6.14 — Pointers to pointers and dynamic multidimensional arrays

BY ALEX ON SEPTEMBER 14TH, 2015 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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; // dereference pointer to int to get int value

int **ptrptr = &ptr;
std::cout << **ptrptr; // first dereference to get pointer to int, second dereference to get in
```

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; // use 0 instead prior to C++11
```

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 **6.5 -- Multidimensional Arrays** 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 right-most array dimension is a compile-time constant, you can do this:

```
int (*array)[5] = new int[10][5];
```

The parenthesis are required here to ensure proper precedence. In C++11 or newer, this is a good place to use automatic type deduction:

```
auto array = new int[10][5]; // so much simpler!
```

Unfortunately, this relatively simple solution doesn't work if the right-most 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 elements, then we'd have to access deallocated memory to delete the array elements. 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. So we won't talk about it further here.

We'll talk more about pass by address and pass by reference in the next chapter.

Pointer to a pointers to a pointer to...

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

```
int ***ptrx3;
```

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

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

```
int ****ptrx4;
```

Or higher, if you wish.

However, in reality these don't see much use because it's not often you need so much 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!

6.15 -- An introduction to std::array

Index

6.13 -- Void pointers

C++ TUTORIAL | PRINT THIS POST

107 comments to 6.14 — Pointers to pointers and dynamic multidimensional arrays

« Older Comments 1 2

Sid22
January 27, 2020 at 8:02 am · Reply

I am trying to implement a simple program that has objects of "Student" class dynamically allocated and then sorted in the array in alphabetical order by their names (as in a dictionary). For example: the object with the string "Ana" as the name will be at an earlier index of the array of objects , whereas the object with "Azra" as the name will come after.

-> The problem:

The logic seems fine and the code compiles on Visul Studio but crashes at runtime midway through the program after taking necessary info for initialization of object members (which has been implemented through

```
get_info()
```

in the default constructor.

MY code:

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

class Student
{
private:
    std::string name;
    int age, marks;
    static int s_num;
public:
    Student()
    {
```

```
        getInfo();
    }
    Student(std::string n,int a, int m):name{n},age{a},marks{m}
    {}
    std::string get_name()
    {
        return name;
    }
    void getInfo()
    {
        std::cout << "\nEnter student#" << s_num << "\'s name:";
        std::cin >> name;

        std::cout << "\nEnter student#" << s_num << "\'s age:";
        std::cin >> age;

        std::cout << "\nEnter student#" << s_num << "\'s marks:";
        std::cin >> marks;

        s_num++;
    }
    void display_info(int student_count)
    {

        std::cout << "\nFor student#" << student_count << ":\n";
        std::cout << "\nName: " << name << "\n";
        std::cout << "\nAge: " << age << "\n";
        std::cout << "\nMarks: " << marks << "\n\n";

    }
 };
int Student::s_num = 1;
void sort_by_name(Student* ptr,int num)
{


    for (int i{ 0 }; i<num;i++)
    {
        for (int j{ 0 }; j < num; j++)
        {
            std::string temp = ptr[j].get_name();
            int a = temp.compare(ptr[j + 1].get_name());
            if (a < 0) //if value of compared string is lower..should come first
            {
                //swapping
                Student temp_obj(ptr[j]);
                ptr[j] = ptr[j + 1];
                ptr[j + 1] = temp_obj;
            }

        }

    }
}

int main()
{
    int num{ 0 };
    std::cout << "\nHow many students are there ?\n";
    std::cin >> num;
    Student* ptr = new Student[num];
    sort_by_name(ptr, num);
```

```
75      for (int i{ 0 }; i < num; i++)
76      {
77          ptr[i].display_info(i + 1);
78      }
79
80      delete[]ptr;
81
82      return 0;
83  }
```

nascardriver
January 27, 2020 at 8:19 am · Reply

Line 51: What's the maximum value of `j`?
Line 54: Which index are you accessing when `j` is at its maximum?

- Initialize variables with brace initialization for higher type safety.
- Use ++prefix. postfix++ is slower.
- `std::string` can be compared with `<`. (`ptr[j].get_name() < ptr[j + 1].get_name()`).

Sid22
January 27, 2020 at 10:08 am · Reply

SOLVED IT!
Made the following changes:

```
1   void sort_by_name(Student* ptr, int num)
2   {
3
4
5       for (int i{ 0 }; i < num; i++)
6       {
7           for (int j{ 1 }; j < num; j++) /*started j from 1*/
8           {
9               std::string temp = ptr[j-1].get_name(); /*ptr[j-1] here now to access
10              int a = temp.compare(ptr[j].get_name()); //also changed a<0 to a>0
11              if (a > 0) //if value of compared string is lower..should come first
12              {
13                  //swapping
14                  Student temp_obj(ptr[j-1]);
15                  ptr[j-1] = ptr[j];
16                  ptr[j] = temp_obj;
17              }
18
19          }
20
21      }
22  }
```

Pankaj
January 20, 2020 at 6:45 pm · Reply

How to de-allocate for allocation,

int (*ptr)[5] = new int [10][5];

auto ptr = new int[10][5];

hellmet
October 23, 2019 at 1:19 am · Reply

I don't get this...

```
int (*array)[5] = new int[10][5];
// I initially thought of writing the above as below
int (*array)[10] = new int[10][5];
```

Could you explain this please?

nascardriver
October 23, 2019 at 3:23 am · Reply

The 5 is the size of the inner array. The size of the outer array is lost when the array decays, just like with one-dimensional arrays.
The syntax is hard to learn, like the syntax of function pointers and array references. Fortunately you don't need them unless you want to have full control over the memory. You'll learn about easier and safer alternatives later.

hellmet
October 23, 2019 at 3:37 am · Reply

Hmmm how seemingly counterintuitive :P
Okay so you mean new int[10] decays into a pointer which means I'm then doing new (*int)[5]? Hence the syntax? But the array is still 10 rows and 5 cols right? I tried to verify, but doesn't seem to crash the code hence unsure of the bounds set up.

nascardriver
October 23, 2019 at 3:46 am · Reply

> Okay so you mean new int[10] decays into a pointer which means I'm then doing new (*int)[5]?
I honestly don't know when the array decays. I never needed it and I don't think I ever will, so I won't look it up.

> the array is still 10 rows and 5 cols right?
Rows and columns are a nice aide, but they can't be used to communicate, because a 2D array can be imagined multiple ways, such that rows and columns could be swapped.
"Outer array" and "inner array" work better. When you grab an element from the outer array, you get an inner array.
The outer array has 10 elements. The inner arrays have 5 elements each.
Trying to verify this by trying to crash or by trying to get weird values is bad. It's undefined behavior, you can't draw any conclusions from the test.
Compile-time errors are more reliable

```
new int[1][2]{
  {0}, {0} // error: excess elements in array initializer
};

new int[1][2]{
  {0, 0} // ok. 1 is the size of the outer array. 2 of the inner array.
};

```

```
new int[2][1]{
  {0}, {0} // ok. 2 is the size of the outer array. 1 of the inner arrays.
};
```

hellmet
October 23, 2019 at 3:56 am · Reply

Oh my god that makes so much sense! The 'inner' and 'outer' array explanation on point! Amazing example with the compiler-error! Thank you very much!

Justin
August 3, 2019 at 11:47 am · Reply

For skilled programmers (even though I'm only learning c++),
I need help with a C2062 error that I don't understand. I started trying to code natural merge sort. Look at this code.

```
#ifndef MERGESORT
#define MERGESORT

//This program uses natural merge sort to sort dynamic int-type arrays

void sortMergeSort(int *array, int arrayLength) {
  //PART 1: Find natural groups of sorted numbers and group them in arrays
  //Find the number of naturally-sorted groups of numbers
  int runs{1};
  for (int i{0}; i + 1 < arrayLength; ++i)
    if (array[i] > array[i + 1])
      ++runs;
  //Make an array of empty arrays to put runs in and an array for their lengths
  int **arrayOfRuns{new int*[runs]};
  int *runsLengths{new int[runs]};
  //Assign the runs and their lengths in the arrays
  for (int i{0}, int run{0}, int start{0}, int runLength{1}; i + 1 < arrayLength; ++i) {
    if (array[i] < array[i + 1])
      ++runLength;
    else {
      arrayOfRuns[run] = new int[runLength];
      for (int j{0}; j < runLength; ++j, ++start)
        arrayOfRuns[run][j] = array[start];
      runsLengths[run] = runLength;
      ++run;
      runLength = 1;
    }
  }
}

