9.9 — Overloading the parenthesis operator

BY ALEX ON OCTOBER 25TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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 data[4][4]{};

public:
    double& operator()(int row, int col);
    const 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 data[row][col];
}

const double& Matrix::operator()(int row, int col) const
{
```

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

    return 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 data[4][4]{};

public:
    double& operator()(int row, int col);
    const 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 data[row][col];
}

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

    return 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)
            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:

```
class Accumulator
{
private:
    int m_counter{ 0 };

public:
    Accumulator()
    {
    }

    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

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

9.10 -- Overloading typecasts

Index

9.8 -- Overloading the subscript operator

C++ TUTORIAL | PRINT THIS POST

87 comments to 9.9 — Overloading the parenthesis operator

« Older Comments 1 2

Ged
February 6, 2020 at 10:16 am · Reply

Why are you using "static_cast<std::string::size_type>(start + count)", cause my line seems much more simpler.

```
std::string operator()(int x, int y)
    {
        assert(x + y <= m_name.length() && "String is out of range");
        std::string temp{};
        for (int index = x; index < x + y; ++index)
            temp += m_name[index];
        return temp;
    }
```

```
    std::string operator()(int start, int length)
    {
        assert(start + length <= static_cast<int>(m_string.length()) && "Mystring::operator(

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

nascardriver
February 7, 2020 at 8:29 am · Reply

`std::string::operator[]` wants a `std::string::size_type`. If you pass it an `int` and your compiler is configured to warn about unsigned conversions, you'll get a warning.

sito

January 31, 2020 at 5:00 am · Reply

Oh i see. So basically the []operator when used with std::string wants a std::string size_type in order to work properly. I now have 2 more questions in order to understand this. First off why didn't my compiler warn me about this? using visual studio2019. Could you instead of having to cast the int parameters just make them as unsigned ints? and then make a loop with an unsigned int variable? it would be one cast less you would have to make in the loop or am I thinking about this the wrong way?

nascardriver
January 31, 2020 at 5:22 am · Reply

> why didn't my compiler warn me about this?
/W4 doesn't enable C4365, which is the warning responsible for signed/unsigned conversions. /Wall includes it, but might enable more warnings than necessary. /w44365 enables this specific warning when you're using /W4.

> Could you instead [...]
If you're going to change the loop variable's type, use the correct one right away, `std::string::size_type`. Keep in mind that this is unsigned, a reverse loop, you can't go negative.

sito
January 27, 2020 at 12:51 pm · Reply

hello! Why do you make the code in the sullution for quiz 1 so complecated? in my opinion it's more complecated than it needs to be. Wouldn't it be better to do it like this? see code below. There is one cast and it's more readable than your example or are there bennefits to doing it your way?

```
#include <string>
#include<cassert>
#include <iostream>
class MyString {
private:
    std::string m_str;
public:
    MyString(std::string str) :
        m_str{ str }
    {
    }
    std::string operator()(int startingIndex, int length) {
        std::string str{};
        int strLength{static_cast<int>(m_str.length()) };
        assert(length > strLength && "wrong length");
        for (int i{ startingIndex }; i<strLength; ++i) {
            str += m_str[i];
            if (i== length)
                break;
        }
        return str;
    }
};

int main()
{
    MyString string{ "Hello, world!" };
    std::cout << string(7, 5) << '\n'; // start at index 7 and return 5 characters
    return 0;
}
```

I might be missing something here but I think this is such a short program where the string isn't long enough to really care about performence in this case.

nascardriver
January 28, 2020 at 2:18 am · Reply

You have a signed to unsigned conversion in your code. Your compiler should have warned you.

Apart from you allowing substrings that end after the end of the string, there's no real difference in the code. What makes you think one or the other is easier to read?

sito
January 29, 2020 at 12:01 pm · Reply

hello! this might just be because I haven't done my homework correctly or that i've simply forgot but there are 2 parts I don't understand, first off what does this part do at the end of the assert statement
&& "Mystring::operator(int, int): Substring is out of range");?
Also why do you use size type in the loop? again this might be simple questions and I have just not done my homework correctly if that's the case I appologise.

nascardriver
January 30, 2020 at 12:28 am · Reply

An assert fails if the condition evaluates to `false`. If an assert fails, you have to understand why it failed. A message helps. We can add a message by adding `&& "message"`, because it doesn't change the condition. A string literal is `true` (Because it's non-zero).
condition && message = x
false && true = false
true && true = true
The message doesn't change the condition.

`operator[]` of `std::string` wants a `std::string::size_type` (An unsigned integer type). If you give it a signed int, the compiler complains about an implicit signed to unsigned conversion (Because this could cause an underflow). The cast silences the warning.

Ryan
December 10, 2019 at 12:11 pm · Reply

For the quiz, couldn't you also use:
ret += m_string[static_cast<unsigned int>(start++)];

Performance and efficiency wise, which version is better?

nascardriver
December 11, 2019 at 5:43 am · Reply

There are several ways of doing this. The compiler should produce the same (optimal) code for every way. Use whichever you find easier to read.

Daniel
November 8, 2019 at 5:27 am · Reply

Hey,
I am just here to note, that example solution doesn't work in case you use, Wall and Werror, since std::string operator[](), takes in unsigned int. If you want it to work, you must use static cast/unsigned ints. It can also be 'resolved' by just removing Werror, from your settinsg.
This whole paragraph also applies to assertion line.

However it might be good idea to note that somewhere in the text or update the example, since in first lectures, you recommend to use Werror Wall etc. (It took me more time than I want to admit to find the bug :D)

