C++ Chapter 6 Arrays and Vectors

Contents

 No AI use.

Time required: 120 minutes

What is Computer Science?

Study of data and algorithms

An algorithm is a set of rules or instructions to be followed. A tool for solving problems.

Programming is Problem Solving.

Solve the problem first.

1. Understand the problem
2. Figure out the algorithms and data
3. Write down the algorithms and data in a way we can understand
4. Translate algorithm into computer language

Online Tutorial

Go through the following tutorials before starting the tutorials

- C++ Arrays
- C++ Arrays and Loops
- C++ Omit Array Size
- C++ Get Array Size
- C++ Multidimensional Arrays
- C++ Arrays Video
- C++ Vectors
- Vectors (Tutorial with Try it Live)
- Vectors and Vector Functions (Video)

Tutorial 1: Stack Frames

Stack frames are an area in memory used to store function calls and their local variables. The stack is comprised of several Stack Frames, with each frame representing a function call.

The size of the stack increases in proportion to the number of functions called, and then shrinks upon return of a completed function.

The Stack works on a LIFO (Last In First Out) basis.

Each stack frame contains:

1. The returning line number
2. Any arguments from the called function
3. Storage space for all the function's (automatic) variables

4. Various bookkeeping data

Consider the following program, containing two functions:

```
#include <stdio.h>
// Function prototypes
void firstFunc();
void secondFunc();
int main()
{
    printf("Hello, World! \n");
    printf("Main is the first function in the stack.\n");
    firstFunc();
        printf("The stack will be empty when the program exits.");
    return 0;
}

void firstFunc()
{
    int myInt = 17;
    printf("This is the second function in the stack.\n");
    printf("This function has an int with a value of: %d \n", myInt);
    secondFunc();
    printf("See you later.");
}

void secondFunc()
{
    int yourInt = 42;
    char myChar = 'd';
    printf("This is the third function in the stack.\n");
    printf("This function has an int: %d, and a char: %c \n", yourInt,
myChar);
}
```

You can use Python Tutor to visualize this.

1. Upon program start an initial stack frame is created for main()

2. firstFunc() is called and a new stack frame is created from unused stack memory, containing:

Line to return to = the line after where it was called from in main() = Line 9

Storage space for an int

3. secondFunc() is called and a new stack frame is created from unused stack memory, containing:

Line to return to = the line after where it was called from in firstFunc() = Line 15

