

Arrays in C++



Computer Engineering
Shahed University
winter 2021 (1399)

Teacher Assistants :
Amirreza Tavakoli

Arrays in C++

- An array in C or C++ is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They are used to store similar type of elements as in the data type must be the same for all elements. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C or C++ can store derived data types such as the structures, pointers etc. Given below is the picturesque representation of an array.

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

Array Length = 9

First Index = 0

Last Index = 8

Why do we need arrays?

- We can use normal variables (v1, v2, v3, ..) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

Array Declaration in C

`int a[3];`

2192	451	13918
------	-----	-------

`int a[3]={1, 2, 3};`

1	2	3
---	---	---

`int a[3]={1, 1, 1};`

1	1	1
---	---	---

`int a[3]={ };`

0	0	0
---	---	---

`int a[3]={ 0 };`

0	0	0
---	---	---

`int a[3]={ 1 };`

1	0	0
---	---	---

`int a[3]={ [0...1]=3 };`

3	3	0
---	---	---

`int a[]={ [0...1]=3 };`

3	3
---	---

`int *a;`

`int* a;`

`int * a;`

`int*a;`

Advantages of an Array in C++:

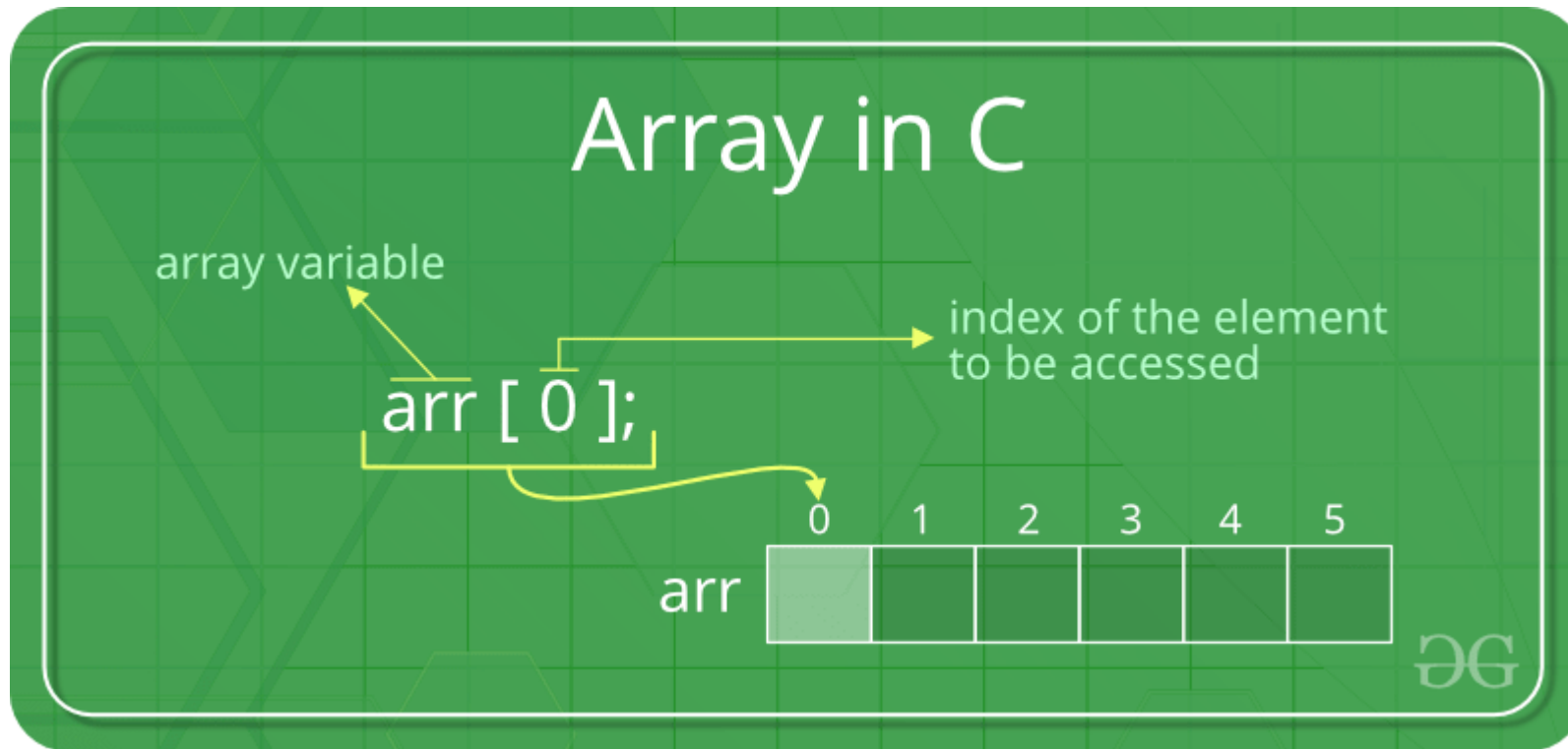
1. Random access of elements using array index.
2. Use of less line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing less line of code.

