C++

OVERVIEW

C++: A Versatile Powerhouse

Created by *Bjarne Stroustrup* , C++ is a high-performance language that seamlessly blends procedural, object-oriented, and generic programming paradigms. As a middle-ground between high and low-level languages, it offers both efficiency and control. A cornerstone in software development, C++ is used to build everything from operating systems to cutting-edge games. Its rich standard library and cross-platform compatibility make it a developer's preferred choice.

Key Features

- ➢ **Static typing:** Ensures type safety during compilation.
- ➢ **Object-oriented:** Supports encapsulation, data hiding, inheritance, and polymorphism.
- ➢ **Standard Library:** Provides a rich set of functions and data structures.
- ➢ **Portability:** Adherence to the ANSI standard ensures compatibility.
- ➢ **Performance:** Offers low-level control and optimization opportunities.

Applications

- o System programming
- o Game development
- o Embedded systems
- o High-performance computing
- o Financial systems
- o Operating systems

When we consider a C++ program, it can be defined as a collection of objects that interact by invoking each other's methods. Let us now briefly explore what a class, object, methods, and instance variables mean.

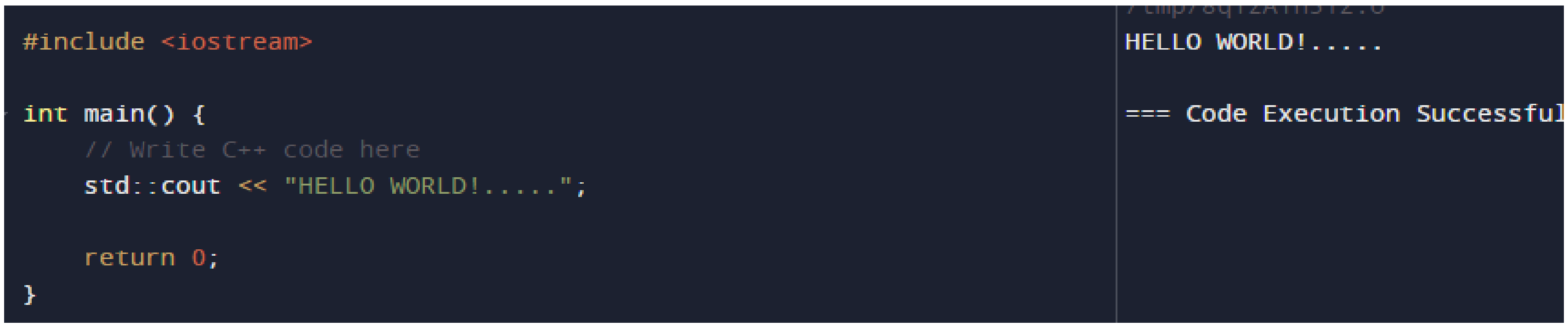
- **Object**: Objects have states and behaviors. Example: A car has states - color, model, brand, as well as behaviors - accelerating, braking, and turning. An object is an instance of a class.

- **Class**: A class can be defined as a template or blueprint that describes the behaviors and states that objects of its type support.

- **Methods**: A method is essentially a behavior. A class can contain multiple methods. It is within methods that the logic is written, data is manipulated, and all actions are performed.

- **Instance Variables**: Each object has its own unique set of instance variables. An object's state is defined by the values assigned to these instance variables.

BASIC HELLO WORLD PRINTING PROGRAM



```
#include <iostream>

int main() {
    // Write C++ code here
    std::cout << "HELLO WORLD!.....";

    return 0;
}
```

```
HELLO WORLD!.....

=== Code Execution Successful
```

The C++ language defines various headers that provide essential or beneficial information for your program. For this program, the header <iostream> is required. The line using namespace std; instructs the compiler to use the std namespace. Namespaces are a relatively recent feature in C++.

Basic Syntax

- The next line, // main() is where program execution begins, is a single-line comment in C++. Single-line comments start with // and continue to the end of the line.

- The line int main() is the main function where the program's execution starts.

- The subsequent line, cout << "This is my first C++ program."; displays the message “HELLO WORLD!....." on the screen.

- The line return 0; ends the main() function and returns the value 0 to the calling process.

Compile & Execute C++ Program:

- Here's how to save, compile, and run the program. Follow these steps:
- Open a text editor and enter the code provided above.
- Save the file as: hello.cpp.
- Open a command prompt and navigate to the directory where you saved the file.
- Type g++ hello.cpp and press enter to compile your code. If your code has no errors, the command prompt will move to the next line and generate an a.out executable file.
- Now, type ./a.out to run your program.
- You will see 'Hello World' printed on the window.

