# 6.14 — Pointers to pointers

BY ALEX ON SEPTEMBER 14TH, 2015 | LAST MODIFIED BY ALEX ON NOVEMBER 11TH, 2015

This lesson is optional, for advanced readers who want to learn more about C++. Future lessons do not build on it.

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
t 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; // 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.

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 be 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. 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**

**Share this:**

Email Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

## 18 comments to 6.14 — Pointers to pointers

Avneet
September 15, 2015 at 9:12 am · Reply

Doesn't this code causes memory leak:

```
bool allocateArray(int **ptr, int size)
{
    *ptr = new int[size];
    return (*ptr != nullptr); // use 0 if not C++11
}

int main()
{
    int *array;
    allocateArray(&array, 10);

    return 0;
}
```

In main (), a pointer to int is declared. It's  address is then sent to allocateArray() that returns a bool. We dereference the pointer (with single asterisk) and get access to a pointer. Dynamically Allocate some memory and then return true, in this case. Now control is back to main (). main () neither uses that memory, nor deallocates it. When 'array' is going out of scope, only reference to that memory is destroyed and results a memory leak. Am I right? Correct me if not. The next code also has a memory leakage problem I think.

And a confusion. We are sending the address of 'array' by value, that means a copy of that address would be passed. Isn't the copy of the address is same as the real address. I mean, array and ptr are pointing to same address right? So, main's 'array' can deallocate the dynamically allocated memory in allocateArray(), where 'ptr' is controlling the dynamic allocation? Confused? My bad english ☹

Alex
September 15, 2015 at 10:34 am · Reply

Yes, the memory is never deallocated. It wasn't relevant to the example so I didn't flesh out that part of the example. However, I think it's good practice, so I've added it. Thanks for the thought.

Consider this code:

```
bool allocateArray(int *ptr, int size)
{
    ptr = new int[size];
    return (ptr != nullptr); // use 0 if not C++11
}

int main()
{
    int *array = 0;
    allocateArray(array, 10);

    return 0;
}
```

This is a pass-by-value example. In this case, ptr gets a copy of array's value. We allocate memory to ptr, but the address of that memory never gets assigned back to array because ptr is a copy.

In the example in the lesson, we are sending the address of array by address, not by value. So the ptr is set to the _address_ of array. Think of it this way:

ptr -> array -> array element 0

In this case, when we dereference ptr, we get array (which itself points to the array). By allocating memory to *ptr, we're essentially changing the address of array, which is why when we return to main, array is still pointing at the allocated memory.

This allows array to deallocate the memory allocated in allocateArray() because array is a valid pointer to that memory.

On second thought, I think I'm going to remove the example altogether. It's a little complicated for this point in the tutorials, and we haven't even hit the lessons on pass by value, pass by address, and pass by reference yet.

**Avneet**
September 15, 2015 at 8:36 pm · Reply

hoooofffff…pretty much confusing. Now after 7 hours, I think I have cleared this. BTW, thanks for these two newly added lessons. They are helpful. 🙂

venkata prasad
October 16, 2015 at 4:46 am · Reply

I read everywhere that if I have a need to do arithematic operations on pointers always choose pass by pointer mentod over pass by reference method.

but what about this code?

```
incrementptr1(int *ptr1)

{
    cout<<ptr++<<endl;

return 0;

}

incrementptr2(int* &ptr2)

{
  cout<<ptr++<<endl;
}
```

int main()

{

int value=5;

incrementptr1(&value);
incrementptr2(&value);

return 0;

}

with the increment2 method I could increment the pointer though I did pass it by reference.

does this work?

Alex
October 16, 2015 at 6:08 pm · Reply

It really depends on what your intent is. If you want to increment the pointer inside the function without affecting the argument, pass the pointer by value (incrementptr1). If you want the function to be able to change the argument, then pass the pointer by reference (incrementptr2).

Anton
November 1, 2015 at 3:30 am · Reply

Regarding dynamic multidimensional arrays Ivor Horton writes in his book "Beginning Visual C++ 2013" about another way to create such matrices:

```
double (*pbeans)[4] {};
pbeans = new double [3][4]; // Allocate memory for a 3x4 array
```

or for one-line initialization

```
double (*pbeans)[4] {new double [3][4]};
```

for three-dimensional matrices:

```
auto pBigArray (new double [5][10][10]); // Allocate memory for a 5x10x10 array
```

and so on …

As for the way to remove the kebab he suggests:

```
delete [] pBigArray; // Release memory for array
pBigArray = nullptr;
```

I'm confused here. This couldn't possibly be a proper way to declare a true dynamic multidimensional array. Firstly, I don't truly understand the syntax and secondly the code:

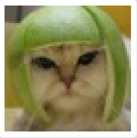
```
double (*pbeans)[4] {};
```

looks like

```
double pbeans*[4] {};
//or
double pbeans*[4]
```

Hayl, I don't even understand what it does! It doesn't look like something that the compiler would accept.