Disadvantages of an Array in C/C++:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Accessing Array Elements:

- Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.



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

int main()
{
    int arr[5];
    arr[0] = 5;
    arr[2] = -10;

    // this is same as arr[1] = 2
    arr[3 / 2] = 2;
    arr[3] = arr[0];

    cout << arr[0] << " " << arr[1] << " " << arr[2] << " " << arr[3];

    return 0;
}
```

No Index Out of bound Checking:

```
// This C++ program compiles fine  
// as index out of bound  
// is not checked in C.
```

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

```
int main()  
{  
    int arr[2];  
  
    cout << arr[3] << " ";  
    cout << arr[-2] << " ";  
  
    return 0;  
}
```

Output :

-449684907 4195777

Multidimensional Arrays in C++

General form of declaring N-dimensional arrays:

```
data_type array_name[size1][size2]....[sizeN];
```

data_type: Type of data to be stored in the array.

Here data_type is valid C/C++ data type

array_name: Name of the array

size1, size2,... ,sizeN: Sizes of the dimensions

Examples:

Two dimensional array:

```
int two_d[10][20];
```

Three dimensional array:

```
int three_d[10][20][30];
```

Size of multidimensional arrays

- Total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.
For example:
- The array **int x[10][20]** can store total $(10 * 20) = 200$ elements.
- Similarly array **int x[5][10][20]** can store total $(5 * 10 * 20) = 1000$ elements.

Two-Dimensional Array

- Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one – dimensional array for easier understanding.
- The basic form of declaring a two-dimensional array of size x, y:

Syntax

```
data_type array_name[x][y];
```

data_type: Type of data to be stored. Valid C/C++ data type.

We can declare a two dimensional integer array say 'x' of size 10,20 as:

```
int x[10][20];
```

- Elements in two-dimensional arrays are commonly referred by $x[i][j]$ where i is the row number and ' j ' is the column number.
- A two – dimensional array can be seen as a table with ' x ' rows and ' y ' columns where the row number ranges from 0 to $(x-1)$ and column number ranges from 0 to $(y-1)$.
- A two – dimensional array 'x' with 3 rows and 3 columns is shown below:

	Column 0	Column 1	Column 2
Row 0	$x[0][0]$	$x[0][1]$	$x[0][2]$
Row 1	$x[1][0]$	$x[1][1]$	$x[1][2]$
Row 2	$x[2][0]$	$x[2][1]$	$x[2][2]$

Initializing Two – Dimensional Arrays

- There are two ways in which a Two-Dimensional array can be initialized.

First Method:

```
int x[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

The above array have 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in the order, first 4 elements from the left in first row, next 4 elements in second row and so on.

