9.x — Chapter 9 comprehensive quiz

BY ALEX ON AUGUST 9TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

In this chapter, we explored topics related to operator overloading, as well as overloaded typecasts, and topics related to the copy constructor.

Summary

Operator overloading is a variant of function overloading that lets you overload operators for your classes. When operators are overloaded, the intent of the operators should be kept as close to the original intent of the operators as possible. If the meaning of an operator when applied to a custom class is not clear and intuitive, use a named function instead.

Operators can be overloaded as a normal function, a friend function, or a member function. The following rules of thumb can help you determine which form is best for a given situation:

- If you're overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function.
- If you're overloading a unary operator, do so as a member function.
- If you're overloading a binary operator that modifies its left operand (e.g. operator+=), do so as a member function if you can.
- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function or friend function.

Typecasts can be overloaded to provide conversion functions, which can be used to explicitly or implicitly convert your class into another type.

A copy constructor is a special type of constructor used to initialize an object from another object of the same type. Copy constructors are used for direct/uniform initialization from an object of the same type, copy initialization (Fraction f = Fraction(5,3)), and when passing or returning a parameter by value.

If you do not supply a copy constructor, the compiler will create one for you. Compiler-provided copy constructors will use memberwise initialization, meaning each member of the copy is initialized from the original member. The copy constructor may be elided for optimization purposes, even if it has side-effects, so do not rely on your copy constructor actually executing.

Constructors are considered converting constructors by default, meaning that the compiler will use them to implicitly convert objects of other types into objects of your class. You can avoid this by using the explicit keyword in front of your constructor. You can also delete functions within your class, including the copy constructor and overloaded assignment operator if desired. This will cause a compiler error if a deleted function would be called.

The assignment operator can be overloaded to allow assignment to your class. If you do not provide an overloaded assignment operator, the compiler will create one for you. Overloaded assignment operators should always include a self-assignment check.

New programmers often mix up when the assignment operator vs copy constructor are used, but it's fairly straightforward:

- If a new object has to be created before the copying can occur, the copy constructor is used (note: this includes passing or returning objects by value).
- If a new object does not have to be created before the copying can occur, the assignment operator is used.

By default, the copy constructor and assignment operators provided by the compiler do a memberwise initialization or assignment, which is a shallow copy. If your class dynamically allocates memory, this will likely lead to problems, as multiple objects will end up pointing to the same allocated memory. In this case, you'll need to explicitly define these in order to do a deep copy. Even better, avoid doing your own memory management if you can and use classes from the standard library.

Quiz Time

1) Assuming Point is a class and point is an instance of that class, should you use a normal/friend or member function overload for the following operators?

1a) point + point
1b) -point
1c) std::cout << point
1d) point = 5;

Show Solution

2) Write a class named Average that will keep track of the average of all integers passed to it. Use two members: The first one should be type `std::int_least32_t`, and used to keep track of the sum of all the numbers you've seen so far. The second should be of type `std::int_least8_t`, and used to keep track of how many numbers you've seen so far. You can divide them to find your average.

2a) Write all of the functions necessary for the following program to run:

```cpp
int main()
{
    Average avg{};

    avg += 4;
    std::cout << avg << '\n'; // 4 / 1 = 4

    avg += 8;
    std::cout << avg << '\n'; // (4 + 8) / 2 = 6

    avg += 24;
    std::cout << avg << '\n'; // (4 + 8 + 24) / 3 = 12

    avg += -10;
    std::cout << avg << '\n'; // (4 + 8 + 24 - 10) / 4 = 6.5

    (avg += 6) += 10; // 2 calls chained together
    std::cout << avg << '\n'; // (4 + 8 + 24 - 10 + 6 + 10) / 6 = 7

    Average copy{ avg };
    std::cout << copy << '\n';

    return 0;
}
```

and produce the result:

```
4
6
12
6.5
7
7
```

Hint: Remember that 8 bit integers are usually `char`s, so `std::cout` treats them accordingly.

Show Solution

2b) Does this class need an explicit copy constructor or assignment operator?

Show Solution

3) Write your own integer array class named IntArray from scratch (do not use std::array or std::vector). Users should pass in the size of the array when it is created, and the array should be dynamically allocated. Use assert statements to guard against bad data. Create any constructors or overloaded operators needed to make the following program operate correctly:

```cpp
#include <iostream>

IntArray fillArray()
{
    IntArray a(5);

    a[0] = 5;
    a[1] = 8;
    a[2] = 2;
    a[3] = 3;
    a[4] = 6;

    return a;
}

int main()
{
    IntArray a{ fillArray() };
    std::cout << a << '\n';

    auto &ref{ a }; // we're using this reference to avoid compiler self-assignment errors
    a = ref;

    IntArray b(1);
    b = a;

    std::cout << b << '\n';

    return 0;
}
```

This programs should print:

```
5 8 2 3 6
5 8 2 3 6
```

Show Solution

4) Extra credit: This one is a little more tricky. A floating point number is a number with a decimal where the number of digits after the decimal can be variable. A fixed point number is a number with a fractional component where the number of digits in the fractional portion is fixed.

In this quiz, we're going to write a class to implement a fixed point number with two fractional digits (e.g. 12.34, 3.00, or 1278.99). Assume that the range of the class should be -32768.99 to 32767.99, that the fractional component should hold any two digits, that we don't want precision errors, and that we want to conserve space.

