The Ultimate

C++

Part 2: Intermediate

Mosh Hamedani

codewithmosh.com

Hi! I am Mosh Hamedani. I'm a software engineer with over 20 years of experience and I've taught millions of people how to code and become professional software engineers through my YouTube channel and coding school (Code with Mosh).

This PDF is part of my **Ultimate C++ course** where you will learn everything you need to know from the absolute basics to more advanced concepts. You can find the full course on my website.

https://codewithmosh.com

https://www.youtube.com/c/programmingwithmosh

https://twitter.com/moshhamedani

https://www.facebook.com/programmingwithmosh/

Table of Content

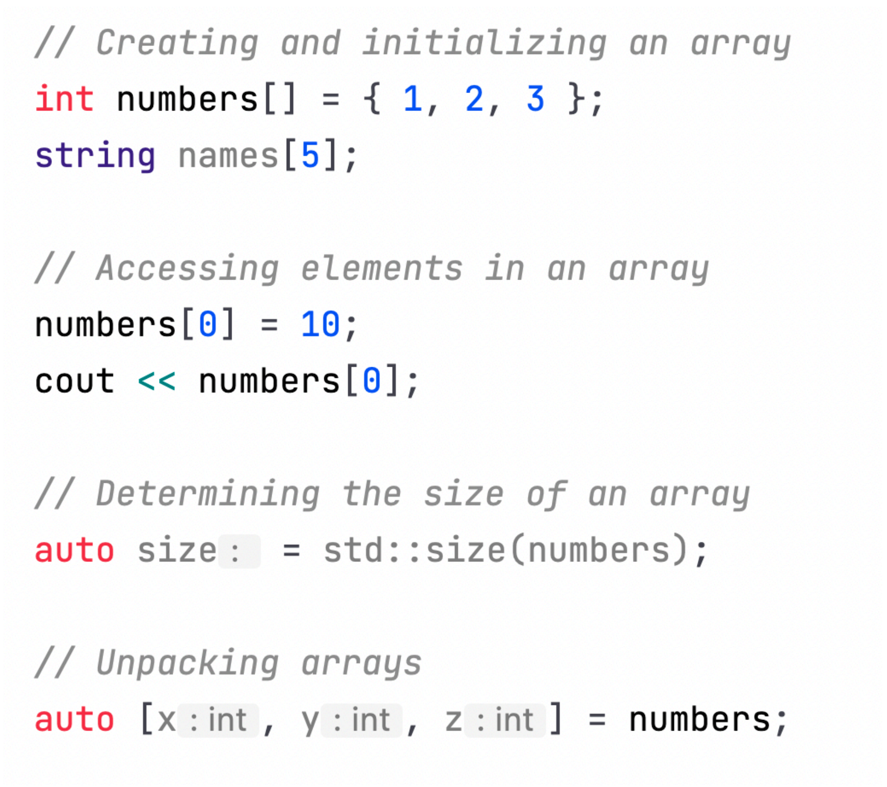
Arrays

Terms

Arrays
Array elements
Bubble sort algorithm
Linear search algorithm
Pointers
Unpacking arrays

Summary

- An *array* is a collection of items stored in sequence.
- We can access the items (also called *elements*) in an array using an index. The index of the first element in an array is 0.
- An array can be unpacked into separate variables.
- Using the **size()** function in the C++ Standard Library (STL) we can determine the size of an array.
- When passing an array to a function, we should always pass its size as well. The reason for this is that array parameters are interpreted as *pointers* (memory addresses). We cannot use a memory address to determine how many elements exist in an array.
- In C++, we cannot assign an array to another. To copy an array, we need to copy individual elements using a loop.
- Similarly, arrays cannot be compared for equality. To compare two arrays, we have to compare their individual elements.
- There are many algorithms for searching and sorting arrays. These algorithms differ based on their performance and memory usage.