Better Method:

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
```

This type of initialization make use of nested braces. Each set of inner braces represents one row. In the above example there are total three rows so there are three sets of inner braces.

Accessing Elements of Two-Dimensional Arrays

- Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

Example:

```
int x[2][1];
```

- The above example represents the element present in third row and second column.
- Note: In arrays if size of array is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is $2+1 = 3$.
- To output all the elements of a Two-Dimensional array we can use nested for loops. We **will require two for loops**. One to traverse the rows and another to traverse columns.

```
// C++ Program to print the elements of a
// Two-Dimensional array
#include<iostream>
using namespace std;

int main()
{
    // an array with 3 rows and 2 columns.
    int x[3][2] = {{0,1}, {2,3}, {4,5}};

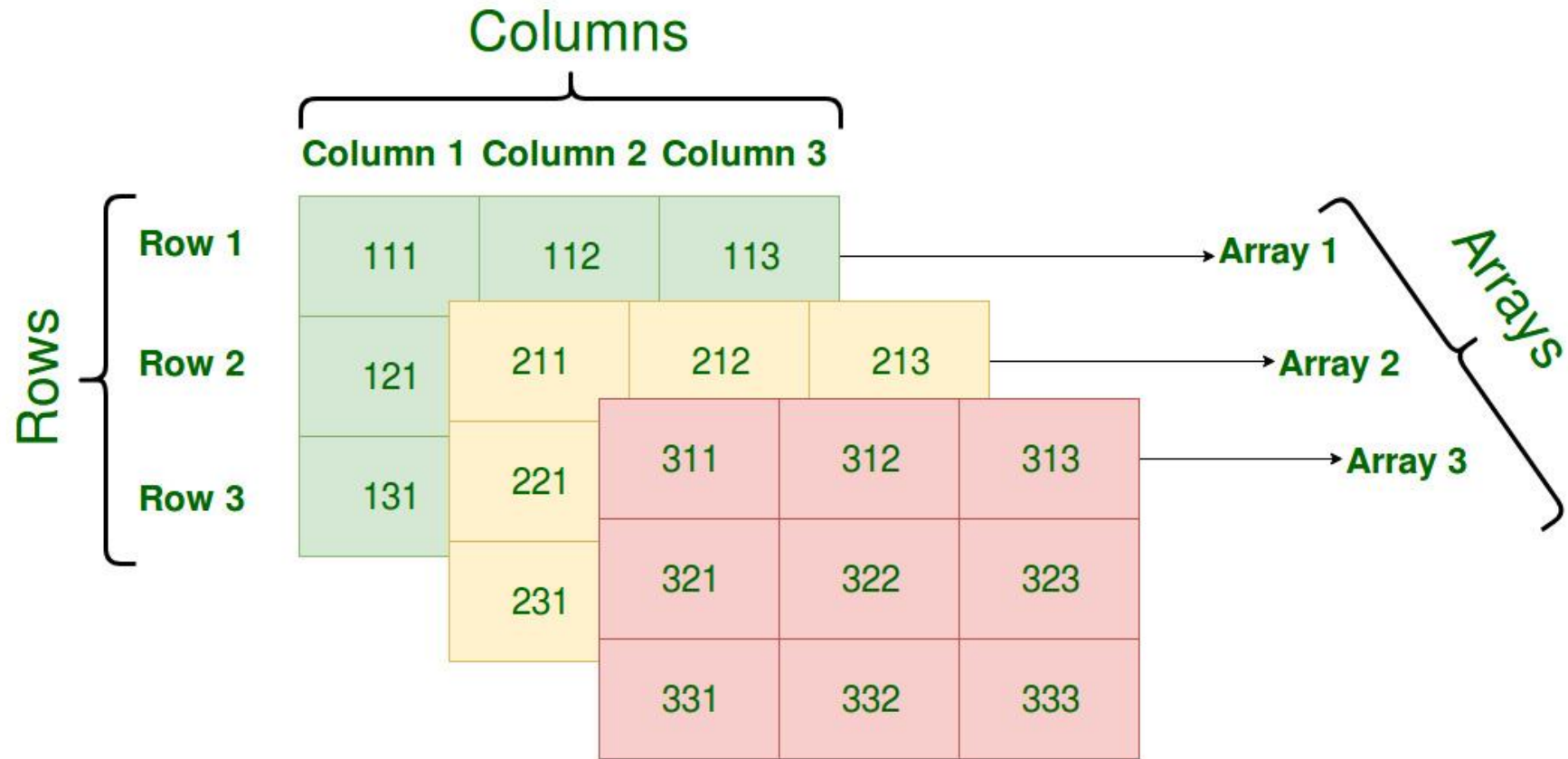
    // output each array element's value
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cout << "Element at x[" << i
                << "][" << j << "]: ";
            cout << x[i][j]<<endl;
        }
    }