4a) What type of member variable(s) do you think we should use to implement our fixed point number with 2 digits after the decimal? (Make sure you read the answer before proceeding with the next questions)

Show Solution

4b) Write a class named FixedPoint2 that implements the recommended solution from the previous question. If either (or both) of the non-fractional and fractional part of the number are negative, the number should be treated as negative. Provide the overloaded operators and constructors required for the following program to run:

```
int main()
{
    FixedPoint2 a{ 34, 56 };
    std::cout << a << '\n';

    FixedPoint2 b{ -2, 8 };
    std::cout << b << '\n';

    FixedPoint2 c{ 2, -8 };
    std::cout << c << '\n';

    FixedPoint2 d{ -2, -8 };
    std::cout << d << '\n';

    FixedPoint2 e{ 0, -5 };
    std::cout << e << '\n';

    std::cout << static_cast<double>(e) << '\n';

    return 0;
}
```

This program should produce the result:

```
34.56
-2.08
-2.08
-2.08
-0.05
-0.05
```

Hint: Although it may initially seem like more work initially, it's helpful to store both the non-fractional and fractional parts of the number with the same sign (e.g. both positive if the number is positive, and both negative if the number is negative). This makes doing math much easier later.
Hint: To output your number, first cast it to a double.

Show Solution

4c) Now add a constructor that takes a double. You can round a number (on the left of the decimal) by using the round() function (included in header cmath).

Hint: You can get the non-fractional part of a double by static casting the double to an integer
Hint: To get the fractional part of a double, you'll first need to zero-out the non-fractional part. Use the integer value to do this.
Hint: You can move a digit from the right of the decimal to the left of the decimal by multiplying by 10. You can move it two digits by multiplying by 100.

The follow program should run:

```
int main()
{
    FixedPoint2 a{ 0.01 };
    std::cout << a << '\n';

    FixedPoint2 b{ -0.01 };
    std::cout << b << '\n';

    FixedPoint2 c{ 5.01 }; // stored as 5.0099999... so we'll need to round this
    std::cout << c << '\n';

    FixedPoint2 d{ -5.01 }; // stored as -5.0099999... so we'll need to round this
    std::cout << d << '\n';

    return 0;
}
```

This program should produce the result

```
0.01
-0.01
5.01
-5.01
```

Show Solution

4d) Overload operator==, operator >>, operator- (unary), and operator+ (binary).

The following program should run:

```
void testAddition()
{
    // h/t to reader Sharjeel Safdar for this function
    std::cout << std::boolalpha;
    std::cout << (FixedPoint2{ 0.75 } + FixedPoint2{ 1.23 } == FixedPoint2{ 1.98 }) << '\n'; /
    std::cout << (FixedPoint2{ 0.75 } + FixedPoint2{ 1.50 } == FixedPoint2{ 2.25 }) << '\n'; /
    std::cout << (FixedPoint2{ -0.75 } + FixedPoint2{ -1.23 } == FixedPoint2{ -1.98 }) << '\n'
    std::cout << (FixedPoint2{ -0.75 } + FixedPoint2{ -1.50 } == FixedPoint2{ -2.25 }) << '\n'
    std::cout << (FixedPoint2{ 0.75 } + FixedPoint2{ -1.23 } == FixedPoint2{ -0.48 }) << '\n';
    std::cout << (FixedPoint2{ 0.75 } + FixedPoint2{ -1.50 } == FixedPoint2{ -0.75 }) << '\n';
    std::cout << (FixedPoint2{ -0.75 } + FixedPoint2{ 1.23 } == FixedPoint2{ 0.48 }) << '\n';
    std::cout << (FixedPoint2{ -0.75 } + FixedPoint2{ 1.50 } == FixedPoint2{ 0.75 }) << '\n';
}

int main()
{
    testAddition();

    FixedPoint2 a{ -0.48 };
    std::cout << a << '\n';

    std::cout << -a << '\n';

    std::cout << "Enter a number: "; // enter 5.678
    std::cin >> a;

    std::cout << "You entered: " << a << '\n';

    return 0;
}
```

And produce the output:

```
true
true
true
true
```

```
true
true
true
true
-0.48
0.48
Enter a number: 5.678
You entered: 5.68
```

Hint: Add your two FixedPoint2 together by leveraging the double cast, adding the results, and converting back to a FixedPoint2.
Hint: For operator>>, use your double constructor to create an anonymous object of type FixedPoint2, and assign it to your FixedPoint2 function parameter

Show Solution

10.1 -- Object relationships

Index

9.15 -- Shallow vs. deep copying

C++ TUTORIAL | PRINT THIS POST

271 comments to 9.x — Chapter 9 comprehensive quiz

« Older Comments 1 2 3 4

Matt

January 6, 2020 at 6:15 am · Reply

Man I'm rusty! Had to review a lot of chapters before picking up in ch.9 where I left off around a year ago. Slow and steady, I'll make it through the tutorials in this website one day.