Semicolons & Blocks in C++

In C++, a semicolon is a statement terminator. This means that each statement must end with a semicolon, indicating the conclusion of a logical entity.

```
x = y;
y = y+1;
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces.
C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line

C++ Identifiers

Identifiers are names given to various program elements, such as variables, functions, classes, and objects. They help you to identify and refer to these elements in your code.

Rules for naming identifiers:

- Identifiers must start with a letter (uppercase or lowercase) or an underscore (_).
- The remaining characters can be letters, digits, or underscores.
- Identifiers are case-sensitive, so myVariable and myvariable are different identifiers.
- Identifiers cannot be the same as any of the reserved keywords in C++.
- Identifiers should be descriptive and meaningful to help you understand your code
- Use camel case or snake case for readability.

Whitespace in C++: The Invisible Architect

Whitespace is the collective term for characters that, while present, are essentially ignored by the C++ compiler. These include spaces, tabs, newlines, and comments.

Purpose of Whitespace:

- **Separation:** Clearly distinguishes different parts of a statement, making the code more readable.
- **Formatting:** Improves code readability by adding visual structure.
- **Comments:** Provides explanations without affecting the code's execution.

Key Points:

Blank Lines: Lines containing only whitespace (and possibly comments) are entirely ignored.

Compiler Focus: Whitespace helps the compiler recognize the boundaries of elements like keywords, identifiers, and operators.

Flexibility: You can use whitespace liberally to format your code for better understanding.

COMMENTS IN C++

Comments are essential for making your code understandable to both yourself and others. They are non-executable text that the compiler ignores. C++ offers two primary types:

Single-line comments

Begin with //
Everything after // on the same line is considered a comment.

Multi-line comments

Begin with /*
End with */
Everything between these delimiters is a comment.

Best practices for using comments:

- Explain the purpose of code sections or functions.
- Describe complex algorithms or logic.
- Provide information about input and output parameters.
- Document changes made to the code.
- Use clear and concise language

DATA TYPES IN C++

Data types define the kind of data a variable can hold and the operations that can be performed on it. C++ offers a rich set of data types to accommodate diverse programming needs.

Fundamental Data Types

These are the basic building blocks of C++:

- **int:** Represents integer values (whole numbers without decimal points).
  - short, long, and long long are variations for different integer sizes.
- **float:** Represents single-precision floating-point numbers (numbers with decimal points).
- **double:** Represents double-precision floating-point numbers (higher precision than float).
- **char:** Represents single characters.
- **bool:** Represents boolean values (true or false).
- **void:** Represents the absence of a type (often used for function return types).

<u>**Modifiers**</u>

You can modify fundamental data types using:

- **signed:** Indicates positive or negative values (default).
- **unsigned:** Indicates non-negative values.
- **short:** Reduces the size of an integer.
- **long:** Increases the size of an integer

Derived Data Types

These data types are created using fundamental data types:

- **Arrays:** Collections of elements of the same data type.
- **Pointers:** Variables that store memory addresses.
- **References:** Aliases for existing variables.
- **Structures:** User-defined composite data types.
- **Unions:** Allow storing different data types in the same memory location.
- **Enumerations:** User-defined data types with named constants.

User-Defined Data Types

You can create your own data types using classes and structs:

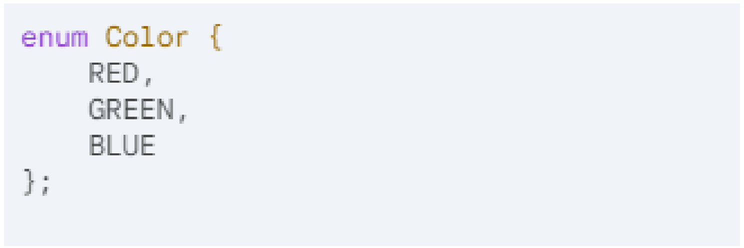
- **Classes:** Encapsulate data and functions.
- **Structs:** Similar to classes but with public members by default.

Choosing the Right Data Type

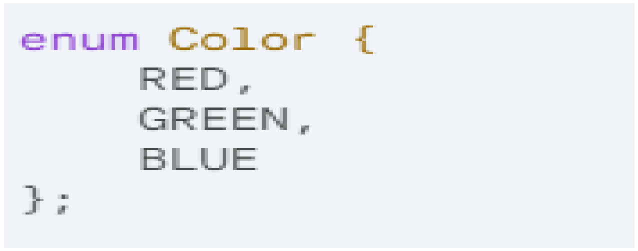
- **Range of values:** The data type should be able to accommodate the expected values.
- **Memory usage:** Choose data types efficiently to avoid unnecessary memory consumption.
- **Precision:** For numerical data, select the appropriate precision based on requirements.

Enumerated Types in C++

An enumerated type is a user-defined data type that consists of a set of named constants. These constants are integral values, but they are treated as identifiers rather than numbers.



```
enum Color {
    RED,
    GREEN,
    BLUE
};
```



```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

In this example, Color is the enumerated type, and RED, GREEN, and BLUE are the enumerators.
By default, RED has the value 0, GREEN has the value 1, and BLUE has the value 2.

Key points to remember:

- Enumerators are implicitly converted to integers.
- You can assign specific integer values to enumerators.
- Enumerated types can be used in switch statements.
- C++11 introduced scoped enums to prevent naming conflicts.

Variable Declaration vs. Definition

Declaration: Informs the compiler about the variable's type and name without allocating memory.

Example: extern int count;

Definition: Allocates memory for the variable and optionally assigns an initial value.

