Script for the vhb lecture

Programming in C++
Part 1

Prof. Dr. Herbert Fischer
Deggendorf Institute of Technology

Table of Contents

4 Arrays and Strings

Chapter 4 deals with the topic of "arrays". You will learn how to program them in C++ and how to handle strings.

4.1 Arrays

You can store any elementary C++ data type in an array. An array is simply a collection of elements of the same data type. An array has a name and, in order to access the individual elements of the array, a position number, also known as an index, is used.

4.1.1 Single-dimensional arrays

Like any other element in C++, an array must also be defined. The definition specifies the name of the array, the data type of the elements, and the number of array elements (array size).

Syntax: type name[arraySize];

Please note that the array size is **always** an integer constant or an integer expression consisting of constants only. The size of the array cannot be changed by a variable or later in the program/source code.

Assuming you need an integer array to store five integer values. You would declare this array as follows:

- int myArray[5];

Since each integer value occupies a memory space of 4 bytes, the array requires a total of 20 bytes.

Now that you have declared an array, you can store values in it by using the array indexing operator [ ]:

- myArray [0] = -200;
- myArray [1] = -100;
- myArray [2] = 0;
- myArray [3] = 100;
- myArray [4] = 200;

As you can see from the example above, arrays in C/C++ have 0 as the index of their first element that is also called base index. When the array is accessed, counting begins at 0. This means that an array with 5 elements contains the indexes 0 to 4. Thus, the index of the last array element is always 1 lower than the number of elements. An array can be built with any data type. However, there are exceptions, such as void or certain classes. In the further course of your program, you can again use the array indexing operator to access the individual elements of the array:

- int result = myArray [3] + myArray [4]; // The result is 300

There is a simpler way to declare and store elements in an array at the same time:

- int myArray [5] = { -200, -100, 0, 100, 200 };

If you know exactly how many elements your array stores, and if you fill up the array when declaring it, you can omit the array size when declaring the array. In this case, the declaration would be as follows:

- `int myArray[] = { -200, -100, 0, 100, 200 };`

This will work because the compiler can determine from the list of assigned values how many elements the array contains and how much memory space needs to be allocated.

Example: Lottery numbers in arrays

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
{
   int lottery[6];
   int i, j;
   bool newNumber;

   srand(time(NULL));                  // Re-generate random numbers based on the time
   for(i=0; i<6; i++)                  // Draw six numbers, one after the other
   {
      do                               // Repeat drawing until the new number is not identical
      {                                // to any of the previous ones
         lottery [i] = rand() % 49 + 1; // Random number between 1 and 49
         newNumber = true;             // Positive basic setting
         for (j=0; j<i; j++)
         {                             // Run through all previously drawn numbers
            if (lottery [j]== lottery [i])
            {                          // A double number was found
               newNumber = false;
            }
         }
      } while (!newNumber);
   }
   for (i=0; i<6; i++)
   {
      cout << lottery [i] << " ";
   }
   cout << endl;
}
```

Explanation:
Using srand, random numbers are first regenerated on the basis of the current time, otherwise they would remain the same with every program call. The outer loop with variable i as an index runs through the number of lottery numbers to be drawn. The drawing itself takes place within the do-*while* loop, because it should be repeated until you find a number that has not been drawn yet. The decision can therefore only be made after the action in the body of the loop. The *do-while* loop is therefore the ideal choice because it checks the condition at the bottom of the loop. The validation of the numbers takes place after the drawing. All numbers drawn are traversed in an inner *for* loop. This means that the index j starts at 0 and remains smaller than i, i. e. smaller than the index of the number drawn just now. If this number is equal to a number drawn previously, the boolean variable newNumber is set to false. This causes the drawing to be repeated. After each drawing, this variable must be set to true, otherwise the loop will never be exited once a double drawing was detected.

The end of an array must not be overwritten. One of the most powerful features of C/C++ is direct access to the memory. Due to this feature, C/C++ does not prevent you from writing to a certain memory location, even if your program should not have access to this memory area. The following code is valid, but will cause your program to crash:

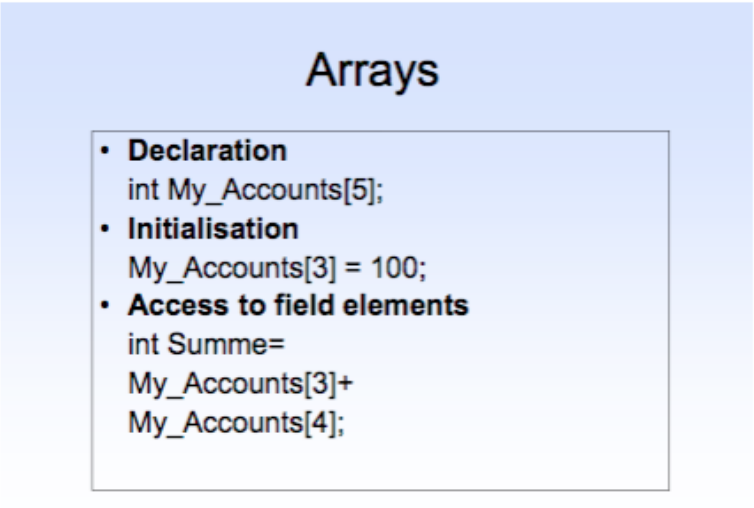
Example:

```
int Array[5];
Array[5] = 10;
```

This is a common mistake caused by arrays having a base index of 0. One is easily tempted to assume that the last element of the array has index 5, whereas it actually has index 4. If you overwrite the last element of an array, you will not know which memory is affected. In the best case, the result is unpredictable. In the worst case, your program or even the operating system will crash. This problem is very difficult to pinpoint, because the affected memory area is accessed much later and the crash occurs only then. This means you should be really careful when you define an array!

Rules for the use of arrays:

- The indices of arrays always start at 0. The index of the first element in an array will be 0, the index of the second will be 1, the index of the third will be 2, and so forth.
- Array sizes must be compile-time constants. At the time of the compilation, the compiler must know how much memory space it needs to allocate for the array. Therefore, you cannot use a variable to declare the size of an array. The code below would therefore not be valid and would cause a compiler error:
  ```
  int x = 10;
  int myArray[x];      // Compiler error
  ```
- Please make sure you do not overwrite the last element of an array.
- Allocate large arrays in the dynamic memory (=heap) rather than in the stack memory (more on this in the second part of the course).
- Arrays that have been allocated in the heap memory can use a variable to declare the array size:
  ```
  int x = 10;
  int myArray[x] = new int[x]; // this is OK (also part of the second part of the course)
  ```



4.1.2 Multi-dimensional arrays

Arrays can be multidimensional. To declare a two-dimensional array with integer elements, you would enter the following code:

- int myArray[3][5];

This allocates memory for 15 integer values (60 bytes in total). In principle, you access the elements of such an array in the same way as in a single-dimensional array, except that you must specify two array indexing operators.

- int x = myArray [1][1] + myArray [2][1];

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
        int table[2][3]={{10,11,12},{2,3,4}};        // Array with 2 rows and 3 columns

        cout<<"In the 1st row, 2nd column you can see " << table [0][1] <<endl;  // Result: 11
        cout<<"In the 2nd row, 1st column you can see " << table [1][0] <<endl;  // Result: 2
        cout<<"In the 2nd row, 2nd column you can see " << table [1][1] <<endl; // Result: 3

        system("pause");
        return 0;
}
```

A further example would be to calculate the annual sales figures. The data (numbers) defined in the array represent the monthly sales figures that are added up.

Example: Sale figures

```
#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
  // Sales figures for 12 months over a period of 2 years
  // [2] → line, i. e. 2 years, and [12] --> column, number of months per year
   float numbers [2] [12] = {{20.5, 22, 25.2, 30, 55, 66, 74, 20.55, 22, 25.2, 30, 55},
                             {20.55, 22, 25.2, 55, 8, 99, 44, 22.5, 66.3, 10, 11, 12.5}};
   // Total sales figures per year over a period of 2 years
   float sales [2] [1];

   for (int i = 0; i < 2; i++) {
      for (int j = 0; j < 12; j++) {
          sales [i][0] += numbers [i][j];          // Add up sales figures of the individual months
                                                   // Assign the value to a variable
      }
   }

   cout << "Total sales 1st year: " << sales[0][0] << endl;
   cout << "Total sales 2nd year: " << sales[1][0] << endl;

   system("pause");
   return 0;
}
```

4.2 Strings (Character strings)

C++ has two types of strings:

- C-style strings: single-dimensional arrays of characters (char-variables)
- C++-strings: *string* class from the standard library

4.2.1 C-Strings

The *char* data type is used for a single character.

➢ char character = 'A';

Make sure that the A is written in single quotes. The compiler literally does not write the character A into the variable, but the character code of the character in the ASCII table.

Note: A character within single quotes is considered as a value.

