Chapter 6

STL Arrays and Vectors

Lets Review

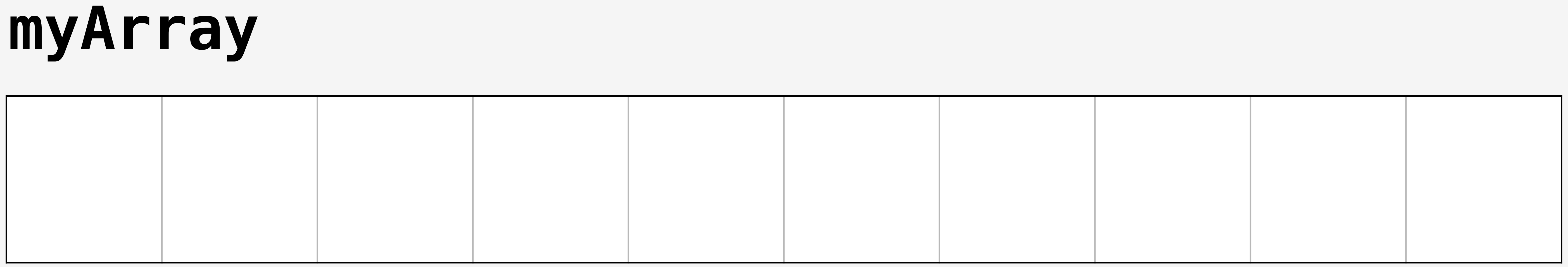
C++ Built-in arrays

Built-in Arrays

Built-in arrays are a sequence of elements that are contained in a contiguous block of memory

- C Arrays do NOT know their own size. Programmer is responsible for managing the size of an array.
- Regular arrays of local scope are left uninitialized - contents of the array are unknown.

```
const int SIZE = 10;
int myArray[SIZE];
```



Array Initialization

```
const int ARRAY_SIZE = 5;

int myArray[ARRAY_SIZE];
myArray[0] = 1;
myArray[1] = 2;
…

// With C++11

int myArray[ARRAY_SIZE] = { 1, 2, 3, 4, 5 };

int myArray[ARRAY_SIZE] { 1, 2, 3, 4, 5 };

int myArray[ARRAY_SIZE] { };
```

example: arrays.cpp

Arrays and Functions

- Arrays are inherently pass ***by reference***
- We can pass any array to a function by indicating `[]`

```
void printArray(int myArray[], const int size);
```

- Any updates made to ***myArray*** are maintained through the rest of the program. We typically pass the ***size*** along as an argument with an array

Returning an array

We cannot return an array that is created locally in a function because it goes out of scope.

We will talk about how we can ***actually*** return an array in Chapter 11 by using pointers.

example: arrays_func.cpp

STL Vectors

Data Structure

Vector Basics

A ***vector*** is a data structure that groups values of the same type under a single name in a list (similar to an array).

- In Fact, the underlying data structure of a ***vector*** is just a dynamically allocated c array
- The ***vector*** container class allows us to use common list operations and better represent a list of values.
- Use vectors with `#include <vector>`

```
vector<dataType> vectorName(numElements);
```

Vector Basics

- Vectors can be initialized as empty

```
vector<int> myVector;
```

- With a number ***n*** of values (all initialized to some empty state - depends on the data type)

```
vector<int> myVector(4);
```

- Or With a specified default value for all elements

```
vector<int> myVector(4, -1)
```

myVector	-1	-1	-1	-1

example: vector_basics.cpp

Accessing Elements

- Accessing elements of an array work by using an ***offset*** value.
- With c arrays we access elements by ***[index]***
- With vectors we access elements by using ***.at(index)*** which is a member function of the vector class that does bounds checking!

```
vector<int> myVector(5);
cout << vector.at(1); // can access elements
vector.at(i) = 10; // can also mutate elements
```

Accessing elements of a vector

When using vectors and stl arrays, ***always*** use the `.at(index)` member function!