Here's what I came up with for quiz #3:

IntArray.h

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <iostream>

class IntArray
{
private: // MEMBER VARIABLES
    int m_size{};
    int* m_array{ nullptr };

private: // MEMBER FUNCTIONS
    void deepCopy(const IntArray& arr);

public: // CONSTRUCTORS AND DESTRUCTORS
    IntArray(int size); // Default Constructor
    IntArray(const IntArray& arr); // Copy Constructor
    ~IntArray(); // Destructor

public: // MEMBER AND FRIEND FUNCTIONS
    IntArray& operator=(const IntArray& arr);

    int& operator[](int x);

    friend std::ostream& operator<<(
        std::ostream& cout, const IntArray& arr);
};

#endif
```

IntArray.cpp

```
#include <cassert>
#include "IntArray.h"
#include <iostream>

// Default Constructor
IntArray::IntArray(int size = 0) : m_size{ size }
{
    assert((size >= 0) && "Size of IntArray must not be negative.");

    if (size == 0) // allow an empty IntArray to be created
        m_array = nullptr;
    else
        m_array = new int[size]{0};
}

// Copy Constructor
IntArray::IntArray(const IntArray& arr) :
    m_size{ arr.m_size },
    m_array { nullptr }
{
    deepCopy(arr);
}
```

```

// Destructor
IntArray::~IntArray()
{
    delete[] m_array;
}

// Safely copies member variables including
// dynamic memory contained in IntArray objects
void IntArray::deepCopy(const IntArray& arr)
{
    m_size = arr.m_size;
    delete[] m_array;
    if (arr.m_array) // deep copy if there is something to copy
    {
        m_array = new int[m_size];
        for (int i{ 0 }; i < m_size; ++i)
        {
            m_array[i] = arr.m_array[i];
        }
    }
    else
    {
        m_array = nullptr;
    }
}

// PUBLIC MEMBER AND FRIEND FUNCTIONS
IntArray& IntArray::operator=(const IntArray& arr)
{
    if (this == &arr)
    {
        return *this;
    }
    deepCopy(arr);
    return *this;
}

int& IntArray::operator[](int x)
{
    assert((x >= 0 && x < m_size) && "IntArray::operator[]"
        "call out of bounds");
    return m_array[x];
}

std::ostream& operator<<(
    std::ostream& cout, const IntArray& arr)
{
    if (arr.m_array) // make sure it's not a nullptr
    {
        for (int x{ 0 }; x < arr.m_size; ++x)
        {
            cout << arr.m_array[x] << ' ';
        }
    }
    return cout;
}
```

main.cpp

```
#include "IntArray.h"
#include <iostream>

```

```
IntArray fillArray()
{
    IntArray a(5); // default constructor
    a[0] = 5; // overload operator[]
    a[1] = 8;
    a[2] = 2;
    a[3] = 3;
    a[4] = 6;

    return a;
}

int main()
{
    IntArray a = fillArray(); // deep copy constructor
    std::cout << a << '\n'; // overload operator<<

    auto& ref = a;
    a = ref; // overload operator=
    std::cout << a << '\n';

    IntArray b(1);
    b = a;
    std::cout << b << '\n';

    b = b;
    std::cout << b << '\n';

    return 0;
}
```

Output:
5 8 2 3 6
5 8 2 3 6
5 8 2 3 6
5 8 2 3 6

Thanks for your continued efforts on this site!

nascardriver
January 8, 2020 at 4:23 am · Reply

Default arguments go in the header. A user of your code (Assuming a library) doesn't care about the definitions, they'll never open a source file. All they need should be in the header.

The 0 in line 13 doesn't do anything. Empty curly braces initialize all elements to 0.

The rest looks good :)

Name
January 2, 2020 at 3:59 am · Reply

```
#include <iostream>
using namespace std;

int idGenerator()
{
    static int ids;
    ids++;
    return ids;
}

class A
{
public:
    int id;
    string name;

    A()
    {
        id = idGenerator();
        cout << "inside A() ; id -> " << id << '\n';
    }

    A(string name)
    {
        this->name = name;
        id = idGenerator();
        cout << "inside A(" << name << ") ; id -> " << id << '\n';
    }

    ~A()
    {
        cout << "inside ~(" << name << ")A ; id -> " << id << '\n';
    }
};

int main()
{
    A a = A("name 1");
    a = A("name 2");
    A b("name 3");
}
```

I was expecting output as:

```
inside A(name 1) ; id -> 1
inside A(name 2) ; id -> 2
inside ~(name 1)A ; id -> 1
inside A(name 3) ; id -> 3
inside ~(name 3)A ; id -> 3
inside ~(name 2)A ; id -> 2
```

but the output turns out to be:

```
inside A(name 1) ; id -> 1
inside A(name 2) ; id -> 2
inside ~(name 2)A ; id -> 2
inside A(name 3) ; id -> 3
inside ~(name 3)A ; id -> 3
inside ~(name 2)A ; id -> 2
```

WHY?

nascardriver
January 2, 2020 at 5:40 am · Reply

Assignment doesn't destroy objects.

Line 39 creates a new temporary `A`.
`a` is assigned that temporary, the member variables are copied to `a`, this doesn't destroy `a`.
The temporary dies at the end of line 39.

Name
January 2, 2020 at 7:53 am · Reply

so if there is a code like this

```
#include <iostream>
using namespace std;

int idGenerator()
{
    static int ids;
    ids++;
    return ids;
}

class A
{
public:
    int id;
    string name;
    int *tempMemory; // new code

    A()
    {
        id = idGenerator();
        tempMemory = new int; // new code
        cout << "inside A() ; id -> " << id << '\n';
    }

    A(string name)
    {
        this->name = name;
        id = idGenerator();
        tempMemory = new int; // new code
        cout << "inside A(" << name << ") ; id -> " << id << '\n';
    }

