13.10 — Overloading the parenthesis operator

ALEX JULY 24, 2021

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(col >= 0 && col < 4);
    assert(row >= 0 && row < 4);

    return m_data[row][col];
}

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

    return m_data[row][col];
}

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

int main()
{
    Accumulator acc{};
    std::cout << acc(10) << '\n'; // prints 10
    std::cout << acc(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.

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

Question #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:

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

    return 0;
}
```

This should print

```
world
```

Show Solution

Next lesson
13.11 Overloading typecasts

Back to table of contents

Previous lesson
13.9 Overloading the subscript operator

149 COMMENTS Newest

Mateusz Kacpersi January 31, 2022 10:11 am

Hello :)

```
double& operator()(int row, int col);
double operator()(int row, int col) const;
```

I dont really understand when i should use '&' ? Why in the second operator() we dont use it ?

0 Reply

> **Alex** Author — Reply to Mateusz Kacpersi — January 31, 2022 6:33 pm
>
> We use `&` when we want a reference. In this case, the non-const version of `operator()` returns a reference to the array element, so that we can do things like this:
>
> ```
> matrix(1, 2) = 4.5;
> ```
>
> If we didn't return a reference, then the operator would return a copy, and we'd assign 4.5 to the copy. Then the next time we accessed matrix(1, 2) it would have the same value as before (not 4.5).
>
> 0 Reply

Omer January 27, 2022 6:07 am

Hi,

In the previous lesson (Overloading the subscript operator) the const version of the operator[] function returned const reference to int, here the const version of the operator() returns just a double without any reference involved, Why? and could've we just return int instead of const reference to int?

0 Reply

> **Alex** Author — Reply to Omer — January 27, 2022 12:20 pm
>
> It can return either. I've updated the prior lesson to return an int since ints are normally passed by value, not const reference.
>
> 0 Reply

George October 7, 2021 1:05 am

Why do we need all this fancy stuff for an index, static_cast<std::string::size_type>(start + count), when simple int will do? Or at least my program compiled and run fine.

0 Reply

> **Alex** Author — Reply to George — October 8, 2021 3:48 pm
>
> Because indexing in C++ is messed up due to prior inconsistent decisions around whether indices should be signed or unsigned. In the vast majority of cases, int will work just fine, though you may get compiler warnings about signed/unsigned conversions.
>
> 0 Reply
>
> > **George** — Reply to Alex — October 9, 2021 9:48 am
> >
> > so what does size_type return, signed or unsigned int?
> >
> > 0 Reply
> >
> > > **Alex** Author — Reply to George — October 10, 2021 1:38 pm
> > >
> > > std::string::size_type is typically an alias for std::size_t, which is an unsigned integral type (not necessarily an int).
> > >
> > > 0 Reply

Sanket Kittad September 21, 2021 10:48 am

Why won't this code work?

```
1  #include<iostream>
2  #include<string>
3  #include<vector>
4  #include<algorithm>
5  using namespace std;
6
7  class myString{
8      string m_string;
9  public:
10     myString(const string & string={}):m_string{string}
11     {}
12     string operator()(int start,int sizy){
13         string ans{};
14         for(int i=start;i<sizy;++i){
15             ans=+m_string[i];
16         }
17         return ans;
18     }
19 };
20 int main()
21 {
22     myString string{ "Hello, world!" };
23     std::cout << string(7, 5) << '\n'; // start at index 7 and return 5 characters
24
25     return 0;
26 }
```

Last edited 5 months ago by Sanket Kittad

0 Reply

> **nascardriver** Sub-admin — Reply to Sanket Kittad — September 27, 2021 8:19 am
>
> What's the error? What does it do? What did you expect it to do?
>
> EDIT: Sorry, I didn't see you already figured it out
>
> *Last edited 5 months ago by nascardriver*
>
> 1 Reply

> **Alex** Author — Reply to Sanket Kittad — September 21, 2021 11:02 am
>
> This is a super easy thing to debug in a debugger. Put a breakpoint on line 12, and inspect the values of start and sizy. It will become quite apparent why your loop isn't executing.
>
> 1 Reply
>
> > **Sanket Kittad** — Reply to Alex — September 21, 2021 11:16 am
> >
> > Yeah I got my mistake, thanks!
> >
> > 0 Reply

UnknownLearner September 3, 2021 5:17 am

for practice purposes, I tried to experiment with built-in fixed arrays. It assumes the input is correct :)