In C++ programs, strings are represented by arrays of the *char* data type. Note that a string is written in double quotes.
For example, you could assign a string to a character array as follows:

➢ char text[] = "This is a string.";

With this you reserve 18 bytes of memory and store the string in this memory area. Depending on how smart you are, you may already have noticed that this string actually contains only 17 characters. The reason why there are 18 bytes reserved is that there is a null character at the end of each string and C/C++ takes this fact into account by leaving space for the null-terminating character in the memory allocation. The *null-terminating character* is a special character represented by \0, which corresponds to a numeric null 0. If the program encounters a 0 in the character field, it interprets this position as the end of the string.

Example:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
        char str[] = "This is a string.";
        cout << str << endl;
        str[7] = '\0';                                  // Single quotes !
        cout << str << endl;
        cout << endl << "Press any key to continue...";
        system("pause");
}
```

Explanation:
At the beginning, the character array contains the string "This is a string.", followed by the terminating null-terminating character. This string is printed on the screen using cout. In the next line, the seventh element of the array is assigned the value \0, the null-terminating character. The string is printed on the screen again. This time, however, only "This is" is being displayed. This is due to the fact that from the computer's point of view, the string in the array ends with the seventh element. The remaining characters are still in memory, but cannot be displayed due to the null-terminating character.

4.2.2 C++ Strings

In C++ the *string* class is defined in the header file <string>.

Declaration:

```
#include <string>
using namespace std;
string str1; // empty string
```

Examples for the definition of strings:

```
string str2 = „I am a string“;
string str3(10, ‚+’);      // Variable str3 holds the value 10 plus sign
string str4(str2,2,4);    // Variable str4 holds the value: am a
```

Strings can be concatenated with the overloaded operator + (Operator overloading see part 2 chapter 5):

```
string str5 = str2 + “for testing purposes“; // Variable str5 holds the value: I am a string for testing purposes
```

Example: Other string features

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
        string str1= "My house";            // ANNOTATION: The index of the first character of a string is 0 (just like
                                            // arrays).
        string str2= str1.substr(3,5);      // Outputs a substring. Deletes 5 characters of the string from the
                                            // forth character (because index starts at 0) (result: house).
        string str3= str1.erase(1,3);       // Deletes characters 1 to 3 from the first character of the string
                                            // (result: mouse)

        cout << str2 << endl;
        cout << str3 << endl;
        string strA="Apple”;
        string strB=“Miss Mar”;
        string strC= strA.replace(0,2,strB); // Replaces the 0th character and the 2 following characters of string strA
        cout << strC << endl <<endl;        // with the passed string strB (result: Miss Marple)

        string str4="abcd";
        cout << str4.length() << endl;      // Outputs the length of the string (result: 4)
        cout << str4.find("d") << endl;     // Returns the index of the matching letter, if no letter exists
                                            // => Output of a higher number than the length of the string
                                            // (result: 3)

        return 0;
}
```

4.3 Sorting

You usually want to output similar data on a larger scale in a sorted order. Thus you will come across the topic of sorting again and again in your life as a programmer. For this reason, the standard C++ library also contains a sorting procedure. For the sake of simplicity, you would probably also use the sorting procedure in practice. For practice purposes in dealing with arrays, however, it will do you no harm to program a simple sorting algorithm yourself.

There are several ways of sorting the elements of an array. We use the so-called bubble sort. It requires two nested loops. The outer loop specifies the number of comparisons. The algorithm repeatedly compares two adjacent elements and swaps them if the right one is smaller than the left one. The name originates from the fact that the larger values rise like bubbles and move to the right.
Since the largest element is placed on the far right of the list after each pass, you now only have to sort a list that is one element shorter than before. This means that you only have to pass through the list as many times as it has elements.

Example: Bubble sorting

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
        int numbers[6] = {43,22,100,500,3,99};

        int i, j;                                       // indices for counting
        int tmp;                                        // auxiliary variable for the swapping of elements
        for (i = 0; i < 6 - 1; ++i){                    // pass through the entire array
                for (j = 0; j < 6 - i - 1; ++j){        // start at the end and look for the smallest element
                if (numbers [j] > numbers [j + 1])      // check if the element is smaller than its predecessor
                {       // swap elements
                        int tmp = numbers [j];
                        numbers [j] = numbers [j + 1];
                        numbers [j + 1] = tmp;
                }
                }
           }

        for(int i=0;i<6;i++)
                cout << numbers [i] << endl;

        system("pause");
        return 0;
}
```