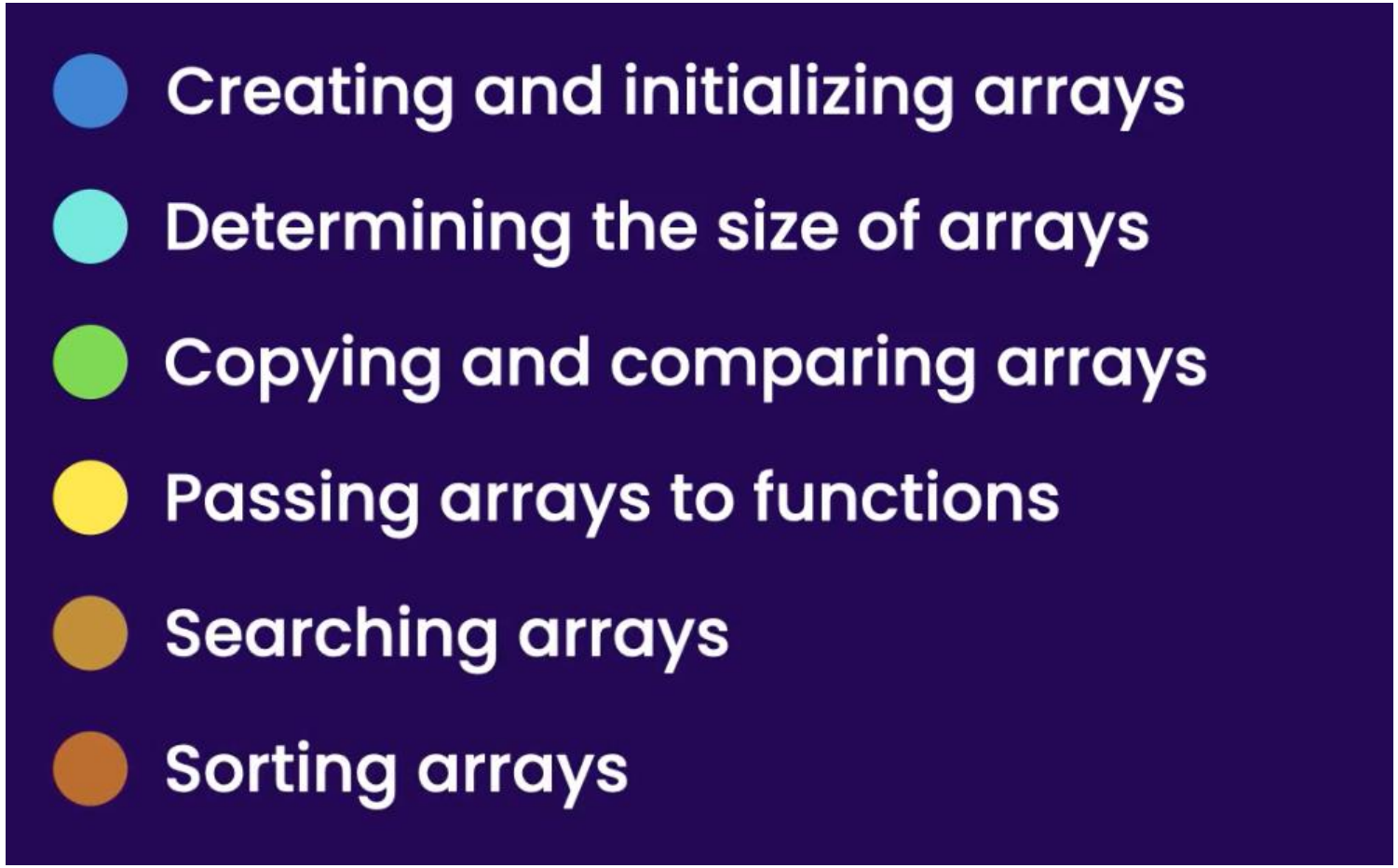
C++ Notes

Part 2-1

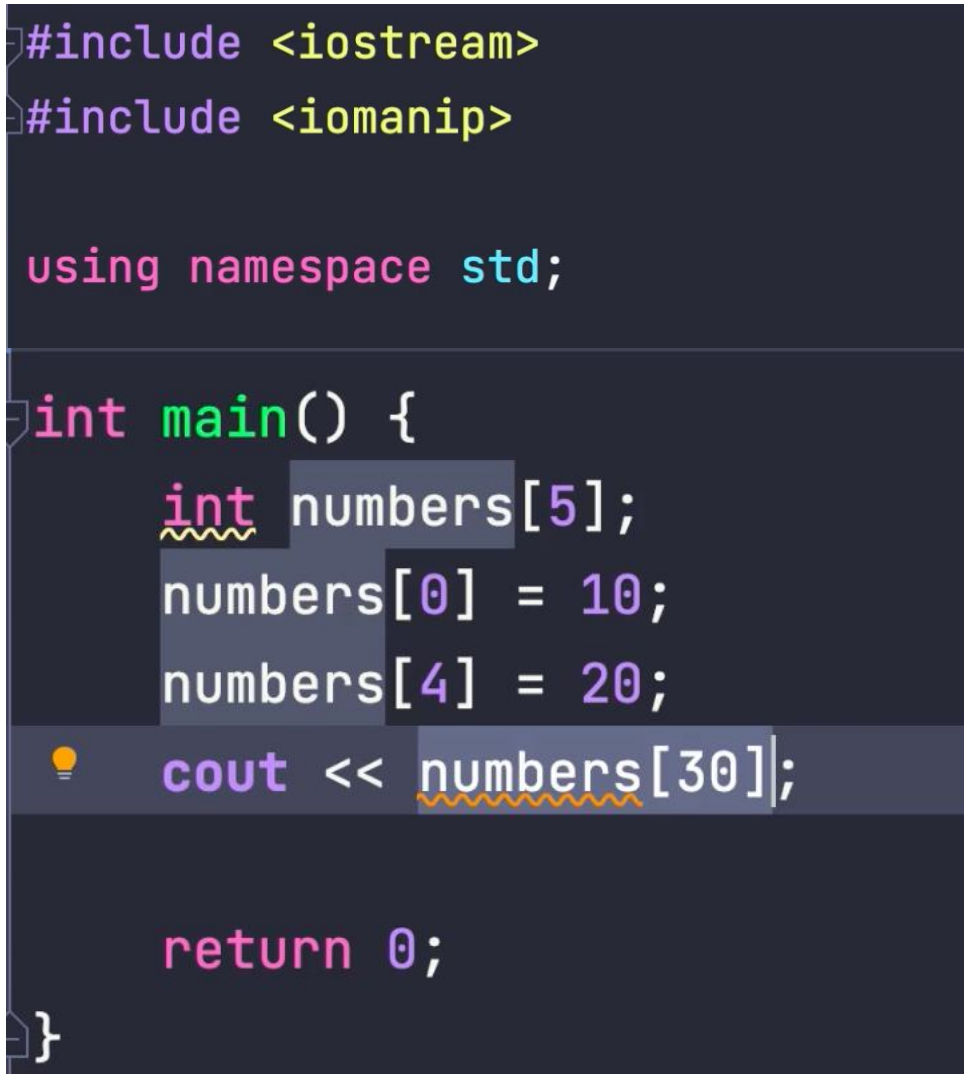
Slides were created from CodewithMosh.com

Topics Covered

Arrays


Creating and initializing arrays
Determining the size of arrays
Copying and comparing arrays
Passing arrays to functions
Searching arrays
Sorting arrays

Creating and Initializing Arrays



```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    int numbers[5];
    numbers[0] = 10;
    numbers[4] = 20;
    cout << numbers[30];

    return 0;
}
```

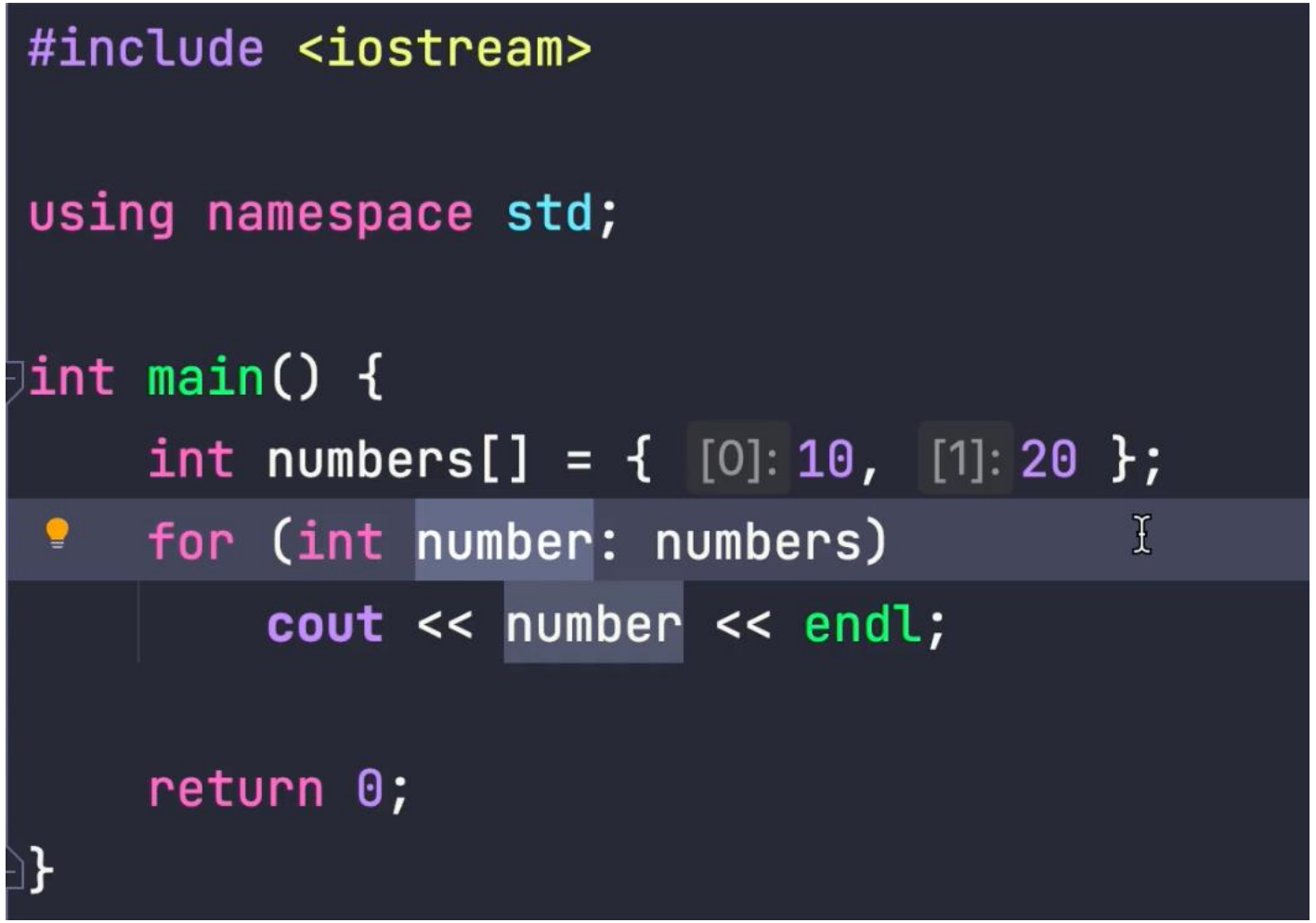
- Array indices start from 0 (zero).
- Assigning a value to an index outside of the index range will not cause any compile issue, but it will produce a garbage value.

Creating and Initializing Arrays

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20 };

    cout << numbers;

    return 0;
}
```

- We do not need to specify the array size if we initialize the array with values.
- The compiler determines the size based on the values provided.

Determining the Size of Arrays



