19.4 — Pointers to pointers and dynamic multidimensional arrays

ALEX[1] SEPTEMBER 9, 2024

This lesson is optional, for advanced readers who want to learn more about C++. No future lessons build on this lesson.

A pointer to a pointer is exactly what you'd expect: a pointer that holds the address of another pointer.

Pointers to pointers

A normal pointer to an int is declared using a single asterisk:

```
int* ptr; // pointer to an int, one asterisk
```

A pointer to a pointer to an int is declared using two asterisks

```
int** ptrptr; // pointer to a pointer to an int, two asterisks
```

A pointer to a pointer works just like a normal pointer — you can dereference it to retrieve the value pointed to. And because that value is itself a pointer, you can dereference it again to get to the underlying value. These dereferences can be done consecutively:

```
int value { 5 };

int* ptr { &value };
std::cout << *ptr << '\n'; // Dereference pointer to int to get int value

int** ptrptr { &ptr };
std::cout << **ptrptr << '\n'; // dereference to get pointer to int, dereference again to get int value
```

The above program prints:

```
5
5
```

Note that you can not set a pointer to a pointer directly to a value:

```
int value { 5 };
int** ptrptr { &&value }; // not valid
```

This is because the address of operator (operator&) requires an lvalue, but &value is an rvalue.

However, a pointer to a pointer can be set to null:

```
int** ptrptr { nullptr };
```

Arrays of pointers

Pointers to pointers have a few uses. The most common use is to dynamically allocate an array of pointers:

```
int** array { new int*[10] }; // allocate an array of 10 int pointers
```

This works just like a standard dynamically allocated array, except the array elements are of type "pointer to integer" instead of integer.

Two-dimensional dynamically allocated arrays

Another common use for pointers to pointers is to facilitate dynamically allocated multidimensional arrays (see 17.12 -- Multidimensional C-style Arrays (https://www.learncpp.com/cpp-tutorial/multidimensional-c-style-arrays/)[2] for a review of multidimensional arrays).

Unlike a two dimensional fixed array, which can easily be declared like this:

```
int array[10][5];
```

Dynamically allocating a two-dimensional array is a little more challenging. You may be tempted to try something like this:

```
int** array { new int[10][5] }; // won't work!
```

But it won't work.

There are two possible solutions here. If the rightmost array dimension is constexpr, you can do this:

```
int x { 7 }; // non-constant
int (*array)[5] { new int[x][5] }; // rightmost dimension must be constexpr
```

The parenthesis are required so that the compiler knows we want `array` to be a pointer to an array of 5 `int` (which in this case is the first row of a 7-row multidimensional array). Without the parenthesis, the compiler would interpret this as `int* array[5]`, which is an array of 5 `int*`.

This is a good place to use automatic type deduction:

```
int x { 7 }; // non-constant
auto array { new int[x][5] }; // so much simpler!
```

Unfortunately, this relatively simple solution doesn't work if the rightmost array dimension isn't a compile-time constant. In that case, we have to get a little more complicated. First, we allocate an array of pointers (as per above). Then we iterate through the array of pointers and allocate a dynamic array for each array element. Our dynamic two-dimensional array is a dynamic one-dimensional array of dynamic one-dimensional arrays!

```
int** array { new int*[10] }; // allocate an array of 10 int pointers – these are our
rows
for (int count { 0 }; count < 10; ++count)
    array[count] = new int[5]; // these are our columns
```

We can then access our array like usual:

```
array[9][4] = 3; // This is the same as (array[9])[4] = 3;
```

With this method, because each array column is dynamically allocated independently, it's possible to make dynamically allocated two dimensional arrays that are not rectangular. For example, we can make a triangle-shaped array:

```
int** array { new int*[10] }; // allocate an array of 10 int pointers – these are our
rows
for (int count { 0 }; count < 10; ++count)
    array[count] = new int[count+1]; // these are our columns
```

In the above example, note that array[0] is an array of length 1, array[1] is an array of length 2, etc...

