10.12 — Dynamically allocating arrays

ALEX JANUARY 18, 2022

In addition to dynamically allocating single values, we can also dynamically allocate arrays of variables. Unlike a fixed array, where the array size must be fixed at compile time, dynamically allocating an array allows us to choose an array length at runtime.

To allocate an array dynamically, we use the array form of new and delete (often called new[] and delete[]):

```cpp
#include <iostream>

int main()
{
    std::cout << "Enter a positive integer: ";
    int length{};
    std::cin >> length;

    int* array{ new int[length]{} }; // use array new.  Note that length does not need to be constant!

    std::cout << "I just allocated an array of integers of length " << length << '\n';

    array[0] = 5; // set element 0 to value 5

    delete[] array; // use array delete to deallocate array

    // we don't need to set array to nullptr/0 here because it's going to go out of scope immediately after this anyway

    return 0;
}
```

Because we are allocating an array, C++ knows that it should use the array version of new instead of the scalar version of new. Essentially, the new[] operator is called, even though the [] isn't placed next to the new keyword.

The length of dynamically allocated arrays has to be a type that's convertible to `std::size_t`. In practice, using an `int` length is fine, since `int` will convert to `std::size_t`.

Author's note

Some might argue that because array new expects a length of type `size_t`, our lengths (e.g. such as `length` in the example above) should either be of type `size_t` or converted to a `size_t` via `static_cast`.

I find this argument uncompelling for a number of reasons. First, it contradicts the best practice to use signed integers over unsigned ones. Second, when creating dynamic arrays using an integral length, it's convention to do something like this:

```cpp
double* ptr { new double[5] };
```

`5` is an `int` literal, so we get an implicit conversion to `size_t`. Prior to C++23, there is no way to create a `size_t` literal without using `static_cast`! If the designers of C++ had intended us to strictly use `size_t` types here, they would have provided a way to create literals of type `size_t`.

The most common counterargument is that some pedantic compiler might flag this as a signed/unsigned conversion error (since we always treat warnings as errors). However, it's worth noting that GCC does not flag this as a signed/unsigned conversion error even when such warnings (-Wconversion) are enabled.

While there is nothing wrong with using `size_t` as the length of a dynamically allocated array, in this tutorial series, we will not be pedantic about requiring it.

Note that because this memory is allocated from a different place than the memory used for fixed arrays, the size of the array can be quite large. You can run the program above and allocate an array of length 1,000,000 (or probably even 100,000,000) without issue. Try it! Because of this, programs that need to allocate a lot of memory in C++ typically do so dynamically.

Dynamically deleting arrays

When deleting a dynamically allocated array, we have to use the array version of delete, which is delete[].

This tells the CPU that it needs to clean up multiple variables instead of a single variable. One of the most common mistakes that new programmers make when dealing with dynamic memory allocation is to use delete instead of delete[] when deleting a dynamically allocated array. Using the scalar version of delete on an array will result in undefined behavior, such as data corruption, memory leaks, crashes, or other problems.

One often asked question of array delete[] is, "How does array delete know how much memory to delete?" The answer is that array new[] keeps track of how much memory was allocated to a variable, so that array delete[] can delete the proper amount. Unfortunately, this size/length isn't accessible to the programmer.

Dynamic arrays are almost identical to fixed arrays

In lesson 10.8 -- Pointers and arrays, you learned that a fixed array holds the memory address of the first array element. You also learned that a fixed array can decay into a pointer that points to the first element of the array. In this decayed form, the length of the fixed array is not available (and therefore neither is the size of the array via sizeof()), but otherwise there is little difference.

A dynamic array starts its life as a pointer that points to the first element of the array. Consequently, it has the same limitations in that it doesn't know its length or size. A dynamic array functions identically to a decayed fixed array, with the exception that the programmer is responsible for deallocating the dynamic array via the delete[] keyword.