```cpp
#include <iostream>

using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20 };
    for (int number: numbers)
        cout << number << endl;

    return 0;
}
```

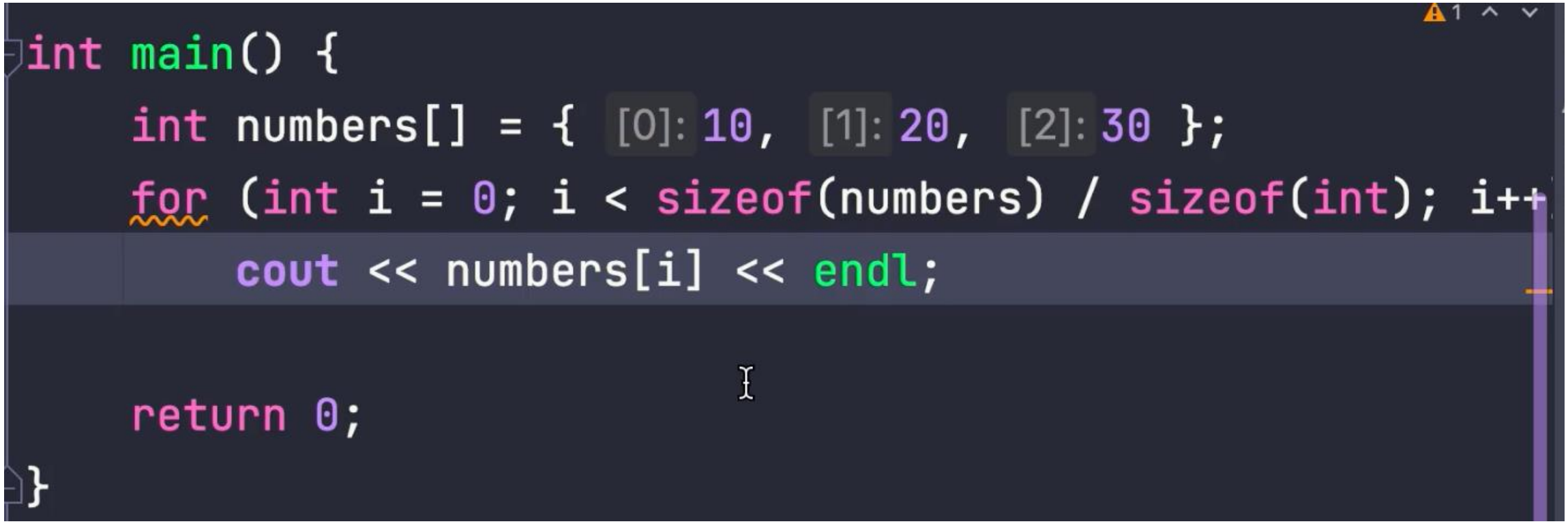
- Range-based for loop.
- The variable number will be assigned the array elements one-by-one.

Determining the Size of Arrays

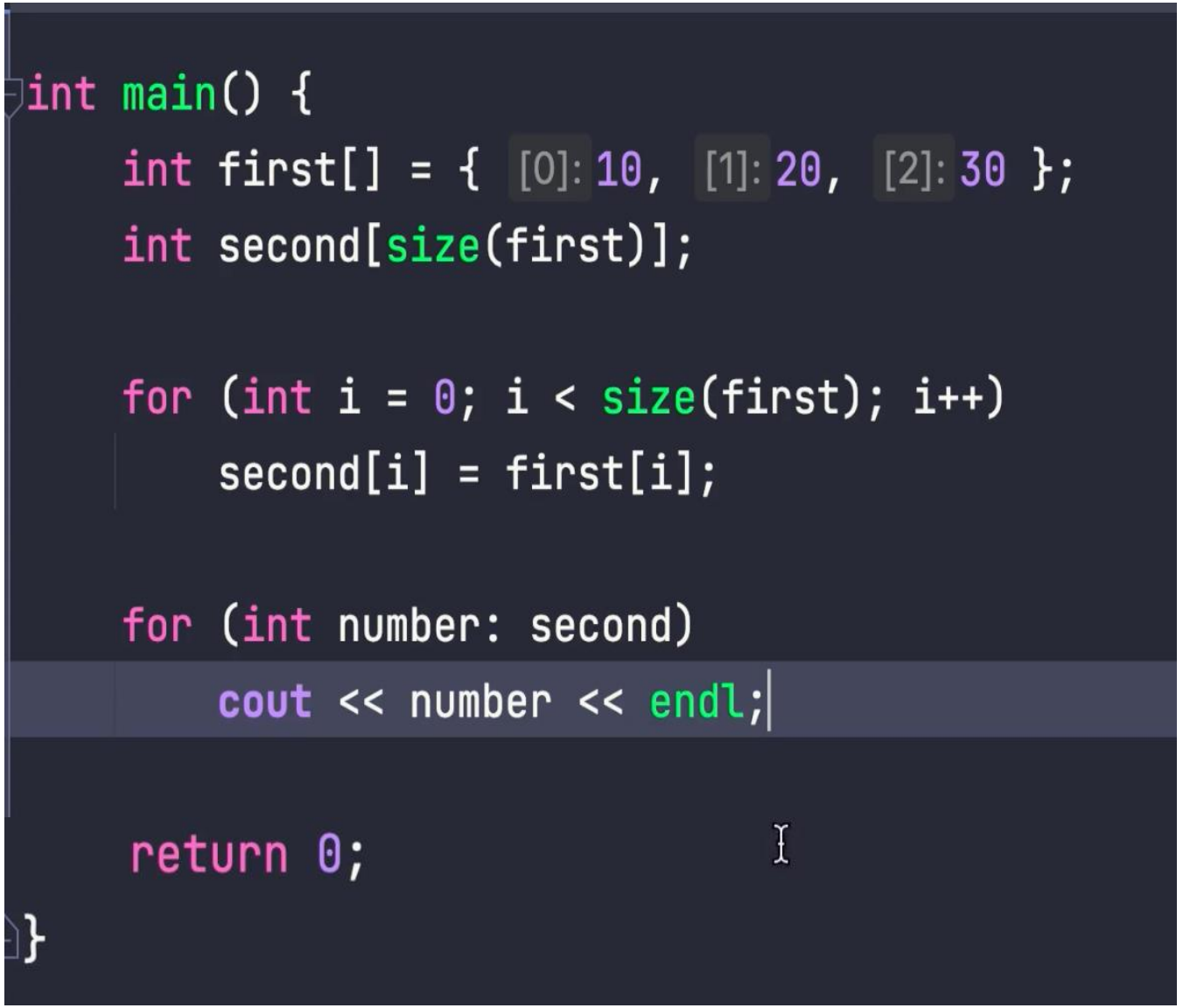
```cpp
#include <iostream>

using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20 };
    for (auto number : int : numbers)
        cout << number << endl;

    return 0;
}
```

- Using **auto** in the for loop, the compiler determines the type of the variable “number” from the values in the array.

Determining the Size of Arrays



Copying Arrays



Determining the Size of Arrays

```cpp
using namespace std;

int main() {
    int first[] = { [0]: 10, [1]: 20, [2]: 30 };
    int second[] = { [0]: 10, [1]: 20, [2]: 30 };

    if (first == second)
        cout << "Equal";

    return 0;
}
```

- ❑ We cannot compare arrays using their names directly.
- ❑ We need to compare each element individually.
- ❑ Array names store their address in the memory.

Determining the Size of Arrays

```cpp
int main() {
    int first[] = { [0]: 10, [1]: 20, [2]: 30 };
    int second[] = { [0]: 10, [1]: 20, [2]: 30 };

    cout << first << endl;
    cout << second << endl;

    return 0;
}
```

- We cannot compare arrays using their names directly.
- We need to compare each element individually.
- Array names store their address in the memory.

```
0x7ffee74a18fc
0x7ffee74a18f0
```

Array addresses in the memory

Determining the Size of Arrays

```cpp
int main() {
    int first[] = { [0]: 10, [1]: 20, [2]: 30 };
    int second[] = { [0]: 10, [1]: 20, [2]: 30 };

    bool areEqual = true;
    for (int i = 0; i < size(first); i++)
        if (first[i] != second[i]) {
            areEqual = false;
            break;
        }

    cout << boolalpha << areEqual;

    return 0;
}
```

- We cannot compare arrays using their names directly.
- We need to compare each element individually.
- Array names store their address in the memory.

Passing Arrays to Functions

```cpp
#include <iostream>

using namespace std;

void printNumbers(int numbers[]) {
    for (int number: numbers)
        cout << number;
}
```

```cpp
// int[] -> int*
void printNumbers(int numbers[]) {
    for (int i = 0; i < size(numbers); i++)
        cout << numbers[i];
}
```

- Integer array number, when passed as an argument, represents a number which is the address of the array in memory.
- It is a pointer.
- We cannot use range loop in this case.

- **size(numbers)** will not work, because numbers is just an address (hexadecimal number)

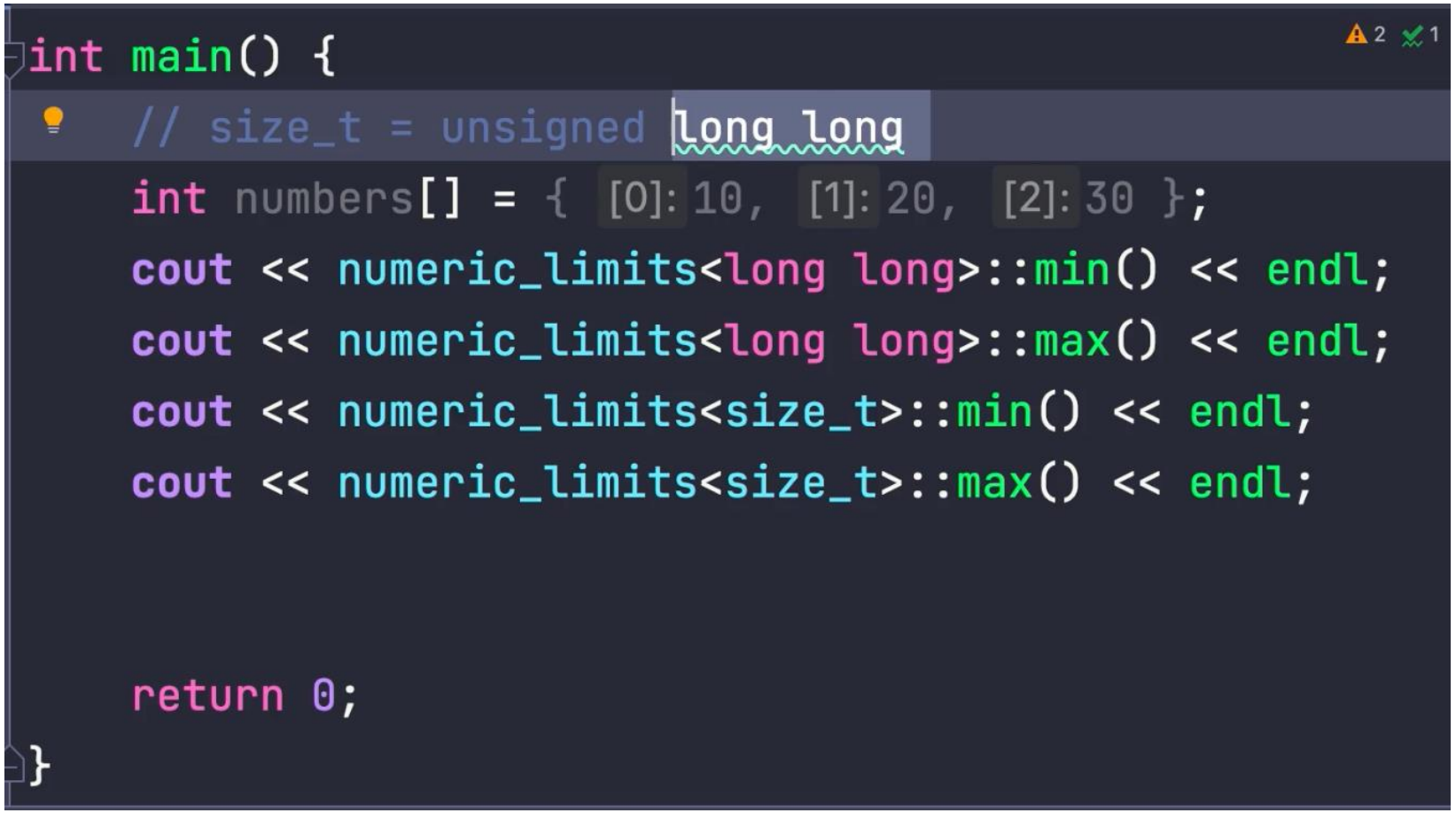
Passing Arrays to Functions

```cpp
#include <iostream>

using namespace std;

// int[] -> int*
void printNumbers(int numbers[], int size) {
    for (int i = 0; i < size; i++)
        cout << numbers[i];
}
int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    printNumbers(numbers, size: size(numbers));

    return 0;
}
```

Solution

- We need to pass the size of the array too.

size_t()



- It is another data type.
- size_t is used in C++ primarily for representing the size of objects in memory.
- It can hold the size of the largest object the system can handle.
- It is an unsigned integer type that is guaranteed to be large enough to hold the size of any object.

Unpacking Arrays

```cpp
int main() {
    int values[3] = { [0]: 10, [1]: 20, [2]: 30 };
    auto [x : int , y : int , z : int ] = values;
    cout << x << ", " << y << ", " << z;


    return 0;
}
```

Equivalent

```cpp
//     int x = values[0];
//     int y = values[1];
//     int z = values[2];
```

- ❑ C++: structured binding
- ❑ JavaScript: destructuring
- ❑ Python: unpacking

Searching Arrays (Linear Search)