```
#include <string>
#include <iostream>
#include <cassert>

class Mystring
{
private:
    std::string m_string;

public:
    Mystring(const std::string string="")
        :m_string(string)
    {
    }

    std::string operator()(int start, int length)
    {
        //asserting, that we can cast to unsigned int without fear:
        assert(start+length >= 0 && "Mystring::operator(int, int): Negative values not allo
        assert(static_cast<unsigned int>(start + length) <= m_string.length() && "Mystring:

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

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

    return 0;
}
```

Daniel
November 8, 2019 at 9:52 am · Reply

btw the code is the updated code, so it works properly even with Werror

nascardriver
November 9, 2019 at 4:15 am · Reply

Lesson updated, thanks for pointing out the error!
Note that `std::string::length` doesn't necessarily return an `unsigned int`. It returns

`std::string::size_type`, which may or may not be `unsigned int`.

hellmet
October 31, 2019 at 9:19 am · Reply

Ahh, I see where this can be useful!

Say I have a matrix that has n dimensions, each with size k1, k2, ... kn
By combining the () with the ... (ellipsis), one can use it to index into the array of n dimensions!

```
1 Matrix matrix(3);        // 3 dim matrix
2 matrix.Allocate(3, 4, 2) // allocate a 3x4x2 matrix (uses the ... operator in the function)
3 matrix(1, 1, 2) = 4.5;   // element at (1, 1, 2)    (also uses the ... operator in the funct
```

Perhaps, that is what numpy does?

Daniel
November 8, 2019 at 5:43 am · Reply

Hey,
I don't think that is what numpy does, since C++ doesn't have typechecked ellipsis (something like fce(double ...),), you also don't know, how much arguments are passed.
However I think there is solution for these problems throught templates, although I am just at this lecture, so I don't recall correctly, how it was done. However NascarDriver or Alex, might confirm, that this either might be solution or rather might not and they really use ellipsis

nascardriver
November 9, 2019 at 4:56 am · Reply

I don't know how NumPy does it. C++ has parameter packs (Basically type-safe variadic parameters), through which this should be possible in a safe way. Variadic arguments shouldn't be used anymore.

Arunreddy
September 13, 2019 at 2:37 am · Reply

matrix(1, 2) = 4.5;
how this sets value, i didn't understand.
please explain,
Thanks.

nascardriver
September 13, 2019 at 2:54 am · Reply

It calls `Matrix::operator()(int row, int col)`, which returns a `double&`. If you forgot about reference, re-read lesson 6.11 and 7.4a.

George Pitchurov
August 20, 2019 at 7:40 am · Reply

I have hard time figuring out why something really odd. Consider this code:

```
int main()
{
   std::string strA("Hello world");
   std::string strB=strA; // String strB assumes the value of strA, but otherwise are disti

   for (int iii=0; iii<10; iii++)   //Loop through first 10 elements
   {
      std::cout<<strA[iii]<<' '<< &strA+iii << '\n'; //Print value and adress of strA eleme
      std::cout<<strB[iii]<<' '<< &strB+iii << '\n------------\n'; //Print value and adress
   }
return 0;
```

If you print that, you'll see that first element of strA and strB are 'H', but have different adresses (not surprising). Then however the second element of strA has the SAME adress as the first element of strB, but has different value: 'e' (which is also shared by the second element of strB). Then the third element of strA has the SAME adress as second element of strB, but again has different value 'l' (instead of 'e', which the second strB element has). And the fourth element of strA has the same adress as the third element of strB, but again different value.

So how is possible that one and the same adress holds different values for two variables of the same type? No assignement is done within the loop, so that to expect any overwrite action, only output statements! How is at all possible one and same adress to be allocated to two different array variables that are not reference to each other?

nascardriver
August 20, 2019 at 10:44 am · Reply

Line 9: You can't store more than 1 character in a char.

You're taking the address of `strA` and `strB`, which are `std::string`. This address isn't the address of the actual string. Pointer arithmetic on `std::string` uses `std::string`'s size. What you want is

```
strA.c_str() + iii
strB.c_str() + iii
```

`c_str` returns the C-style string.

« Older Comments 1 2