10.7 — std::initializer_list

BY ALEX ON MARCH 9TH, 2017 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

Consider a fixed array of integers in C++:

```
int array[5];
```

If we want to initialize this array with values, we can do so directly via the initializer list syntax:

```
int main()
{
    int array[5] { 5, 4, 3, 2, 1 }; // initializer list
    for (int count=0; count < 5; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This prints:

```
5 4 3 2 1
```

This also works for dynamically allocated arrays:

```
int main()
{
    int *array = new int[5] { 5, 4, 3, 2, 1 }; // initializer list
    for (int count = 0; count < 5; ++count)
        std::cout << array[count] << ' ';
    delete[] array;

    return 0;
}
```

In the previous lesson, we introduced the concept of container classes, and showed an example of an IntArray class that holds an array of integers:

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

class IntArray
{
private:
    int m_length;
    int *m_data;

public:
    IntArray():
        m_length(0), m_data(nullptr)
    {
    }

    IntArray(int length):
        m_length(length)
    {
        m_data = new int[length];
    }

    ~IntArray()
```

```
    {
        delete[] m_data;
        // we don't need to set m_data to null or m_length to 0 here, since the object will be
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};
```

What happens if we try to use an initializer list with this container class?

```
int main()
{
    IntArray array { 5, 4, 3, 2, 1 }; // this line doesn't compile
    for (int count=0; count < 5; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This code won't compile, because the IntArray class doesn't have a constructor that knows what to do with an initializer list. As a result, we're left initializing our array elements individually:

```
int main()
{
    IntArray array(5);
    array[0] = 5;
    array[1] = 4;
    array[2] = 3;
    array[3] = 2;
    array[4] = 1;

    for (int count=0; count < 5; ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

That's not so great.

Prior to C++11, list initialization could only be used with static or dynamic arrays. However, as of C++11, we now have a solution to this problem.

Class initialization using std::initializer_list

When a C++11 compiler sees an initializer list, it automatically converts it into an object of type std::initializer_list. Therefore, if we create a constructor that takes a std::initializer_list parameter, we can create objects using the initializer list as an input.

std::initializer_list lives in the <initializer_list> header.

There are a few things to know about std::initializer_list. Much like std::array or std::vector, you have to tell std::initializer_list what type of data the list holds using angled brackets. Therefore, you'll never see a plain std::initializer_list. Instead, you'll see something like std::initializer_list<int> or std::initializer_list<std::string>.

Second, std::initializer_list has a (misnamed) size() function which returns the number of elements in the list. This is useful when we need to know the length of the list passed in.

Let's take a look at updating our IntArray class with a constructor that takes a std::initializer_list.

```
#include <cassert> // for assert()
#include <initializer_list> // for std::initializer_list
#include <iostream>

class IntArray
{
private:
    int m_length {};
    int *m_data {};

public:
    IntArray()
    {
    }

    IntArray(int length) :
        m_length(length)
    {
        m_data = new int[length];
    }

    IntArray(const std::initializer_list<int> &list) : // allow IntArray to be initialized via
        IntArray(static_cast<int>(list.size())) // use delegating constructor to set up initia
    {
        // Now initialize our array from the list
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }
    }

    ~IntArray()
    {
        delete[] m_data;
        // we don't need to set m_data to null or m_length to 0 here, since the object will be
    }

    IntArray(const IntArray&) = delete; // to avoid shallow copies
    IntArray& operator=(const IntArray& list) = delete; // to avoid shallow copies

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() { return m_length; }
};

int main()
{
    IntArray array{ 5, 4, 3, 2, 1 }; // initializer list
    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This produces the expected result:

```
5 4 3 2 1
```

It works! Now, let's explore this in more detail.

Here's our IntArray constructor that takes a std::initializer_list<int>.