```cpp
#include <iostream>

using namespace std;

int find(int numbers[], int size, int target) {
    for (int i = 0; i < size; i++)
        if (numbers[i] == target)
            return i;
    return -1;
}

int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    cout << find(numbers, size: size(numbers), target: 10);

    return 0;
}
```

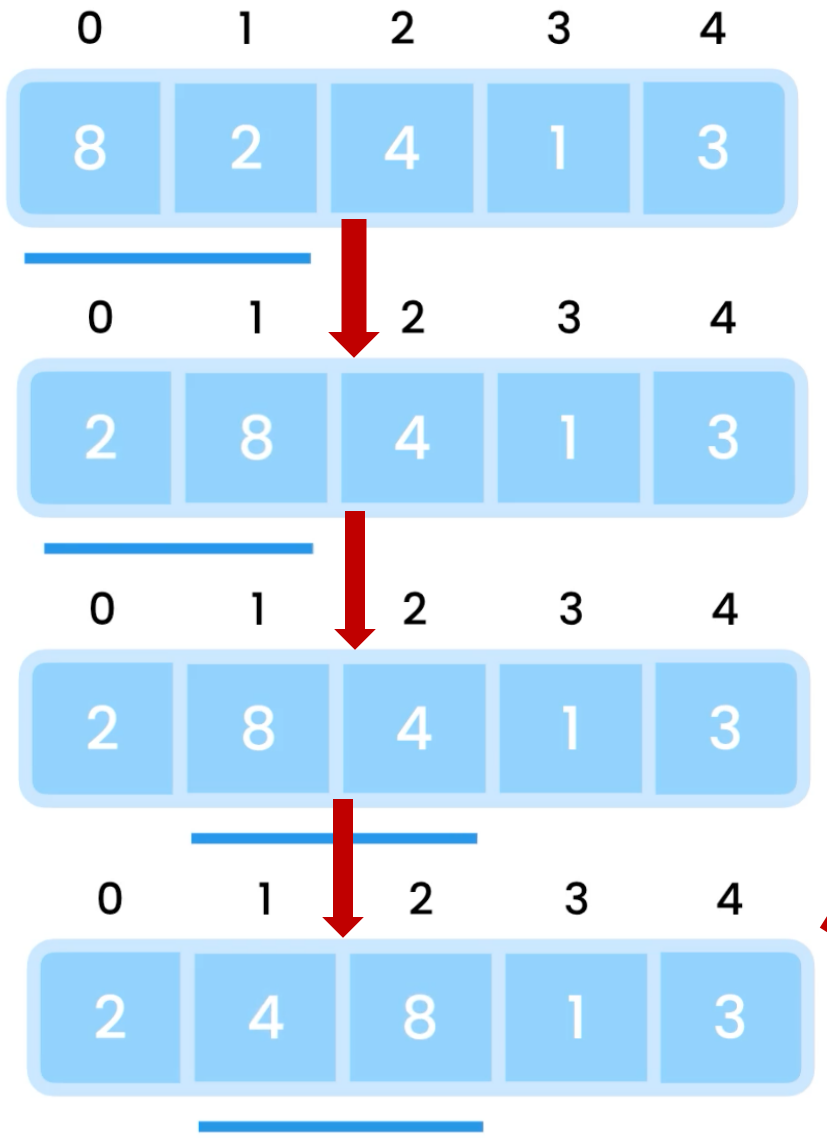
- Create a function for finding a value in an array.
- If the target value exists, return its index; otherwise return -1.

Sorting Arrays

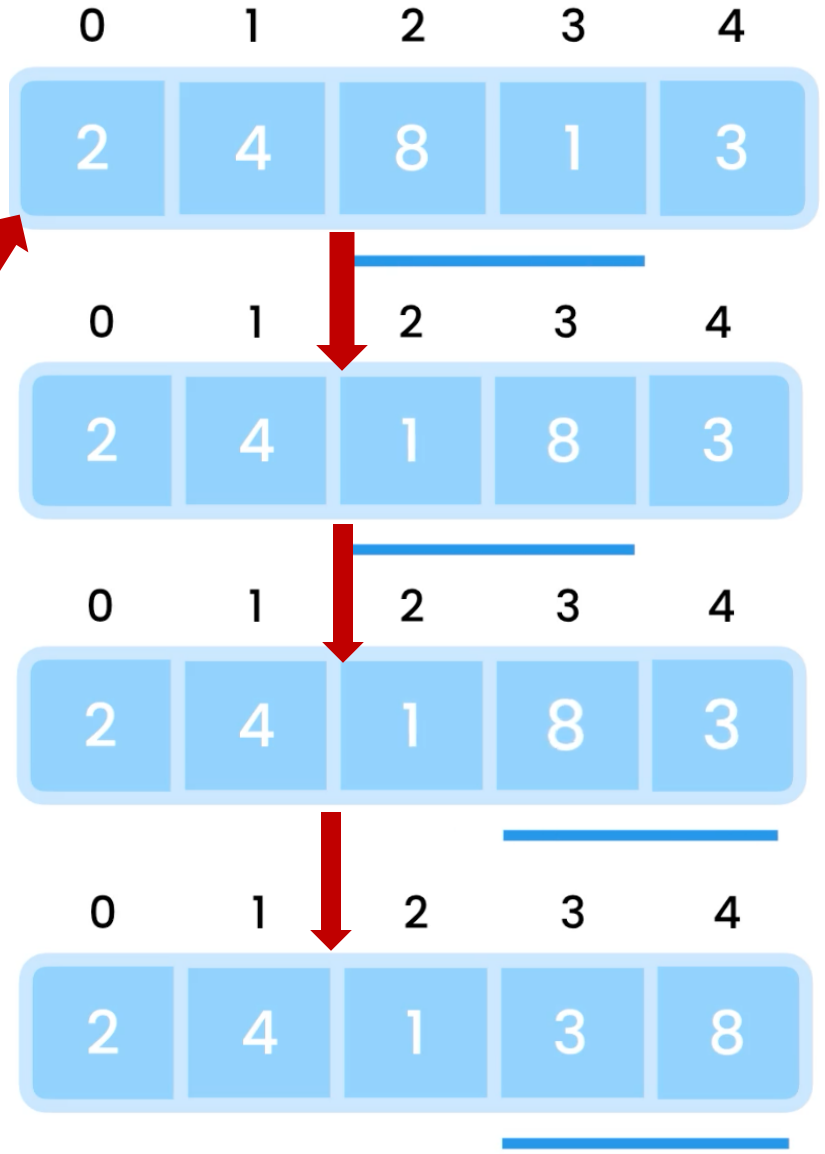
Common Algorithms

- ❑ Bubble sort
- ❑ Selection sort
- ❑ Insertion sort
- ❑ Merge sort
- ❑ Quick sort

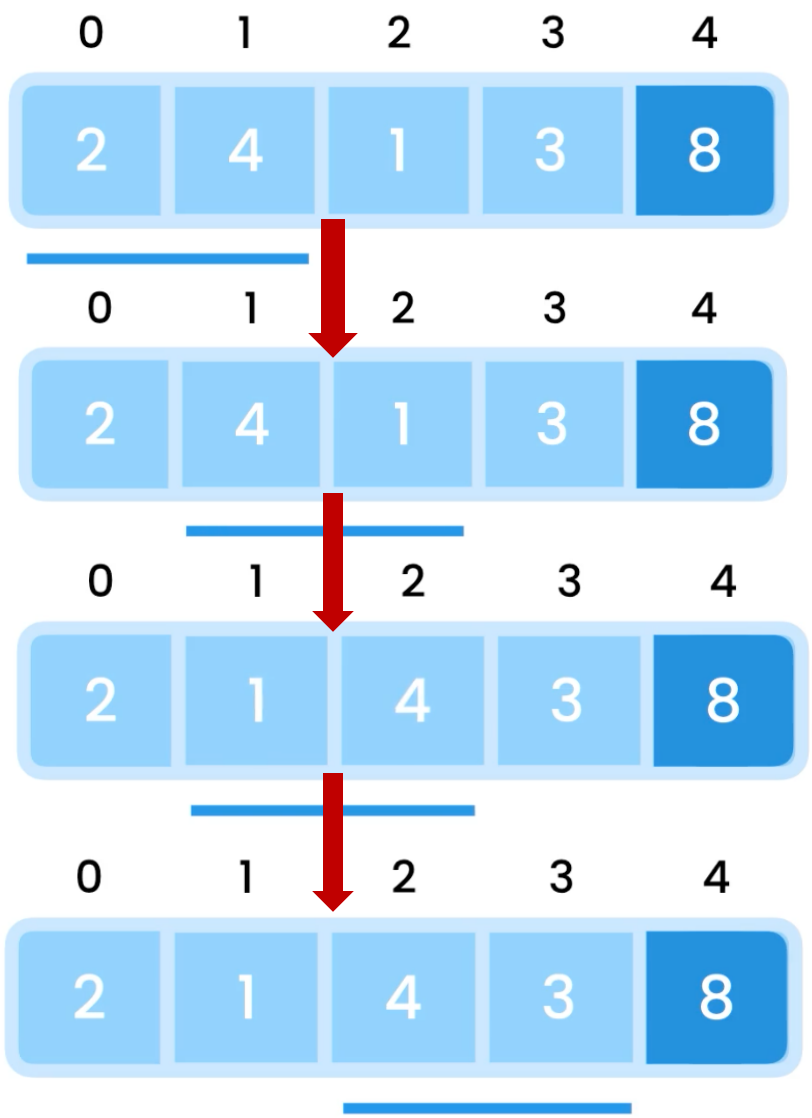
Bubble Sort



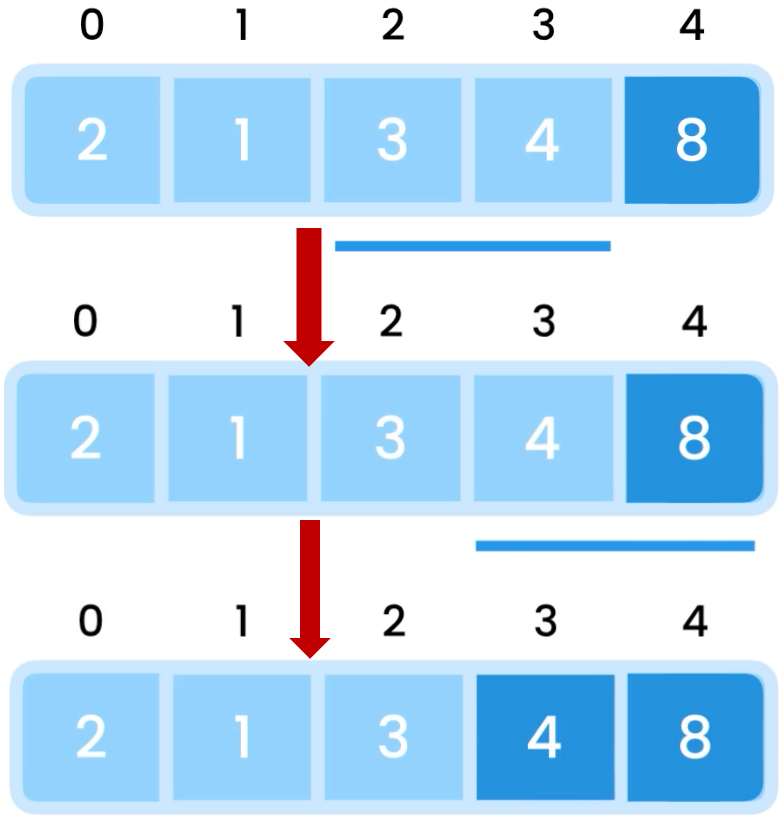
First iteration (first pass)



Bubble Sort

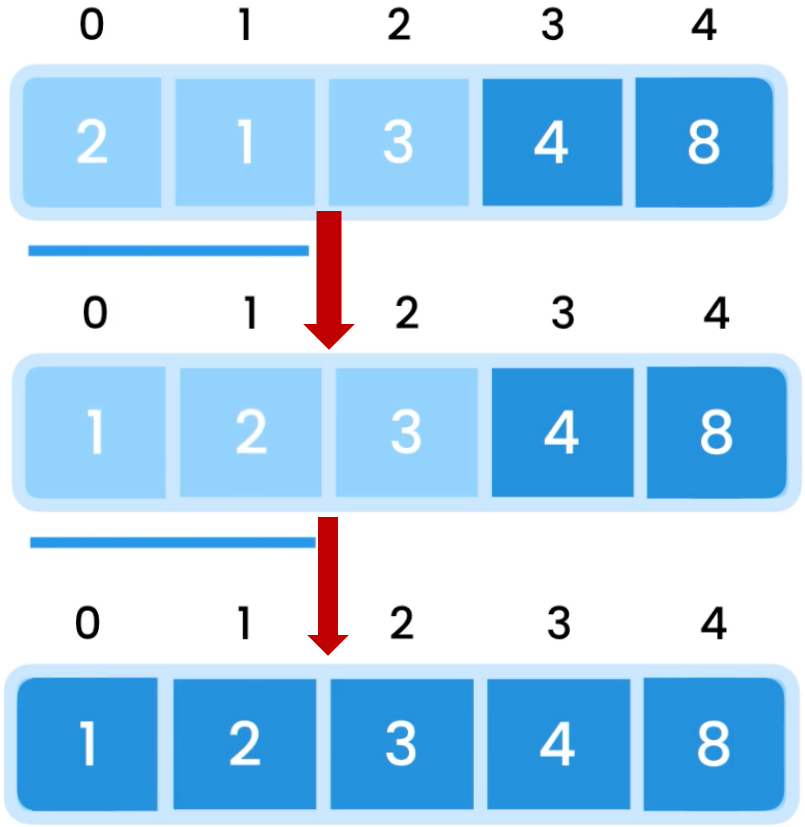


Second iteration (second pass)



Bubble Sort

Third iteration (third pass)



Bubble Sort

```cpp
#include <iostream>

using namespace std;

void swap(int numbers[], int i, int j) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}
```

```cpp
void sort(int numbers[], int size) {
    for (int pass = 0; pass < size; pass++) {
        for (int i = 1; i < size; i++)
            if (numbers[i] < numbers[i - 1])
                swap(numbers, i, j: i - 1);
    }
}
```

```cpp
int main() {
    int numbers[] = { [0]: 30, [1]: 20, [2]: 10 };
    sort(numbers, size: size(numbers));
    for (int number: numbers)
        cout << number << endl;

    return 0;
}
```

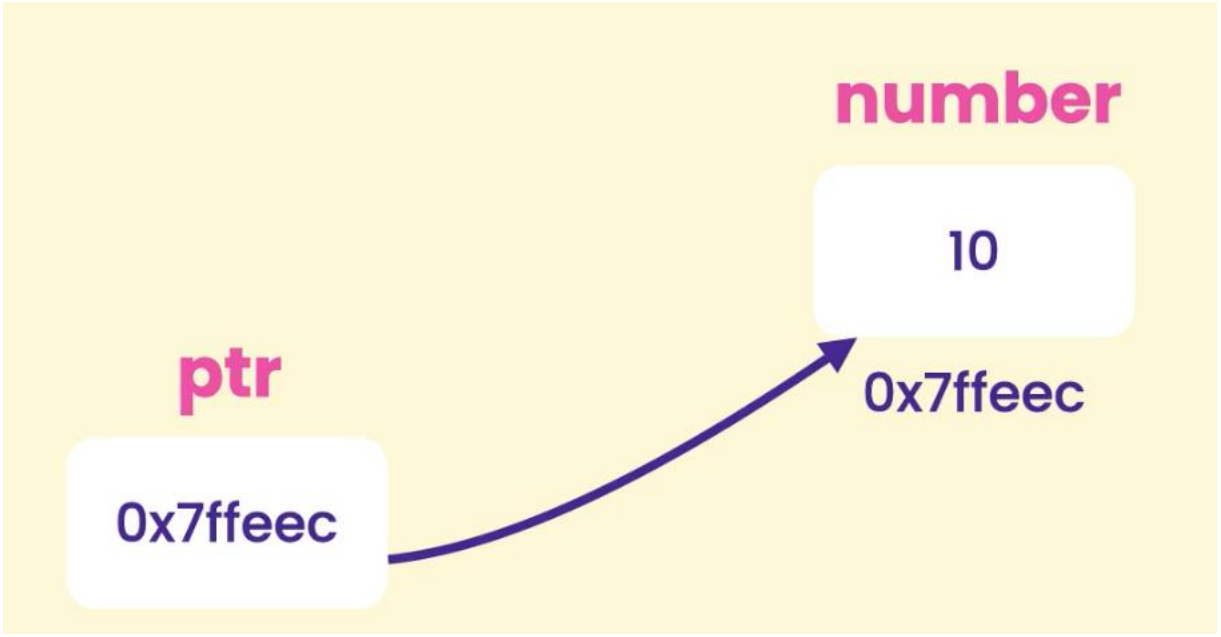
```cpp
int main() {
    const int rows = 2;
    const int columns = 3;