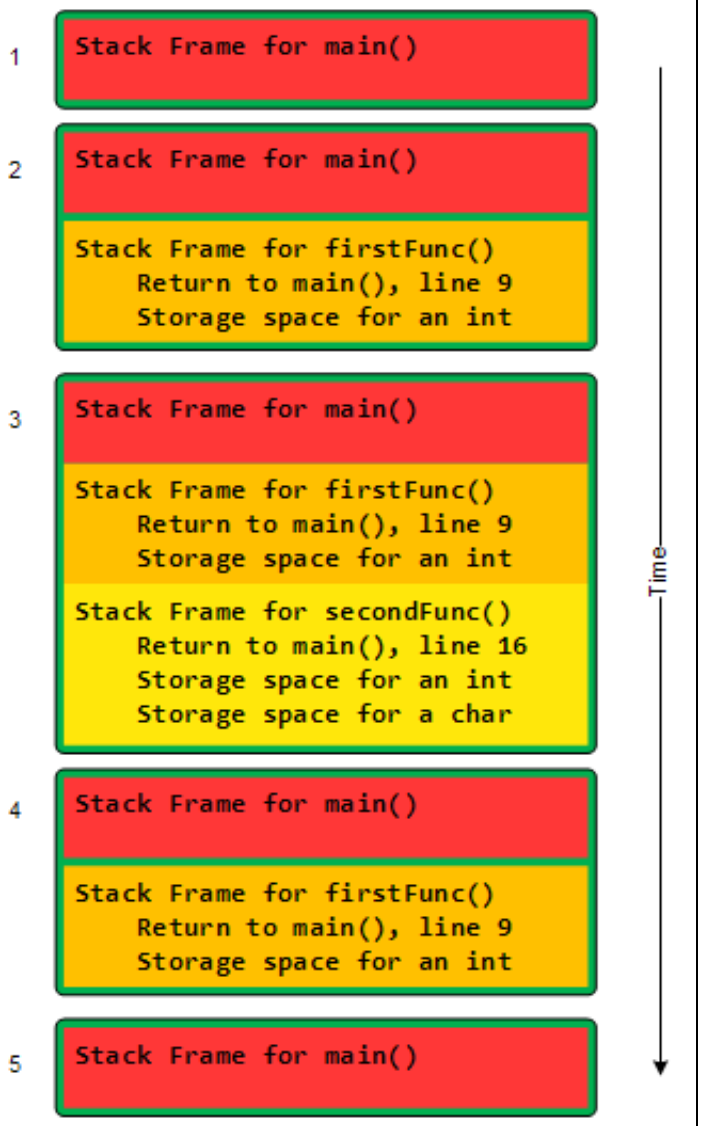
Storage space for an int

Storage space for a char

4. When secondFunc() returns, it's frame is used to determine where to return to (line 15 of firstFunc()), then deallocated and the space returned to the stack

5. When firstFunc() returns, it's frame is used to determine where to return to (line 9 of main()), then deallocated and the space returned to the stack

When main() returns, the program ends.



Example run:

```
Hello, World!
Main is the first function in the stack.
This is the second function in the stack.
This function has an int with a value of: 17
This is the third function in the stack.
This function has an int: 42, and a char: d
See you later.The stack will be empty when the program exits.
```

Please run the above program. Attach a screen shot showing the run.

Arrays

We will learn how to store multiple values of the same type by using a single variable. This language feature will enable you to write programs that manipulate larger amounts of data.

An array is a sequence of values; the values in the array are called elements. You can make an array of ints, doubles, Strings, or any other type, but all the values in an array must have the same type.

There two types of arrays in C++, traditional C style arrays, and the modern std::array which was introduced in C++11. We will used the modern std::array.

1. Size Safety and Awareness:

Traditional array: Loses its size information when passed to functions, requiring manual size tracking

```
int oldArray[5] = {1, 2, 3, 4, 5};
// Size information is lost in functions
void process(int arr[]) {
    // sizeof(arr) won't work here!
}
```

std::array: Maintains its size information and provides a size() method

```
std::array<int, 5> modernArray = {1, 2, 3, 4, 5};
modernArray.size(); // Always returns 5
```

2. Bounds Checking:

Traditional array: No built-in bounds checking, can lead to buffer overflows

```
int oldArray[5] = {1, 2, 3, 4, 5};
oldArray[10] = 1; // Dangerous! No runtime error, leads to undefined
behavior
```

std::array: Provides .at() method with bounds checking

```
std::array<int, 5> modernArray = {1, 2, 3, 4, 5};
modernArray.at(10); // Throws std::out_of_range exception
```

3. Assignment and Copying:

Traditional array: Cannot be assigned or copied directly

```
int arr1[5] = {1, 2, 3, 4, 5};
int arr2[5];
arr2 = arr1; // Compilation error!
```

std::array: Supports direct assignment and copying

```
std::array<int, 5> arr1 = {1, 2, 3, 4, 5};
std::array<int, 5> arr2;
arr2 = arr1; // Works perfectly fine
```

4. Modern array provides additional member functions

```
std::array<int, 5> arr = {1, 2, 3, 4, 5};
arr.front();  // First element
arr.back();   // Last element
arr.empty();  // Check if empty
arr.fill(0);  // Fill with value
```

The main advantage of std::array is that it provides a safer, more feature-rich alternative to traditional arrays while maintaining the same performance characteristics (zero overhead abstraction).