Initializing dynamically allocated arrays

If you want to initialize a dynamically allocated array to 0, the syntax is quite simple:

```cpp
int* array{ new int[length]{} };
```

Prior to C++11, there was no easy way to initialize a dynamic array to a non-zero value (initializer lists only worked for fixed arrays). This means you had to loop through the array and assign element values explicitly.

```cpp
int* array = new int[5];
array[0] = 9;
array[1] = 7;
array[2] = 5;
array[3] = 3;
array[4] = 1;
```

Super annoying!

However, starting with C++11, it's now possible to initialize dynamic arrays using initializer lists!

```cpp
int fixedArray[5] = { 9, 7, 5, 3, 1 }; // initialize a fixed array before C++11
int* array{ new int[5]{ 9, 7, 5, 3, 1 } }; // initialize a dynamic array since C++11
// To prevent writing the type twice, we can use auto. This is often done for types with long names.
auto* array{ new int[5]{ 9, 7, 5, 3, 1 } };
```

Note that this syntax has no operator= between the array length and the initializer list.

For consistency, fixed arrays can also be initialized using uniform initialization:

```cpp
int fixedArray[]{ 9, 7, 5, 3, 1 }; // initialize a fixed array in C++11
char fixedArray[]{ "Hello, world!" }; // initialize a fixed array in C++11
```

Explicitly stating the size of the array is optional.

Resizing arrays

Dynamically allocating an array allows you to set the array length at the time of allocation. However, C++ does not provide a built-in way to resize an array that has already been allocated. It is possible to work around this limitation by dynamically allocating a new array, copying the elements over, and deleting the old array. However, this is error prone, especially when the element type is a class (which have special rules governing how they are created).

Consequently, we recommend avoiding doing this yourself.

Fortunately, if you need this capability, C++ provides a resizable array as part of the standard library called std::vector. We'll introduce std::vector shortly.

Quiz time

Question #1

Write a program that:

- Asks the user how many names they wish to enter.
- Dynamically allocates a `std::string` array.
- Asks the user to enter each name.
- Calls `std::sort` to sort the names (See 10.4 -- Sorting an array using selection sort and 10.9 -- Pointer arithmetic and array indexing)
- Prints the sorted list of names.

`std::string` supports comparing strings via the comparison operators < and >. You don't need to implement string comparison by hand.

Your output should match this:

```
How many names would you like to enter? 5
Enter name #1: Jason
Enter name #2: Mark
Enter name #3: Alex
Enter name #4: Chris
Enter name #5: John

Here is your sorted list:
Name #1: Alex
Name #2: Chris
Name #3: Jason
Name #4: John
Name #5: Mark
```

A reminder

You can use `std::getline()` to read in names that contain spaces (see lesson 4.13 -- An introduction to std::string).

A reminder

To use `std::sort()` with a pointer to an array, calculate begin and end manually

```cpp
std::sort(array, array + arrayLength);
```

Hide Solution

```cpp
#include <algorithm> // std::sort
#include <iostream>
#include <string>

int getNameCount()
{
    std::cout << "How many names would you like to enter? ";
    int length{};
    std::cin >> length;

    return length;
}

// Asks user to enter all the names
void getNames(std::string* names, int length)
{
    for (int i{ 0 }; i < length; ++i)
    {
        std::cout << "Enter name #" << i + 1 << ": ";
        std::getline(std::cin >> std::ws, names[i]);
    }
}

// Prints the sorted names
void printNames(std::string* names, int length)
{
    std::cout << "\nHere is your sorted list:\n";

    for (int i{ 0 }; i < length; ++i)
        std::cout << "Name #" << i + 1 << ": " << names[i] << '\n';
}

int main()
{
    int length{ getNameCount() };

    // Allocate an array to hold the names
    auto* names{ new std::string[length]{} };

    getNames(names, length);

    // Sort the array
    std::sort(names, names + length);

    printNames(names, length);

    // don't forget to use array delete
    delete[] names;
    // we don't need to set names to nullptr/0 here because it's going to go out
    // of scope immediately after this anyway.

    return 0;
}
```

Next lesson
10.13 For-each loops

Back to table of contents

Previous lesson
10.11 Dynamic memory allocation with new and delete