6.4 — Sorting an array using selection sort

BY ALEX ON JULY 3RD, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

A case for sorting

Sorting an array is the process of arranging all of the elements in the array in a particular order. There are many different cases in which sorting an array can be useful. For example, your email program generally displays emails in order of time received, because more recent emails are typically considered more relevant. When you go to your contact list, the names are typically in alphabetical order, because it's easier to find the name you are looking for that way. Both of these presentations involve sorting data before presentation.

Sorting an array can make searching an array more efficient, not only for humans, but also for computers. For example, consider the case where we want to know whether a name appears in a list of names. In order to see whether a name was on the list, we'd have to check every element in the array to see if the name appears. For an array with many elements, searching through them all can be expensive.

However, now assume our array of names is sorted alphabetically. In this case, we only need to search up to the point where we encounter a name that is alphabetically greater than the one we are looking for. At that point, if we haven't found the name, we know it doesn't exist in the rest of the array, because all of the names we haven't looked at in the array are guaranteed to be alphabetically greater!

It turns out that there are even better algorithms to search sorted arrays. Using a simple algorithm, we can search a sorted array containing 1,000,000 elements using only 20 comparisons! The downside is, of course, that sorting an array is comparatively expensive, and it often isn't worth sorting an array in order to make searching fast unless you're going to be searching it many times.

In some cases, sorting an array can make searching unnecessary. Consider another example where we want to find the best test score. If the array is unsorted, we have to look through every element in the array to find the greatest test score. If the list is sorted, the best test score will be in the first or last position (depending on whether we sorted in ascending or descending order), so we don't need to search at all!

How sorting works

Sorting is generally performed by repeatedly comparing pairs of array elements, and swapping them if they meet some predefined criteria. The order in which these elements are compared differs depending on which sorting algorithm is used. The criteria depends on how the list will be sorted (e.g. in ascending or descending order).

To swap two elements, we can use the std::swap() function from the C++ standard library, which is defined in the algorithm header. For efficiency reasons, std::swap() was moved to the utility header in C++11.

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

int main()
{
    int x{ 2 };
    int y{ 4 };
    std::cout << "Before swap: x = " << x << ", y = " << y << '\n';
    std::swap(x, y); // swap the values of x and y
    std::cout << "After swap:  x = " << x << ", y = " << y << '\n';
}
```

This program prints:

```
Before swap: x = 2, y = 4
After swap:  x = 4, y = 2
```

Note that after the swap, the values of x and y have been interchanged!

Selection sort

There are many ways to sort an array. Selection sort is probably the easiest sort to understand, which makes it a good candidate for teaching even though it is one of the slower sorts.

Selection sort performs the following steps to sort an array from smallest to largest:
1) Starting at array index 0, search the entire array to find the smallest value
2) Swap the smallest value found in the array with the value at index 0
3) Repeat steps 1 & 2 starting from the next index

In other words, we're going to find the smallest element in the array, and swap it into the first position. Then we're going to find the next smallest element, and swap it into the second position. This process will be repeated until we run out of elements.

Here is an example of this algorithm working on 5 elements. Let's start with a sample array:

{ 30, 50, 20, 10, 40 }

First, we find the smallest element, starting from index 0:

{ 30, 50, 20, **10**, 40 }

We then swap this with the element at index 0:

{ **10**, 50, 20, **30**, 40 }

Now that the first element is sorted, we can ignore it. Now, we find the smallest element, starting from index 1:

{ *10*, 50, **20**, 30, 40 }

And swap it with the element in index 1:

{ *10*, **20**, **50**, 30, 40 }

Now we can ignore the first two elements. Find the smallest element starting at index 2:

{ *10*, *20*, 50, **30**, 40 }

And swap it with the element in index 2:

{ *10*, *20*, **30**, **50**, 40 }

Find the smallest element starting at index 3:

{ *10*, *20*, *30*, 50, **40** }

And swap it with the element in index 3:

{ *10*, *20*, *30*, **40**, **50** }

Finally, find the smallest element starting at index 4:

{ *10*, *20*, *30*, *40*, **50** }

And swap it with the element in index 4 (which doesn't do anything):

{ *10*, *20*, *30*, *40* **50** }

Done!

{ 10, 20, 30, 40, 50 }

Note that the last comparison will always be with itself (which is redundant), so we can actually stop 1 element before the end of the array.

Selection sort in C++

Here's how this algorithm is implemented in C++:

```
#include <algorithm> // for std::swap, use <utility> instead if C++11
#include <iostream>

int main()
{
    constexpr int length{ 5 };
    int array[length]{ 30, 50, 20, 10, 40 };

    // Step through each element of the array
    // (except the last one, which will already be sorted by the time we get there)
    for (int startIndex{ 0 }; startIndex < length - 1; ++startIndex)
    {
        // smallestIndex is the index of the smallest element we've encountered this iteration
        // Start by assuming the smallest element is the first element of this iteration
        int smallestIndex{ startIndex };

        // Then look for a smaller element in the rest of the array
        for (int currentIndex{ startIndex + 1 }; currentIndex < length; ++currentIndex)
        {
            // If we've found an element that is smaller than our previously found smallest
            if (array[currentIndex] < array[smallestIndex])
                // then keep track of it
                smallestIndex = currentIndex;
        }

        // smallestIndex is now the smallest element in the remaining array
                // swap our start element with our smallest element (this sorts it into the co
        std::swap(array[startIndex], array[smallestIndex]);
    }

    // Now that the whole array is sorted, print our sorted array as proof it works
    for (int index{ 0 }; index < length; ++index)
        std::cout << array[index] << ' ';

    std::cout << '\n';

    return 0;
}
```

The most confusing part of this algorithm is the loop inside of another loop (called a **nested loop**). The outside loop (startIndex) iterates through each element one by one. For each iteration of the outer loop, the inner loop (currentIndex) is used to find the smallest element in the remaining array (starting from startIndex+1). smallestIndex keeps track of the index of the smallest element found by the inner loop. Then smallestIndex is swapped with startIndex. Finally, the outer loop (startIndex) advances one element, and the process is repeated.

Hint: If you're having trouble figuring out how the above program works, it can be helpful to work through a sample case on a piece of paper. Write the starting (unsorted) array elements horizontally at the top of the paper.

Draw arrows indicating which elements startIndex, currentIndex, and smallestIndex are indexing. Manually trace through the program and redraw the arrows as the indices change. For each iteration of the outer loop, start a new line showing the current state of the array.

Sorting names works using the same algorithm. Just change the array type from int to std::string, and initialize with the appropriate values.

std::sort

Because sorting arrays is so common, the C++ standard library includes a sorting function named `std::sort`. `std::sort` lives in the <algorithm> header, and can be invoked on an array like so:

```
#include <algorithm> // for std::sort
#include <iostream>
#include <iterator> // for std::size

int main()
{
    int array[]{ 30, 50, 20, 10, 40 };

    std::sort(std::begin(array), std::end(array));

    for (int i{ 0 }; i < static_cast<int>(std::size(array)); ++i)
        std::cout << array[i] << ' ';

    std::cout << '\n';

    return 0;
}
```

By default, std::sort sorts in ascending order using operator< to compare pairs of elements and swapping them if necessary (much like our selection sort example does above).

We'll talk more about `std::sort` in a future chapter.

Quiz time

Question #1

Manually show how selection sort works on the following array: { 30, 60, 20, 50, 40, 10 }. Show the array after each swap that takes place.

Show Solution

Question #2

Rewrite the selection sort code above to sort in descending order (largest numbers first). Although this may seem complex, it is actually surprisingly simple.

Show Solution

Question #3

This one is going to be difficult, so put your game face on.

Another simple sort is called "bubble sort". Bubble sort works by comparing adjacent pairs of elements, and swapping them if the criteria is met, so that elements "bubble" to the end of the array. Although there are quite a few ways to optimize bubble sort, in this quiz we'll stick with the unoptimized version here because it's simplest.

Unoptimized bubble sort performs the following steps to sort an array from smallest to largest:
A) Compare array element 0 with array element 1. If element 0 is larger, swap it with element 1.
B) Now do the same for elements 1 and 2, and every subsequent pair of elements until you hit the end of the array. At this point, the last element in the array will be sorted.
C) Repeat the first two steps again until the array is sorted.

Write code that bubble sorts the following array according to the rules above:

```
1 int array[]{ 6, 3, 2, 9, 7, 1, 5, 4, 8 };
```

Print the sorted array elements at the end of your program.

Hint: If we're able to sort one element per iteration, that means we'll need to iterate as many times as there are numbers in our array to guarantee that the whole array is sorted.
Hint: When comparing pairs of elements, be careful of your array's range.

Show Solution

Question #4

Add two optimizations to the bubble sort algorithm you wrote in the previous quiz question:

- Notice how with each iteration of bubble sort, the biggest number remaining gets bubbled to the end of the array. After the first iteration, the last array element is sorted. After the second iteration, the second to last array element is sorted too. And so on... With each iteration, we don't need to recheck elements that we know are already sorted. Change your loop to not re-check elements that are already sorted.
- If we go through an entire iteration without doing a swap, then we know the array must already be sorted. Implement a check to determine whether any swaps were made this iteration, and if not, terminate the loop early. If the loop was terminated early, print on which iteration the sort ended early.

Your output should match this:

```
Early termination on iteration 6
1 2 3 4 5 6 7 8 9
```

Show Solution

6.5 -- Multidimensional Arrays

Index

6.3 -- Arrays and loops

C++ TUTORIAL | PRINT THIS POST

446 comments to 6.4 — Sorting an array using selection sort

« Older Comments 1 ... 5 6 7

Panda23
February 7, 2020 at 11:14 am · Reply

So this is the code for the program to sort an array in ascending format without messing with the original array. So I used an array of pointers. And the code worked fine at first. But then I introduced the lines in the main function (which I have commented out for you to see) that compile successfully but give a run time error on Visual studio compiler. The error says "Debug Assertation failed". What am I doing wrong?

```
1
2
3   #include<iostream>
4   void sort_ascen(int**, int);
5   void swap_array_of_ptr(int**, int, int);
6   void display_1(int** static_array, int length)
7   {
8       std::cout << "\nArray Elements: \n";
9
10      for (int i{ 0 }; i < length; i++)
11      {
12          std::cout << static_array[i][0] << ",";
13      }
14      std::cout << "\n";
15  }
16
17  int main()
18  {
19      int length=10;
20      int arr[10];
21      int** array_of_ptr=  new int*[10];
```

```
22
23      for (int i = 0; i < length; i++)
24      {
25          std::cout << "Enter the element for Array[" << i << "]: ";
26          std::cin >> arr[i];
27          array_of_ptr[i] = &arr[i];
28      }
29      display_1(array_of_ptr, 10);
30      sort_ascen(array_of_ptr, length);
31      display_1(array_of_ptr, length);
32
33  /*
34      for (int i{ 0 }; i < length; i++)
35      {
36          delete[] array_of_ptr[i];
37      }
38      delete[]array_of_ptr;
39
40  */
41
42      return 0;
43  }
44
45  void swap_array_of_ptr(int** arr,int i, int j)
46  {
47      //we are sorting the array of pointers here
48      int* temp=arr[i];
49      arr[i] = arr[j];
50      arr[j] = temp;
51
52  }
53  void sort_ascen(int** array, int length)
54  {
55      for (int i{ 0 }; i < length-1; ++i)
56      {
57          int min = i;
58              for (int j{ i + 1 }; j < length; j++)
59              {
60                  if (array[min][0] > array[j][0])
61                  {
62                      min = j;
63                  }
64              }
65              swap_array_of_ptr(array, i, min);
66      }
67  }
```

nascardriver
February 8, 2020 at 3:42 am · Reply

You're calling `delete[]` on an object that wasn't dynamically allocated. `array_of_ptr[i]` points to `arr[i]`, which has automatic storage duration, ie. you can't free it manually.

- Initialize variables with brace initialization for higher type safety.
- Use ++prefix. postfix++ is slower.
- Use single quotation marks for characters, they're faster.

Panda23
February 8, 2020 at 1:09 pm · Reply

Oh so I should just do :

delete[]array_of_ptr;

Got it.

One more question:Project vs Solution

I have searched on stack overflow and various other sites....but I cannot quite grasp the difference between a Project and a Solution in the Visual Studio IDE.

What I understand so far:

-> Solution is a container of sorts that can contain multiple Projects and multiple assemblies dont have to be generated. But what does this actually mean?

-> Its all a bit confusing because, though I use Visual Studio daily, I still have a very basic idea of a project as a combination of header files, cpp files and executable files.

->So what does the statement "multiple projects in one solution" means in light of this? Do some projects use files and codes from other projects when they are from one solution?

Something tells me there are more details to these. Please explain and/or point me to a place where I can learn more about these things! I would be really grateful!

spazzlo
February 6, 2020 at 11:35 am · Reply

I'm trying to have some code mix up the array so I can sort it again, but it doesn't seem to be mixing anything up!