    return 0;
}
```

Output:

Element at x[0][0]: 0
Element at x[0][1]: 1
Element at x[1][0]: 2
Element at x[1][1]: 3
Element at x[2][0]: 4
Element at x[2][1]: 5

Three-Dimensional Array



Initializing Three-Dimensional Array:

- Initialization in Three-Dimensional array is same as that of Two-dimensional arrays. The difference is as the number of dimension increases so the number of nested braces will also increase.

First Method:

```
int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23};
```

Better Method:

```
int x[2][3][4] =  
{  
    { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },  
    { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }  
};
```

Accessing elements in Three-Dimensional Arrays

- Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

```

// C++ program to print elements of Three-Dimensional
// Array
#include<iostream>
using namespace std;

int main()
{
    // initializing the 3-dimensional array
    int x[2][3][2] =
    {
        { {0,1}, {2,3}, {4,5} },
        { {6,7}, {8,9}, {10,11} }
    };

    // output each element's value
    for (int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 2; ++k)
            {
                cout << "Element at x[" << i << "][" << j << "][" << k << "] = " << x[i][j][k] << endl;
            }
        }
    }
    return 0;
}

```

Output:

```

Element at x[0][0][0] = 0
Element at x[0][0][1] = 1
Element at x[0][1][0] = 2
Element at x[0][1][1] = 3
Element at x[0][2][0] = 4
Element at x[0][2][1] = 5
Element at x[1][0][0] = 6
Element at x[1][0][1] = 7
Element at x[1][1][0] = 8
Element at x[1][1][1] = 9
Element at x[1][2][0] = 10
Element at x[1][2][1] = 11

```

Other number of dimension

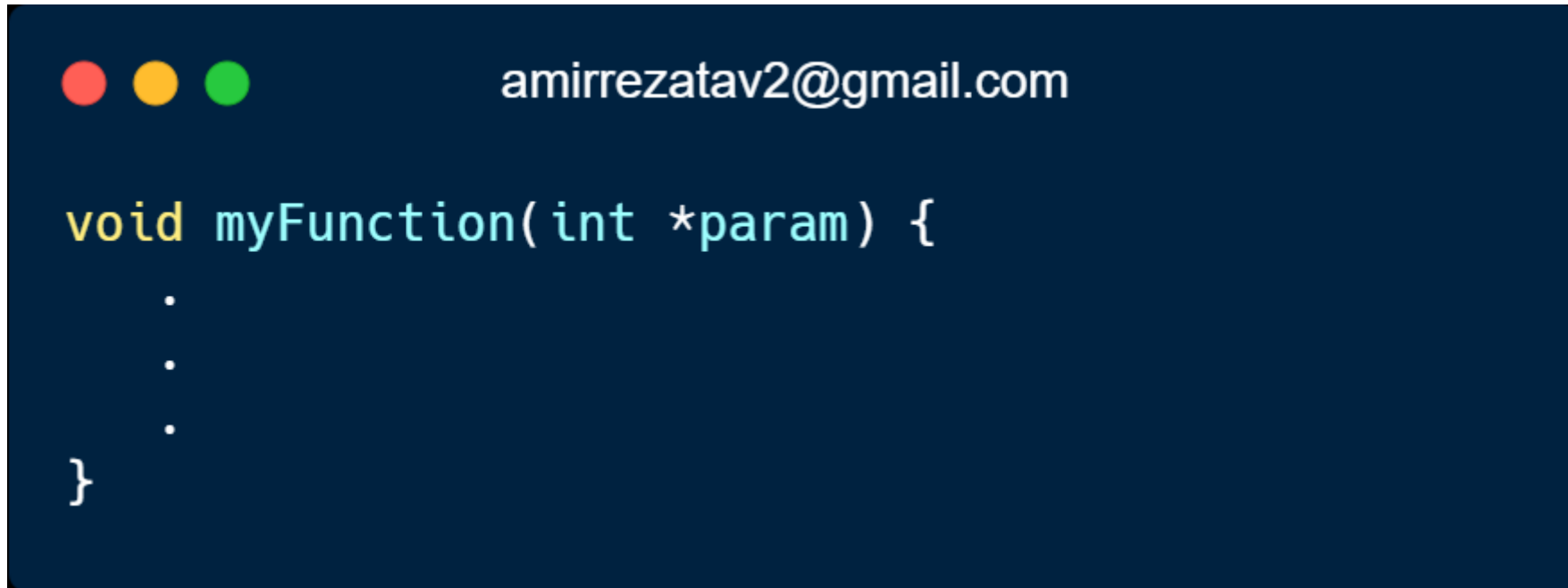
- In similar ways, we can create arrays with any number of dimension.
- However the complexity also increases as the number of dimension increases.
- The most used multidimensional array is the Two-Dimensional Array.

C++ Passing Arrays to Functions

- C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.
- If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

C++ Passing Arrays to Functions

- Way-1
 - Formal parameters as a pointer as follows –

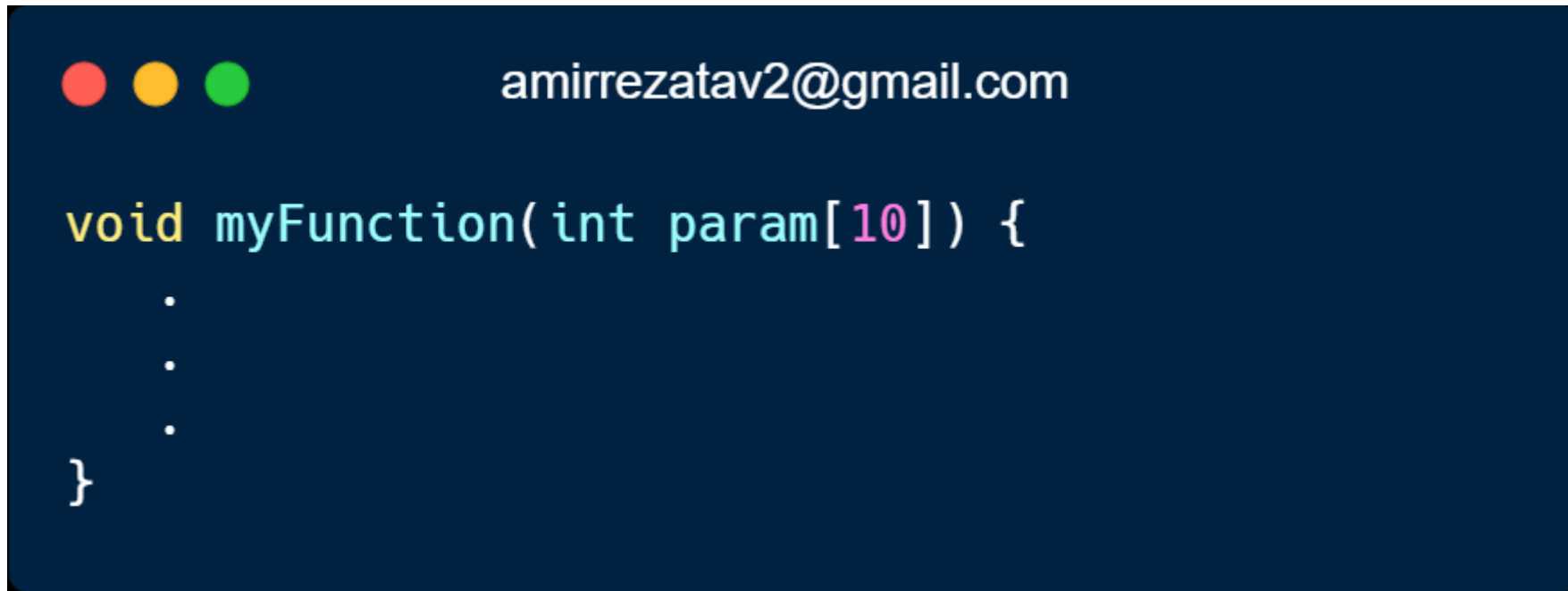