#endif
```

This is the error that I get:
(17,59): error C2062: type 'int' unexpected
I think that it's saying that there's something wrong with the declaration of runLength, but I don't know what's wrong in my code.

Also, how do I shorten the long for loop? It's 89 characters wide!

nascardriver
August 4, 2019 at 7:52 am · Reply

Declare 1 variable per line. You could remove all the `int`s from line 17, except the first one, but declaring 1 variable per line is more readable.

Justin
August 4, 2019 at 9:29 am · Reply

I declared all of the variables except i out of the for loop, and it all works now. Thanks for helping me with that error.

Atas
June 27, 2019 at 12:43 am · Reply

Hello! Is there any way to rewrite this line of code:

```
1 int (*array)[5] = new int[10][5];
```

such that the '*' is next to the type, not the variable name? I much prefer my *-s and &-s next to the type name.

The following

```
1 int* array[5] = new int[10][5];
```

doesn't compile and I can not quite grasp why and what exactly is happening. The syntax for declaring arrays is confusing. It says "cannot covert 'int (*)[5]' to 'int *[5]'". I presume int (*)[5] is an array of 5 pointers to int, but then what is int *[5]?

nascardriver
June 27, 2019 at 4:37 am · Reply

> int (*)[5] is an array of 5 pointers to int
int (*)[5] is a pointer to an array of 5 ints.
int *[5] is an array of 5 pointers to ints.

```
1 // Using @auto
2 auto *array{ new int[10][5] };
3
4 // Using a type alias
5 using Atas = int (*)[5];
6 Atas array{ new int[10][5] };
```

Atas
June 27, 2019 at 5:32 am · Reply

> int (*)[5] is a pointer to an array of 5 ints.
But doesn't this make the lhs of the statement

```
1 int (*array)[5] = new int[10][5];
```

a pointer to an array of 5 ints? Could you please spell out the precendence on the rhs of the assignment as well, that's the most confusing part I think. I read it as follows: the second set of square brackets [5] is a declaration of static array of 5 elements, where an element is the rest of the expression, i.e. new int[10]. Putting it all together I interpret the rhs as

```
(new int[10])[5] = (int*)[5]
```

Is that anywhere near what's actually going on? :-)

nascardriver
June 27, 2019 at 6:05 am · Reply

It's the other way around.

```
auto *array{ new int[10][5] };
```

The length of `array` is 10, the length of `array[i]` is 5.

Those are all the same

```
auto *array{ new int[10][5] };

using Inner = int[5];

auto *array3{ new Inner[10] };

using Outer = Inner[10];

auto *array2{ new Outer };
```

Atas
June 27, 2019 at 6:57 am · Reply

Thank you! It's slowly starting to click.

cdecde57
June 8, 2019 at 4:23 pm · Reply

I am having a problem printing a 2d array that is dynamically allocated like in this lesson.
Please help.

```
    int hight{ 0 };         //These 4 lines I get
    int width{ 0 };         //user input.
    std::cin >> hight;  //
    std::cin >> width;    //

    int **arrays{ new(std::nothrow)int*[hight]};    //These three lines dynamically
    for (int loop{ 0 }; loop < hight; ++loop)        //allocate a 2d array using user input
        arrays[loop] = new(std::nothrow)int[width]; //

    arrays[hight][width] = { 0 }; // Here I set all their values to 0

    for (int ht{ 0 }; ht < hight; ++ht)            //These 6 lines we print the array.
    {                                             //
        for (int wd{ 0 }; wd < width; ++wd)        //
            std::cout << arrays[ht][wd] << "  ";//
        std::cout << "\n\n";                      //
    }                                             //
```

nascardriver
June 9, 2019 at 12:30 am · Reply

Line 10: No, you're accessing invalid indexes, resulting in undefined behavior. If you want to 0-initialize your arrays, add empty curly braces to the end of line 8.

If you have problems with something, always state the expected- and observed behavior. If you receive compiler warnings or errors, share those too.

cdecde57
June 10, 2019 at 6:53 am · Reply

Alright. Thanks! It works now. I also understand why it now so thanks!

I really read through this lesson and I think I understand how this is actually working now.

We make an array. Then put that array into a loop to give values to each of the arrays in the array like normal, but instead giving it a value of like 3 we say that it is another array so it's like an array within an array.

Correct me if I am wrong. I am just trying to figure out how essentially it works so I can do well in it.

Thanks!

nascardriver
June 10, 2019 at 6:55 am · Reply

Sounds like you understood it :)

« Older Comments 1 2