6.9a — Dynamically allocating arrays

BY ALEX ON AUGUST 18TH, 2015 | LAST MODIFIED BY ALEX ON APRIL 13TH, 2018

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 a
fter 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.

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 **6.8 -- 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:

```
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.

```
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!

```
int fixedArray[5] = { 9, 7, 5, 3, 1 }; // initialize a fixed array in C++03
int *array = new int[5] { 9, 7, 5, 3, 1 }; // initialize a dynamic array in C++11
```

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

For consistency, in C++11, fixed arrays can also be initialized using uniform initialization:

```
int fixedArray[5] { 9, 7, 5, 3, 1 }; // initialize a fixed array in C++11
char fixedArray[14] { "Hello, world!" }; // initialize a fixed array in C++11
```

One caveat, in C++11 you can not initialize a dynamically allocated char array from a C-style string:

```
char *array = new char[14] { "Hello, world!" }; // doesn't work in C++11
```

If you have a need to do this, dynamically allocate a std::string instead (or allocate your char array and then strcpy the string in).

Also note that dynamic arrays must be declared with an explicit length:

```
int fixedArray[] {1, 2, 3}; // okay: implicit array size for fixed arrays

int *dynamicArray1 = new int[] {1, 2, 3}; // not okay: implicit size for dynamic arrays

int *dynamicArray2 = new int[3] {1, 2, 3}; // okay: explicit size for dynamic arrays
```

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

1) Write a program that:
* Asks the user how many names they wish to enter.
* Asks the user to enter each name.
* Calls a function to sort the names (modify the selection sort code from lesson **6.4 -- Sorting an array using selection sort**)
* Prints the sorted list of names.

Hint: Use a dynamic array of std::string to hold the names.
Hint: std::string supports comparing strings via the comparison operators < and >

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
```

Quiz solutions

1) **Hide Solution**

```cpp
#include <iostream>
#include <string>
#include <utility> // for std::swap, if you're not C++11 compatible, #include <algorithm> instead

void sortArray(std::string *array, int length)
{
    // Step through each element of the array
    for (int startIndex = 0; startIndex < length; ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered so far.
        int smallestIndex = startIndex;

        // Look for smallest element remaining in the array (starting at startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)
        {
            // If the current element is smaller than our previously found smallest
            if (array[currentIndex] < array[smallestIndex])
                // This is the new smallest number for this iteration
                smallestIndex = currentIndex;
        }

        // Swap our start element with our smallest element
        std::swap(array[startIndex], array[smallestIndex]);
    }
}

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

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

    // Ask user to enter all the names
    for (int i = 0; i < length; ++i)
    {
        std::cout << "Enter name #" << i + 1 << ": ";
        std::cin >> names[i];
    }

    // Sort the array
    sortArray(names, length);

    std::cout << "\nHere is your sorted list:\n";
    // Print the sorted array
    for (int i = 0; i < length; ++i)
        std::cout << "Name #" << i + 1 << ": " << names[i] << '\n';

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

    return 0;
}
```

6.10 -- Pointers and const

Index

6.9 -- Dynamic memory allocation with new and delete

Share this:

307 comments to 6.9a — Dynamically allocating arrays

« Older Comments 1 … 3 4 5

ayush

July 5, 2018 at 7:23 am · Reply

```
#include<iostream>
#include<string>
#include<algorithm>
using namespace std;
int sortarray(int *array,int length)
{
    for(int startindex=0;startindex<=0;++startindex)
    {
        int small=startindex;
        for(int currentindex=startindex+1;currentindex<=length;++currentindex)
        {
            if(array[currentindex]<array[small])
            small=currentindex;
        }
        swap(array[startindex],array[small]);
    }
}

int main()
{
    int length;

```