Creating Arrays

To create an array, you declare a reference variable with an array type, then create the array itself. Array types look like other C++ types, except they are followed by square brackets ([]).

For example, the following lines declare that **counts** is an "integer array" and **values** is a "double array".

```
#include <array>
std::array<int, 5> counts{};
// Constant for size of array
const int SIZE = 5;
std::array<double, SIZE> counts{};
```

The number of elements depends on the value of size (at the time the array is created).

You can use any integer expression for the size of an array. The value must be nonnegative.

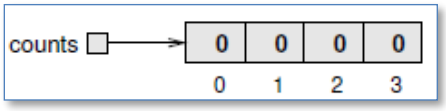
You can initialize an array with a comma-separated sequence of elements enclosed in braces, like this:

```
std::array<int, 5> counts = {1, 2, 3, 4, 5};
```

This statement creates an array variable, **myArray**, and creates an array with five elements.

Accessing Arrays

When you create an array without assigning values, the elements are initialized to garbage values. The following diagram shows a memory diagram of the counts array.



The arrow indicates that the value of counts is a reference to the array. You should think of the array and the variable that refers to it as two different things. As you'll soon see, we can assign a different variable to refer to the same array, and we can change the value of counts to refer to a different array.

The boldface numbers inside the boxes are the elements of the array. The lighter numbers outside the boxes are the indexes used to identify each location in the array. As with strings, the index of the first element is 0, not 1.

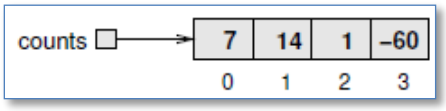
The .at() member function selects elements from an array. Using the .at() member function is a safer way to access the array, it uses bounds checking that will throw an exception that can be handled.

```
std::array<int, 3> arr = {1, 2, 3};
try {
    // This will throw an exception - index 5 is out of bounds
    arr.at(5);
} catch (const std::out_of_range& e) {
    std::cout << "Error: " << e.what() << std::endl;
}
```

You can access an array anywhere in an expression:

```
counts.at(0) = 7;
counts.at(1) = counts.at(0) * 2;
counts.at(2)++;
counts.at(3) -= 60;
```

The following diagram shows the results of these statements.



Arrays and For Loops

For loops are commonly used to iterate through an array one element at a time.

```
/**
 * Name: ArrayForLoop.cpp
 * Written by:
 * Written on:
 * Purpose: Demonstrate array and for loop
 */
#include <iostream>
#include <array>

int main()
{
    const int SIZE = 5;
    // Create and initialize array
    std::array<int, SIZE> counts = {1, 2, 3, 4, 5};
    // Iterate through array one element at a time
    for (int i = 0; i < SIZE; i++)
    {
        std::cout << counts.at(i) << std::endl;
    }
    return 0;
}
```

Example run:

```
1
2
3
4
5
```

The loop variable i is used as the index element to iterate through the array.

For the counts array, the only legal indexes are 0, 1, 2, 3, and 4. If the index is negative or greater than 3, the result is a random integer.

Many computations can be implemented by looping through the elements of an array and performing an operation on each element. Looping through the elements of an array is called traversing or iterating.

```
/**
 * Name: ArraSquares.cpp
 * Written by:
 * Written on:
 * Purpose: Demonstrate array and for loop
 */
#include <iostream>
#include <array>
int main()
{
    const int SIZE = 5;
    // Creat and initialize an array
    std::array<int, SIZE> anArray = {1, 2, 3, 4, 5};

    // Square each element of an array
    for (int i = 0; i < SIZE; i++)
    {
        // Print original element
        std::cout << anArray.at(i);
        // Square current element
        anArray.at(i) = anArray.at(i) * anArray.at(i);
        // Print squared element
        std::cout << " " << anArray.at(i) << std::endl;
    }
    return 0;
}
```

Example run:

```
1 1
2 4
3 9
4 16
5 25
```

This example traverses an array and squares each element. At the end of the loop, the array has the values {1, 4, 9, 16, 25}.