```
#include<iostream>

class Mystring
{
private:
    char m_str[15]{};
    char m_newStr[15]{};
public:
    Mystring(const char str[])
    {
        for (size_t i = 0; i < sizeof(m_str) / sizeof(*m_str); ++i)
        {
            m_str[i] = str[i];
        }
    }
    char* operator()(int beginAt, int toEnd)
    {
        for (int i = 0; i < toEnd; ++i)
        {
            m_newStr[i] = m_str[beginAt + i];
        }
        return &m_newStr[0];
    }
};

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

    return 0;
}
```

0 Reply

Jack August 18, 2021 11:52 am

Is overloading operator() sufficient for using the object where a lambda / function-pointer would go (e.g. in standard library functions like find_if())?

0 Reply

Waldo Lemmer July 22, 2021 10:55 pm

- Section "Having fun with functors" should #include <iostream>
- If you were to use at() at Q#3, you wouldn't need the 2 asserts

0 Reply

Kacper Bazan June 18, 2021 3:40 pm

I added some extra logic to my for loop so the user can capture up till the end of a string, even if the length variable exceeds the bounds of the string.

```
std::string operator()(int index, int length)
    {
        std::string sub_string{};

        for (int count = 0; count < length && (index + count <= m_string.length()); ++count)
        {
            sub_string += m_string[index + count];
        }
        return sub_string;
    }
```

0 Reply

> **nascardriver** Sub-admin — Reply to Kacper Bazan — June 30, 2021 7:11 am
>
> `index + count <= m_string.length()` this calculation will run after every loop iteration. Calculating anything in the compare part of a for-loop is usually not a good idea and should only be done if you can't restructure.
>
> You can move the calculation out of the loop by first calculating how often the loop needs to run
>
> ```
> int iterations{ std::min(length, static_cast(m_string.length()) - index) };
>
> for (int count{ 0 }; count < iterations; ++count)
> ```
>
> or calculate where the loop ends
>
> ```
> int start{ index };
> int end{ std::min(index + length, static_cast(m_string.length())) };
>
> for (int i{ start }; i < end; ++i)
> ```
>
> 0 Reply

Haldhar Patel June 9, 2021 1:36 am

Can we use push_back() in place of "+=" to append something to a string?

is this code from quiz

```
std::string ret{};
        for (int count{ 0 }; count < length; ++count)
        {
            ret += m_string[static_cast<std::string::size_type>(start + count)];
        }
```

different from this

```
std::string string{};
for(std::size_t i{index};i<index+length;++i){
    string.push_back(m_string[i]);
}
```

0 Reply

> **Alex** Author — Reply to Haldhar Patel — June 10, 2021 11:10 am
>
> Yes. However, push_back() can only append a single character, whereas += can handle a single character or a string, and is thus more flexible. Personally, I think += is also easier to read, as it avoids additional parenthesis.
>
> 0 Reply

Sam March 27, 2021 11:04 pm

Here is my implementation for the quiz.

```
class MyString
{
private:
  std::string m_string{};

public:
  MyString(const std::string& str) : m_string(str) {}

  std::string operator()(int start, int length=0)
  {
    // ensure valid index
    assert(start >= 0);

    // ensure the subscript is valid
    assert(length + start <= static_cast<int>(m_string.size()));

    // store the substring
    std::string sub_string{};

    int index{ 0 };

    // loop over the length
    while (index != length)
    {
      // append the character to the substr and increment
      sub_string += m_string[start++];

      ++index;
    }

    // check if empty
    if (!sub_string.empty())
      return sub_string;
    else
      return "error";
  }
};
```

0 Reply