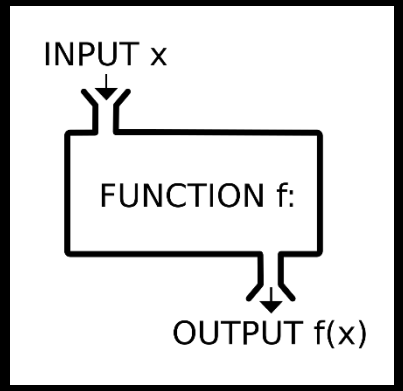
Functions, Arrays, Vectors

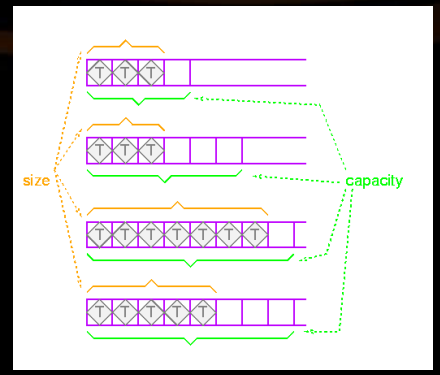
Creating and Using Functions, Arrays in C++, STL Vectors

SoftUni Foundation

Georgi Georgiev

A guy that knows C++





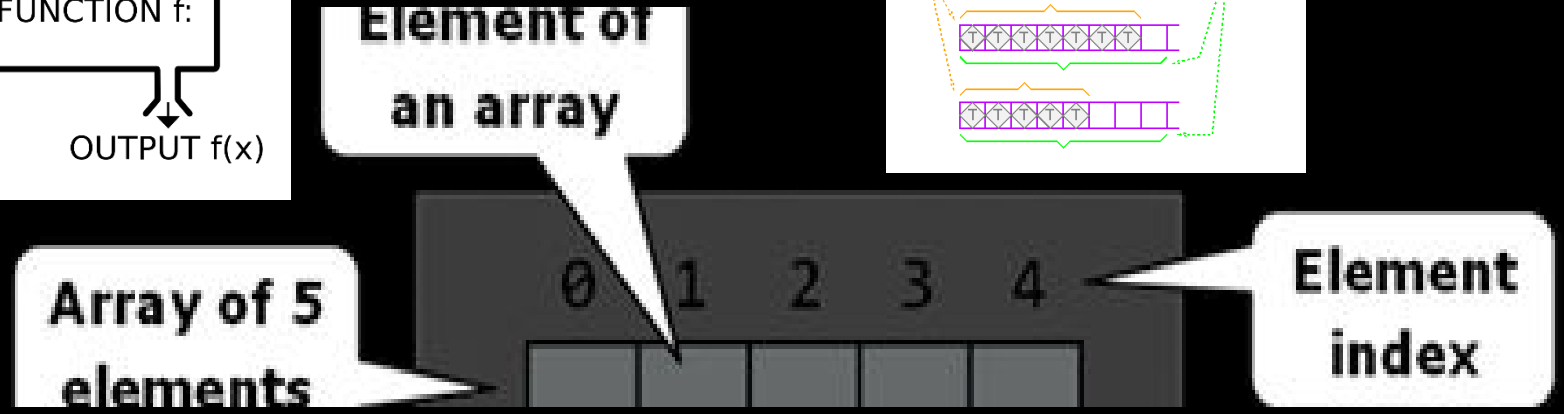
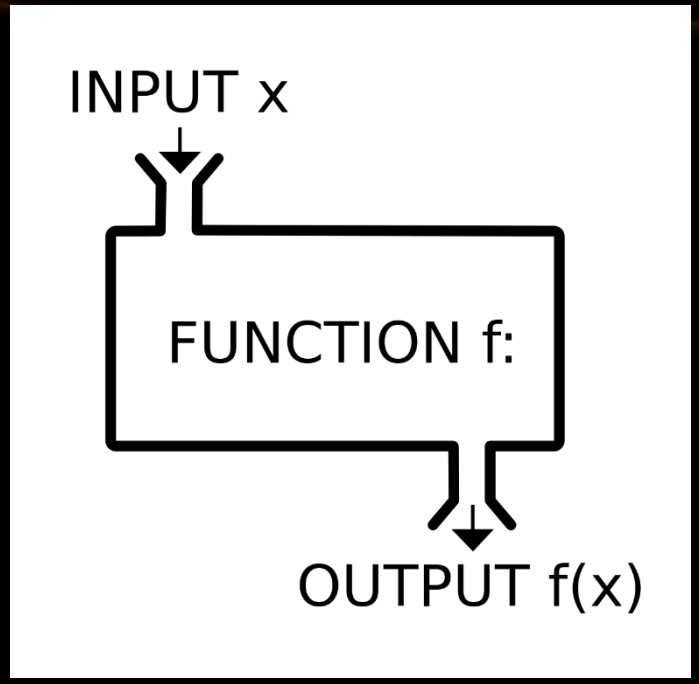