```
1   IntArray(const std::initializer_list<int> &list): // allow IntArray to be initialized via
2       IntArray(static_cast<int>(list.size())) // use delegating constructor to set up initia
3   {
4       // Now initialize our array from the list
5       int count = 0;
6       for (int element : list)
7       {
8           m_data[count] = element;
9           ++count;
10      }
11  }
```

On line 1: As noted above, we have to use angled brackets to denote what type of element we expect inside the list. In this case, because this is an IntArray, we'd expect the list to be filled with int. Note that we also pass the list by const reference, so we don't make an unnecessary copy of the std::initializer_list when it's passed to our constructor.

On line 2: We delegate allocating memory for the IntArray to the other constructor via a delegating constructor (to reduce redundant code). This other constructor needs to know the length of the array, so we pass it list.size(), which contains the number of elements in the list. Note that list.size() returns a size_t (which is unsigned) so we need to cast to a signed int here.

The body of the constructor is reserved for copying the elements from the list into our IntArray class. For some inexplicable reason, std::initializer_list does not provide access to the elements of the list via subscripting (operator[]). The omission has been noted many times to the standards committee and never addressed.

However, there are easy ways to work around the lack of subscripts. The easiest way is to use a for-each loop here. The for-each loops steps through each element of the initialization list, and we can manually copy the elements into our internal array.

One caveat: Initializer lists will always favor a matching initializer_list constructor over other potentially matching constructors. Thus, this variable definition:

```
1   IntArray array { 5 };
```

would match to IntArray(const std::initializer_list<int> &), not IntArray(int). If you want to match to IntArray(int) once a initializer_list constructor has been defined, you'll need to use copy initialization or direct initialization.

Class assignment using std::initializer_list

You can also use std::initializer_list to assign new values to a class by overloading the assignment operator to take a std::initializer_list parameter. This works analogously to the above. We'll show an example of how to do this in the quiz solution below.

Note that if you implement a constructor that takes a std::initializer_list, you should ensure you do at least one of the following:

1. Provide an overloaded list assignment operator
2. Provide a proper deep-copying copy assignment operator

Here's why: consider the above class (which doesn't have an overloaded list assignment or a copy assignment), along with following statement:

```
array = { 1, 3, 5, 7, 9, 11 }; // overwrite the elements of array with the elements from th
```

First, the compiler will note that an assignment function taking a std::initializer_list doesn't exist. Next it will look for other assignment functions it could use, and discover the implicitly provided copy assignment operator. However, this function can only be used if it can convert the initializer list into an IntArray. Because { 1, 3, 5, 7, 9, 11 } is a std::initializer_list, the compiler will use the list constructor to convert the initializer list into a temporary IntArray. Then it will call the implicit assignment operator, which will shallow copy the temporary IntArray into our array object.

At this point, both the temporary IntArray's m_data and array->m_data point to the same address (due to the shallow copy). You can already see where this is going.

At the end of the assignment statement, the temporary IntArray is destroyed. That calls the destructor, which deletes the temporary IntArray's m_data. This leaves our array variable with a hanging m_data pointer. When you try to use array->m_data for any purpose (including when array goes out of scope and the destructor goes to delete m_data), you'll get undefined results (and probably a crash).

Rule: If you provide list construction, it's a good idea to provide list assignment as well.

Summary

Implementing a constructor that takes a std::initializer_list parameter (by reference to prevent copying) allows us to use list initialization with our custom classes. We can also use std::initializer_list to implement other functions that need to use an initializer list, such as an assignment operator.

Quiz time

1) Using the IntArray class above, implement an overloaded assignment operator that takes an initializer list.

The following code should run:

```
int main()
{
    IntArray array { 5, 4, 3, 2, 1 }; // initializer list
    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    std::cout << '\n';

    array = { 1, 3, 5, 7, 9, 11 };

    for (int count = 0; count < array.getLength(); ++count)
        std::cout << array[count] << ' ';

    return 0;
}
```

This should print:

```
5 4 3 2 1
1 3 5 7 9 11
```

Show Solution


10.x -- Chapter 10 comprehensive quiz


Index


10.6 -- Container classes

C++ TUTORIAL | PRINT THIS POST

123 comments to 10.7 — std::initializer_list

« Older Comments 1 2

cnoob
January 6, 2020 at 1:57 pm · Reply

hi! In the Quiz line 43:

```
1 IntArray& operator=(const std::initializer_list<int> &list)
```

If the std::initializer_list is not const, the compiler says that Im refering to an explicitly deleted function. Why? I know that we dont want to change the list so it should be const, but why cant the compiler tell the deleted operator from the other without the new one being const, even though their parameters are different anyway?

nascardriver
January 8, 2020 at 5:21 am · Reply

```
1 std::initializer_list<int> &
```

can't bind to an r-value initializer list, so the constructor ignores this function.

The compiler has to find some other way of compiling

```
array = { 1, 3, 5, 7, 9, 11 };
```

The only other way is to construct an `IntArray` from the initializer list, then use `operator=(const IntArray& list)`. That would work, but this operator is deleted.

cnoob
January 8, 2020 at 2:22 pm · Reply

Now I get it, thanks!

arcad
December 27, 2019 at 2:58 pm · Reply

In the quiz exercise, why can't we just call the constructor that takes ::initializer_list as a parameter, instead of using another loop in the implementation of the assignment operator?

nascardriver
December 28, 2019 at 2:33 am · Reply

Calling the constructor would create a new object, not modify the current one. You could add another function that sets the current object from an `std::initializer_list` and call that from the constructor and assignment operator.

arcad
December 30, 2019 at 11:04 am · Reply

Thanks!

Ryan
December 12, 2019 at 6:38 am · Reply

For example if we had.

```
int main()
{
    IntArray array { 5, 4, 3, 2, 1 }; // initializer list
    array = { 5, 4, 3, 2, 1 }; // assignment operator
    return 0;
}
```

This would just recopy the same numbers into the list again! I'd say that's inefficient.
There is no implicit conversion for an int array to an initializer list so (this == &list) would not work.

The function can be changed a bit to improve the efficiency.

```
int count{ 0 };
for (int element : list)
{
    if(m_array[count] != element)
        m_array[count] = element;
    ++count;
}
return *this;
```

However I was wondering, how we could do an overloading type to std::inilizer_list such that if we did,

```
if(this == &list)
      return *this;
```

it would work.

nascardriver
December 12, 2019 at 7:17 am · Reply

Self-assignments are extremely rare. We're checking for self-assignments to prevent errors, not for efficiency.
Without testing, I'd say your code is less efficient than an implementation that copies the values unconditionally. A quick test confirms my assumption, at least on my system.
For custom types, your version prevents the type from doing its own self-assignment check, which might be a lot faster than yours.

```
if(this == &list)
      return *this;
```

This can never work, because you're comparing pointers of different types. You can use `std::equal` to compare lists.

```
if (std::equal(list.begin(), list.end(), this->m_array))
{
  std::cout << "same list\n";
}
```

Ryan
December 12, 2019 at 8:21 am · Reply

But isn't self-assignment considered dangerous or time wasting in assignment operators 9.14.
I know that
if (length != m_length)
pretty much prevents the object to point to a garbage address, such that it ensures an identical object's array does not get go out of the heap. Am i correct?
So comparable to what I did, what does the compiler do in it's own self-assignment check.

Also out of curiosity, would a copy and swap idiom algorithm be an good alternative?

nascardriver
December 12, 2019 at 8:31 am · Reply

Self-assignment is dangerous and time-wasting. You can fix the dangerous part, but not the time-wasting part. The only real way to not waste time is to not assign an object to itself in the first place.

> if (length != m_length)
This check is optional. If the old and new array have the same length, we can re-use the memory. The check could also be `if (length < = m_length)`. That way we'd waste some memory, but we don't have to re-allocate. > what does the compiler do in it's own self-assignment check
There's no self-assignment check in the compiler. Built-in types don't need self-assignment checks, because a self-assignment isn't dangerous for them.

> would a copy and swap idiom algorithm be an good alternative?
Yes. If I recall correctly, Alex wants to integrate this into the lessons.

hellmet
November 21, 2019 at 7:04 am · Reply

Hmm I think I'm missing a nuance here. One prefers uniform initialization to prevent copies. But, in the std::initializer_list constructor, I'm copying the elements one by one. I can't think of a way to prevent those copies though. We have n copies here, else, without uniform init, we'd have 2n copies. Well better this than 2n I guess!

Can't I cheat by assigning references to the elements in the constructor? Lifespan extension through reference? But I think that was only for r-value const references though, so the 'cheat' wouldn't work here, unless the array itself as const).

Ken
November 5, 2019 at 2:49 pm · Reply

In the solution of the quiz, when you return *this do you return pointer to the entire class?

nascardriver
November 6, 2019 at 3:57 am · Reply

`this` is a pointer, `*this` is a reference. If you're unsure about what `this` is, re-read lesson 8.8.

sareo
October 8, 2019 at 6:19 pm · Reply

Hi!
My original answer to this quiz was:

```
1  IntArray& operator=(const std::initializer_list<int>& list)
2      {
3          m_length = static_cast<int>(list.size());
4          delete[] m_data;
5          m_data = new int[m_length];
6
7          int count = 0;
8          for (int element : list)
9          {
10             m_data[count] = element;
11             ++count;
12         }
13         return *this;
14     }
```

And I get a warning "Buffer overrun while writing to 'm_data': the writable size is 'm_length*4' bytes, but '8' bytes might be written."
When I replace lines 3-5 with your answer, that warning goes away.

```
1          int length{ static_cast<int>(list.size()) };
2      if (length != m_length)
3          {
4              delete[] m_data;
5              m_length = length;
6              m_data = new int[length];
7          }
```

I'm not understanding what is relevantly different between the two. Can you tell me why that warning is occurring, and how the second piece of code fixes it?

nascardriver
October 9, 2019 at 2:44 am · Reply

There is no buffer overrun in your code. The warning is wrong.

sareo
October 9, 2019 at 3:26 am · Reply

Phew, I was all kinds of confused! Thanks!

Atas
July 11, 2019 at 4:27 am · Reply

"Rule: If you provide list construction, it's a good idea to provide list assignment as well"

Why should one prefer overloading list assignment over regular assignment? The latter seems more general, isn't that's a good thing?

nascardriver
July 11, 2019 at 5:30 am · Reply

If you provide a list constructor but not a list assignment operator and you assign a list, you're calling the list constructor and then the copy/move constructor, which is slower than immediately calling a list assignment operator.

Dimbo1911
July 4, 2019 at 12:11 am · Reply

Good morning, how does this code seem?

```
1  #include <cassert>
2  #include <iostream>
3  #include <initializer_list>
4
5  class IntArray
6  {
7  private:
8      int m_length{ };
9      int *m_data{ };
10
11 public:
12     IntArray()
13     {
14     }
15
16     IntArray(int length) : m_length{ length }
17     {
18         m_data = new int[m_length];
19     }
20
21     IntArray(const std::initializer_list<int> &list) :
22         IntArray(static_cast<int>(list.size()))
```

```
    {
        int count{ 0 };
        for(auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }
    }

    ~IntArray()
    {
        delete[] m_data;
    }

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

    IntArray& operator= (const std::initializer_list<int> &list)
    {
        if(m_length != static_cast<int>(list.size()))
        {
            delete[] m_data;
            m_length = static_cast<int>(list.size());
            m_data = new int[m_length];
        }

        // probati i sa advanced for petljom
        int count{ 0 };
        for(auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }

        return *this;
    }

    int& operator[] (int index)
    {
        assert(index >= 0 && index < m_length && "Index out of bounds!");
        return m_data[index];
    }

    int getLength(){ return m_length; }
};



int main()
{
    IntArray array{ 5, 4, 3, 2, 1 };

    for(int count{ 0 }; count < array.getLength(); ++count)
        std::cout << array[count] << " ";
    std::cout << '\n';

    array = { 1, 3, 5, 7, 9, 11 };

    for(int count{ 0 }; count < array.getLength(); ++count)
        std::cout << array[count] << " ";

    return 0;
```

```
85 }
```

nascardriver
July 4, 2019 at 5:44 am · Reply

Good day,
- Line 18: Initialize in the member initializer list.
- Line 22: Brace initialization.
- Line 24-29, 50+: Can be replaced with `std::copy(list.begin(), list.end(), m_data);`. This isn't covered on learncpp.
- `IntArray::getLength` should be const.
- Line 76, 82: Use single quotation marks for characters.

Dimbo1911
July 4, 2019 at 9:24 pm · Reply

Hello nascar, thanks for your input.
>Line 22: Brace initialization. : do you mean the