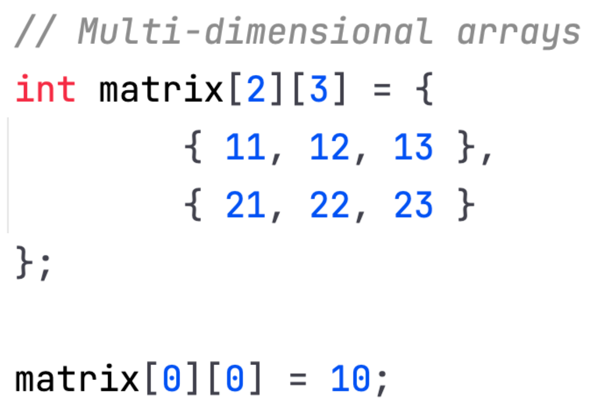


```
// Creating and initializing an array
int numbers[] = { 1, 2, 3 };
string names[5];

// Accessing elements in an array
numbers[0] = 10;
cout << numbers[0];

// Determining the size of an array
auto size : = std::size(numbers);

// Unpacking arrays
auto [x : int, y : int, z : int] = numbers;
```



```
// Multi-dimensional arrays
int matrix[2][3] = {
        { 11, 12, 13 },
        { 21, 22, 23 }
};

matrix[0][0] = 10;
```

Pointers

Terms

Address-of operator
Dereference operator
Free store
Heap memory
Indirection operator
Memory leak
Null pointer
Pointer
Reference parameters
Shared pointers
Smart pointers
Stack memory
Unique pointers

Summary

- Using the *address-of operator* **(&)** we can get the address of a variable.
- A *pointer* is a variable that holds the memory address of another variable.
- Using the *indirection* or *dereference operator* **(*)** we can get the content of a memory address stored in a pointer.
- One of the applications of pointers is to efficiently pass objects between function calls. Reference parameters are a safer and simpler alternative for the same purpose.
- A *null pointer* is a pointer that doesn't point to any objects.
- Local variables are stored in a part of memory called the *stack memory*. The memory allocated to these variables is automatically released when they go out of scope.

- We can use the **new** operator to dynamically allocate memory on a different part of memory called the *heap* (or *free store*).

- When allocating memory on the heap, we should always deallocate it using the **delete** operator. If we don't, our program's memory usage constantly increases. This is known as a *memory leak.*

- *Smart pointers* in the STL are the preferred way to work with pointers because they take care of releasing the memory when they go out of scope.

- There are two types of smart pointers: *unique* and *shared*.

- A unique pointer owns the memory address it points to. So we cannot have two unique pointers pointing to the same memory location.

- If we need multiple pointers pointing to the same memory location, we have to use shared pointers.

```cpp
// Declaring and using pointers
int number = 10;
int* ptr = &number;
*ptr = 20;

// Pointer to constant data (const int)
const int x = 10;
const int* ptr = &x;

// Constant pointer
int x = 10;
int* const ptr = &x;

// Constant pointer to constant data
int x = 10;
const int* const ptr = &x;

// Dynamic memory allocation using raw pointers
int* numbers = new int[10];
delete[] numbers;

// Dynamic memory allocation using smart pointers
#include <memory>
auto numbers : unique_ptr<int[]>  = make_unique<int[]>(10);
```

Strings

Terms

C String	Escape sequences
C++ String	Raw string
Character literal	String literal
Concatenate	Substring

Summary

- A *C-string* (also called *C-style String*) is an array of characters terminated with the null terminator (`\0`).
- C-strings cannot be copied, compared, or concatenated (combined). That's why you should avoid them in new code and prefer to use C++ strings.
- C++ strings are represented using the **string** class in the STL. This class internally uses a character array to hold a string. But it hides away all the complexity around C-strings. It dynamically resizes the array when needed and provides useful methods for working with strings.
- A *string literal* is a sequence of characters enclosed with double quotation marks.
- A *character literal* is a character enclosed with single quotation marks.
- We use escape sequences to represent special characters within string and character literals. Examples of escape sequences are `\\`, `\"`, `\'`, `\n` and `\t`.
- In *raw strings*, escape sequences are not processed.

```
// Working with C-strings
char name[5] = "Mosh";
char copy[5];

cout << strlen(name);

strcpy(copy, name);

if (strcmp(name, copy) == 0)
    cout << "Equal";


// Working with C++ strings
string name = "Mosh";

cout << name.length();

string copy = name;

if (name == copy)
    cout << "Equal";

// Modifying strings
string name = "Mosh";
name.append(" Hamedani");
name.insert(0, "I am ");
name.erase(0, 2);
name.clear();
name.replace(0, 2, "**");
```

```
// Searching strings
string name = "Mosh";
int index;
index = name.find('a');
index = name.rfind('a');
index = name.find_first_of(",.;");
index = name.find_last_of(",.;");
index = name.find_first_not_of(",.;");
```

```
// Extracting substrings
string name = "Mosh Hamedani";
string substr;
substr = name.substr();
substr = name.substr(3);
substr = name.substr(3, 5);
```

```
// Working with characters
string name = "Mosh Hamedani";
bool b;
b = isupper(name[0]);
b = islower(name[0]);
b = isdigit(name[0]);
b = isalpha(name[0]);

name[0] = toupper(name[0]);
name[0] = tolower(name[0]);
```

```
// Conversions
string str = "10";
int i = stoi(str);
double d = stod(str);
string s = to_string(10);

// Escape sequences
string message = "Hello\nWorld";
string columns = "first\tlast";
string path = "c:\\folder\\file.txt";


// Raw strings
string path = R"(c:\folder\file.txt)";
```