Table of Contents

SoftUni Foundation

1. Functions – Calling, Declaring, Defining
2. C++ Arrays
   - Arrays in Programming
   - Declaring & Initializing
   - Usage with Functions
3. STL Vectors
   - Dynamic-size arrays
   - Using `std::vector`

SoftUni Foundation



Functions

Calling, Defining, Implementing, Overloads

What is a Function

SoftUni Foundation

- A named block that performs a specific task
  - Can be called from other code
  - Can take parameters and return a value
- Similar to math functions like `cos(α)`, `sin(α)`, `tan(α)`, etc.
  - All of these are actually functions you can use in C++
- Also known as methods, procedures, subroutines
- `main()` is a function

Calling Functions

SoftUni Foundation

- Using functions is almost like using variables, however:
  - You write **()** after them, which could contain parameters
- Most functions return a value – you can use it in an expression
  - Just like you would use a variable's value
  - **void** functions don't have values, they just do something

```
#include<iostream>
#include<cmath>
int main() {
    double degrees = 60;
    double radians = degrees * M_PI / 180.0;
    std::cout << sin(radians) << std::endl;
    std::cout << cos(radians) << std::endl;
}
```


SoftUni
Foundation
Calling Functions
LIVE DEMO

Declaring and Defining Functions

SoftUni Foundation

- Declare the function return type, its name and parameters
- Define the body
  - always in a block
  - contains the actual function code
- Call them same as you call other functions

```
#include<iostream>
int getMax(int a, int b) {
    int maxValue;
    if (a > b) {
        maxValue = a;
    } else {
        maxValue = b;
    }

    return maxValue;
}

int main() {
    std::cout << getMax(5, 7) << std::endl;
    return 0;
}
```


SoftUni
Foundation
Declaring and Defining Functions
LIVE DEMO

Function Declaration

SoftUni Foundation

- Declaration – function's name, return type and parameters
  - Can be separate from definition (which includes the code block)
- Syntax: `returnType functionName(parameters);`
- Parameters: empty, single, or several separated by `,` (comma)
  - Syntax: `dataType [parameterName [= defaultValue]]`

```
int getMax(int a, int b);
```

```
int getMax(int, int);
```

```
void printAlphabet(char start = 'a', char end = 'a');
```

Function Declaration vs. Definition

SoftUni Foundation

- Declaration – tells the compiler there is such a function
  - Can be anywhere
  - Can appear multiple times
  - Same visibility rules as for variables
- Definition – tells the compiler what the function does
- Can be declared but not defined – compilation error if called

```
#include<iostream>
int getMax(int, int);

int main() {
    std::cout << getMax(5, 7);
    return 0;
}

int getMax(int a, int b) {
    int maxValue;
    if (a > b) { maxValue = a;}
    else { maxValue = b; }
    return maxValue;
}
```


SoftUni
Foundation
Declaring vs. Defining Functions
LIVE DEMO

Parameters & Default Values

SoftUni Foundation

- Parameters are just variables living in the function's block
- Parameters with default values can be omitted by the caller
  - If omitted are initialized with the default value
  - Must be last in the parameter list

```
void printAlphabet(char first = 'a', char last = 'z') {
    for (char i = first; i <= last; i++) { cout << i; }
    cout << endl;
}
int main() {
    printAlphabet(); printAlphabet('p'); printAlphabet('p', 'x');
    return 0;
}
```

Passing By Value vs. Passing By Reference

SoftUni Foundation

- Parameters are normally copies of their originals
  - Changing them does NOT change the caller's variables
  - "Passing by value"
- To access the caller's variables directly, use references
  - Syntax: **`dataType& param`**
  - "Passing by reference"

```
int square(int num) {
    num = num * num;
    return num;
}
void swap(int& a, int& b) {
    int oldA = a; a = b; b = oldA;
}
int main() {
    int x = 5;
    std::cout << square(x); //25
    std::cout << x; //5
    int y = 42;
    swap(x, y);
    std::cout << x; //42
    return 0;
}
```

