21.10 — Overloading the parenthesis operator

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:

```
class Matrix
{
private:
    double data[4][4]{};
};
```

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:

```
#include <cassert> // for assert()

class Matrix
{
private:
    double m_data[4][4]{};

public:
    double& operator()(int row, int col);
    double operator()(int row, int col) const; // for const objects
};

double& Matrix::operator()(int row, int col)
{
    assert(row >= 0 && row < 4);
    assert(col >= 0 && col < 4);

    return m_data[row][col];
}

double Matrix::operator()(int row, int col) const
{
    assert(row >= 0 && row < 4);
    assert(col >= 0 && col < 4);

    return m_data[row][col];
}
```

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

```
#include <iostream>

int main()
{
    Matrix matrix;
    matrix(1, 2) = 4.5;
    std::cout << matrix(1, 2) << '\n';

    return 0;
}
```

which produces the result:

```
4.5
```

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

```
#include <cassert> // for assert()
class Matrix
{
private:
    double m_data[4][4]{};

public:
    double& operator()(int row, int col);
    double operator()(int row, int col) const;
    void operator()();
};

double& Matrix::operator()(int row, int col)
{
    assert(row >= 0 && row < 4);
    assert(col >= 0 && col < 4);

    return m_data[row][col];
}

double Matrix::operator()(int row, int col) const
{
    assert(row >= 0 && row < 4);
    assert(col >= 0 && col < 4);

    return m_data[row][col];
}

void Matrix::operator()()
{
    // reset all elements of the matrix to 0.0
    for (int row{ 0 }; row < 4; ++row)
    {
        for (int col{ 0 }; col < 4; ++col)
        {
            m_data[row][col] = 0.0;
        }
    }
}
```

And here's our new example:

```
#include <iostream>

int main()
{
    Matrix matrix{};
    matrix(1, 2) = 4.5;
    matrix(); // erase matrix
    std::cout << matrix(1, 2) << '\n';

    return 0;
}
```

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:

```
#include <iostream>

class Accumulator
{
private:
    int m_counter{ 0 };

public:
    int operator() (int i) { return (m_counter += i); }

    void reset() { m_counter = 0; } // optional
};

int main()
{
    Accumulator acc{};
    std::cout << acc(1) << '\n'; // prints 1
    std::cout << acc(3) << '\n'; // prints 4

    Accumulator acc2{};
    std::cout << acc2(10) << '\n'; // prints 10
    std::cout << acc2(20) << '\n'; // prints 30

    return 0;
}
```

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

The nice thing about functors is that we can instantiate as many separate functor objects as we need, and use them all simultaneously. Functors can also have other member functions (e.g. `reset()`) that do convenient things.

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

Question #1

Write a class named MyString that holds a `std::string`. Overload `operator<<` to output the string. Overload `operator()` to return the substring that starts at the index of the first parameter (as a `MyString`). The length of the substring should be defined by the second parameter.

The following code should run:

```
int main()
{
    MyString s { "Hello, world!" };
    std::cout << s(7, 5) << '\n'; // start at index 7 and return 5 characters

    return 0;
}
```

This should print

```
world
```

Hint: You can use `std::string::substr` to get a substring of a std::string.

Show Solution

Question #2

This quiz question is extra credit.

> Step #1

Why is the above inefficient if we don't need to modify the returned substring?

Show Solution

> Step #2

What might we do instead?

Show Solution

> Step #3

Update `operator()` from the prior quiz solution to return the substring as a `std::string_view` instead.

Hint: `std::string::substr()` returns a `std::string`. `std::string_view::substr()` returns a `std::string_view`. Be very careful not to return a dangling `std::string_view!`.

Show Hint

Show Hint

Show Solution