Deallocating a dynamically allocated two-dimensional array using this method requires a loop as well:

```
for (int count { 0 }; count < 10; ++count)
    delete[] array[count];
delete[] array; // this needs to be done last
```

Note that we delete the array in the opposite order that we created it (elements first, then the array itself). If we delete array before the array columns, then we'd have to access deallocated memory to delete the array columns. And that would result in undefined behavior.

Because allocating and deallocating two-dimensional arrays is complex and easy to mess up, it's often easier to "flatten" a two-dimensional array (of size x by y) into a one-dimensional array of size x * y:

```
// Instead of this:
int** array { new int*[10] }; // allocate an array of 10 int pointers – these are our
rows
for (int count { 0 }; count < 10; ++count)
    array[count] = new int[5]; // these are our columns

// Do this
int *array { new int[50] }; // a 10x5 array flattened into a single array
```

Simple math can then be used to convert a row and column index for a rectangular two-dimensional array into a single index for a one-dimensional array:

```
int getSingleIndex(int row, int col, int numberOfColumnsInArray)
{
     return (row * numberOfColumnsInArray) + col;
}

// set array[9,4] to 3 using our flattened array
array[getSingleIndex(9, 4, 5)] = 3;
```

Passing a pointer by address

Much like we can use a pointer parameter to change the actual value of the underlying argument passed in, we can pass a pointer to a pointer to a function and use that pointer to change the value of the pointer it points to (confused yet?).

However, if we want a function to be able to modify what a pointer argument points to, this is generally better done using a reference to a pointer instead. This is covered in lesson 12.11 -- Pass by address (part 2) (https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/)[3].

Pointer to a pointer to a pointer to...

It's also possible to declare a pointer to a pointer to a pointer:

```
1 | int*** ptrx3;
```

This can be used to dynamically allocate a three-dimensional array. However, doing so would require a loop inside a loop, and is extremely complicated to get correct.

You can even declare a pointer to a pointer to a pointer to a pointer:

```
1 | int**** ptrx4;
```

Or higher, if you wish.

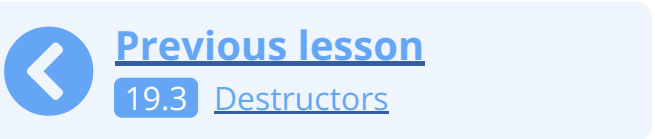
However, in reality these don't see much use because it's not often you need so many levels of indirection.

Conclusion

We recommend avoiding using pointers to pointers unless no other options are available, because they're complicated to use and potentially dangerous. It's easy enough to dereference a null or dangling pointer with normal pointers — it's doubly easy with a pointer to a pointer since you have to do a double-dereference to get to the underlying value!

4

5



6

7

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[10] are connected to your provided email address.

256 COMMENTS

Newest

C.Moresi

June 6, 2025 10:24 am PDT

For those struggling with pointers (Please Alex, correct me if I'm wrong):

A pointer stores the address of another variable. But a pointer is still a variable itself, and it is stored at its own memory address. For example:

```
1 int a {5};
2 int* ptr {&a};
3 int** ptrptr {&ptr};
```

- Variable a -> Address 0x100 -> value "5"
- Variable ptr -> Address 0x200 -> value "0x100"
- Variable ptrptr -> Address 0x300 -> value "0x200"

That is why it's possible to store a pointer in another pointer.

0 Reply

paulisprouki

Reply to C.Moresi[11] July 4, 2025 3:54 pm PDT

well yes of course, anything that can be accessed using a name has to be stored somewhere. Computers are not smart they are just made very very optimized.

0 Reply

Kania

April 3, 2025 7:11 am PDT

Ah, yes—pointers, my arch-nemesis.