For Each Loop

Traversing arrays is so common, C++ provides an alternative syntax that makes the code more compact. This is like the enhanced for loop in Java.

We could rewrite the loop like this:

```
/**
 * Name: ArrayForEachLoop.cpp
 * Written by:
 * Written on:
 * Purpose: Demonstrate array and for loop
 */
#include <iostream>
#include <array>

int main()
{
    const int SIZE = 5;
    // Create and initialize array
    std::array<int, SIZE> counts = {1, 2, 3, 4, 5};
    // Iterate through array one element at a time
    for (int count : counts)
    {
        std::cout << count << std::endl;
    }
    return 0;
}
```

This statement is called a range based for loop, also known as the "for each" loop. You can read the code as, "for each value in values". It's conventional to use plural nouns for array variables and singular nouns for element variables.

Notice how the single line for (int value : values) replaces the first two lines of the standard for loop. It hides the details of iterating each index of the array, and instead, focuses on the values themselves.

Using the enhanced for loop, we can write an accumulator very concisely:

```
std::array<int, 5> scores = {1, 2, 3, 4, 5};
for (int score : scores){
    counts.at(score)++;
}
```

Tutorial 2: MoreArrayFun

Arrays hold multiple values. This example shows a couple of ways to initialize and iterate through an array.

```cpp
/**
 * Filename: MoreArrayFun.cpp
 * Written by:
 * Written on:
 * Demonstration of CPP Arrays
 */

#include <iostream>
#include <string>
#include <array>
int main()
{
    // Constant for Array size
    const int ARRAY_SIZE = 10;

    // Create empty array of Integers
    std::array<int, ARRAY_SIZE> numberArray{};

    // Hard code array elements
    std::array<std::string, 4> names = {"Bob", "Sally", "John", "Ed"};
```

```
    std::cout << "Fill array with integers with for loop" << std::endl;
    // Fill array with for loop
    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        numberArray.at(i) = i;
        std::cout << numberArray[i] << " ";
    }

    // Access individual array elements
    numberArray.at(1) = 25;
    numberArray.at(3) = numberArray.at(4) * 2;
    numberArray.at(8)++;
    numberArray.at(2) += 20;

    std::cout << "\nnumberArray after accesing the array." << std::endl;
    // Go through array one at a time and print the element
    // Use the .size() to determine the length of the array
    for (int i = 0; i < numberArray.size(); i++)
    {
        std::cout << numberArray.at(i) << " ";
    }

    std::cout << "\nFor each loop with string array" << std::endl;
    // For Each loop goes through each element in an array
    for (std::string a : names)
    {
        std::cout << a << " ";
    }
    return 0;
}
```

Example run:

```
Fill array with integers with for loop
0 1 2 3 4 5 6 7 8 9
numberArray after accesing the array.
0 25 22 8 4 5 6 7 9 9
For each loop with string array
Bob Sally John Ed
```

Vectors

In C++, vectors are dynamic arrays that can grow or shrink in size. They are part of the C++ Standard Library and are defined in the `<vector>` header. Here's how you can use vectors in C++:

Include the **<vector>** header at the beginning of your C++ code to use vectors.

```
#include <vector>
```

Declare a vector: You can declare a vector using the following syntax:

```
// Replace 'type' with the type of elements you want to store in the vector,
// Replace 'name' with the desired name of the vector.
std::vector<type> name;
```

For example, to create a vector of integers:

```
// Declares an empty vector of integers
std::vector<int> my_vector;
```

Initialize a vector: You can initialize a vector with elements using various methods, such as:

```
// Initialize with values
std::vector<int> my_vector = {1, 2, 3, 4, 5};

// Initialize with a specific size and default value
// Creates a vector of size 5 with all elements initialized to 0
std::vector<int> my_vector(5, 0);

// Creates a copy of 'another_vector'
std::vector<int> my_vector(another_vector);
```