    ~A()
    {
        delete tempMemory; // new code
        cout << "inside ~(" << name << ")A ; id -> " << id << '\n';
    }
};

int main()
{
    A a = A("name 1");
    a = A("name 2");
    A b("name 3");
}
```

this should led to memory leak?

nascardriver
January 2, 2020 at 8:17 am · Reply

Not a leak, but a double delete, which causes undefined behavior.
Line 43 allocates memory and the assignment operator copies the pointer (Shallow copy). The temporary will delete the memory, then the destruction of `a` will try to delete the same memory again.

Ryan
December 11, 2019 at 4:54 am · Reply

so when we do std::cout << a;
in the function friend std::ostream& operator<<(std::ostream &out, const IntArray &array)

The array copies the contents in a? And we need a deep copy, because the compiler default copy initializer doesn't work for dynamic arrays?

nascardriver
December 11, 2019 at 6:02 am · Reply

No copy is made, `array` is a reference see lesson 6.11 and 7.3.

Ryan
December 11, 2019 at 9:21 am · Reply

Sorry, my bad. I was using some random online compiler. And the compiler is pretty weird and hard to understand, such that I thought it was using the copy initalizer after the overloading<< function.

So just to be clear. As IntArray fillToArray() object hadn't been created yet at run-time, it used the deep copy to copy it to object "a"?

Whereas when copying "a" to "b", as "a" had been created at that point in time, it used the assignment operator?

nascardriver
December 12, 2019 at 1:45 am · Reply

```
IntArray a = fillArray();
```

`fillArray` uses the copy constructor (which creates a deep copy) to copy `fillArray::a` to the caller (`main::a`).
In practice, this constructor call will most certainly be optimized away and `fillArray` will construct its `a` right into `main::a` without creating a copy.

```
b = a;
```

uses `operator=` because `b` already exists. The right side of the assignment doesn't matter.

R
December 9, 2019 at 4:34 am · Reply

Is it actually safe to use

```
m_base { static_cast<int_least16_t>(d) }
```

in 4(c) solution without applying rounding?
Will it not lose digits occasionally for same reason 5.01 might lose a digit in the fractional part?

nascardriver
December 9, 2019 at 4:47 am · Reply

That's exactly what we want to happen. The decimals before the point are stored separately from the decimals behind the point. By casting to an int, we extract the part before the point.

R
December 9, 2019 at 12:22 pm · Reply

Thanks, I took another look at how double values are stored and now I get it.

Anastasia
September 4, 2019 at 4:40 am · Reply

Hi!
Here's my 3 (with some wannabe std::vector<int> functionality :D).

@nascardriver, only code up to line 120 in `IntArray.cpp` (up to `empty()` function) is related to the quiz, so if you don't have time/ don't want to look at the rest - just ignore it. Although it would certainly help me a lot to see your suggestions and corrections about everything, any feedback is greatly appreciated!

And no pointer arithmetics there anymore :)

IntArray.h

```
#ifndef INT_ARRAY_H
#define INT_ARRAY_H

#include <iostream>

class IntArray {
    int* m_array { nullptr };
    int m_length {};
    int m_places_left {};

    void freeMemory();
    void getMemory(int);
    void makeDeepCopy(const IntArray&);

    bool outOfMemory() const;

public:
    IntArray(int = 1);
    ~IntArray();

    IntArray(const IntArray&);
    IntArray& operator= (const IntArray&);

    void copyElements(int*) const;
    void fill(int, int, int);

    bool empty() const;

    const int* begin() const;
```

```cpp
    const int* end() const;
    int back() const;
    int length() const;
    int capacity() const;

    int& operator[](int); // write/read index
    const int& operator[](int) const; // read-only index

    friend std::ostream& operator<< (std::ostream&, const IntArray&);

    // random overloads
    IntArray& operator<< (int); // push value
    IntArray& operator>> (int); // erase value (first occurence)

    IntArray& operator+ (int); // reserve memory
    void operator!();  // shrink_to_fit

    void operator+= (const IntArray&); // push elements of b into a
    void operator-= (const IntArray&); // erase values present in b from a
};

#endif
```

IntArray.cpp

```cpp
#include "IntArray.h"
#include <cassert>

std::ostream& operator<< (std::ostream& stream, const IntArray& array)
{
    stream << '[' << ' ';

    for (const auto& element : array)
        stream << element << ' ';

    stream << ']';

    return stream;
}

IntArray::IntArray(int memory_amount)
    : m_places_left { memory_amount }
{
    getMemory(memory_amount);
    fill(0, memory_amount, 0);
}

IntArray::~IntArray()
{
    freeMemory();
}

IntArray::IntArray(const IntArray& origin)
    : m_length { origin.m_length }
{
    makeDeepCopy(origin);
}

IntArray& IntArray::operator= (const IntArray& origin)
{
    if (&origin != this) {

        freeMemory();
        makeDeepCopy(origin);
    }

    return *this;
}

int& IntArray::operator[](int index)
{
    assert(index >= 0 && index < m_length);

    return m_array[index];
}

const int& IntArray::operator[](int index) const
{
    assert(index >= 0 && index < m_length);

    return m_array[index];
}

void IntArray::freeMemory()
{
    delete[] m_array;
    m_array = nullptr;
    m_length = 0;
    m_places_left = 0;
}

void IntArray::getMemory(int amount)
{
    assert(amount > 0);

    int* new_array { new int[static_cast<size_t>(amount)] };

    if (m_array) {

        copyElements(new_array);

        int new_length { m_length };

        freeMemory();

        m_length = new_length;
    }

    m_array = new_array;

    m_places_left = amount - m_length;
}

void IntArray::makeDeepCopy(const IntArray& origin)
{
    m_length = origin.m_length;
    getMemory(m_length);
    origin.copyElements(m_array);
}

void IntArray::copyElements(int* copy) const
{
    for (int element { 0 }; element < m_length; ++element)
        copy[element] = m_array[element];
}
```