```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <random>
#include <ctime>

int main() {

    std::mt19937 mersenne{std::mt19937(static_cast<unsigned int>(std::time(nullptr)))};

    int toSort[5] {};

    const int arrayLen{static_cast<int>(std::size(toSort))};

    std::uniform_int_distribution mix_rand{0, arrayLen};

    for(int i{0}; i < arrayLen; i++) {
        toSort[i] = i;
    }

    for(int index1{0}; index1 < arrayLen; index1++) {
        for(int index2{0}; index2 < arrayLen; index2++) {
            if(mix_rand(mersenne) == 0) {
                std::swap(toSort[index1], toSort[index2]);
            }
        }
    }

    for(int i{0}; i < arrayLen; i++) {
        std::cout << i;
        std::cout << " ";
```

```
    }

    // some sorting code down here, not relevant since I've tried disabling it and it still
}
```

What I expect it to output would be a random list of numbers, like 2, 4, 3, 1, 0
But what actually prints out is 0, 1, 2, 3, 4

nascardriver
February 7, 2020 at 8:42 am · Reply

You're printing `i`, not the array elements.

spazzlo
February 7, 2020 at 10:33 am · Reply

Ah, that makes sense! Thanks for the help. I've fixed it, and now the program works.

sherrigan
February 2, 2020 at 5:33 am · Reply

Hello hope you are having a good day!
So I am writing a sort function which should push all the 0s present in an array to the right of the array, also preserving the order of the other elements of the array that are not zero. My array is:

```
array{1,0,1,0,1,2,2,1,2,1,2,2,1,2/**/,0,0,1,2,0,0,0,1,2,2,0,2}
```

The /**/ indicate the place where my logic seems to breaking because the output produced is:

Array Elements:
1,1,1,2,2,1,2,1,2,2,1,2,0,1,0,0,0,0,0,0,0,0,0,0,0,

I cannot figure out why?

My code:

```
#include <iostream>
void swap(int* ptr, int i1,int i2)
{
    int temp{ptr[i1]};
    ptr[i1]=ptr[i2];
    ptr[i2]= temp;
}

void display(int* static_array, int length)
{
    std::cout << "\nArray Elements: \n";

        for (int i{ 0 }; i < length; i++)
        {
            std::cout << static_array[i] << ",";
        }
}


}
void sort_arr(int *arr, int l,int num_to_move)
{
    int count=l-1;
```

```
    for(int i{0};i<l;i++)
{
    if(arr[i]==num_to_move)
    {
        int j;
    int index_safekeep=i;

    for( j=i;j<l-1 ;j++)
    {
        arr[j]=arr[j+1];
    }
    arr[count--]=num_to_move;

    }
}

}

int main()
{
int arr[]{1,0,1,0,1,2,2,1,2,1,2,2,1,2/**/,0,0,1,2,0,0,0,1,2,2,0,2};

int l=sizeof(arr)/sizeof(arr[0]);

sort_arr(arr,l,0);
display(arr, l);

    return 0;
}
```

nascardriver
February 2, 2020 at 5:53 am · Reply

Hello!

- Initialize variables with brace initialization for higher type safety (Lesson 1.4).
- Use `std::size` to get an array's size (Lesson 6.2).
- Use your editor's auto-formatter.
- Name variables descriptively. Avoid abbreviations.
- Use ++prefix unless you need postfix++, it's faster than postfix++.

Change your array to `{ 0, 0, 1 }`. That should give you an idea of what's going wrong.

sherrigan
February 2, 2020 at 7:41 am · Reply

Thanks! Solved the problem!
Any tips on how to come up with potential test cases - for problems related to array sortings and arragments - that have the chance of exposing any shortcoming in my logic that I might have?I

nascardriver
February 2, 2020 at 8:03 am · Reply

Look at what the algorithm is supposed to do, cover its edge cases. In this example, 1 and 2 element arrays, zeroes at the beginning and end. Then add some cases that have no reason to fail, zeroes in the middle and a random array.

kitabski
January 25, 2020 at 6:14 am · Reply

Good day!!!
I will really appriciate if someone could clarify one point...

Here is the code:

```
#include <iostream>

int main() {
    using namespace std;

    constexpr int length{9}; // setting array length
    int array[length] {11, 43, 78, 93, 24, 37, 85, 56, 62}; // initializing the array

    for (int outer{0}; outer < length-1; ++outer) { // run the loop according to numbers in

        //short minNum {0}; // minimum will be assigned to minNum variable

        for (int inner {outer}; inner < length; ++inner) { // loop to find minimum number
            cout << "outer " << outer << ' ' << "inner " << inner << '\n'; // check
        }

    }


    for (int i {0}; i < length; ++i) { // printing sorted array
        cout << array[i] << ' ';
    }

    return 0;
}
```

Once initializing int "outer" in the first loop to 0 than within the brackets it should immediately become 1, but it remains 0 till the outer loops starts running the second time.
What have i missed?

nascardriver
January 25, 2020 at 6:32 am · Reply

The iteration expression is executed _after_ every cycle.