Structures

Terms

Enumerations
Enumerators
Methods
Nested structures
Operator overloading
Strongly-typed enums
Structures
Structure members

Summary

- We use *structures* to define custom data types.
- Members of a structure can be variables or functions (also called *methods*).
- Structures can be nested to represent more complex types.
- To compare two structures, we have to compare their individual members.
- We can provide operators for our structures using a technique called *operator overloading*.
- Just like the built-in data types, structures can be used as function parameters or their return type.
- Using an *enumeration*, we can group related constants into a single unit. Members of this unit are called *enumerators*.
- *Strongly-typed enumerations* define a scope for their members. This allows two enums having members with the same name.

```
// Defining a structure
struct Movie {
    string title;
    int releaseYear = 1900;
};


// Creating an instance of a structure
Movie movie = { "Terminator 1", 1984 };

// Unpacking a structure
auto [title : string , releaseYear : int ] = movie;


// Operator overloading
bool operator==(const Movie& first, const Movie& second) {
    return (
        first.title == second.title &&
        first.releaseYear == second.releaseYear
    );
}

// Pointer to a structure
void showMovie(Movie* movie) {
    // Structure pointer operator
    cout << movie->title;
}
```

```cpp
// Classic (unscoped) enumeration
enum Action {
    list,
    add,
    update
};


// Strongly-typed enumeration
enum class Operation {
    list,
    add,
    update
};


// Using enumerations
void doSomething(Operation operation) {
    if (operation == Operation::add) {
        // ...
    }
    else if (operation == Operation::list) {
        // ...
    }
}
```

Streams

Terms

Binary files	Output streams
Buffer	Streams
File streams	String streams
Input streams	Text files

Summary

- A *stream* is an abstraction for a data source or destination. Using streams, we can read data or write it to a variety of places (eg terminal, files, network, etc) in the same way.

- In the C++ STL, we have many stream classes for different purposes. All these classes inherit their functionality from **ios_base**.

- A *buffer* is a temporary storage in memory used for reading or writing data to streams.

- If an error occurs while reading data from a stream, the invalid data stays in the buffer and will be used for subsequent reads. In such situations, first we have to put the stream into a clean state using the **clear()** method. Then, we should clear the data in the buffer using the **ignore()** method.

- In the C++ STL, we have three stream classes for working with files. (**ifstream** for reading from files, **ofstream** for writing to files, and **fstream** for reading and writing to files).

- *Binary files* store data the same way it is stored in memory. They are more efficient for storing large amount of numeric data but they're not human readable.

- Using *string streams* we can convert data to a string or vice versa.

```
// Writing to a stream
cout << "Hello World";

// Reading from a stream
int number;
cin >> number;

// Handling read errors
if (cin.fail()) {
    cout << "Error";
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
}


// Writing to a text file
ofstream file;
file.open("file.txt", ios::app);
if (file.is_open()) {
    file << "Hello World";
    file.close();
}
```

```
// Reading from a text file
ifstream file;
file.open("file.txt");
if (file.is_open()) {
    string str;
    while(!file.eof()) {
        getline(file, str);
    }
    file.close();
}


// Writing to a binary file
int numbers[3] = { 1, 2, 3 };

ofstream file;
file.open("file.bin", ios::app | ios::binary);
if (file.is_open()) {
    file.write(
        reinterpret_cast<char*>(&numbers),
        sizeof(numbers));
    file.close();
}
```

```
// Reading from a binary file
ifstream file;
file.open("file.bin", ios::binary);
if (file.is_open()) {
    int number;
    while (file.read(
        reinterpret_cast<char*>(&number),
        sizeof(number))) {
        cout << number << endl;
    }
    file.close();
}
```