Returning Values from Functions

SoftUni Foundation

- The return keyword determines the function's return value
- **return** immediately exits the function
  - Non-**void** functions must have a **return** followed by a value

```
int getMax(int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}
```


SoftUni
Foundation
Parameters and Returning Values
LIVE DEMO

Overloaded Functions

- Same function name and return type, different parameter list
  - Different number or types of parameters

```
int getMax(int a, int b) {
    if (a > b) {
        return a;
    }
    return b;
}

int getMax(int a, int b, int c) {
    return getMax(a, getMax(b, c));
}
```


SoftUni
Foundation
Overloaded Functions
LIVE DEMO

static Variables Inside Functions

SoftUni Foundation

- static variables live through entire program, initialized once
- static variables can be used inside functions to track state
  - E.g. how many times a function was called

```
double movingAverage(int nextNumber) {
    static int count = 0;
    static int sum = 0;
    sum += nextNumber;
    count++;
    return sum / (double)count;
}
```


SoftUni
Foundation
Static Variables Inside Functions
LIVE DEMO

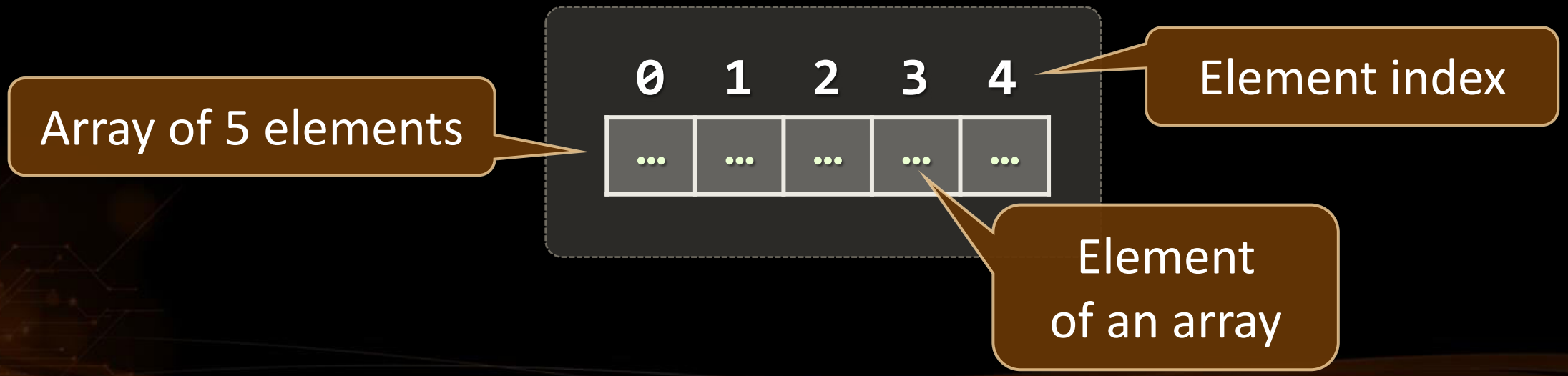
SoftUni Foundation

Arrays

Multiple Values Inside One Variable

What are Arrays?

- In programming, an array is a sequence of elements
  - Elements are numbered from `0` to `Length-1`
  - Elements are of the same type (e.g. integers)
  - Arrays have fixed size – cannot be resized



Creating C++ Arrays

SoftUni Foundation

- Declaring: `dataType identifier [arraySize];`
- C++ arrays have some special initialization syntax
  - `dataType identifier[N] = {elem0, elem1, ..., elemN-1}`
  - There can be less than `N` elements, but not more
  - `N` can be omitted – number of elements assumed as size
- Example: `int fibonacci[5] = {1, 1, 2, 3, 5};`