    // 2x3
    int matrix[rows][columns] = {
            [0]: { [0]: 11,  [1]: 12,  [2]: 13},
            [1]: { [0]: 21,  [1]: 22,  [2]: 23}
    };

    for (int row = 0; row < rows; row++) {
        for (int col = 0; col < columns; col++)
            cout << matrix[row][col] << endl;
    }

    return 0;
}
```

Multi-dimentional Arrays

```cpp
using namespace std;

const int rows = 2;
const int columns = 3;

void printMatrix(int matrix[rows][columns]) {
    for (int row = 0; row < rows; row++) {
        for (int col = 0; col < columns; col++)
            cout << matrix[row][col] << endl;
    }
}
int main() {
    // 2x3
    int matrix[rows][columns] = {
            [0]: { [0]: 11,  [1]: 12,  [2]: 13},
            [1]: { [0]: 21,  [1]: 22,  [2]: 23}
    };
    printMatrix(matrix);

    return 0;
}
```

Multi-dimensional Arrays

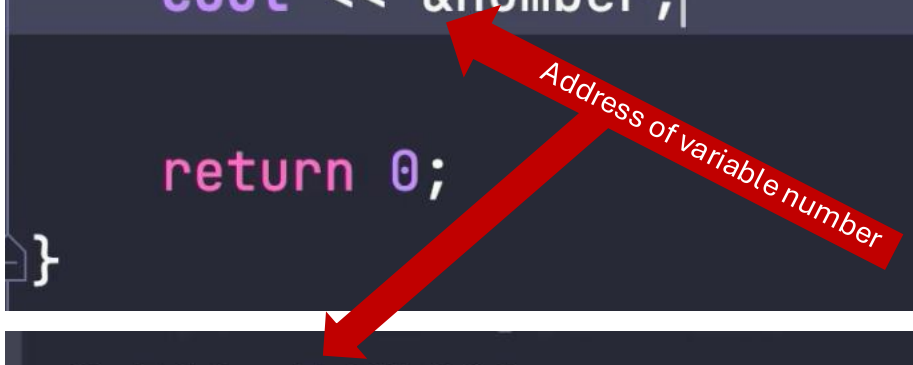
Pointers



- A pointer holds the address of another variable in memory.
- Reasons for using pointers
  - Efficiently passing large objects.
  - Dynamic memory allocation.
  - Enabling polymorphism.

Declaring and Using Pointers

```cpp
#include <iostream>

using namespace std;

int main() {
    int number = 10;
    cout << &number;

    return 0;
}
```

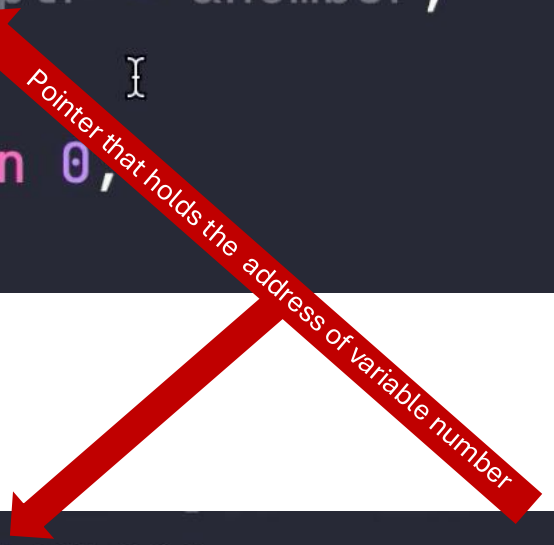



```
0x7ffee3d1b908
Process finished with exit code 0
```

- A pointer holds the address of another variable in memory.
- Output of the cout statement.

Declaring and Using Pointers

```
int main() {
    int number = 10;
    cout << &number;

    int* ptr = &number;

    return 0;
}
```



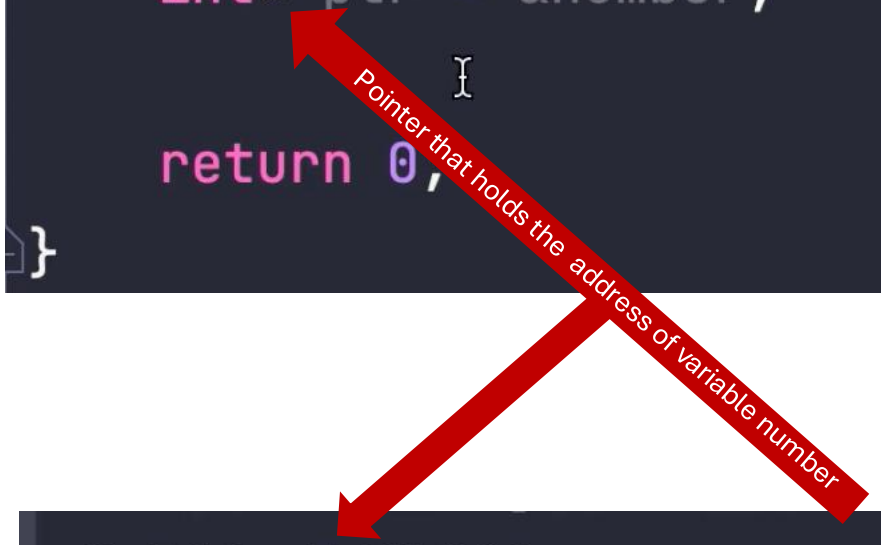

```
0x7ffee3d1b908
Process finished with exit code 0
```

- A pointer holds the address of another variable in memory.
- A * symbol after the type represents a pointer.
- Pointer type and viable type should match.

- Output of the cout statement.

Declaring and Using Pointers

```
int main() {
    int number = 10;
    cout << &number;

    int* ptr = &number;

    return 0;
}
```




```
0x7ffee3d1b908
Process finished with exit code 0
```

- A pointer holds the address of another variable in memory.
- A * symbol after the type represents a pointer.
- Pointer type and viable type should match.
- Pointers should be initialized. Otherwise, we might access a part of the memory that we are not supposed to.

- Output of the cout statement.

Declaring and Using Pointers

```
int main() {
    int number = 10;

    int* ptr;
    cout << ptr;

    return 0;
}
```

Pointer not initialized

- Pointers should be initialized. Otherwise, we might access a part of the memory that we are not supposed to.
- Program may terminate and give “memory access violation” error.

```
int main() {
    int number = 10;

    int* ptr = nullptr;
    cout << ptr;

    return 0;
}
```

Initialized to null pointer

Declaring and Using Pointers

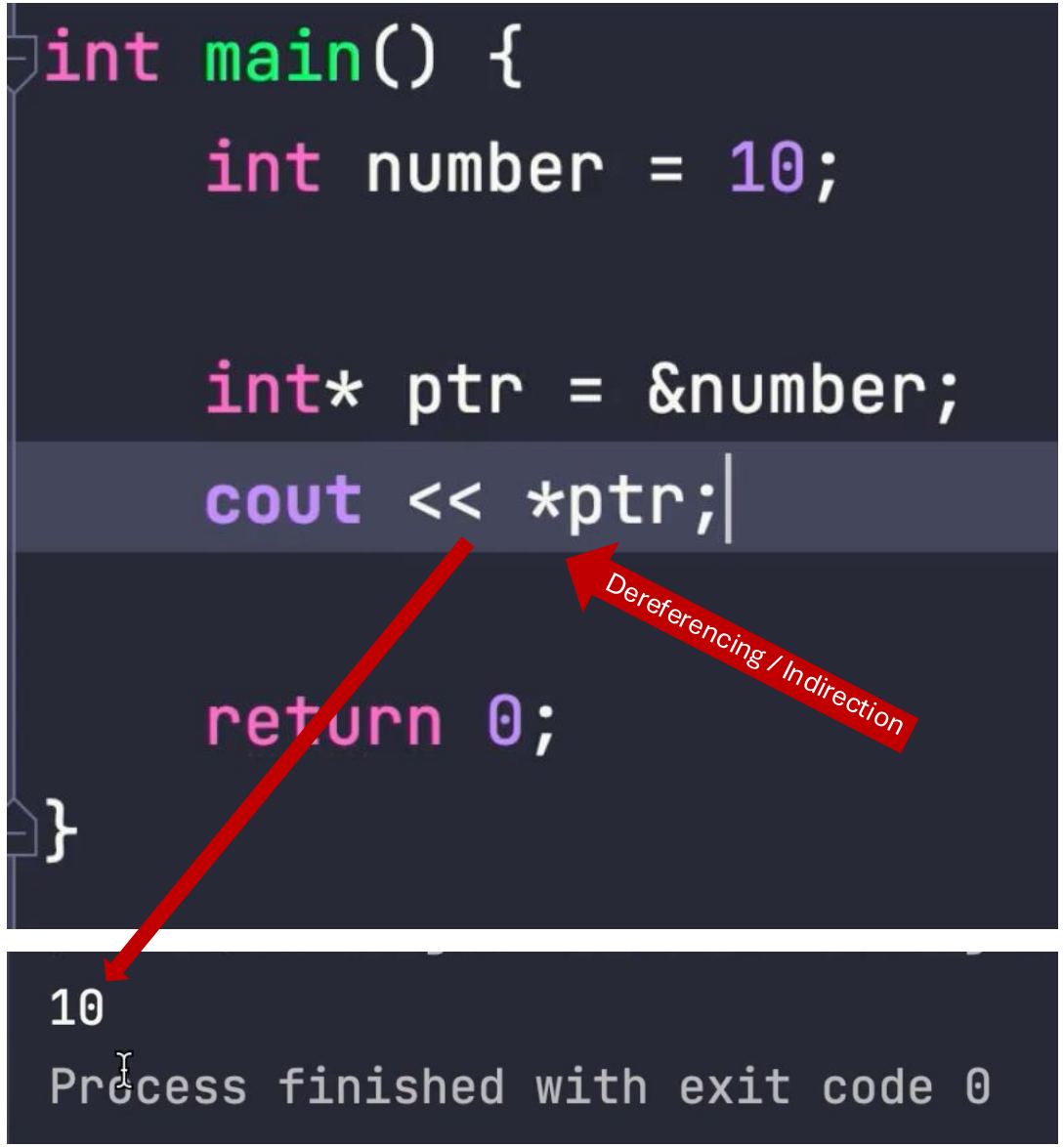
- Pointers should be initialized. Otherwise, we might access a part of the memory that we are not supposed to.
- Program may terminate and give “memory access violation” error.

```
int main() {
    int number = 10;

    int* ptr = nullptr;
    cout << ptr;

    return 0;
}
```

Initialized to null pointer

```
int main() {
    int number = 10;

    int* ptr = nullptr;
    if (ptr != nullptr)
    cout << ptr;

    return 0;
}
```

Pointers



- * in front of the pointer is called “dereferencing” or “indirection”.
- It is used to access the value that pointer points to.

- Output of the cout statement.

Pointers

```
int main() {
    int number = 10;

    // The address-of operator
    int* ptr = &number;
    // Indirection (de-referencing) operator
    *ptr = 20;

    cout << number;

    return 0;
}
```

- * in front of the pointer is called “dereferencing” or “indirection”.
- It is used to access the value that pointer points to.

Pointers

```c
int main() {
    int x = 10;
    int y = 20;
    int* ptr = &x;
    *ptr *= 2;
    ptr = &y;
    *ptr *= 3;

    return 0;
}
```

- **Exercise:** Determine the values printed on the terminal.

Pointers and Constants

Pointers and Constants

Case 1: Data is Constant

```
int main() {
    const int x = 10;
    int* ptr = &x;

    return 0;
}
```

- This declaration will give an error.
- Since the x is constant int, then the pointer should be constant int.
- Types should match.