```

void IntArray::fill(int begin, int end, int value)
{
    for (int element { begin }; element <= end; ++element) {
        m_array[element] = value;

        if (element > m_length) {
            ++m_length;
            --m_places_left;
        }

        if (outOfMemory() && element != end)
            *(this) + (end - element);
    }
}

bool IntArray::outOfMemory() const
{
    return (m_places_left <= 0);
}

bool IntArray::empty() const
{
    return (m_length <= 0);
}

const int* IntArray::begin() const
{
    assert(m_array);

    return &m_array[0];
}

const int* IntArray::end() const
{
    assert(m_array);

    return &m_array[m_length];
}

int IntArray::back() const
{
    return m_array[m_length - 1];
}

int IntArray::length() const
{
    return m_length;
}

int IntArray::capacity() const
{
    return m_places_left + m_length;
}

IntArray& IntArray::operator+ (int amount)
{
    if (amount > 0)
        getMemory(amount + capacity());

    return *this;
}
```

```

void IntArray::operator!()
{
    getMemory(m_length);
}

IntArray& IntArray::operator<< (int value)
{
    if (outOfMemory())
        *(this) + 10;

    m_array[m_length++] = value;
    --m_places_left;

    return *this;
}

IntArray& IntArray::operator>> (int value)
{
    int index_to_erase {};

    while (m_array[index_to_erase] != value) {
        ++index_to_erase;

        if (index_to_erase >= m_length)
            return *this;
    }

    int index { index_to_erase };

    for (; index < m_length - 1; ++index)
        m_array[index] = m_array[index + 1];

    --m_length;
    ++m_places_left;

    return *this;
}

void IntArray::operator+= (const IntArray& that)
{
    if (capacity() < m_length + that.m_length)
        *(this) + (m_length + that.m_length - capacity());

    for (int element { 0 }; element < that.m_length; ++element)
        *(this) << that.m_array[element];
}

void IntArray::operator-= (const IntArray& that)
{
    for (int element { 0 }; element < that.m_length; ++element)
        *(this) >> that.m_array[element];
}
```

alexMain.cpp

```
#include "IntArray.h"
#include <iostream>

IntArray fillArray()
{
    IntArray a { 5 };
    a[0] = 5;
    a[1] = 8;
    a[2] = 2;
    a[3] = 3;
    a[4] = 6;

    return a;
}

int main()
{
    IntArray a { fillArray() };
    std::cout << a << '\n';

    auto& ref { a }; // we're using this reference
    // to avoid compiler self-assignment errors
    a = ref;

    IntArray b { 1 };
    b = a;

    std::cout << b << '\n';

    return 0;
}
```

I don't like my `fill()` function (lines 102-115) - neither using it in the constructor, nor its implementation (although I tried to make it as useful as possible). But since Alex was using `[]` operator on the memory which would be inaccessible to this operator in my implementation (because I don't initialize it when allocating (for several reasons)) I had to add it :/ Any thoughts on what could I've possibly done instead are welcome.
Also, Valgrind is complaining about 4 bytes (size of an int on my system) being written by `fill()` to an address it shouldn't access to (valgrind thinks so, anyway). 2 times actually (because the default constructor which calls this function was invoked 2 times). I don't know whether it's something I should be worried about.

Not quiz related:
I know that my operator overloadings may not have much logical sense, but because this is what the chapter was about, this was what I wanted to practice. I've made all the overloadings member functions, because all of them are modifying the left operand(this). I'd like to know whether it was the right approach (I don't mean here the operators required by the quiz -- with those I didn't have much choice). And of course I'd very appreciate any suggestions on how to make the implementations more efficient as well.

edit: fixed an error -- lines 112-113 should be

```
if (outOfMemory() && element != end)
    *(this) + (end - element);
```

not:

```
if (outOfMemory() && element != end)
    getMemory(end - element);
```

nascardriver
September 4, 2019 at 8:08 am · Reply

`m_places_left`. Interesting, I've only every seen "capacity". You need to update `m_places_left` whenever the array changes, capacity would stay the same.
You already noticed that your operators aren't quite self-explanatory. I suggest you to get into the habit of documenting your code using documentation comments. They help the user of your code (Yourself, if the project is big enough) and can provide rich autocomplete if your editor supports it.

```
/**
 * A description of what the function does. Blub blubber bla.
 * Bounds-checks are only performed when assertions are enabled.
 * This is the style of documentation I'm used to from JS/TS. I
 * don't know if it's compatible with a CPP doc generator.
 * @param index Index of the element to get.
 * @return Element at @index
 */
int& operator[](int index);

/**
 * Comment about implementation decisions. Usually empty because the
 * code is self-explanatory.
 */
int& operator[](int index)
{
  // ...
}
```

- If you share your code as a library, the user will only see the header. If there are no parameter names in the header, your library is very hard to use.
- Use `std::size_t`, `size_t` is C.
- In the `std` library, whenever you supply an end index, that index is exclusive. Your `fill` uses an inclusive end index (Should be mentioned in a doc-comment). That's fine, but might cause trouble (At least thinking time) when your class is used alongside something from the `std` library.
- When you supply operators to do something, also add a named function. Otherwise you have to do things like in line 113 when you have a pointer.
- Line 131, 138: That's undefined behavior if `m_places_left` is 0. This is a case where pointer arithmetic should be used.

