!).

## Having fun with functors

Operator() is also commonly overloaded to implement **functors** (or **function object**), which are classes that operate like functions. The advantage of a functor over a normal function is that functors can store data in member variables (since they are classes).

Here's a simple functor:

```

1  class Accumulator
2  {
3  private:
4      int m_counter = 0;
5
6  public:
7      Accumulator()
8      {
9      }
10
11     int operator() (int i) { return (m_counter += i); }
12 };
13
14 int main()
15 {
16     Accumulator acc;
17     std::cout << acc(10) << std::endl; // prints 10
18     std::cout << acc(20) << std::endl; // prints 30
19
20     return 0;
21 }

```

Note that using our Accumulator looks just like making a normal function call, but our Accumulator object is storing an accumulated value.

You may wonder why we couldn't do the same thing with a normal function and a static local variable to preserve data between function calls. We could, but because functions only have one global instance, we'd be limited to using it for one thing at a time. With functors, we can instantiate as many separate functor objects as we need and use them all simultaneously.

## Conclusion

Operator() is sometimes overloaded with two parameters to index multidimensional arrays, or to retrieve a subset of a one dimensional array (with the two parameters defining the subset to return). Anything else is probably better written as a member function with a more descriptive name.

Operator() is also often overloaded to create functors. Although simple functors (such as the example above) are fairly easily understood, functors are typically used in more advanced programming topics, and deserve their own lesson.

### Quiz time

1) Write a class that holds a string. Overload operator() to return the substring that starts at the index of the first parameter. The length of the substring should be defined by the second parameter.

Hint: You can use array indices to access individual chars within the std::string

Hint: You can use operator+= to append something to a string

The following code should run:

```
1  int main()
2  {
3      Mystring string("Hello, world!");
4      std::cout << string(7, 5); // start at index 7 and return 5 characters
5
6      return 0;
7  }
```

This should print

world

### Hide Solution

```
1  #include <string>
2  #include <iostream>
3  #include <cassert>
4
5  class Mystring
6  {
7  private:
8      std::string m_string;
9
10 public:
11     Mystring(const std::string string="")
12         :m_string(string)
13     {
14     }
15
16     std::string operator()(int start, int length)
17     {
18         assert(start + length <= m_string.length() && "Mystring::operator(int, int): Substring is out o
19 f range");
20
21         std::string ret;
22         for (int count = 0; count < length; ++count)
23             ret += m_string[start + count];
24         return ret;
25     }
26 };
27
28 int main()
29 {
30     Mystring string("Hello, world!");
31     std::cout << string(7, 5); // start at index 7 and return 5 characters
32
33     return 0;
34 }
```