Common ***vector*** member functions

`.push_back(item)` - used to append an element to the end of a vector

- If there is no memory at the end of the vector to store the new element, the vector doubles the ***capacity*** of the vector, inserts the new item, then updates the size

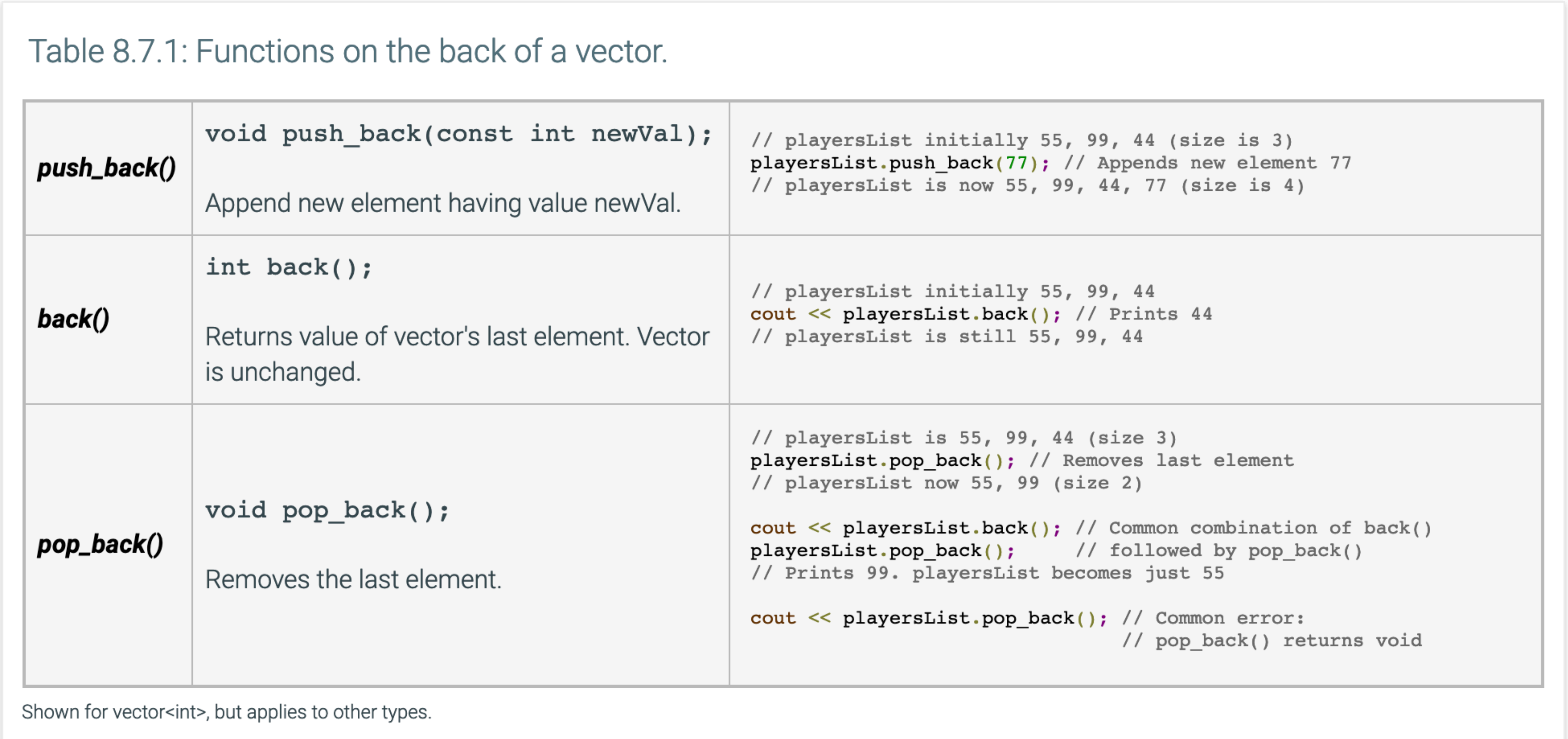


Table 8.7.1: Functions on the back of a vector.

push_back()	`void push_back(const int newVal);`
	Append new element having value newVal.
		`// playersList initially 55, 99, 44 (size is 3)`
		`playersList.push_back(77); // Appends new element 77`
		`// playersList is now 55, 99, 44, 77 (size is 4)`

back()	`int back();`
	Returns value of vector's last element. Vector is unchanged.
		`// playersList initially 55, 99, 44`
		`cout << playersList.back(); // Prints 44`
		`// playersList is still 55, 99, 44`

pop_back()	`void pop_back();`
	Removes the last element.
		`// playersList is 55, 99, 44 (size 3)`
		`playersList.pop_back(); // Removes last element`
		`// playersList now 55, 99 (size 2)`
		
