19.4 — Pointers to pointers and dynamic multidimensional arrays

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 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 here to ensure proper precedence. 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).

Pointer to a pointer to a pointer to…

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

```
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:

```
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!