```
int main() {
    const int x = 10;
    const int* ptr = &x;

    return 0;
}
```

Pointers and Constants

Case 1: Data is Constant

```
int main() {
    const int x = 10;
    const int* ptr = &x;

    *ptr = 20;

    return 0;
}
```

- *ptr = 20 statement will give a compilation error.
- The variable x is declared as constant and cannot be changed.

Pointers and Constants

Case 1: Data is Constant

```
int main() {
    const int x = 10;
    const int* ptr = &x;

    int y = 20;
    ptr = &y;

    return 0;
}
```

- *ptr = 20 statement will give a compilation error.
- The variable x is declared as constant and cannot be changed.
- The pointer ptr is not constant, it can be set the address of y later on.

Pointers and Constants

Case 2: Pointer is Constant

```
int main() {
    int x = 10;
    int* const ptr = &x;

    int y = 20;
    ptr = &y;

    return 0;
}
```

Compilation error

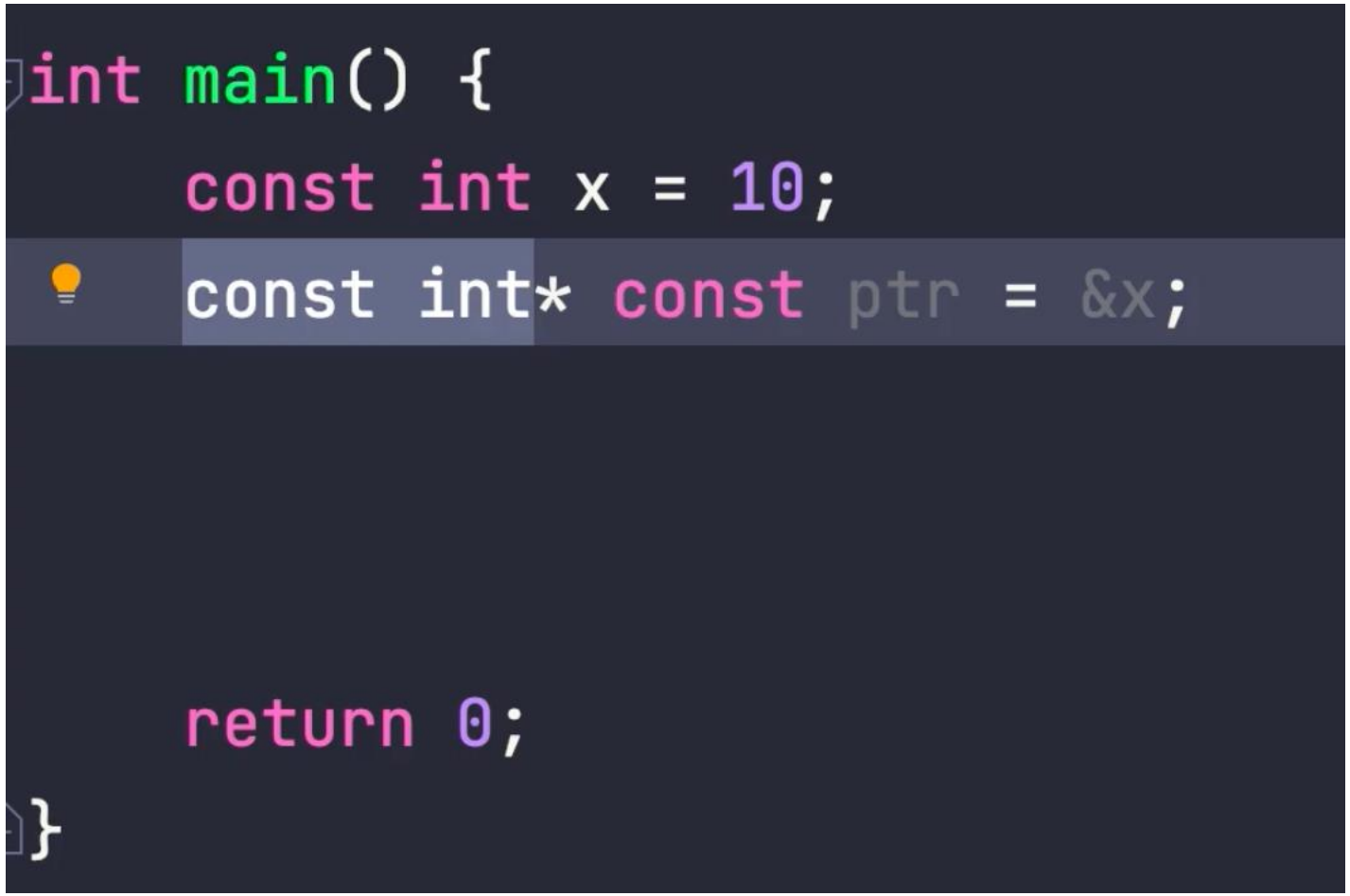
- The pointer ptr is constant.
- It cannot be changed.
- It has to be initialized during the declaration.

```
int main() {
    int x = 10;
    int* const ptr;

    return 0;
}
```

Compilation error

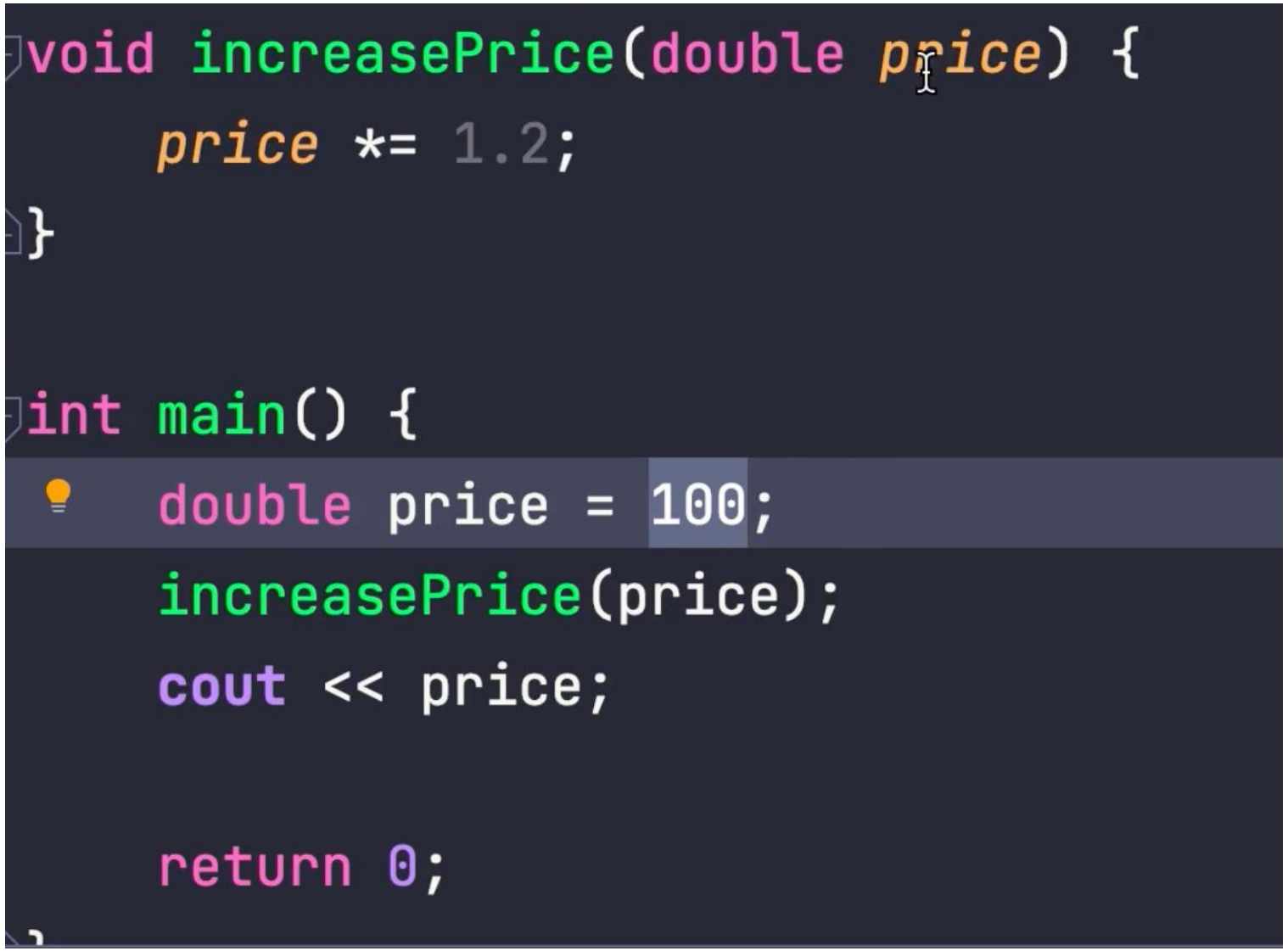
Pointers and Constants

Case 3: Both the Data and the Pointer are Constant



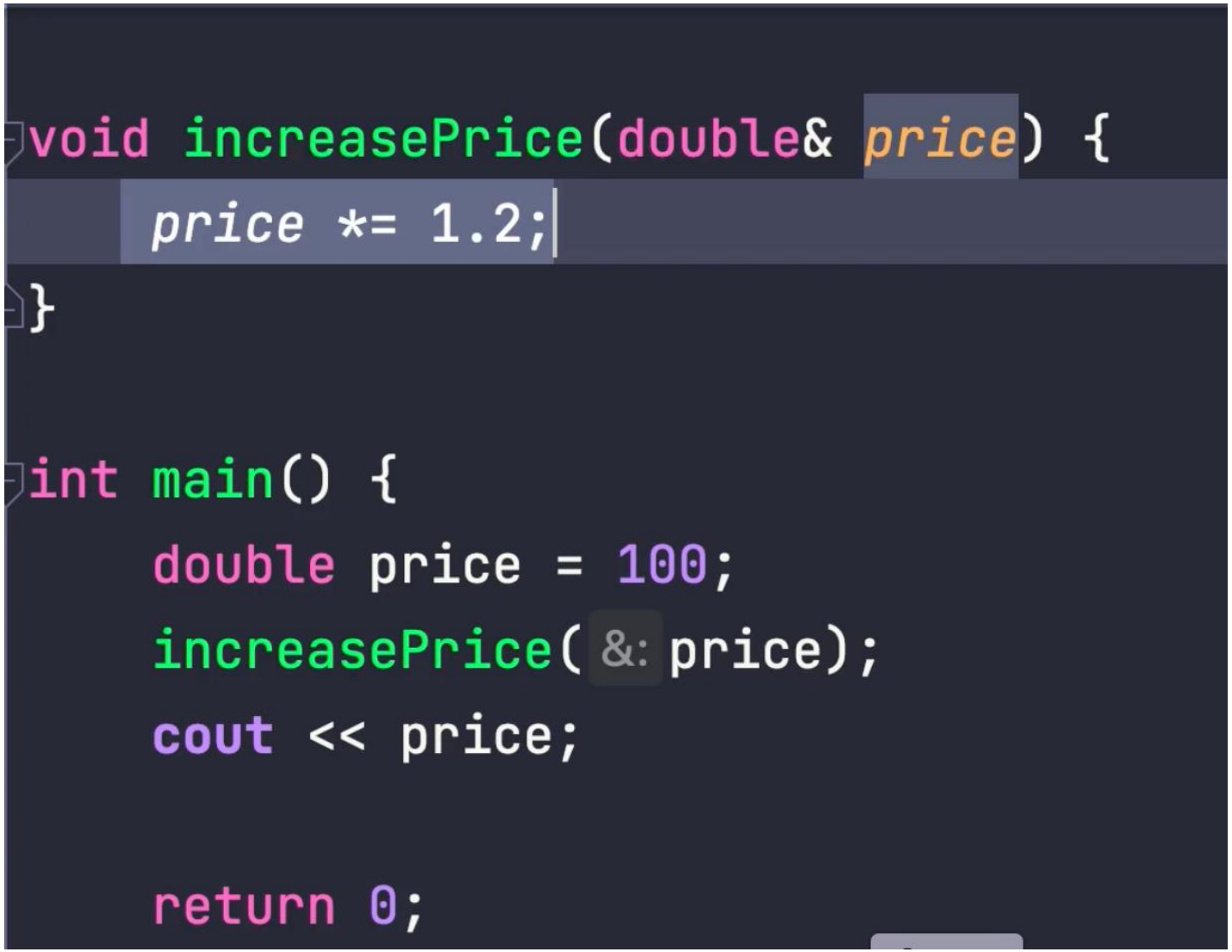
- A constant pointer, ptr, is pointing to a constant integer, x.

Passing Pointers to Functions



- The value of price is copied to the parameter price in the function.
- This is called-by-value.
- In the main function, the value of price does not change after the function call.

Passing Pointers to Functions



```
void increasePrice(double& price) {
    price *= 1.2;
}

int main() {
    double price = 100;
    increasePrice( &: price);
    cout << price;

    return 0;
```

- This is call-by-reference.
- The address of the price is passed to the argument.
- The change on the parameter price in the function changes the value in the main function.

Passing Pointers to Functions

```
void increasePrice(double* price) {
    *price *= 1.2;
}

int main() {
    double price = 100;
    increasePrice( price: &price);
    cout << price;

    return 0;
```

- This is using a pointer.
- Using call-by-reference is a better choice.

```cpp
#include <iostream>

using namespace std;

void swap(int* first, int* second) {
    int temp = *first;
    *first = *second;
    *second = temp;
}

int main() {
    int x = 10;
    int y = 20;
    swap( first: &x,  second: &y);
    cout << x << ", " << y;

    return 0;
}
```

Passing Pointers to Functions

EXERCISE

Implement the swap function for swapping two variables using pointers.

```cpp
void swap(int* first, int* second);
```

The Relationship Between Array and Pointers

```cpp
using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    cout << numbers;

    return 0;
}
```

- The array variable numbers is technically a pointer that points the address of the first element.

```
0x7ffeece758fc
Process finished with exit code 0
```

- Output of the cout statement.

The Relationship Between Array and Pointers

```
using namespace std;

    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    cout << *numbers;

    return 0;
}
```

- The array variable numbers is technically a pointer that points the address of the first element.

```
10
Process finished with exit code 0
```

- Output of the cout statement.

The Relationship Between Array and Pointers

```
using namespace std;

    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    int* ptr = numbers;
    cout << ptr;


    return 0;
}
```

- The array variable numbers is technically a pointer that points the address of the first element.

```
0x7ffeece758fc
Process finished with exit code 0
```

- Output of the cout statement.

The Relationship Between Array and Pointers

```
using namespace std;

    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    int* ptr = numbers;
    cout << *ptr;


    return 0;
}
```

- The array variable numbers is technically a pointer that points the address of the first element.