Index	*0*	*1*	*2*	*3*	*4*
Value	1	1	2	3	5

C++ Array Declaration & Initialization

SoftUni Foundation

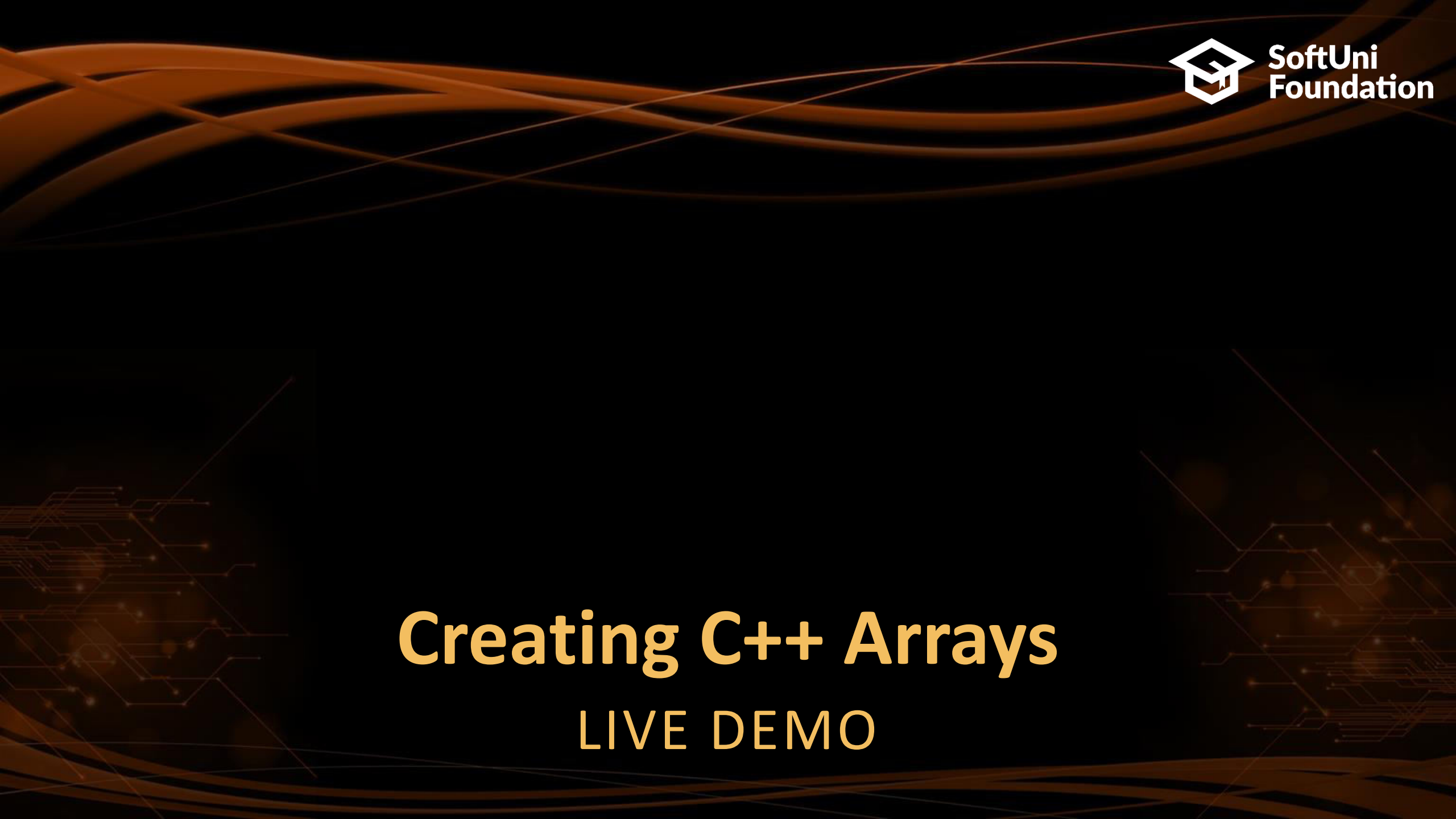
- When size is known compile-time:
  - {} available to initialize elements
  - If {} has less elements than the array, remaining are set to default

```
double arrayWithDefaults[3] = {3.14};
```

- {} initialization is available for non-compile-time sizes in C++11

```
int arrSize;
cin >> arrSize;
double arr[arrSize]{};
```

- Other initialization rules are the same as for primitives (e.g. statics/globals are default initialized)


SoftUni
Foundation
Creating C++ Arrays
LIVE DEMO

Accessing Array Elements

SoftUni Foundation

- The indexing operator `[]` gives access to any array element
  - `array[index] = value; arrayType value = arrayName[index]`
  - E.g. to access the first element of `arr`, do `arr[0]`
- Once you access the element, treat it as a normal variable

```
double point2d[2] = {};

std::cin >> point2d[0] >> point2d[1];

double distToCenter = sqrt(point2d[0] * point2d[0] + point2d[1] * point2d[1]);
point2d[0] /= distToCenter;
point2d[1] /= distToCenter;

std::cout << point2d[0] << " " << point2d[1] << std::endl;
```