```
for (int i{ 0 }; i < 3; ++i)
{
  std::cout << i << '\n';
}

// 0
// 1
// 2
```

kitabski
January 25, 2020 at 6:39 am · Reply

Thanks a lot, nascardriver!!!
Just to clarify...

Does it mean after closing brace brackets?

nascardriver
January 25, 2020 at 7:04 am · Reply

`i` doesn't exist after the for-loop. I suggest you re-read lesson L.5.7.

kitabski
January 25, 2020 at 6:27 pm · Reply

Thank you a lot, nascardriver!!!!
As it turned out i just had to read more carefully:

First, we declare a loop variable named count, and assign it the value 0.

Second, count < 10 is evaluated, and since count is 0, 0 < 10 evaluates to true. Consequently, the statement executes, which prints 0.

Third, ++count is evaluated, which increments count to 1. Then the loop goes back to the second step.

Thanks again for the best tutorial and immediate help!!!
You guys are awesome!!!

Sinethemba
January 21, 2020 at 12:37 am · Reply

Hi. I have managed to solve question 4 as in below. Please feel free to comment, your input would be highly appreciated.

```
#include <iterator> // for std::size()
#include <utility>  // for std::swap()
#include <iostream>

int main()
{
    int array[]{ 6, 3, 2, 9, 7, 1, 5, 4, 8 };
    constexpr int length{ static_cast<int>(std::size(array)) };
    int currentIndex{ 0 };
    bool swapped{ true };
    // we assume that the number of required iterations is the (total size - 1)
    int index{ length - 1 };

    // step through all the elements of the array except the last element
    for (int startIndex{ 0 }; startIndex < (length - 1); ++startIndex)
    {
        int swapCount{};    // keeps track of the number of swaps per iteration

        // terminate early if we haven't made any swaps
        if ((currentIndex >= index) && swapped == false)
        {
            std::cout << "Early termination on iteration " << startIndex << '\n';
            break;
        }
        else
        {
            // compare all alternate element except 1 less than last previous element
            for (currentIndex = 0 ; currentIndex < index; ++currentIndex)
```

```
            {
                // is the current element greater than the following element
                if (array[currentIndex] > array[currentIndex + 1])
                {
                    // swap the elements
                    std::swap(array[currentIndex], array[currentIndex + 1]);
                    swapCount += 1; // increment swapCount by 1 if a swap occured
                }
            }
            // if there were any swaps, flag swapped as true otherwise flag as 0 or false
            swapped = static_cast<bool>(swapCount);
            --index;
        }
    }
    // print out the bubble sorted array in ascending order
    for (int index{ 0 }; index < length; ++index)
        std::cout << array[index] << ' ';
    std::cout << '\n';

    return 0;
}
```

nascardriver
January 21, 2020 at 5:02 am · Reply

You don't care how many swaps were made, so you don't need to count them. Setting `swapped` to `true` when you swap is enough. Then you can check `swapped` _after_ the inner loop.

kavin
January 20, 2020 at 6:42 am · Reply

Hi, i was able to solve the 1st 3 quizzes without problems. But for 4th quiz i was not able to figure out how to implement bool. So, i checked with your solution and implemented bool to printout early termination for my code. Is there any other way to implement early termination ? Btw, is my solution style ok or ?

```
#include <algorithm>
#include <iostream>
#include<iterator>

int main()
{

    int array[]{ 3, 2, 1,  4, 5, 6, 9, 8, 7};
    constexpr int length{ static_cast<int>(std::size(array)) };
    int tempLength{ length }; // to reduce extra inner loops


    for (int startIndex{ 0 }; startIndex < length - 1; ++startIndex)
    {
        bool swap{ false };

        //assume a temporary variable with value 0
        int tempIndex{ 0 };
        for (int currentIndex{ tempIndex + 1 }; currentIndex < tempLength; ++currentIndex)
        {
            if (array[tempIndex] > array[currentIndex])
            {
```

```
                std::swap(array[tempIndex], array[currentIndex]);
                swap = true;
            }
            tempIndex = currentIndex; //to compare next 2 pairs
        }
        if (!swap)
        {
            std::cout << "Early termination on iteration: " << startIndex + 1 << '\n';
            break;
        }
        --tempLength; // to skip already sorted values
    }