		`cout << playersList.back(); // Common combination of back()`
		`playersList.pop_back();     // followed by pop_back()`
		`// Prints 99. playersList becomes just 55`
		
		`cout << playersList.pop_back(); // Common error:`
		`                                // pop_back() returns void`

Shown for vector<int>, but applies to other types.

example: vector_push_back.cpp

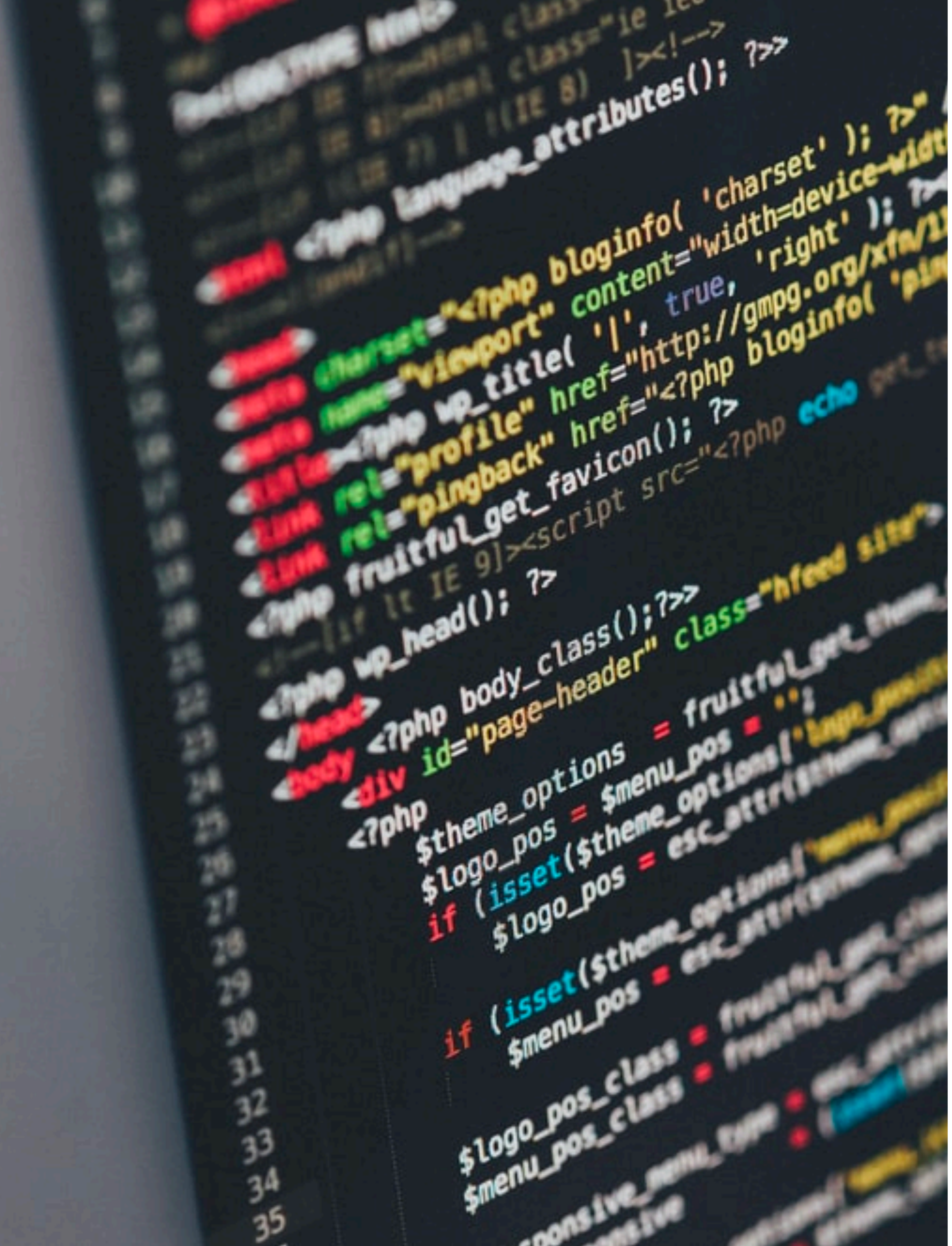
Iterating through vectors

Iterating through a vector is the same as iterating through an array, except that with vectors, it knows its own size - because it is an ***object.***

```
// Iterating through myVector
for (i = 0; i < myVector.size(); ++i) {
    // Loop body accessing myVector.at(i)
}
```