SoftUni
Foundation
Accessing Array Elements
LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- You are John Snow. What will the following two code lines do?

```
double arr[2] = {};
arr[3] = 42;
```

a) cause a compile-time error

b) cause a runtime error due to index being out of bounds

c) summon demons

d) you know nothing

C++ PITFALL: ARRAY OUT-OF-BOUNDS ACCESS

The C++ standard doesn't define out-of-bounds array access behavior. C++ arrays don't store information on their length.

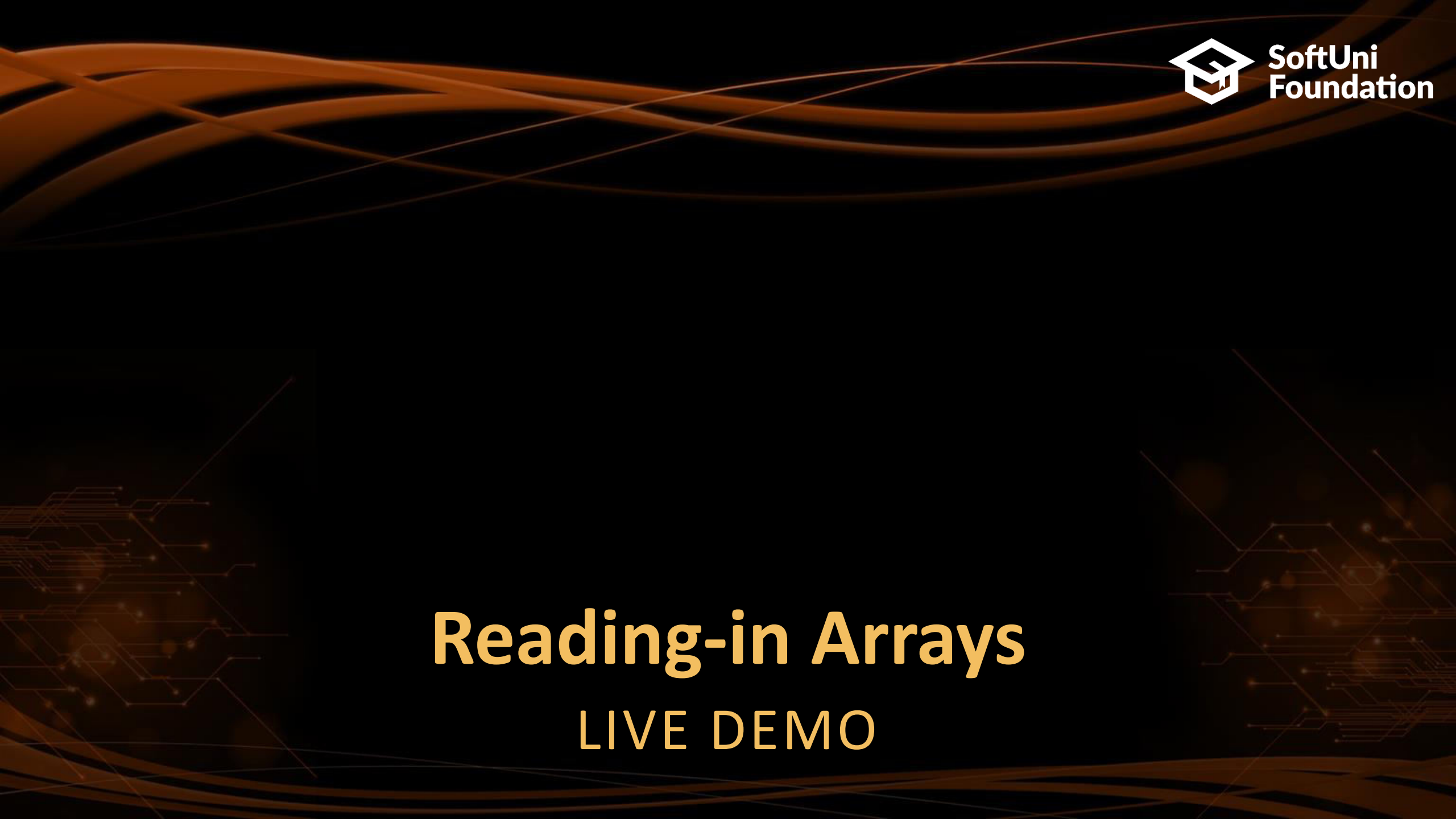
Program will usually attempt to access memory even if out of the array. If that part of memory is accessible to the program, it will execute whatever it is told. Otherwise an error might happen (0xC00005 on Windows)