```
10
Process finished with exit code 0
```

- Output of the cout statement.

The Relationship Between Array and Pointers

```
using namespace std;

    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    int* ptr = numbers;
    cout << ptr[1];


    return 0;
}
```

- The array variable numbers is technically a pointer that points the address of the first element.

```
20
Process finished with exit code 0
```

- Output of the cout statement.

The Relationship Between Array and Pointers

```cpp
using namespace std;

void printNumbers(int numbers[]) {
    numbers[0] = 0;
}

int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    printNumbers(numbers);
    cout << numbers[0];

    return 0;
}
```

- In C++, arrays as function parameters are always passed-by-reference for efficiency.

```
0
Process finished with exit code 0
```

- Output of the cout statement.

Pointer Arithmetic

```cpp
#include <iostream>

using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    int* ptr = numbers;
    ptr++;
    cout << *ptr;

    return 0;
}
```

- Pointer ptr points the first element of numbers.
- ptr++ points the second element.
- Its value is incremented by the size of the data type (by 4 in the case of int data type)

```
20
Process finished with exit code 0
```

- Output of the cout statement.

Pointer Arithmetic

```cpp
using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    int* ptr = numbers;
    cout << *(ptr + 1);
    cout << ptr[1];
    cout << numbers[1];

    return 0;
}
```

- All the expressions in cout statements are identical.

```
10
Process finished with exit code 0
```

- Output of the cout statement.

Pointer Arithmetic

```cpp
using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    int* ptr = numbers;
    cout << *(ptr + 1);
    cout << ptr[1];
    cout << numbers[1];

    return 0;
}
```

- All the expressions in cout statements are identical.
- The bracket notation ptr[1] is easy to use and follow.

```
20
Process finished with exit code 0
```

- Output of the cout statement.

Comparing Pointers

```
using namespace std;
int main() {
    int x = 10;
    int y = 20;

    int* ptrX = &x;
    int* ptrY = &x;

    if (ptrX == ptrY)
        cout << "Same";

    return 0;
```

```
Same
Process finished with exit code 0
```

- We are comparing the address of the variable x stored in pointers ptrX and ptrY.
- Output of the cout statement.

Comparing Pointers

```
using namespace std;
int main() {
    int x = 10;
    int y = 20;

    if (ptrX != nullptr)
        cout << *ptrX;

    return 0;
}
```

- It is a good practice to check if a pointer is a null pointer or not before using it.

```
10
Process finished with exit code 0
```

- Output of the cout statement.

Comparing Pointers

EXERCISE

Given this array:

```
int numbers[] = { 10, 20, 30 };
```

Create a pointer that points to the last element in this array.

Use a while loop to iterate over this array, and print the numbers in reverse order.

Comparing Pointers

```cpp
using namespace std;

int main() {
    int numbers[] = { [0]: 10, [1]: 20, [2]: 30 };
    int* ptr = &numbers[size(numbers) - 1];
    while (ptr >= numbers) {
        cout << *ptr << endl;
        ptr--;
    }

    return 0;
}
```

```
30
20
10

Process finished with exit code 0
```

EXERCISE

Given this array:

```
int numbers[] = { 10, 20, 30 };
```

Create a pointer that points to the last element in this array.

Use a while loop to iterate over this array, and print the numbers in reverse order.

- Output of the cout statement.

```cpp
using namespace std;
int main() {
    // Stack
    // int numbers[1000];
    int x;

    // Heap (Free Store)
    int* numbers = new int[10];
    int* number = new int;
    delete number;
    delete[] numbers;
    number = nullptr;
    numbers = nullptr;

    return 0;
}
```

Dynamic Memory Allocation

- Static variables are created in the stack memory location and deleted automatically once the function ends.
- Dynamic variables are created in the heap memory location and are not deleted automatically.
- User is responsible for the deletion of variables in the heap.
- If not properly deleted, a “memory leak” will occur and the program will crash.

Dynamically Resizing Arrays

```
using namespace std;
int main() {
    int capacity = 5;
    int* numbers = new int[capacity];
    int entries = 0;

    while (true) {
        cout << "Number: ";
        cin >> numbers[entries];
        if (cin.fail()) break;
        entries++;
        if (entries == capacity) {
            capacity *= 2;
            int* temp = new int[capacity];
            for (int i = 0; i < entries; i++)
                temp[i] = numbers[i];
            delete[] numbers;
            numbers = temp;
        }
    }
```

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

    delete[] numbers;

    return 0;
}
```

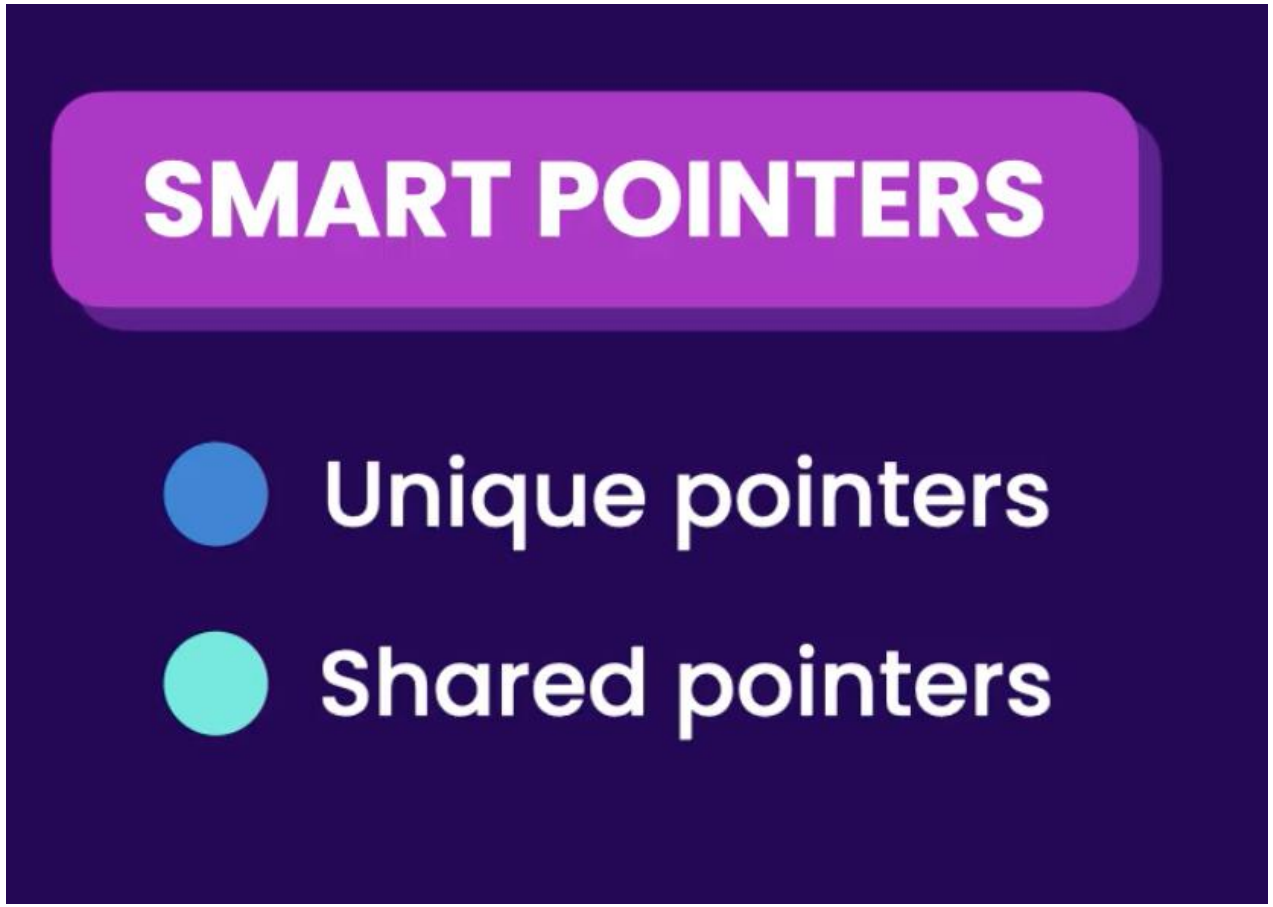
Smart Pointers

```
using namespace std;

int main() {
    int* x = new int;
    delete x;
    delete x;


    return 0;
}
```

- Deleting a pointer more than once will crash the program.
- With smart pointers, we do not need to worry about deleting them.

```
HelloWorld(18860,0x11a910e00) malloc: *** error for object 0x
HelloWorld(18860,0x11a910e00) malloc: *** set a breakpoint in
```

Smart Pointers



- Deleting a pointer more than once will crash the program.
- With smart pointers, we do not need to worry about deleting them.

Working with Unique Pointers

```cpp
#include <iostream>
#include <memory>

using namespace std;

int main() {
    unique_ptr<int> x( p: new int);
    cout << x;

    return 0;
}
```

- A unique pointers owns the memory location it points to.
- No two unique pointers can share the same memory location.
- **Limitation:** We cannot do arithmetic operations on unique pointers.
- unique_ptr< > is a generic class. It uses raw pointers.
- x is an object and an instance of the class.
- It contains a method to delete the pointer automatically.

Working with Unique Pointers

```cpp
#include <iostream>
#include <memory>

using namespace std;

int main() {
    unique_ptr<int> x( p: new int);
    unique_ptr<int> y = make_unique<int>();
    *x = 10;
    cout << *x;

    return 0;
}
```

- A unique pointers owns the memory location it points to.
- No two unique pointers can share the same memory location.
- **Limitation:** We cannot do arithmetic operations on unique pointers.
- unique_ptr< > is a generic class. It can work with different data types.
- It contains a method to delete the pointer automatically.
- make_unique<> function creates the unique pointer (Another way).

Working with Unique Pointers

```cpp
#include <iostream>
#include <memory>

using namespace std;

int main() {
    auto x : unique_ptr<int> = make_unique<int>();
    auto y : unique_ptr<int> = make_unique<int>();
    *x = 10;
    cout << *x;

    return 0;
}
```

- A unique pointers owns the memory location it points to.
- No two unique pointers can share the same memory location.
- **Limitation:** We cannot do arithmetic operations on unique pointers.
- unique_ptr< > is a generic class. It can work with different data types.
- It contains a method to delete the pointer automatically.
- **make_unique<>** function creates the unique pointer (Another way).
- **auto** function lets the compiler infer the type returned from the make_unique function.

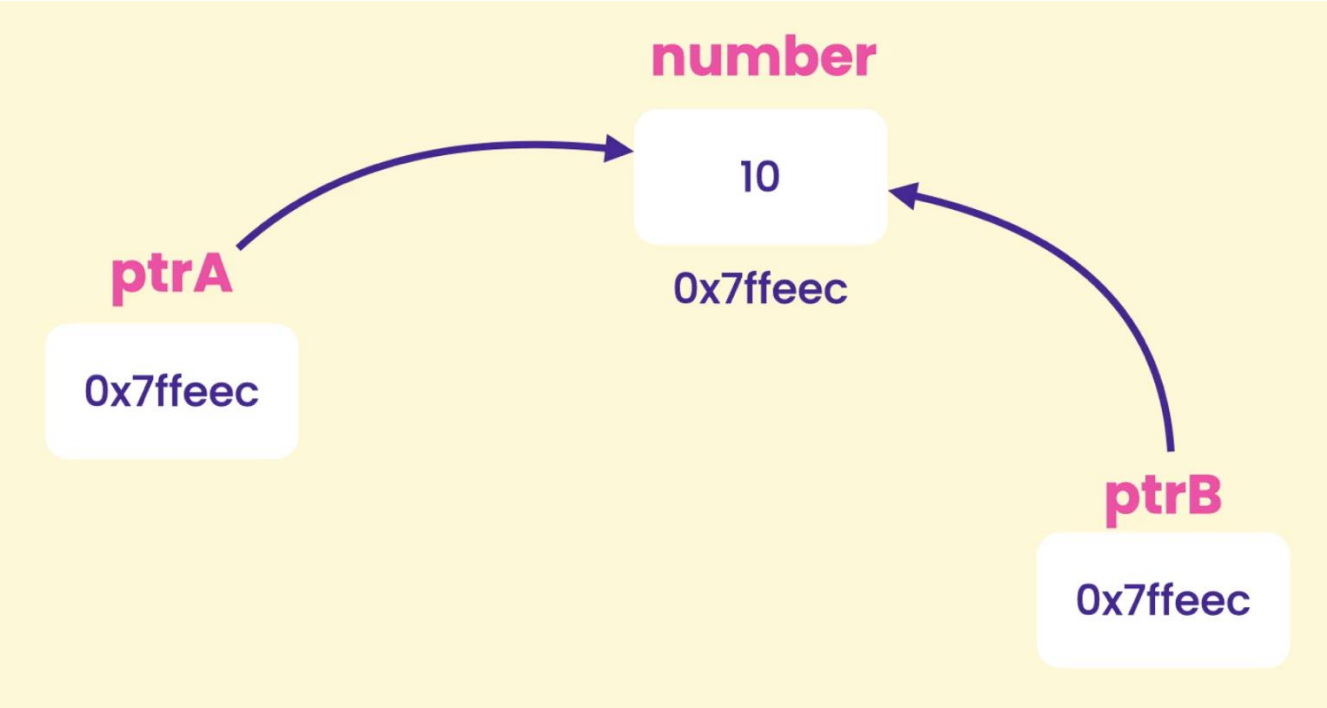
Working with Unique Pointers

```cpp
#include <iostream>
#include <memory>

using namespace std;
```

```cpp
int main() {
    auto numbers : unique_ptr<int[]> = make_unique<int[]>( n: 10);
    auto y : unique_ptr<int> = make_unique<int>();
    numbers[0] = 10;
    cout << numbers[0];

    return 0;
}
```

- A unique pointers owns the memory location it points to.
- No two unique pointers can share the same memory location.
- **Limitation:** We cannot do arithmetic operations on unique pointers.
- unique_ptr< > is a generic class. It can work with different data types.
- It contains a method to delete the pointer automatically.
- **make_unique<>** function creates the unique pointer (Another way).
- **auto** function lets the compiler infer the type returned from the make_unique function.