```
void myFunction(int *param) {  
    .  
    .  
    .  
}
```

The image shows a terminal window with a dark blue background. At the top left, there are three colored circles (red, yellow, green) representing window control buttons. To their right, the email address "amirrezatav2@gmail.com" is displayed in white. Below this, C++ code is shown in a monospaced font. The code defines a function named "myFunction" that takes a pointer to an integer ("int *param") as its parameter. The function body contains three lines of code, each starting with a dot ".". The function is closed with a closing curly brace "}".

C++ Passing Arrays to Functions

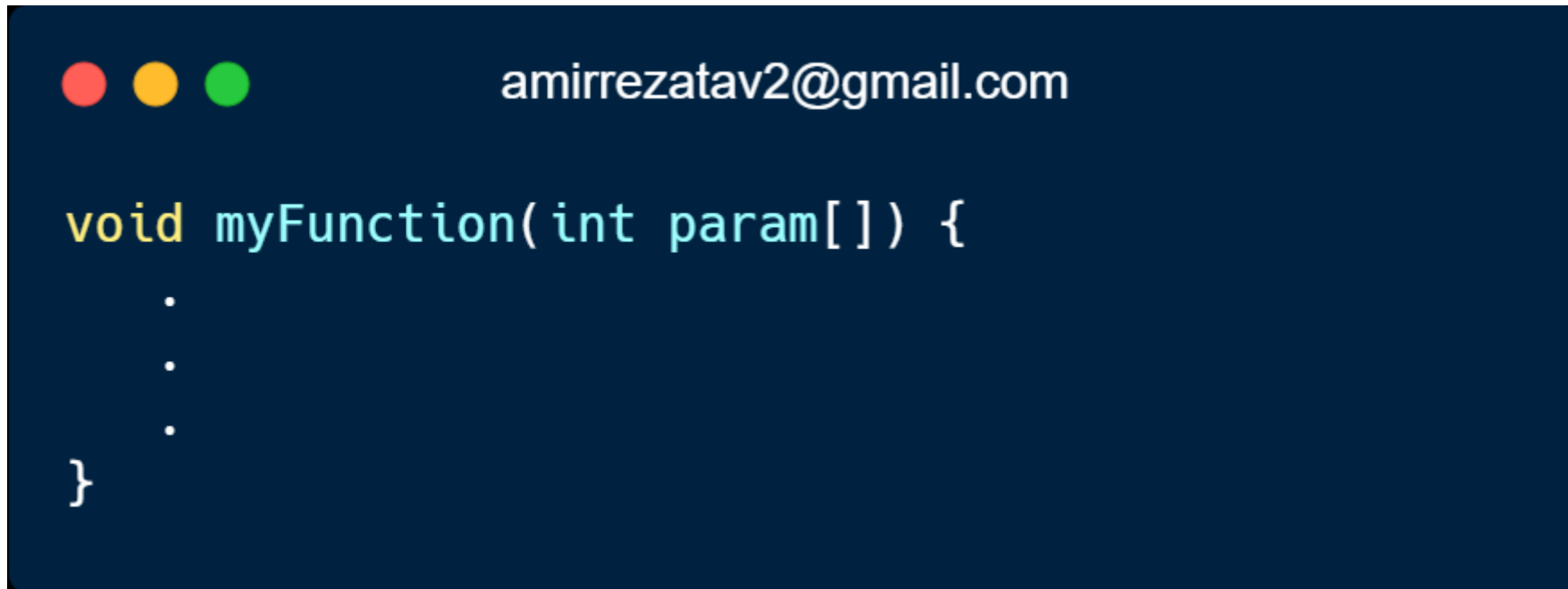
- Way-2
 - Formal parameters as a sized array as follows –