    //print our sorted array
    for (int index{ 0 }; index < length; ++index)
        std::cout << array[index] << ' ';

    std::cout << '\n';

    return 0;
}
```

nascardriver
January 20, 2020 at 7:23 am · Reply

Hi!

Instead of `swap`, you could use a variable that counts how many swaps where made and check if that variable is greater than 0 at the end. This isn't recommended, but several other people did this so it might be easier to understand. If you look through the comments, you'll find it.

Your code looks great :) There's an extra space in- and above your array declaration and a missing space at your include. Your auto-formatter would have fixed those.

kavin
January 20, 2020 at 8:12 am · Reply

Thank you @nascardriver. I am using auto format by edit-->advanced-->format document [or] selection all and using edit-->advanced-->format selection. It formats all code except #include(s) and spacing between statements. Which setting should i change ?

nascardriver
January 20, 2020 at 8:14 am · Reply

I don't have VS, I can't help you there. Seems odd that the formatter doesn't add/remove spaces. Look through the settings and google.

Suyash
January 20, 2020 at 2:48 am · Reply

Here is my complete implementation of the Bubble Sort algorithm (after implementing both the optimizations)... Works as expected but as usual, I would love to hear from you about any unintentional mistakes...

(This exercise reminded me of my algorithms course in college... Good times...)

In main.cpp (Compiler is C++17 compatible)

```
#include <iostream>
#include <iterator>  // For std::size
#include <utility>   // For std::swap

void bubbleSort(int array[], const int length);
void printArray(const int array[], const int length);

int main()
{
    int array[]{ 6, 3, 2, 9, 7, 1, 5, 4, 8 };
    bubbleSort(array, static_cast<int>(std::size(array)));
    printArray(array, static_cast<int>(std::size(array)));

    return 0;
}

// Implement Bubble Sort where elements are sorted in ascending order.
void bubbleSort(int array[], const int length)
{
    // Stores the index position which will be considered the final
    // comparison to be made at the end of each sub-iteration.
    int final_index{ length };

    // Iterate through the elements of an Array (`length` - 1) times
    // because performing sort operations on the last element is
    // redundant.
    for (int counter{ 0 }; counter < length - 1; ++counter)
    {
        // Reset `current_index` to point to the first element
        // of the `array` at the beginning of each sub-iteration.
        int current_index{ 0 };
        // Assume that the largest value in the `array` is located
        // at index 0.
        int max_value{ array[current_index] };
        // Assume that `array` is sorted at the beginning of
        // each sub-iteration.
        bool is_sorted{ true };


        // Iterate through the rest of the array and perform swaps if
        // the `max_value` is larger than the element stored on the
        // `next_index` of the `array`, to ensure that the largest element
        // bubbles up to the end of the `array` at the end of each sub-iteration.
        for (int next_index{ current_index + 1 }; next_index < final_index; ++next_index)
        {
            // If `max_value` is larger than the element stored on the
            // `next_index`, perform a swap; otherwise, assign the next element
            // in the `array` to `max_value`.
            // This will ensure that `max_value` will continue to hold
            // the largest value at the end of each potential swap step.
            if (max_value > array[next_index])
            {
                std::swap(array[current_index], array[next_index]);
                is_sorted = false;
            }
            else
            {
                max_value = array[next_index];
            }

```

```
            // Assign the value of `next_index` to `current_index`
            // to ensure that `current_index` & 'next_index` continues
            // to point to two successive indices of the `array` at the
            // start of next sub_iteration.
            current_index = next_index;
        }

        // Optimization 1: Decrement `final_index` by 1 at the
        // end of each iteration because the largest element in
        // the `array` will be already at `final_index`.
        --final_index;

        // Optimization 2: If no swapping has taken place in the
        // previous sub-iteration, then it can be safely assumed
        // that the `array` has been completely sorted.
        if (is_sorted)
        {
            std::cout << "Early termination on iteration " << counter + 1 << '\n';
            break;
        }
    }
}

// Print the elements in the Array.
void printArray(const int array[], const int length)
{
    for (int index{ 0 }; index < length; ++index)
    {
        std::cout << array[index] << ' ';
    }
    std::cout << '\n';
}
```

nascardriver
January 20, 2020 at 5:20 am · Reply

Looking good!

I find `max_value` confusing, because it's always `array[next_index - 1]`. Do whatever you find easier to understand.

« Older Comments 1 ... 5 6 7