Working with Shared Pointers



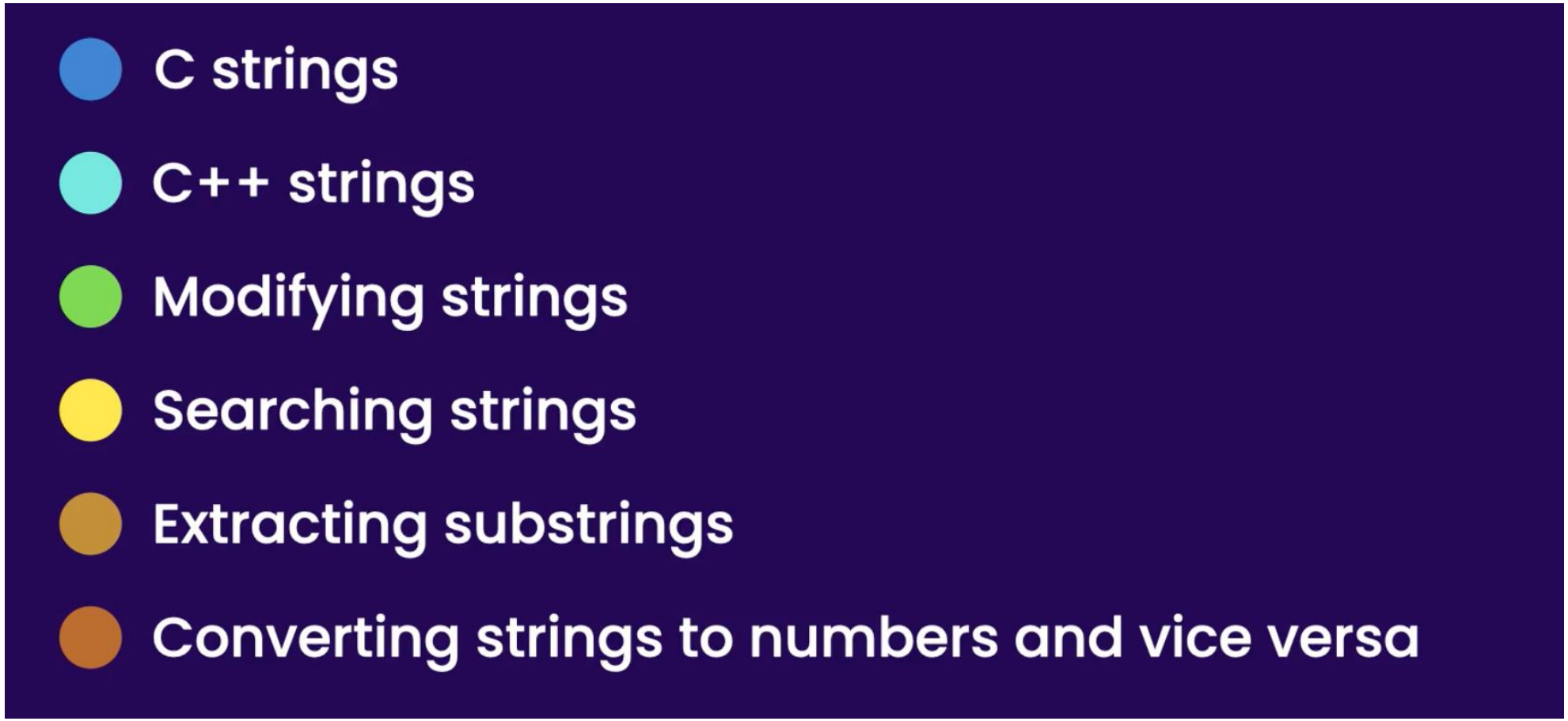
- Two pointers can share the same memory location.

Working with Shared Pointers

```cpp
#include <iostream>
#include <memory>

using namespace std;

int main() {
    auto x : shared_ptr<int> = make_shared<int>();
    *x = 10;

    shared_ptr<int> y(x);
    if (x == y)
        cout << "Equal";

    return 0;
}
```

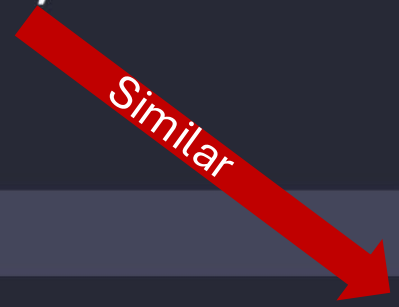
- Two pointers can share the same memory location.

Strings


C strings
C++ strings
Modifying strings
Searching strings
Extracting substrings
Converting strings to numbers and vice versa

C Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    // Null terminator (\0)
    // String literal
    char name[5] = "Mosh";
    // Character literal
    name[0] = 'm';
    cout << name[0];

    return 0;
}
```




```
name[5] = {[0]: 'M', [1]: 'o', [2]: 's', [3]: 'h', [4]: '\0'}
```

- ❑ String type always adjusts its size automatically.
- ❑ Functions working on strings are in <string> but it is already included in <iostream>, so we do not need to include in the header.
- ❑ Functions working on C strings are in <cstring> file, but it is already included in <iostream>, so we do not need to include in the header.

```cpp
#include <iostream>

using namespace std;

int main() {
    // Null terminator (\0)
    // String literal
    char name[5] = "Mosh";
    // Character literal
    name[0] = 'm';
    cout << strlen( s: name);

    return 0;
}
```

C Strings

- **strlen ()** returns the length of the string.

C Strings

- **strcat ()** concatenates two strings.
- The first string should have enough space to hold the second string.

```cpp
#include <iostream>

using namespace std;

int main() {
    char name[50] = "Mosh";
    char lastName[] = "Hamedani";
    // Concatenate (combine)
    strcat( s1: name,  s2: lastName);
    cout << name;

    return 0;
}
```

```cpp
#include <iostream>

using namespace std;

int main() {
    char name[50] = "Mosh";
    char lastName[] = "Hamedani";
    strcpy( dst: name,  src: lastName);
    cout << name;

    return 0;
}
```

C Strings

- **strcopy ()** copies the second string into the first strings.
- The first string should have enough space to hold the second string.

```cpp
#include <iostream>

using namespace std;

int main() {
    char name[50] = "Mosh";
    char lastName[] = "Hamedani";
    if (strcmp(name, lastName) == 0)
        cout << "Equal";

    return 0;
}
```

C Strings

- **strcmp ()** compares two strings.
- If the value returned is zero, then the two strings are equal.
- If the first value comes before the second value alphabetically, then the function will return a negative value.
- If the first value comes after the second value alphabetically, then the function will return a positive value.

```cpp
#include <iostream>

using namespace std;
```


```cpp
#include <iostream>

using namespace std;
```


```cpp
int main() {
    string name = "Mosh";
    name[0] = 'm';
    cout << name;

    return 0;
}
```

C++ Strings

- **string class** internally uses character array and hides all the implementation complexity and details.
- Has many more functions than C strings.

```cpp
#include <iostream>

using namespace std;


int main() {
    string name = "Mosh";
    name[0] = 'm';
    cout << name.length();

    return 0;
}
```

C++ Strings

- **name** is an **object** (an **instance**) of **string class** and there are many functions that can be used.
- Functions are called with their name after a ”.” after the object name.
- The **length** function returns the size of the string.

```cpp
#include <iostream>

using namespace std;


int main() {
    string name = "Mosh";
    name[0] = 'm';
    name += " Hamedani"

    return 0;
}
```

C++ Strings

- **name** is an **object** (an **instance**) of **string class** and there are many functions that can be used.
- Functions are called with their name after a "." after the object name.
- **Example:** Appending a string to another string.
- Resizing the string array, name, is done automatically.

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh";
    string another = name;
    if (name == another)
        cout << "Same";

    return 0;
}
```

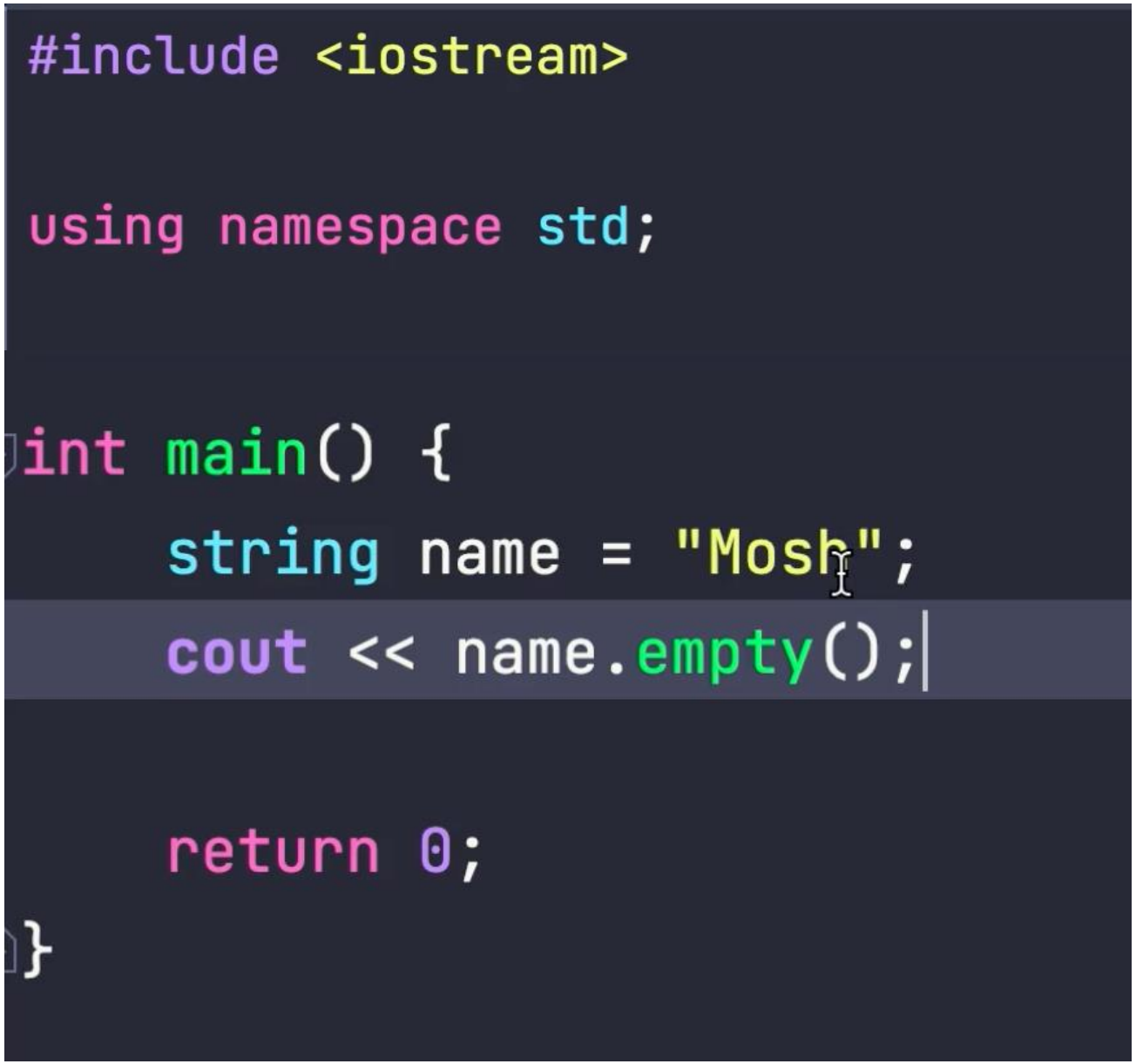
C++ Strings

- **name** is an **object** (an **instance**) of **string class** and there are many functions that can be used.
- Functions are called with their name after a ”.” after the object name.
- **Example:** Copying and comparing strings.