If you want to check it out it is found in chapter 4 ("Arrays, Strings and Pointers") in section [i]???[b]num[/b]???[/i] ("Dynamic Memory Allocation") in the last subsection [i]???[b]num.num[/b]???[/i] ("Dynamic Allocation of Multidimensional Arrays").



Alex
November 5, 2015 at 9:29 pm · Reply

That's super neat. I've updated the lesson to incorporate this. Thanks for the tip!

```
double (*pbeans)[4] {};
```

means (in English) define a pointer that points to an object of type double[4]. In other words, *pbeans (or pbeans[0]) has type double[4].

The {} aren't necessary, but can be used to initialize the array to the default value for the type (in this case, the default value for double is 0.0).

cesare
November 11, 2015 at 2:34 pm · Reply

I need a little clarification about what you say here:

"With this method, because each array column is be 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:"

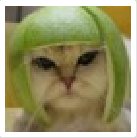
```
int **array = new int*[10]; // allocate an array of 10 int pointers — these are our rows
for (int count = 0; count < 10; ++count)
    array[count] = new int[count]; // these are our columns
```

And then:
"In the above example, note that array[0] is an array of length 1, array[1] is an array of length 2, etc…"

Well, from my point of view, during the first iteration of the loop, the value of 'count' equals 0, so the instruction run like this:

```
array[0] = new int[0]; // these are our columns
```

On the left hand side it clearly refer to the first position in the array, but on the right hand side it seems to allocate an array with length 0 not length 1.
Am I missing something or is it just a typo?



Alex
November 11, 2015 at 3:04 pm · Reply

Typo. I've updated the example to:

```
for (int count = 0; count < 10; ++count)
    array[count] = new int[count+1]; // these are our columns
```

Narendra Kangralkar
November 21, 2015 at 9:18 am · Reply

Below is the better way of allocating memory to 2-D array. It requires only 2 calls to allocate/deallocate memory regardless of number of rows and columns.

```
#include <iostream>

using namespace std;

int main(void)
{
    int **array = 0;
    const int rows = 10;
    const int cols = 5;

    // Allocate memory to store 10 rows
    array = new int*[rows];

    // Allocate memory to store elements in 10 rows and 5 columns
    array[0] = new int[rows * cols];

    // Point each row to column
    for (int i = 1; i < rows; ++i)
        array[i] = array[0] + (i * cols);

    // Access element in normal way
    array[2][2] = 20;

    cout << "array[2][2] = " << array[2][2] << endl;

    delete [] array[0];
    delete [] array;
    return 0;
}
```

raha
January 3, 2016 at 9:41 am · Reply

Hello, thank you so much for the great website, I have a problem in calling a matrix to a function, my main function is as following,

```
int main()
{
  Parameter* param = new Parameter;
  Objective* obj = new Objective;

  parse(param);

  // initialize data matrix X and label vector
y

  double **X = new double*[param->m];
  for (int i=0; i<param->m; i++)
    X[i] = new double [param->d];

  double y[param->m];

  ReadData(X,y,param);


  LogReg(X,y,param,obj);

  exit(0);

}
```

I've defined matrix X and vector y, Im filling these two by function ReadData, and everything is fine by now, but when I try to pass matrix X to function LogReg, which has defined as "double LogReg(double **X, double *y, Parameter* param, Objective* obj)", Im getting "segmentation fault", it happens when I want to use the elements of matrix X such X[i][j], but when I use the elements of vector y such y[i], everything is fine, the thing is why I can pass X to function ReadData(which has been defined in the same file as main) but I can not do the same thing with function "LogReg" which has been define in a header file..

Thank you so much for your help..

raha
January 3, 2016 at 12:19 pm · Reply

The problem has solved. thanks!

Alex
January 4, 2016 at 3:22 pm · Reply

It’s not clear to me from what you’ve provided here why this might be happening.

Federico
January 31, 2016 at 2:58 pm · Reply

I have a question:

```
#include <iostream>

using namespace std;

int * funz()
{
    int var = 77;
    return &var;
}

void gunz(int ** ptr)
{
    int var = 11;
    *ptr = &var;
}

int main()
{

    int var = 1;

    gunz(&ptr);
    cout << *ptr << endl;

    ptr = funz();
    cout << *ptr << endl;

    return 0;

}
```

For the function funz i have a warning: address of local variable ‘var’ returned [-Wreturn-local-addr]
and this is ok, but not for function gunz.
The output is:
11
segmentation fault

It isn’t the same situation?

Thanks and sorry for my english.

Alex
February 1, 2016 at 5:36 pm · Reply

How does gunz(&ptr); even compile? I don’t see where you’re declaring ptr.

Federico
February 2, 2016 at 1:03 am · Reply

Sorry there is an error.
The main is:

```
int main()
{

    int * ptr; //not int var = 1

    gunz(&ptr);
    cout << *ptr << endl;

    ptr = funz();
    cout << *ptr << endl;

    return 0;

}
```

Alex
February 2, 2016 at 1:15 pm · Reply

If either of these work, it's only circumstantial. In both cases, you end up with a hanging pointer, and accessing that pointer will lead to indeterminate results.

Federico
February 3, 2016 at 12:21 am · Reply

Thank you