```
void myFunction(int param[10]) {  
    .  
    .  
    .  
}
```

C++ Passing Arrays to Functions

- Way-3
 - Formal parameters as an unsized array as follows –



```
void myFunction(int param[]) {  
    .  
    .  
    .  
}
```

A terminal window with a dark blue background. The title bar at the top shows three colored circles (red, yellow, green) on the left and the email address "amirrezataav2@gmail.com" on the right. The code is written in a monospaced font, with "void" in yellow and the rest in white.

C++ Passing Arrays to Functions

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

// function declaration:
double getAverage(int arr[], int size);

int main() {
    // an int array with 5 elements.
    int balance[5] = { 1000, 2, 3, 17, 50 };
    double avg;

    // pass pointer to the array as an argument.
    avg = getAverage(balance, 5);

    // output the returned value
    cout << "Average value is: " << avg << endl;

    return 0;
}
```

Output:
Average value is: 214.4

C++ Pointer to an Array

It is most likely that you would not understand this chapter until you go through the chapter related C++ Pointers.

So assuming you have bit understanding on pointers in C++, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration –

```
double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p the address of the first element of balance –

```
double *p;
```

```
double balance[10];
```

```
p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of first element in p, you can access array elements using *p, *(p+1), *(p+2) and so on.

C++ Pointer to an Array

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

int main() {
    // an array with 5 elements.
    double balance[5] = { 1000.0, 2.0, 3.4, 17.0, 50.0 };
    double* p;

    p = balance;

    // output each array element's value
    cout << "Array values using pointer " << endl;

    for (int i = 0; i < 5; i++) {
        cout << "*(p + " << i << ") : ";
        cout << *(p + i) << endl;
    }
    cout << "Array values using balance as address " << endl;

    for (int i = 0; i < 5; i++) {
        cout << "*(balance + " << i << ") : ";
        cout << *(balance + i) << endl;
    }