C++ Strings

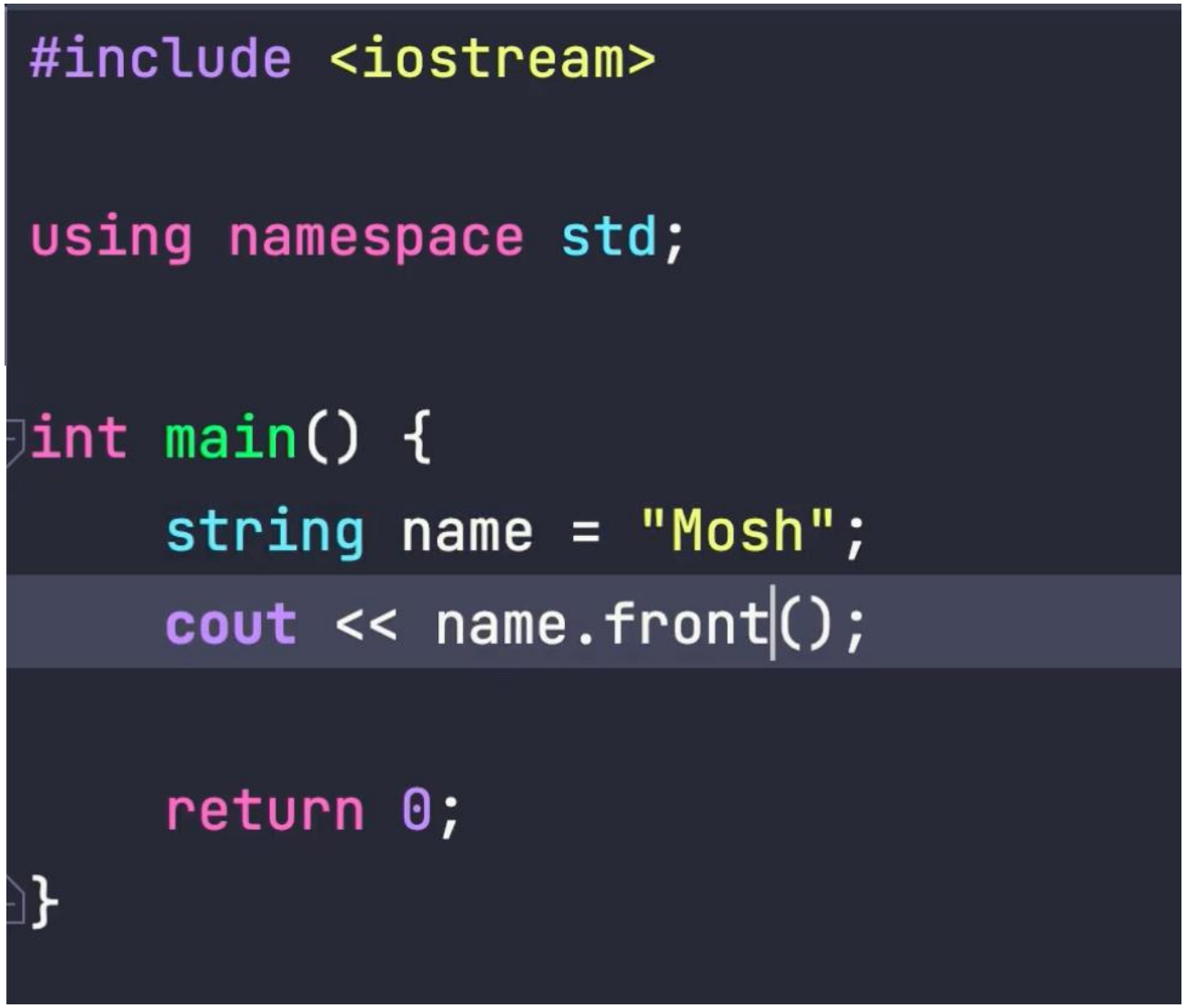
- **name** is an **object** (an **instance**) of **string class** and there are many functions that can be used.
- Functions are called with their name after a ”.” after the object name.

C++ Strings



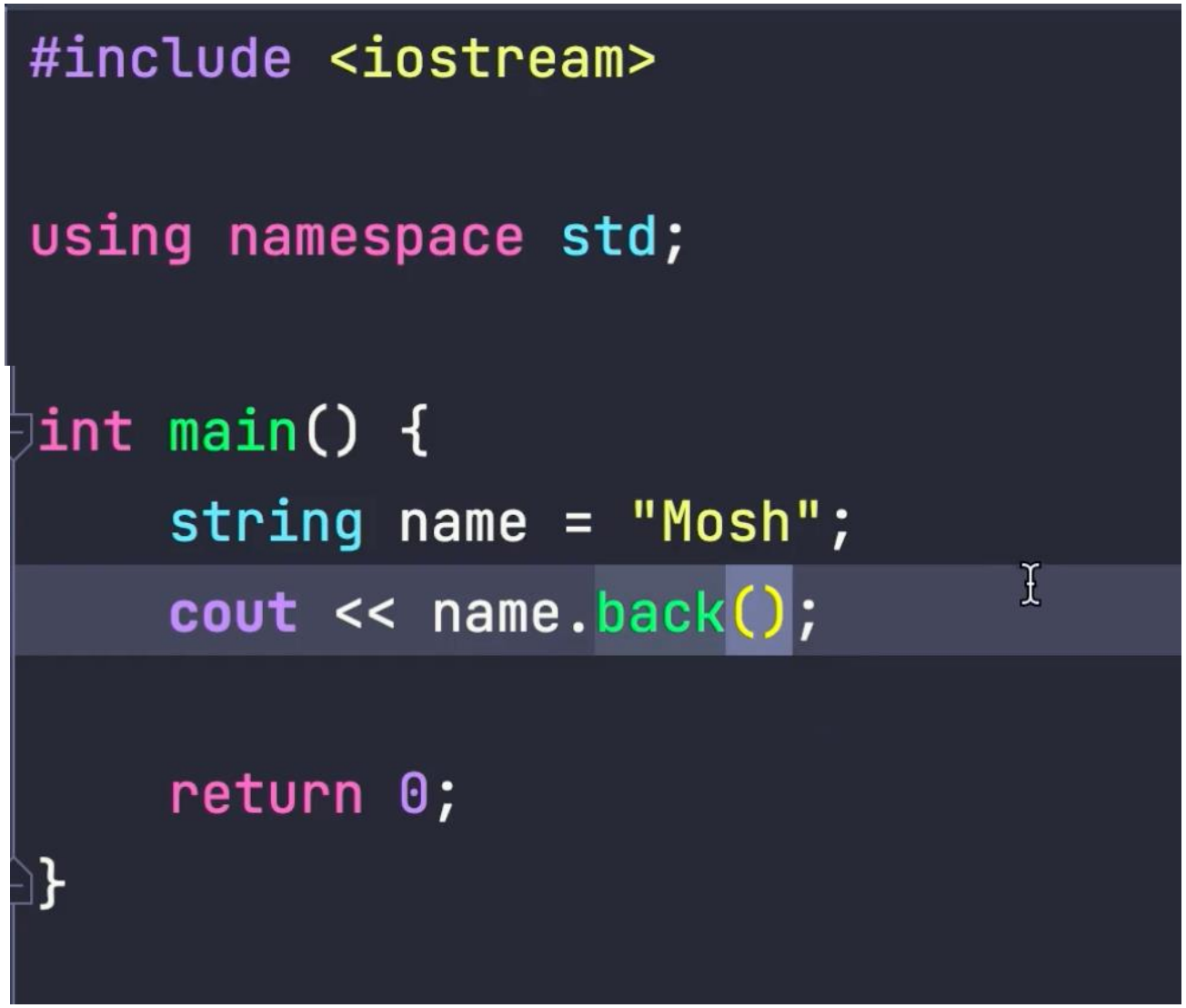
- **name** is an **object** (an **instance**) of **string class** and there are many functions that can be used.
- Functions are called with their name after a "." after the object name.
- Function **empty** returns **true (1)** if the string is empty, otherwise, returns false (0).

C++ Strings



- **name** is an **object** (an **instance**) of **string class** and there are many functions that can be used.
- Functions are called with their name after a ”.” after the object name.
- Function **front** returns the **first** character of the string.

C++ Strings



- **name** is an **object** (an **instance**) of **string class** and there are many functions that can be used.
- Functions are called with their name after a ”.” after the object name.
- Function **back** returns the **last** character of the string.

Modifying Strings

- **append** function attaches the argument string to the object string that makes the call.

Modifying Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh";
    name.insert( pos: 0,  s: "I am ");
    cout << name;

    return 0;
}
```

- **insert** function inserts the argument string to the object string that makes the call starting from the specified position.

Modifying Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh";
    name.erase( pos: 0,  n: 2);
    cout << name;

    return 0;
}
```

- **erase** function deletes a number of characters starting from the specified position.

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh";
    name.clear();
    name = "";
    cout << name;

    return 0;
}
```

Modifying Strings

- **clear** function empties the string.
- It is same as name = “”.

Modifying Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh";
    name.replace( pos: 0,  n1: 2,  s: "MO");
    cout << name;

    return 0;
}
```

- **replace** function replaces a number of characters with specified characters, starting from the specified position.

Searching Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    cout << name.find( c: 'a');

    return 0;
}
```

- The function **find** searches the content in the string starting from the specified position.
- If no position is specified to start, then it starts from the beginning.

```
6
Process finished with exit code 0
```

- Output of the cout statement.

Searching Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    cout << name.find( c: 'a', pos: 7);

    return 0;
}
```

- The function **find** searches the content in the string starting from the specified position.
- If no position is specified to start, then it starts from the beginning.

```
10
Process finished with exit code 0
```

- Output of the cout statement.

Searching Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    if (name.find( c: 'A') == -1)
        cout << "Doesn't Exist!";

    return 0;
}
```

- The function **find** searches the content in the string starting from the specified position.
- If no position is specified to start, then it starts from the beginning.

Searching Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    cout << name.rfind( c: 'a');

    return 0;
}
```

- The function **rfind** searches the content in the string starting from the last position.

```
10
Process finished with exit code 0
```

- Output of the cout statement.

Searching Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh; Hamedani";
    cout << name.find_first_of( s: ",.;");

    return 0;
}
```

```
4
Process finished with exit code 0
```

- The function **find_first_of** returns the first position of any occurrence of the characters in the argument string.
- The function **find_last_of** returns the last position of any occurrence of the characters in the argument string.
- Output of the cout statement.

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    string copy = name.substr();
    name = "";
    cout << copy;

    return 0;
}
```

Extracting Substrings

- The function **substr** without any parameter returns the whole string.

Extracting Substrings

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    string copy = name.substr( pos: 5,  n: 3);
    cout << copy;

    return 0;
}
```

- The function **substr** returns the substring from the starting position specified in the first argument and the length specified in the second argument.

```
Ham
Process finished with exit code 0
```

- Output of the cout statement.

Extracting Substrings

- **Example:** Extracting first and last name.

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Smith Hamedani";
    auto index = name.rfind( c: ' ');
    string firstName = name.substr( pos: 0, n: index);
    string lastName = name.substr( pos: index + 1);
    cout << "(" << firstName << ")";
    cout << "(" << lastName << ")";

    return 0;
}
```

Working with Characters

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    islower( c: name[0])

    return 0;
}
```

- **islower:** Checks if the character is lower case.

Working with Characters

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    cout << isupper( c: name[0]);

    return 0;
}
```

- **isupper:** Checks if the character is upper case.

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "Mosh Hamedani";
    cout << isalpha( c: name[0]);

    return 0;
}
```

Working with Characters

- **isalpha:** Return true (1) if character is alphabetic.

Working with Characters

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = "1Mosh Hamedani";
    cout << isdigit( c: name[0]);

    return 0;
}
```

- **isdigit:** Return true (1) if character is a digit.

Working with Characters

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = " Mosh Hamedani";
    cout << isspace( c: name[0]);

    return 0;
}
```

- **isspace:** Return true (1) if character is a space.

Working with Characters

- **toupper:** Converts character to upper case.
- If the character is a non-alphabetic character, then the character is returned.

Working with Characters

```cpp
#include <iostream>

using namespace std;

int main() {
    string name = " Mosh Hamedani";
    cout << (char) tolower( c: 'A');

    return 0;
}
```

- **tolower:** Converts character to lower case.

String-Numeric Conversion Functions

```cpp
#include <iostream>

using namespace std;

int main() {
    double price = stod( str: "19.99");
    cout << price;

    return 0;
}
```

```
19.99
Process finished with exit code 0
```

- **stod:** Converts a nuber in a string to a double number.

- Output of the cout statement.

String-Numeric Conversion Functions

```cpp
#include <iostream>

using namespace std;

int main() {
    double price = stod( str: "19.x99");
    cout << price;

    return 0;
}
```

```
19
Process finished with exit code 0
```

- **stod:** Converts a number in a string to a double number.

- Output of the cout statement.

String-Numeric Conversion Functions

```cpp
#include <iostream>

using namespace std;

int main() {
    double price = stod( str: "x19.99");
    cout << price;

    return 0;
}
```

- **stod:** Converts a number in a string to a double number.
- If the string starts with a non-digit, then the program will crash with an exception.
  - Output of the cout statement.

```
libc++abi: terminating with uncaught exception of type std::i
```

String-Numeric Conversion Functions

```cpp
#include <iostream>

using namespace std;

int main() {
    string str = to_string( val: 19);
    cout << str;

    return 0;
}
```

- **to_string:** Converts a number into a string.

```
19
Process finished with exit code 0
```

- Output of the cout statement.

Escape Sequences

```cpp
#include <iostream>

using namespace std;

int main() {
    string str = "c:\\my folder";
    cout << str;

    return 0;
}
```

```
c:\my folder
Process finished with exit code 0
```

- \ character is used to include \ (or “) in a string.

- Output of the cout statement.

Escape Sequences

```cpp
#include <iostream>

using namespace std;

int main() {
    string str = "\"Hello World\"";
    cout << str;

    return 0;
}
```

- \ character is used to include \ (or “) in a string.

```
"Hello World"
Process finished with exit code 0
```

- Output of the cout statement.

Escape Sequences

```cpp
#include <iostream>

using namespace std;

int main() {
    char ch = '\'';
    string str = "'Hello World'";
    cout << ch;

    return 0;
}
```

```
'

Process finished with exit code 0
```

- \ character is used to include \ (or “) in a string.
- \ is not needed for ‘ character in a string.
- \ is needed for ‘ character in a **char array**.

- Output of the cout statement.

Escape Sequences

```cpp
#include <iostream>

using namespace std;

int main() {
    string str = "Hello\nWorld";
    cout << str;

    return 0;
}
```

- **\n** puts a new line character into the string.

```
Hello
World
Process finished with exit code 0
```

- Output of the cout statement.

Escape Sequences

```cpp
#include <iostream>

using namespace std;

int main() {
    string str = "Hello\tWorld";
    cout << str;

    return 0;
}
```

- **\t** puts a tab into the string.

```
Hello	World
Process finished with exit code 0
```

- Output of the cout statement.

Raw Strings

```cpp
#include <iostream>

using namespace std;

int main() {
    string str = "\"c:\\folderA\\folderB\\file.txt\"";
    cout << str;

    return 0;
}
```

Using raw string

```cpp
int main() {
    string str = R"("c:\folderA\folderB\file.txt")";
    cout << str;

    return 0;
}
```

- We do not need to use escape sequences for special characters if we use raw strings.