```
1 IntArray{ static_cast<int>(list.size()) }
```

Because if I do that, i get error saying: constructor delegates to itself

nascardriver
July 5, 2019 at 12:36 am · Reply

My bad, use direct initialization.

Abrar Rahman Protyasha
May 25, 2019 at 9:10 pm · Reply

Hey, just a quick question about the quiz solution:

Would we not see undefined behavior in this section if someone passes an empty initializer list {}, where list.size() = 0? I'm just not sure what the behavior of new [] is when the size passed to new is 0.

```
1         if (m_length != static_cast<int>(list.size())) {
2             delete[] m_arr;
3             m_arr = new int[list.size()];
4             if(!m_arr) {
5                 m_arr = nullptr;
6                 return *this;
7             }
8         }
```

Abrar Rahman Protyasha
May 25, 2019 at 9:12 pm · Reply

Actually, I didn't think through the if clause inside, I answered myself here. Regardless, could you please tell me what's the return value when we call new int[0]? I would assume it's nullptr but then you're manually assigning nullptr again so I'm not sure. Thanks again!

nascardriver

May 26, 2019 at 1:51 am · Reply

0-sized arrays are legal when dynamically allocating arrays (But not for static arrays!).
@new will return the address of the newly allocated empty array.
Although empty arrays are not particularly useful, they won't crash.
Line 5 of your quoted code doesn't do anything, since @m_arr is a @nullptr at that point already.

Deepak
May 17, 2019 at 10:21 pm · Reply

Hi,
Suppose i have a class with below variables

```
1 int data;
2 std::vector<int> arr;
```

How to Initialize both while creating the object

```
1 Point pointobj(100, { 1, 2, 3, 4, 5 });
```

I tried writing the constructor as below,but getting error.

```
1 Point(std::initializer_list list1,const std::initializer_list<int> & list) : data(list1),arr
```

could you please tell me how this can be handled

nascardriver
May 18, 2019 at 1:56 am · Reply

@list1 is an @std::initializer_list. You're missing the template argument, and '100' isn't an initializer list.

You don't need an initializer list at all to make your call work.

```
1  #include <vector>
2
3  class C
4  {
5  private:
6    int m_iData{};
7    std::vector<int> m_arr{};
8
9  public:
10   // You could replace @std::vector with @std::initializer_list, but I find
11   // this more obvious.
12   C(int iData, const std::vector<int> &arr)
13       : m_iData{ iData },
14         m_arr{ arr }
15   {
16   }
17 };
18
19 int main(void)
20 {
21   C c{ 1, { 1, 2, 3, 4 } };
22
23   return 0;
24 }
```

Deepak
May 19, 2019 at 3:19 am · Reply

Thanks @nascardriver
So if this work without having the std::initializer_list then in which scenario std::initializer_list is used?

nascardriver
May 19, 2019 at 3:24 am · Reply

If you wanted to do something like this

```
C c{ 1, 2, 3, 4 };
```

This can be achieved with an @std::initializer_list constructor

```
C(const std::initializer_list<int> &arr)
    : m_iData{ 0 },
      m_arr{ arr }
{
}
```

Deepak
May 19, 2019 at 4:02 am · Reply

Thanks @nascardriver

Pawan Kataria
April 30, 2019 at 3:38 am · Reply

Hi,

While studying delegating constructors(https://www.learncpp.com/cpp-tutorial/8-6-overlapping-and-delegating-constructors/), I read,

"A few additional notes about delegating constructors. First, a constructor that delegates to another constructor is not allowed to do any member initialization itself. So your constructors can delegate or initialize, but not both."

But here we are delegating from one constructor to another to allocate memory and then initializing it members in the constructor which just did a delegation.

Which part I understood wrong ??