    return 0;
}
```

Output:

Array values using pointer

***(p + 0) : 1000**

***(p + 1) : 2**

***(p + 2) : 3.4**

***(p + 3) : 17**

***(p + 4) : 50**

Array values using balance as address

***(balance + 0) : 1000**

***(balance + 1) : 2**

***(balance + 2) : 3.4**

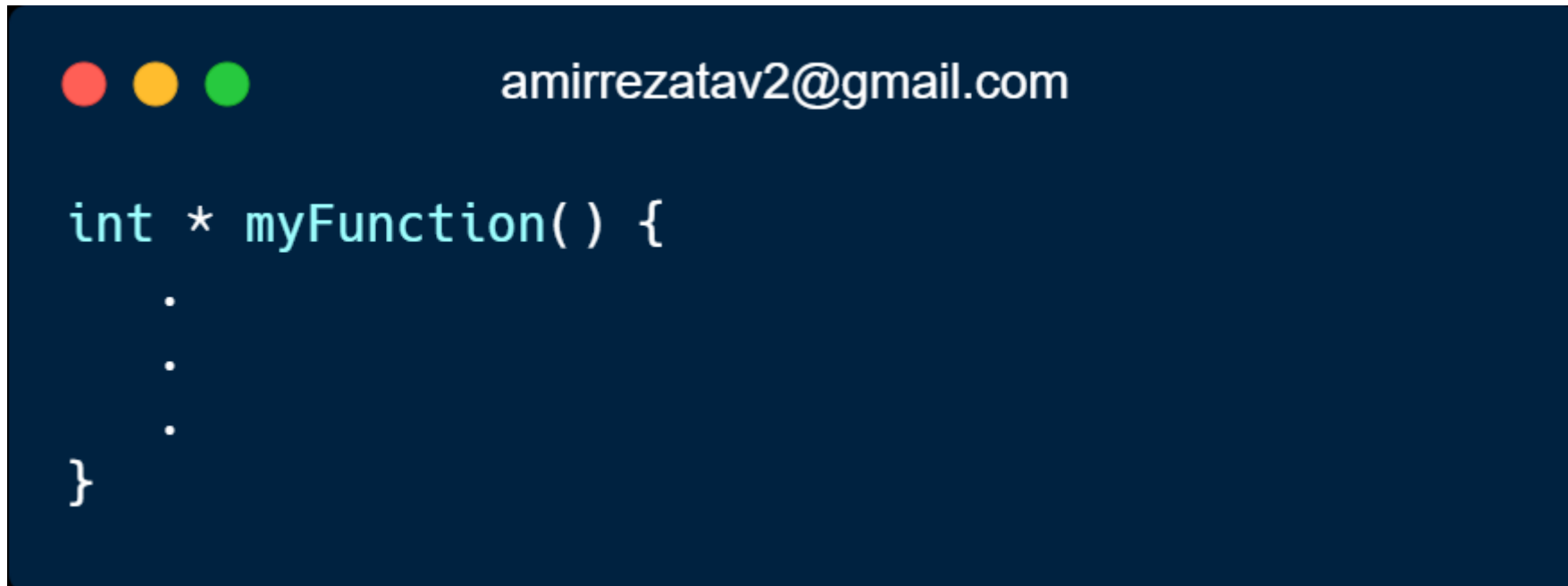
***(balance + 3) : 17**

***(balance + 4) : 50**

Return Array from Functions in C++

C++ does not allow to return an entire array as an argument to a function. However, you can return a pointer to an array by specifying the array's name without an index.

If you want to return a single-dimension array from a function, you would have to declare a function returning a pointer as in the following example –



```
int * myFunction() {  
    .  
    .  
    .  
}
```


Return Array from Functions in C++

```
#include <iostream>
#include <ctime>

using namespace std;

// function to generate and retrun random numbers.
int* getRandom() {

    static int r[10];

    // set the seed
    srand((unsigned)time(NULL));

    for (int i = 0; i < 10; ++i) {
        r[i] = rand();
        cout << r[i] << endl;
    }

    return r;
}

// main function to call above defined function.
int main() {

    // a pointer to an int.
    int* p;

    p = getRandom();

    for (int i = 0; i < 10; i++) {
        cout << "*(p + " << i << ") : ";
        cout << *(p + i) << endl;
    }

    return 0;
}
```

Output:

```
624723190
1468735695
807113585
976495677
613357504
1377296355
1530315259
1778906708
1820354158
667126415
*(p + 0) : 624723190
*(p + 1) : 1468735695
*(p + 2) : 807113585
*(p + 3) : 976495677
*(p + 4) : 613357504
*(p + 5) : 1377296355
*(p + 6) : 1530315259
*(p + 7) : 1778906708
*(p + 8) : 1820354158
*(p + 9) : 667126415
```

Good luck!