0 Reply

Diego

March 2, 2025 11:52 am PST

Instead of using this "complex" ptrs to make multidimensional arrays (matrices), can't I just do something like

```
std::vector<std::vector<std::vector<int>>> matrix3D;

matrix3D.resize(depth);
//nested for
matrix3D.resize(rows);
//nested for
matrix3D.resize(cols);

//nested for
matrix3D[i][j][k] = number;
```

I guess if we are talking only about matrices maybe this is the most optimal solution, but are there uses when we need new/delete, maybe ptrs to classes or something else?

0 Reply

20leunge

September 7, 2024 7:42 pm PDT

Correct me if I'm wrong, but for the above code block:

```
int x { 7 }; // non-constant
int (*array)[5] { new int[x][5] }; // rightmost dimension must be constexpr
```

The purpose of the parentheses is to specify that array is a pointer to (seven) arrays of size 5, and not an array of size 5 that contains pointers to integers, which would be denoted by

```
int* (array[5]) { new int[5][7] };
```

From what I understand, the precedence of operators makes

```
int* array[5] { new int[5][7] };
```

equivalent to the parentheses around array[5]

Last edited 10 months ago by 20leunge

1 Reply

Alex Author

Reply to 20leunge [12] September 9, 2024 9:42 pm PDT

This is correct.

3 Reply

Alex(student)

August 6, 2024 3:25 am PDT

I am quite confused on what is going on here

```
#include <iostream>
#include <array>
int main()
{
    int x { 8 }; // non-constant
    auto array{ new int[x][8]{} }; // 2d 8x8 matrix

    array[1][1] = 2;
    array[1][7] = 3;
    std::cout << array[1][1];
    std::cout << array[1][7];

    array[9][9] = 4;
    std::cout << array[9][9];
    delete[] array;
}
```

I did not get warnings with -Wall, The output is 2 3 4, eventhough array[9][9] should be out of range. But with -fsanitize=address I get warnings and it does not compile.

```
SUMMARY: AddressSanitizer: heap-buffer-overflow /cpp/arrays5-2d-dynamic.cpp:12 in main
Shadow bytes around the buggy address:
  0x510fffffff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x510fffffff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x511000000000: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
  0x511000000080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x511000000100: 00 00 00 00 00 00 00 00 fa fa fa fa fa fa fa fa
=>0x511000000180:[fa]fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x511000000200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x511000000280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x511000000300: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x511000000380: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x511000000400: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
```

But for some reason if I comment out the array[9][9] part, and also `new int[x][8]` without the {}, it complains with -fsanitize... that "==20337==LeakSanitizer has encountered a fatal error."

Seems like array[9][9] is allocating to an area it is not supposed to, but also if I add a for loop

```
array[2][0] = 6;
for(int i{0}; i<12; ++i)
{
    std::cout << array[0][i] << ' ';
}
```

it prints

`0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 3 6 0` so it seems its continuing to the next column ... ?

Last edited 11 months ago by Alex(student)

0 Reply

Alex Author

Reply to Alex(student) [13] August 6, 2024 2:54 pm PDT

Yes, `array[9][9]` is out of bounds. 2d arrays in C++ are laid out in row-major order. See https://www.learncpp.com/cpp-tutorial/multidimensional-c-style-arrays/. This means the element in memory after `[0][7]` is `[1][0]`.

0 Reply

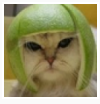
Alex(student)

Reply to Alex [14] August 7, 2024 2:01 am PDT

Ok, it also seems [0][8] == [1][0], or is this undefined behaviour? Considering the for-loop trial on a 8x8 matrix I did

```
array[2][0] = 6;
for(int i{0}; i<12; ++i)
{
    std::cout << array[0][i] << ' ';
}
```

0 Reply



Alex Author

Reply to Alex(student) [15] August 10, 2024 3:36 pm PDT

I believe it's technically undefined behavior. But I imagine it will work on most compilers.