Reading-in an Array

SoftUni Foundation

- Arrays are often read-in from some input, instead of initialized
- That's the point of arrays – to store arbitrary amounts of data
- Common approach: run a loop to read in a number of elements
  - Example: read-in a specified number of elements from console

```
int main() {
    int actualCount; cin >> actualCount;
    int numbers[actualCount];
    for (int i = 0; i < actualCount; i++) {
        cin >> numbers[i];
    }
    return 0;
}
```

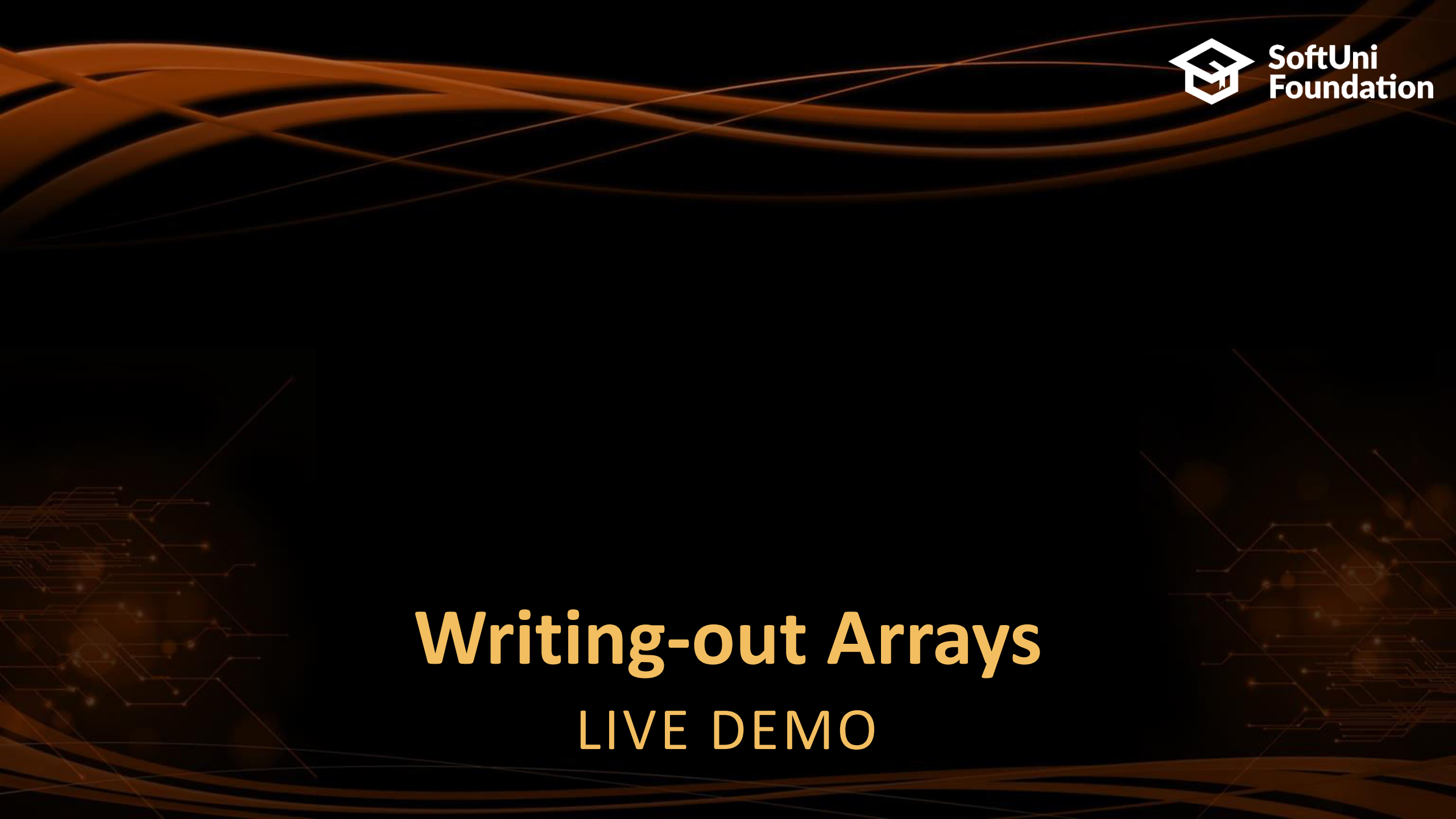

SoftUni
Foundation
Reading-in Arrays
LIVE DEMO

Writing-out an Array

SoftUni Foundation

- You will commonly need to display all elements of an array
- Common approach: loop over the elements, print each
- Note: need to know how long the array is – keep a variable

```
int numRoots = 100;
double squareRoots[numRoots] = {};
for (int i = 0; i < numRoots; i++) {
    squareRoots[i] = sqrt(i);
}
for (int i = 0; i < numRoots; i++) {
    cout << squareRoots[i] << endl;
}
```


SoftUni
Foundation
Writing-out Arrays
LIVE DEMO

Arrays as Function Parameters

SoftUni Foundation

- Array parameters are declared the same way arrays are declared
  - `dataType name[size_0]`, additional `[size_i]` per dimension
  - First dimension size can be omitted

```
void print(int a[], int n) {
  for (int i = 0; i < n; i++) {
    cout << a[i] << " ";
  }
  cout << endl;
}
```

```
int main() {
  int numbers[] = {1,2,3};
  print(numbers, 3);