Accessing elements: You can access elements in a vector using the index (starting from 0), just like an array using the at() member function.

```
// Using at() method
std::cout << numbers.at(1) << std::endl;
```

Modifying elements: You can modify elements in a vector by assigning a new value to a specific index.

```
// Modifies the value of the first element to 42
my_vector.at(0) = 42;
```

Vector methods: Vectors have various built-in methods, such as `size()`, `push_back()`, `pop_back()`, `insert()`, `erase()`, `clear()`, `resize()`, `swap()`, and others. You can refer to the C++ documentation for the complete list of vector methods and their usage.

- **.size() -** Returns the current number of elements in the vector.

- **.capacity() -** Returns the maximum number of elements the vector can hold before resizing.
- **.clear() -** Removes all elements from the vector, leaving it with a size of 0.
- **.pop_back()** - Removes the last element from the vector.
- **.erase(iterator)** - Removes the element at a specified position.
- **.front()** – Returns the first element of the vector.
- **.back()** – Returns the last element of the vector.
- **.insert(new_element, index)** – Inserts a new element at the index.

Here's an example that demonstrates basic usage of vectors in C++:

```
/**
 * Filename: VectorBasics.cpp
 * Written by:
 * Written on:
 * Demonstration of CPP Vectors
 */
#include <iostream>
#include <vector>

int main()
{
    // Declare and initialize a vector
    std::vector<int> my_vector = {1, 2, 3, 4, 5};

    // Access and modify elements
    std::cout << "Element at index 2: " << my_vector.at(2) << std::endl;
    my_vector.at(2) = 42;

    // Iterate through the vector with standard for loop
    std::cout << "Vector elements: ";
    for (int i = 0; i < my_vector.size(); ++i)
    {
        std::cout << my_vector.at(i) << " ";
    }
    std::cout << std::endl;

    // Adds a new element at the end of the vector
    my_vector.push_back(6);
    // Remove the last element of the vector
    my_vector.pop_back();
    // Removes the third element
    my_vector.erase(my_vector.begin() + 2);

    std::cout << "Vector size: " << my_vector.size() << std::endl;
    std::cout << "Vector capacity: " << my_vector.capacity() << std::endl;

    return 0;
}
```

Example run:

```
Element at index 2: 3
Vector elements: 1 2 42 4 5
Vector size: 4
Vector capacity: 10
```

Tutorial 3: MoreVectorFun

Vectors are much like arrays. Vectors are mutable, which means that the size can be changed after they are created.

```cpp
/**
 * Filename: MoreVectorFun.cpp
 * Written by:
 * Written on:
 * Demonstration of CPP Vectors
 */

#include <iostream>
#include <string>
#include <vector>

int main()
{
    const int NUM_PEOPLE = 2;
    // Create two parallel vectors
    std::vector<std::string> names;
    std::vector<int> hourlyPay;

    // Variable to store input from user
    std::string tempName;
    int tempHourlyPay;
```

```
23      // Fill up each vector with information
24      for (int i = 0; i < NUM_PEOPLE; i++)
25      {
26          std::cout << "Please enter a person's name: ";
27          // getline gets a string, assigns value to variable
28          std::getline(std::cin, tempName);
29
30          std::cout << "Please enter " << tempName << "'s hourly pay: ";
31          std::cin >> tempHourlyPay;
32          // Consume newline character left over from getting integer
33          std::cin.get();
34
35          // Add variable value to the end of the vector
36          names.push_back(tempName);
37          hourlyPay.push_back(tempHourlyPay);
38      }
39
40      // Create separation between input and output
41      std::cout << std::endl;
42
43      // Iterate through vector to display information
44      for (int i = 0; i < NUM_PEOPLE; i++)
45      {
46          std::cout << names.at(i) << "'s hourly pay is "
47                    << hourlyPay.at(i) << std::endl;
48      }
49
50      return 0;
51  }
```