```
// Array
   &m_array[m_length]
=  &(*(m_array + m_length))
//   ^^^^^^^^^^^^^^^^^^^^^ UB

// Pointer arithmetic
(m_array + m_length)
// No indirection, no problem.
```

- If you're using the initial value of a variable, explicitly initialize it (Line 182).
- Line 191: Should be declared in the loop's header.
- Line 204, 205: The duplicate calculation can be removed by calculating the missing capacity first and comparing it to 0.
- Line 207, 208: You know that you won't run out of memory and you can calculate the new sizes. `operator<<` is wasteful in this scenario.

> neither using it [fill] in the constructor
> since Alex was using `[]` operator on the memory which would be inaccessible to this operator
But he writes to it, that's ok. If you add a doc-comment stating that the memory is uninitialized, there's no problem.
The constructor doesn't need to use `getMemory`. All you need is line 71 (modified to default-initialize into `m_array`).

> nor its [fill's] implementation
- Line 107-113 should be done before the loop starts.

> Valgrind is complaining
`fill`'s `end` is inclusive.
Please, whenever you get an error or something, post the full error message with line numbers.

> I've made all the overloadings member functions [...]. I'd like to know whether it was the right approach

Yes. None of them could have a lhs operand that's not an instance of your class. Unless you think `5 + arr` makes sense. I'd say your `operator+` should be commutative.

> more efficient
You're overusing `freeMemory`. None of the places you're using `freeMemory` in actually needs to do everything that `freeMemory` does.
If you apply my comments, I don't see anything more as of now.

Anastasia
September 4, 2019 at 10:02 am · Reply

> `m_places_left`. Interesting
It was the first thing that came to my mind as a control tool of the allocated memory. It probably seems as a naive approach (not only this, but I'd guess all of my implementations), because it was my first attempt of writing something of the sort.

> If there are no parameter names in the header, your library is very hard to use.
Can I provide parameter names just for some of the functions, those that are not self-explanatory (like `fill()` or op.overloadings), not for all?

> Use `std::size_t`, `size_t` is C.
Fixed. I just casted to the most reasonable type (which accidentally happened to exist) that came to my mind in order to get rid of the warning. It's awful programming, I know :D

> When you supply operators to do something, also add a named function
I'm sorry that it wasn't very clear, can I just add a commentary next time (instead of duplicating code with named functions)?

> Line 131, 138: That's undefined behavior if `m_places_left` is 0. This is a case where pointer arithmetic should be used.
Thank you, I'd have never thought it's UB! Maybe it's safer (for now) to use pointers everywhere again? Because I just can't see the situations like that...

> If you're using the initial value of a variable, explicitly initialize it (Line 182).
- Line 191: Should be declared in the loop's header.
line 182 -- fair enough, line 191 -- it was intentional, otherwise the loop's header would be too long. I could use the same variable (index_to_erase) in the second loop as well, but I wanted to differentiate between those two things (finding the right index and shifting all the others), for clarity. I could also make the name of `index_to_erase` shorter, but again, I wanted it be self-explanatory, so I decided to move it outside of the header - it seemed readable and clear to me.

> Line 204, 205: The duplicate calculation can be removed by calculating the missing capacity first and comparing it to 0.
- Line 207, 208: You know that you won't run out of memory and you can calculate the new sizes. `operator<<` is wasteful in this scenario.

I simplified it to this:

```cpp
void IntArray::operator+= (const IntArray& that)
{
    for (const auto &element : that)
        *(this) << element;
}
```

And same for the `operator-=`. You're right `<<` can take care of memory anyway.

> The constructor doesn't need to use `getMemory`. All you need is line 71 (modified to default-initialize into `m_array`).
I see your point, but I don't like the idea of the memory requested by the user being initialized right away -- they are supposed to push the values (or copy them) first, before accessing them. This way

(intializing it with 0s) I'm forcing the user to use `[]` (like Alex did) to overwrite the values (this memory it meant to be overwritten, why bother filling it -- I did it, but just for the quiz).

> Line 107-113 should be done before the loop starts.
Thank you, I think I also should reserve memory (if needed) before running the loop, instead of constantly checking it :/

> Valgrind is complaining -- post the full error message with line numbers
Oh, there's no line numbers - just memory addresses. It's not very clear and quite long. But in case it would help to identify the problem (2 identical errors like the one below):

```
==11316== Invalid write of size 4
==11316==    at 0x1097BD: IntArray::fill(int, int, int) (in /home/an/Documents/Cp
==11316==    by 0x1096A6: IntArray::IntArray(int) (in /home/an/Documents/Cpp/Less
==11316==    by 0x1092A0: fillArray() (in /home/an/Documents/Cpp/Lessons/IntArray
==11316==    by 0x1093BE: main (in /home/an/Documents/Cpp/Lessons/IntArray/a.out)
==11316==  Address 0x4da2c94 is 0 bytes after a block of size 20 alloc'd
==11316==    at 0x483950F: operator new[](unsigned long) (vg_replace_malloc.c:423
==11316==    by 0x10971B: IntArray::getMemory(int) (in /home/an/Documents/Cpp/Les
==11316==    by 0x109692: IntArray::IntArray(int) (in /home/an/Documents/Cpp/Less
==11316==    by 0x1092A0: fillArray() (in /home/an/Documents/Cpp/Lessons/IntArray
==11316==    by 0x1093BE: main (in /home/an/Documents/Cpp/Lessons/IntArray/a.out)
==11316==
[ 5 8 2 3 6 ]
```

> [right approach] Yes. None of them could have a lhs operand that's not an instance of your class. Unless you think `5 + arr` makes sense. I'd say your `operator+` should be commutative.
I'm glad something I did right. And I don't think `5 + arr` makes sense... If by saying that it should you are intending that my overloading of the `operator+` was silly - I totally agree :D

> You're overusing `freeMemory`
I may need in the future... It's more `reset` than `freeMemory` though :/

Thank you so much for all the suggestions, I'll try to apply everything (to this and my future codes as well).
And you are right - I need to learn to properly comment my code, thanks for your advice about that as well!

Anastasia
September 4, 2019 at 1:24 pm · Reply

Please, ignore the error output. Valgrind is happy after I moved the assignment line (`m_array[element] = value;`) in `fill()` function under the things making sure that m_length and memory are in order (as you said). Thank you!

nascardriver
September 5, 2019 at 12:08 am · Reply

> Can I provide parameter names just for some of the functions, those that are not self-explanatory
Apart from the copy constructor and copy assignment operator, I don't think any parameters are self-explanatory.

> When you supply operators to do something, also add a named function\