  return 0;
}
```

Arrays & Functions – Specifics

SoftUni Foundation

- Functions work on the ORIGINAL array the caller uses
  - If the function changes an element, the caller's array is modified
  - You can imagine array elements are passed by reference
- Functions CAN'T return C++ “static” arrays created in them
  - Arrays are essentially memory addresses
  - The memory they point to is freed when the function exits
  - We will later discuss other ways to return sequences of elements

C++11 Range-Based for Loop

SoftUni Foundation

- Tired of writing for loops with indices to iterate over an array?
- C++11 added a loop for that use-case
- Syntax (for arrays): **`for (dataType element : array)`**
  - Body will execute once for each element in the array
  - On each iteration, **`element`** will be the next item in the array

```
int numbers[] = {13, 42, 69};
for (int number : numbers) {
    cout << number << endl;
}
```

C++11 <array> Header

SoftUni Foundation

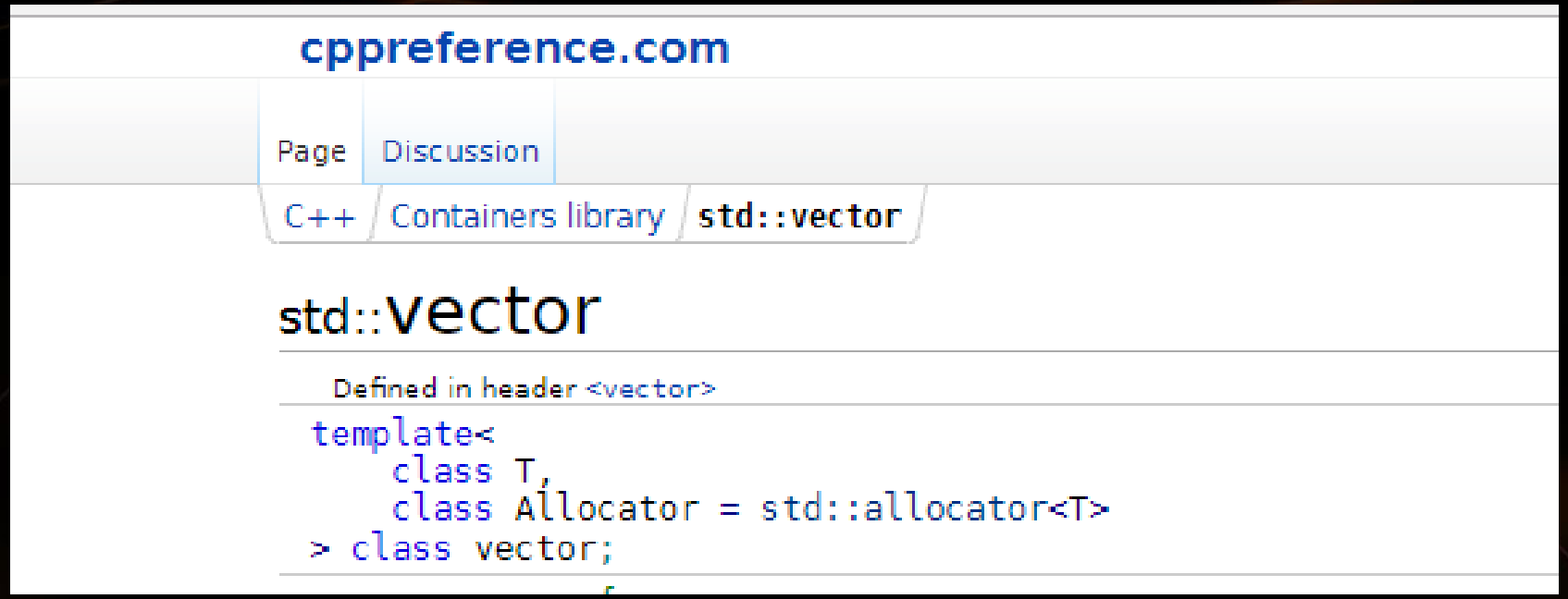
- C++11 provides an alternative array, which is a bit smarter
- The `array` class knows its size, can be returned from functions
  - `#include<array>`
  - Declaring: `array<int, 5> arr;` is the same as `int arr[5];`
  - Declaring & Initializing:
    ```
    array<int, 5> arr = {1, 2, 3, 4, 5};
    ```
  - `arr.size()` gives you the size of the array
  - Accessing elements: use the `[]` operator like with normal arrays