We can also use what's called a ***range-based for loop*** which only works with vectors and other container classes that support iterators (more on this later)

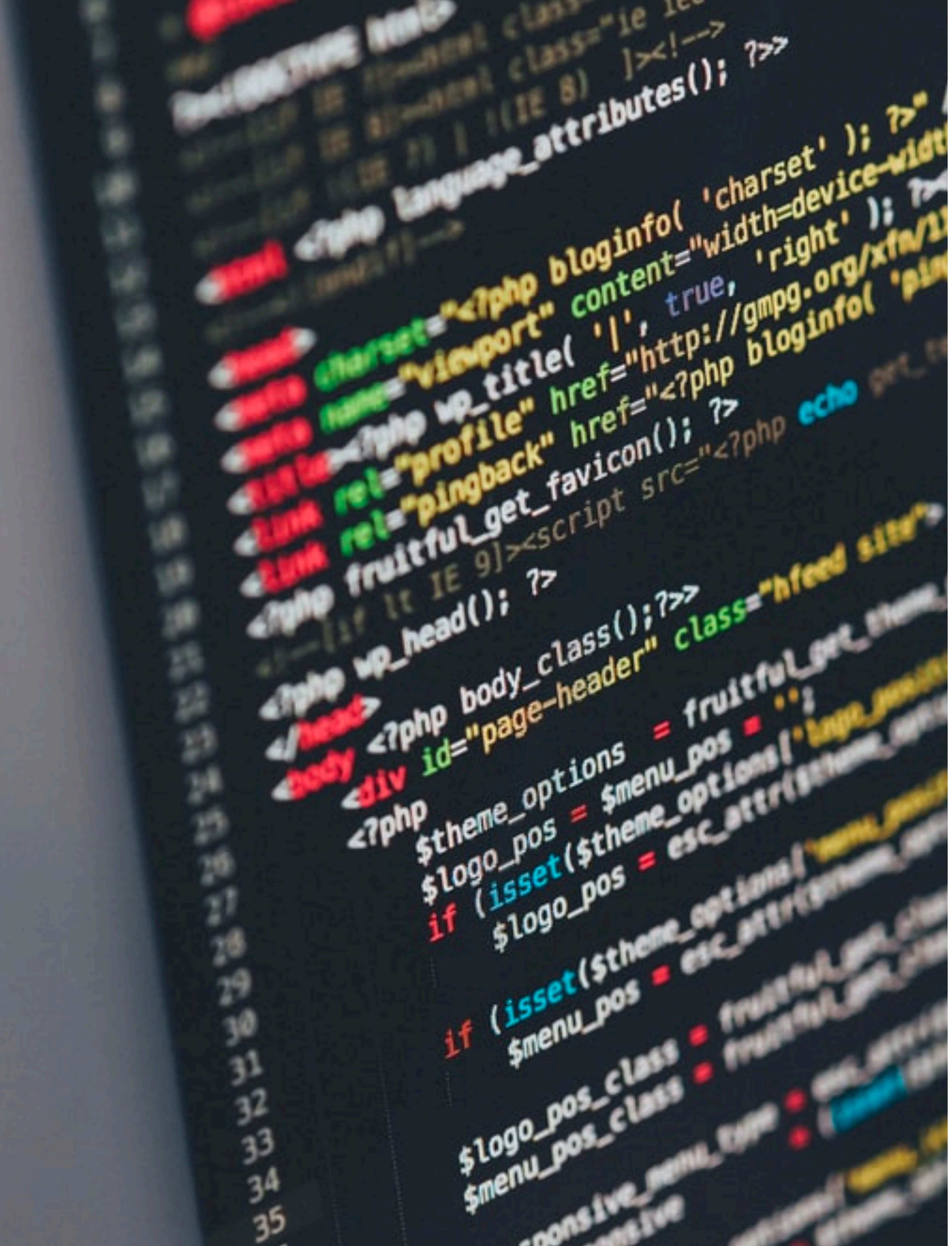
example: findMax.cpp



Normalize a vector

Write a function, that normalizes a vector by finding the minimum value, and subtracting the min from all other values so that the new minimum is 0.

examples: normalize_vector.cpp



Reverse a vector

Write a function that reverses a vector (using pass by reference) by using swaps

examples: vector_reverse.cpp