```
IntArray(const std::initializer_list<int> &list) :IntArray(static_cast<int>(list.size())) //
    {
        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element; // initializing here
            ++count;
        }
    }
```

nascardriver
April 30, 2019 at 3:45 am · Reply

Hi!

The constructor's body can do whatever it wants, but there can not be any initializations in the member initializer list.

```
IntArray(const std::initializer_list<int> &list)
  :  IntArray{ static_cast<int>(list.size()) },
     m_data{ 1, 2, 3 } // Illegal
{}
```

Pawan Kataria
April 30, 2019 at 4:56 am · Reply

Hi @nascardriver,

Understood, it's clear now.

Thanks a lot !!

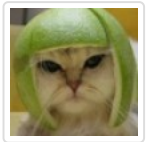
Anthony
April 28, 2019 at 10:58 am · Reply

Hi Alex,

Just to be clear: If the copy constructor and the assignment operator are not deleted, then they should (both) be overloaded and the initializer constructor made explicit? I've done this below:

```
class IntArray {
  private:
    int *m_arr { nullptr };
    int m_length {};
  public:
    IntArray(int n) {
        if (n >= 0) {
            m_length = n;
            m_arr = new int[m_length];
        }
        else
            m_arr = nullptr;
    }
    explicit IntArray(const std::initializer_list<int> &list) : IntArray(list.size()) {  //
        int i {};
        for (const auto &r : list) {
            m_arr[i] = r;
            ++i;
        }
    }
    IntArray(const IntArray &arr) {  // deep-copy constructor
        m_arr = new int[m_length];
        if(!m_arr)
            m_arr = nullptr;
        else {
            m_length = arr.m_length;
            for (int i = 0; i < m_length; ++i)
                m_arr[i] = arr.m_arr[i];
        }
    }
    // IntArray(const IntArray&) = delete;  // to avoid shallow copies
    // IntArray& operator= (const IntArray &arr) = delete;  // to avoid shallow copies
    IntArray& operator= (const IntArray &arr) {
```

```
34          delete[] m_arr;  // the delete[] to prevent a memory leak. Easy to forget!
35          m_arr = new int[arr.m_length];
36          if(!m_arr)
37              m_arr = nullptr;
38          else {
39              m_length = arr.m_length;
40              for (int i {}; i < m_length; ++i)
41              m_arr[i] = arr.m_arr[i];
42          }
43          return *this;
44      }
45      IntArray& operator= (const std::initializer_list<int> &list) {  // overload the already
46          if (m_length != static_cast<int>(list.size())) {
47              delete[] m_arr;
48              m_arr = new int[list.size()];
49              if(!m_arr) {
50                  m_arr = nullptr;
51                  return *this;
52              }
53          }
54          m_length = list.size();
55          int i {};
56          for (const auto &r : list) {
57              m_arr[i] = r;
58              ++i;
59          }
60          return *this;
61      }
62      int& operator[] (int n) const {  // has to be a (non-const) reference to be able to bac
63          return m_arr[n];
64      }
65      ~IntArray() {
66          delete[] m_arr;
67      }
68      ...
69  };
```



Alex
May 2, 2019 at 5:01 pm · Reply

No. If you have a constructor that takes a std::initializer_list, you should have a corresponding list-assignment or a copy assignment function. Otherwise you'll get a shallow copy if you do a list assignment.

The prior recommendation to make your std::initializer_list constructor explicit doesn't prevent this by itself, so that recommendation has been revoked, but it's a good idea to mark all single-parameter constructors as explicit anyway, so no harm there.

Anthony
May 2, 2019 at 6:31 pm · Reply

> No. If you have a constructor that takes a std::initializer_list, you should have a corresponding list-assignment or a copy assignment function. Otherwise you'll get a shallow copy if you do a list assignment.

But if you're too lazy to have the corresponding list-assignment or copy assignment functions, you should delete them so that they can't be used to shallow copy by accident?

Alex
May 6, 2019 at 1:43 pm · Reply

Yep.

Louis Cloete
April 18, 2019 at 2:43 am · Reply

@Alex, your solution again wouldn't compile with the stricter compiler error levels you recommended in 0.11. Here is what you should change:

In the IntArray(const std::initializer_list &list) constructor, you need to call the delegating constructor with IntArray(static_cast<int>(list.size()), else you will get a -Werror warning treated as an error about a narrowing conversion which might change the sign. Thus:

```
IntArray(const std::initializer_list<int> &list):
    IntArray(static_cast<int>(list.size()))
{
    // omitted to reduce distraction and focus on the important part
}
```

Likewise in the operator=(const std::initializer_list &list) method. I solved the problem like this:

```
IntArray& IntArray::operator=(const std::initializer_list<int> &list)
{
    int length { static_cast<int>(list.size() };
    if (length != this->getLength())
    {
        delete[] m_data;
        m_length = length;
        m_data = new int[length];
    }

    // omitted to save space and lessen distraction ;)
}
```

Further, I get a -Werror warning saying the class implements pointer members, but doesn't override IntArray(const IntArray&) or operator=(const IntArray&). I did this:

```
class IntArray
{
    // omitted

    IntArray(const IntArray&) = delete;
    operator=(const IntArray&) = delete;
};
```

and the compiler was happy!

Alex
April 22, 2019 at 12:33 pm · Reply

Thanks, and updated. I'm sure there are other instances of examples that were compiled before the new settings recommendations were put in place. If you find other examples that generate warnings, please point them out so I can update them. Much appreciated!

Louis Cloete
April 22, 2019 at 2:15 pm · Reply

Will do as I find them! Thanks to you for the awesome tutorials!

hassan magaji
March 7, 2019 at 5:18 am · Reply

hi Alex,
about the following code(i might be wrong):

```
for (auto &element : list) // reference of elements
// should'nt be
for (auto element : list) // copy of elements
```

since copying fundamental data types is faster than referencing them.

Alex
March 8, 2019 at 4:29 pm · Reply

Updated. Thanks!

Jon
January 31, 2019 at 12:18 am · Reply

Hello! On the way to figuring out the quiz I had a question I couldn't quite answer. I originally had my operator overload function written like the following, which does print the expected output but also results in a runtime error because it accidentally results in the class Destructor trying to delete unallocated memory later on. I understand how to fix it and get the program to run correctly.

My question is, with the following INCORRECT code, since I delete[] m_data and free the dynamically allocated memory but forget to reallocate the m_data array via the "new" keyword, where and how is my for-each loop setting up the (1, 3, 5, 7, 9, 11) array that still prints correctly?

In a new memory address on the stack, instead of the heap as originally intended? Or is it able to still store and retrieve values in/from the same heap addresses even though they remain unallocated (which doesn't make sense to me)? I just want to make sure I understand what's really going on under the hood. Thank you!

```
    IntArray& operator=(const std::initializer_list<int> &list)
    {
        if (list.size() != static_cast<size_t>(m_length))
        {
            delete[] m_data;

            m_length = list.size();
        }

        int count = 0;
        for (auto &element : list)
        {
            m_data[count] = element;
            ++count;
        }

        return *this;
    }
```

Alex
January 31, 2019 at 5:46 pm · Reply

Deallocating memory returns it back to the OS to be reallocated for another purpose. It typically doesn't clear the contents of that memory. So if you use a pointer to access the deallocated memory, the original contents of that memory may (or may not) still be there.

Jon
January 31, 2019 at 5:53 pm · Reply

And using a pointer you can still assign to that memory as well apparently?

Alex
January 31, 2019 at 6:08 pm · Reply

Yup, and you'll get undefined behavior.

Jon
January 31, 2019 at 8:20 pm · Reply

Thanks Alex, it all makes sense now!

« Older Comments 1 2