SoftUni Foundation

C++11 Range-Based for Loop and array Class

LIVE DEMO

SoftUni Foundation



STL Vectors

C++ Dynamically-Sized Arrays

STL Vector Basics

SoftUni Foundation

- The C++ `std::vector` class is a resizable array
  - Has normal array-like access – `[]` operator
  - Knows it's size (`.size()`), can add elements (`.push_back()`)
- `#include<vector>`
- Acts like a normal variable
  - Can be assigned like a normal variable
  - Can be passed to a function as a value or as a reference
  - Can be returned from a function

Initializing a Vector

SoftUni Foundation

- Declaration Syntax: `std::vector<T> name;`
- The vector is initially empty – items need to be added
  - Call `push_back(T element)` on the vector to add elements

```
std::vector<double> squareRoots;
for (int i = 0; i < 100; i++) {
    squareRoots.push_back(sqrt(i));
}
```

- Can be initialized directly in C++11 with `{}` syntax
  - `std::vector<int> numbers {13, 42, 69};`


SoftUni
Foundation
Initializing STL Vectors
LIVE DEMO

STL Vectors as Function Parameters

SoftUni Foundation

- Vectors parameters are like normal variable parameters
  - Passed by value (copy). Can be passed by reference with &

```
void print(std::vector<int> nums) {
  for (int i = 0; i < nums.size(); i++)
    cout << numbers[i] <<" ";
  cout << endl;
}
void printMultiplied(vector<int> nums,
                     int multiplier) {
  for (int i = 0; i < nums.size(); i++)
    nums[i] *= multiplier;
  print(nums);
}
```

```
int main() {
  vector<int> nums {1, 2, 3};
  printMultiplied(nums, 10);
  // 10, 20, 30
  print(nums);
  // 1, 2, 3
  return 0;
}
```

SoftUni Foundation

STL Vectors as Function Parameters

LIVE DEMO

Returning STL Vectors from Functions

SoftUni Foundation

- Vectors act as normal variables when returned
  - Function returns a copy (C++11 optimizes by "moving" memory)

```
vector<double> getSquareRoots(int from, int to) {
    vector<double> roots;
    for (int i = from; i <= to; i++) {
        roots.push_back(sqrt(i));
    }
    return roots;
}
int main() {
    print(getSquareRoots(4, 25));
    return 0;
}
```

```
void print(vector<double> numbers) {
    for (int number : numbers) {
        cout << number << " "
    }
    cout << endl;
}
```

SoftUni Foundation

Returning STL Vectors from Functions

LIVE DEMO

Summary

- Functions are named, reusable blocks of code
  - Can accept parameters and return values
- Arrays are sequences of numbered elements
  - C++ "static" arrays are a language-feature
  - But lack some basic abilities (knowing size and returning from functions)
- STL Vectors are objects that represent arrays which can resize
  - Act as normal variables when passed to/returned from functions
  - Can mostly be used instead of arrays for basic tasks

Functions, Arrays, Vectors

SoftUni Foundation

Questions?