```
IntArray* pArray{ /* comes from somewhere */ };

// I want to push an element
```

```

pArray->operator<<(3);

*pArray << 3;
```

Neither of these is particularly attractive. It'd be nicer to have a named function.

```
pArray->push(3);
```

The operators then just forward to call to the named function.

> Maybe it's safer (for now) to use pointers everywhere again?
If you want to have the address of an element, you can use pointer arithmetic. If you want to have the element itself, use array syntax.

> I just can't see the situations like that
You're accessing an element that doesn't exist. If you know the size of your array, this is relatively easy to spot. You have to learn it at some point, so you might as well do it right from the beginning.

> the loop's header would be too long
You seem to be targeting a very short line length. Sooner or later you'll have to start splitting lines

```
for (int index{ index_to_erase };
     index < m_length - 1;
     ++index)
{
  m_array[index] = m_array[index + 1];
}
```

You shouldn't let a naming or formatting policy decrease your code's quality.

> You're right `<<` can take care of memory anyway.
Nooo! That's worse than before.
If `that` has 10000 elements, you'll be calling `outOfMemory` 10000 times and and reallocate up to 1000 times. In the previous version you at least didn't have the reallocations.

```
// Untested
void IntArray::operator+=(const IntArray& that)
{
  int iNewLength{ this->m_length + that.m_length };

  // Take care of all allocations first
  if (iNewLength > this->capacity())
  {
    this->getMemory(iNewLength);
  }

  // You already have a function to copy an array
  that.copyElements(this->m_array + this->m_length);

  this->m_length = iNewLength;
}
```

> I don't like the idea of the memory requested by the user being initialized right away
If it's initialization vs no initialization and assignment, initialization wins.

> This way (intializing it with 0s) I'm forcing the user to use `[]` (like Alex did) to overwrite the values
I don't see how that's related or where you're forcing the user to do anything.
If your goal is to get close to `std::vector`, the fill constructor should default-initialize the elements and they should be accessible.

> Oh, there's no line numbers
Are you building in debug mode?

```
==9506== Invalid write of size 4
==9506==    at 0x40251D: IntArray::fill(int, int, int) (src/main.cpp:156)
==9506==    by 0x402405: IntArray::IntArray(int) (src/main.cpp:71)
==9506==    by 0x402C0B: fillArray() (src/main.cpp:271)
==9506==    by 0x402D0E: main (src/main.cpp:283)
```

valgrind-3.15.0

> I moved the assignment line [...]
Your fill is wrong and/or you're using it wrong. Rubber duck your code with an array of length 1.

```
IntArray arr(1); // I expect this to create an array with one element, a 0.
// This calls IntArray(1)
// That calls fill(0, 1, 0);
// That runs for (int element { 0 }; element <= 1; ++element)
// How often does this loop run?
// How many elements are in the array when the loop is finished?
// Why did valgrind complain?
```

Anastasia
September 5, 2019 at 2:38 am · Reply

> Apart from the copy constructor and copy assignment operator, I don't think any parameters are self-explanatory.
I my case yes, they aren't... But I wanted to know whether it would be considered weird (or bad practice) if I provide parameter names just for some functions, but not for all of them.

> You shouldn't let a naming or formatting policy decrease your code's quality.
I think it's an experience thing - at the moment it's hard to spot things that are undesirable and negatively impacting the code quality. Is it okay to split for-loop's header like that?

> Nooo! That's worse than before.
Oops, sorry :(I like your version a lot - it's clear what it does and why and it doesn't duplicate code (using `copyElements()` didn't even come to my mind).

> I don't see how that's related or where you're forcing the user to do anything.
I probably just wanted to force them to use `<<` or `+=` :D

> Are you building in debug mode?
Nope, I didn't set the flag while compiling...Sorry for that. I'm neglecting and ignoring debugging for the most part, using only `cerr`s and occasionally basic valgrind memcheck when it's about allocated memory :/ Need to re-read chapter 3 about debugging - but I believe it's all about IDEs...

> Your fill is wrong
I think I got it, thanks. I didn't like what I did there from the very beginning.

The rest is clear - thank you for all the replies!

nascardriver
September 5, 2019 at 2:51 am · Reply

> I wanted to know whether it would be considered weird (or bad practice) if I provide parameter names just for some functions, but not

for all of them.
I'd say yes.

> Is it okay to split for-loop's header like that?
No. `for`-loops have a place for declarations of new variables so you should use it. Only declare the iterator outside of the loop if you need it after the loop (eg. to check if the loop terminated early).

> Need to re-read chapter 3 about debugging
This was just about building in debug mode. When you build in debug mode, identifiers and file mappings are preserved, so debuggers can resolve addresses to your source code. In release mode, all such information is stripped, because your computer doesn't need it. -g is the flag you need.

Anastasia
September 5, 2019 at 2:58 am · Reply

> `for`-loops have a place for declarations of new variables so you should use it.
I mean having everything in the header, but splitting it in several lines (like you did).

> -g is the flag you need.
Thanks a lot!

nascardriver
September 5, 2019 at 3:02 am · Reply

cpp-wise yes. The only thing preventing you from doing so could be your style convention. If it enforces a line length limit, it might also tell you when and where to split lines. If there is no rule for line splitting, split wherever you please as long as it's readable and consistent.

Anastasia
September 5, 2019 at 3:09 am · Reply

Ok, I'll try to do it this way next time, if necessary.

Thank you once again for taking the time to explain and clarify everything, I learned so much!

noobmaster
August 18, 2019 at 7:51 am · Reply

For solution number 3, why can't we initialize int *m_array with new int[m_length] instead of nullPtr(line 8) and then comment out line 16 and 24 ?

nascardriver
August 18, 2019 at 7:58 am · Reply

`m_length` is 0 in line 8. You've got to wait until you know how the the array is supposed to be.

DecSco
August 2, 2019 at 3:19 am · Reply

Here's an option for the operator+ without casting to double:

```
1  FixedPoint2 operator+ ( const FixedPoint2& a, const FixedPoint2& b )
2  {
3      // store preliminary results in temporary variables
4      int16_t base { static_cast<int16_t>(a.m_base + b.m_base) };
5      // for storing the decimal, int8 is enough. For calculating, it isn't.
6      int16_t decimal { static_cast<int16_t>(a.m_decimal + b.m_decimal) };
7
8      /*
9      If the two numbers have the same sign, we can check for overflow directly.
10     If the two numbers have different signs, overflow occurs when
11     the number with the larger absolute has the smaller absolute decimal.
12     Due to commutativity, (a > -b) && (a.m_decimal < -b.m_decimal) covers
13     both cases such as 2.1 - 1.2 as well as -1.2 + 2.1.
14     */
15     if ( (decimal <= -100) || (a > -b) && (a.m_decimal < -b.m_decimal) )
16         return FixedPoint2(base-1, decimal+100);
17
18     if ( (decimal >=  100) || (a < -b) && (a.m_decimal > -b.m_decimal) )
19         return FixedPoint2(base+1, decimal-100);
20
21     /*
22     the base case also handles a == -b as well as a.m_decimal == -b.m_decimal
23     */
24     return FixedPoint2(base, decimal);
25 }
```

Feedback welcome!

nascardriver
August 2, 2019 at 6:05 am · Reply

- Don't use `std::int*_t` it's not guaranteed to exist.
- Enable compiler warnings, read them, fix them.
- Line 16, 19, 24: Brace initialization.

You can use a sign variable to combine the 2 ifs

```
FixedPoint2 operator+(const FixedPoint2& a, const FixedPoint2& b)
{
  auto iBase{ static_cast<decltype(FixedPoint2::m_base)>(a.m_base + b.m_base) };
  auto iDecimal{ static_cast<decltype(FixedPoint2::m_decimal)>(a.m_decimal + b.m_deci

  // There's also @std::copysign, but it uses floating point conversions.
  int iSign{ (iDecimal < 0) ? -1 : 1 };

  if ((std::abs(iDecimal) >= 100) || (((iSign * a) < (-iSign * b)) && ((iSign * a.m_d
  {
    return {
      static_cast<decltype(FixedPoint2::m_base)>(iBase + iSign),
      static_cast<decltype(FixedPoint2::m_decimal)>(iDecimal - (iSign * 100))
    };
  }
  else
  {
    return { iBase, iDecimal };
  }
}
```

Your version is easier to understand, but I spent more time on this code than I'd like to admit so I might as well share it.

DecSco
August 2, 2019 at 7:28 am · Reply

Thanks! I somehow completely forgot that the return statement itself creates an object!
The quiz instructed to use those int*_t types - the solution differs from that already, though. What about int_least8_t? Same issue?
I wanted to write code without using abs(), and I don't want to admit how long it took me, either :D

nascardriver
August 2, 2019 at 7:42 am · Reply

All the `std::int*_t` types aren't guaranteed. They only exist if the implementation supports them. Fast and least types are always available.

DecSco
July 31, 2019 at 9:42 am · Reply

The initialisation for the FixedPoint2 constructor from a double can (should?) be moved to an initialiser list:

```
FixedPoint2(double d) :
    m_base { static_cast<int_least16_t>(d) },
    m_decimal {static_cast<std::int_least8_t>(round((d - m_base) * 100)) }
{}
```

Alex
August 3, 2019 at 9:23 pm · Reply

Updated as suggested. Thanks!

DecSco
July 31, 2019 at 8:01 am · Reply

Does it make any difference, precision wise, which of the member variables you cast to a double?

```
std::ostream& operator<< ( std::ostream& out, const Average& a )
{
    return out << a.m_sum / static_cast<double>( a.m_count ) ;
}
```

I thought that, because you lose precision for high integer numbers when converting them to a double, it would be better to cast the lower number to a double. But then I thought that probably, the floating point division will produce the same result either way, because both numbers are converted, right?

nascardriver
July 31, 2019 at 9:22 am · Reply

It doesn't make a difference.

« Older Comments 1 2 3 4