Example run:

```
Please enter a person's name: Bill
Please enter Bill's hourly pay: 75
Please enter a person's name: Mike
Please enter Mike's hourly pay: 1

Bill's hourly pay is 75
Mike's hourly pay is 1
```

Tutorial 4: Vector Average

```cpp
/**
 * Filename: VectorAverage.cpp
 * Written by:
 * Written on:
 * Average a vector of numbers
 */
#include <iostream>
#include <vector>

int main()
{
    double sum = 0.0;
    double average = 0.0;
    const int NUMBER_OF_ENTRIES = 5;
    std::vector<double> numbers(NUMBER_OF_ENTRIES);

    std::cout << "Please enter " << NUMBER_OF_ENTRIES << " numbers: ";

    // Allow the user to enter in the values
    for (int i = 0; i < NUMBER_OF_ENTRIES; i++)
    {
        // Insert the number into the vector
        std::cin >> numbers.at(i);
        // Keep a running total
        sum += numbers.at(i);
    }

    // Calculate the average
    average = sum / NUMBER_OF_ENTRIES;

    // Display the results
    std::cout << "The average of ";
    for (int i = 0; i < NUMBER_OF_ENTRIES - 1; i++){
        std::cout << numbers.at(i) << ", ";
    }

    std::cout << numbers.at(NUMBER_OF_ENTRIES - 1) << " is "
              << average << "\n";

    return 0;
}
```

Example run:

```
Please enter 5 numbers: 10
25
36
2222
2
The average of 10, 25, 36, 2222, 2 is 459
```

Assignment 1: Array Game Scores

Write a C++ program that uses an array to store the scores of 5 players in a game. Allow the user to input scores for each player and display the total score at the end.

1. Store Scores:
    a. Use an array to store scores for 5 players in a game.
2. Custom Calculations and Statistics:
    a. After all scores are entered, the program should display:
        i. The total score.
        ii. The highest and lowest scores.
        iii. The average score.
        iv. How many players scored above and below the average.

Example run:

```
Enter the scores for each player:
Score for player 1: 3
Score for player 2: 5
Score for player 3: 7
Score for player 4: 9
Score for player 5: 12

Game Statistics:
Total Score: 36
Highest Score: 12
Lowest Score: 3
Average Score: 7.2
Number of players scoring above average: 2
Number of players scoring below average: 3
```

Assignment 2: Vector Game Scores

Write a C++ program that tracks game scores and demonstrates the basic operations of vectors. Include functions to add elements, remove elements, find the size, and display the elements of a vector.

1. Basic Operations with Validation and Feedback:
    a. Add Elements: Include a function that allows the user to add an element to the vector, but only if the element isn't already present.
    b. Remove Elements: Create a function that removes an element based on user input. Display an error if the element doesn't exist in the vector.
    c. Find Vector Size: Display the size of the vector after each addition or removal, with comments explaining how size changes in each case.
    d. Display Elements: Include a function to print the vector elements.

Example run:

```
Game Score Tracker
1. Add Score
2. Remove Score
3. Display Number of Scores
4. Display All Scores
5. Exit
Choose an option: 1
Enter score to add: 23
Score 23 added successfully.
Current number of scores: 1

Game Score Tracker
1. Add Score
2. Remove Score
3. Display Number of Scores
4. Display All Scores
5. Exit
Choose an option: 3
Total number of scores: 1

Game Score Tracker
1. Add Score
2. Remove Score
3. Display Number of Scores
4. Display All Scores
5. Exit
Choose an option: 4
Scores: 23

Game Score Tracker
1. Add Score
2. Remove Score
3. Display Number of Scores
4. Display All Scores
5. Exit
Choose an option: 2
Enter score to remove: 1
Error: Score 1 not found.
Current number of scores: 1
```

Assignment Submission

1. Attach the program files.
2. Attach screenshots showing the successful operation of the program.
3. Submit in Blackboard.