1 Reply

stillloveit

June 12, 2024 6:50 am PDT

"Conclusion

We recommend avoiding using pointers to pointers unless no other options are available, because they're complicated to use and potentially dangerous. "

I'm SO glad x'D

1 Reply

Dck

March 24, 2024 12:00 am PDT

Hi Alex, for the first method of defining a 2D array, it seems like we can deallocate the array as a whole rightaway without deallocating each row first, right?

```
int (*array)[5] { new int[10][5] };
delete[] array;
```

When I tried to deallocate each row first, my compiler errors out saying it can't deallocate expression of type `int[5]`.

Also another question, when would we want to dynamically allocate our 2D array in this way instead of using nested `std::vector`? Thank you for your help!

Last edited 1 year ago by Dck

0 Reply

Alex Author

Reply to Dck [16] March 24, 2024 2:42 pm PDT

Yes, I believe that's corect. Basically, however you allocate things using new, you need to deallocate them using delete. So if there is only one new statement, there should only be one delete statement.

The advantage of doing this is that you end up with an actual multidimensional array that you can index using two indices (e.g. `array[1][2]`), whereas `std::vector` is always single-dimensional.

In C++23, you can use `std::mdspan` to view a `std::vector` as a multidimensional array, which should be preferred. I also implemented a custom view class for multidimensional std::arrays here: https://www.learncpp.com/cpp-tutorial/multidimensional-stdarray/

0 Reply

Ethan

February 7, 2024 10:31 am PST

```
int** array { new int*[10] }; // allocate an array of 10 int pointers — these are our
rows
for (int count { 0 }; count < 10; ++count)
    array[count] = new int[5]; // these are our columns
```

This may be a bit pedantic, but if the goal is to make an array with 10 rows and 5 columns, then each row has 5 elements and each column has 10 elements. I don't think the two comments 'these are our rows' and 'these are our columns' in the above block are accurate to whats going on. Perhaps the first comment is fine. array[5] is a pointer to the 6th row, so everything makes sense so far. But then the

second comment feels off. Consider the 3rd iteration in the loop: array[2] = new int[5]. The new int[5] in this interation is an array containing the 5 pointers to the elements the 3rd row, ie it is a row. For this reason I feel like "these are our columns" is not accurate.

I'm not sure the correct wording to suggest. Perhaps "allocate an array of 10 int pointers — each to a specific row" and "each row is an array of 5 pointers"?

0 Reply

MoonVoid

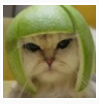
December 29, 2023 8:14 am PST

Hi Alex,
I'm very confused seeing the code in the paragraph.

```
1 int x { 7 }; // non-constant
2 int (*array)[5] { new int[x][5] }; // rightmost dimension must be constexpr
```

What does int(*array)[5] mean?
I searched about this and learned that it seems like a Array Pointer. A strange concept.
We have learnt that a C-style array cannot store its array size, but what does the [5] mean in this array pointer?

0 Reply



Alex Author

Reply to MoonVoid [17] December 31, 2023 4:25 pm PST

`array` is a pointer to an array of 5 integers.

The second dimension of the array isn't explicitly defined in the declaration of `arraty`, but you can still index things as expected because pointers can be indexed using the subscript operator.

0 Reply

Damian

November 18, 2023 11:13 pm PST

In the section: "it's often easier to "flatten" a two-dimensional array (of size x by y) into a one-dimensional array of size x * y:" it may be useful for the reader to get a back reference to std::mdspan.

Thanks for these great tutorials nonetheless!

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/multidimensional-c-style-arrays/
3. https://www.learncpp.com/cpp-tutorial/pass-by-address-part-2/
4. https://www.learncpp.com/cpp-tutorial/void-pointers/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/destructors/
7. https://www.learncpp.com/pointers-to-pointers/
8. https://www.learncpp.com/cpp-tutorial/bit-manipulation-with-bitwise-operators-and-bit-masks/
9. https://www.learncpp.com/cpp-tutorial/introduction-to-stdarray/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/#comment-610737
12. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/#comment-601738
13. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/#comment-600574
14. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/#comment-600599
15. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/#comment-600612
16. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/#comment-595038
17. https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/#comment-591480
18. https://g.ezoic.net/privacy/learncpp.com