Example: int count = 0;

Scope of Variables:
Variables can be declared at different levels, affecting their visibility:

- **Global variables:** Declared outside any function, accessible from anywhere.
- **Local variables:** Declared within a function, accessible only within that function.
- **Block-level variables:** Declared within a block (e.g., {}), accessible only within that block.

Initialization:
Assigning a value to a variable at the time of declaration is called initialization.

Example: int age = 25;

Lvalues and Rvalues: A Deeper Dive

Lvalues:

- Refer to memory locations.
- Can appear on both the left and right sides of an assignment.
- Examples: variables, array elements, dereferenced pointers.

Rvalues:

- Represent data values without a specific address.
- Can only appear on the right side of an assignment.
- Examples: literals, expressions, function return values (unless explicitly specified as references).

Additional Considerations:

- **Const-qualified lvalues:** These are lvalues that cannot be modified.
- **Xvalues:** A more nuanced concept introduced in C++11, representing entities that can be moved from.
- **Rvalue references:** Used to bind to rvalues and enable move semantics.

Grasping the distinction between lvalues and rvalues is crucial for:

- Efficient code optimization.
- Correct use of references and pointers.
- Understanding move semantics and return value optimization.

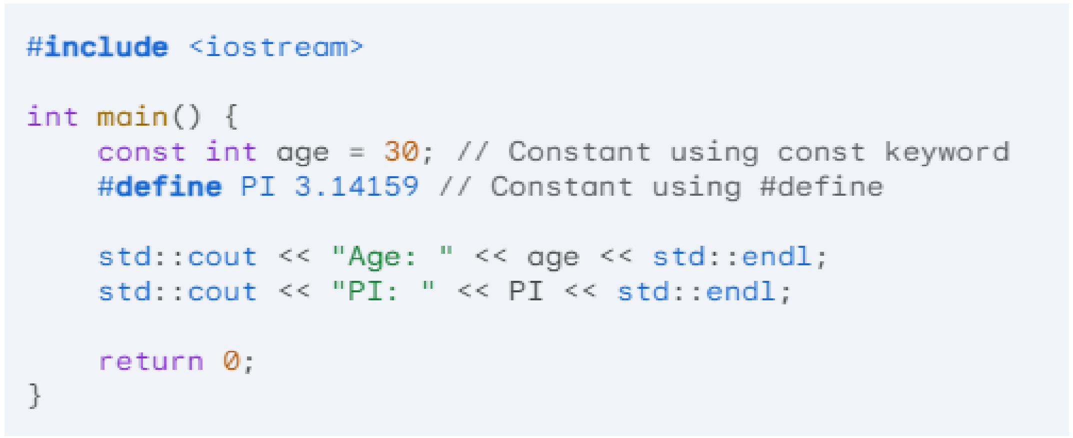
CONSTANTS/LITERALS

Constants: These are similar to variables but their values cannot be modified after they are initialized. They are often used to store values that are known at compile time and are used throughout the program.

Literals: These are fixed values that are directly inserted into the code. They cannot be changed during program execution.

Defining Constants
In C++, you can define constants using the const keyword or the #define preprocessor directive.

- const int MAX_VALUE = 100;
- #define PI 3.14159



```
#include <iostream>

int main() {
    const int age = 30; // Constant using const keyword
    #define PI 3.14159 // Constant using #define

    std::cout << "Age: " << age << std::endl;
    std::cout << "PI: " << PI << std::endl;

    return 0;
}
```

- Constants improve code readability and maintainability.
- Using const is generally preferred over #define for defining constants.T
- he value of a constant cannot be changed after it is defined.

Types of Literals

Integer Literals

These represent integer values. They can be expressed in decimal, octal, or hexadecimal format.

- **Decimal:** Normal base-10 numbers (e.g., 123, -456)
- **Octal:** Preceded by 0 (e.g., 0173)
- **Hexadecimal:** Preceded by 0x or 0X (e.g., 0xAF)

Floating-Point Literals

These represent real numbers with decimal points. They can be expressed in decimal or scientific notation.

- **Decimal:** Contains a decimal point (e.g., 3.14, -0.01)
- **Scientific notation:** Uses the exponent form (e.g., 1.23e5)

Character Literals

These represent single characters enclosed in single quotes. (e.g., 'a', 'Z', '$')

String Literals

These represent sequences of characters enclosed in double quotes. (e.g., "Hello, world!“)

Example: Boolean Literals

These represent boolean values (true or false).

Storage Classes in C++

Storage classes in C++ determine the lifetime, scope, linkage, and storage location of variables and functions. They provide essential information about how variables are managed within a program.

Types of Storage Classes

1. auto Storage Class

Default storage class for local variables.

Variables with auto storage class are created when the block or function is entered and destroyed when it exits.

Rarely used explicitly as it's the default behavior.

2. register Storage Class

Suggests to the compiler to store the variable in a CPU register for faster access.

Not guaranteed to be used by the compiler.

Use with caution as excessive use might degrade performance.

3. static Storage Class

Can be used for both local and global variables.

For local variables:

Retains its value between function calls.

Not initialized on every function call.

For global variables:

Restricts the variable's scope to the current file.

4. extern Storage Class

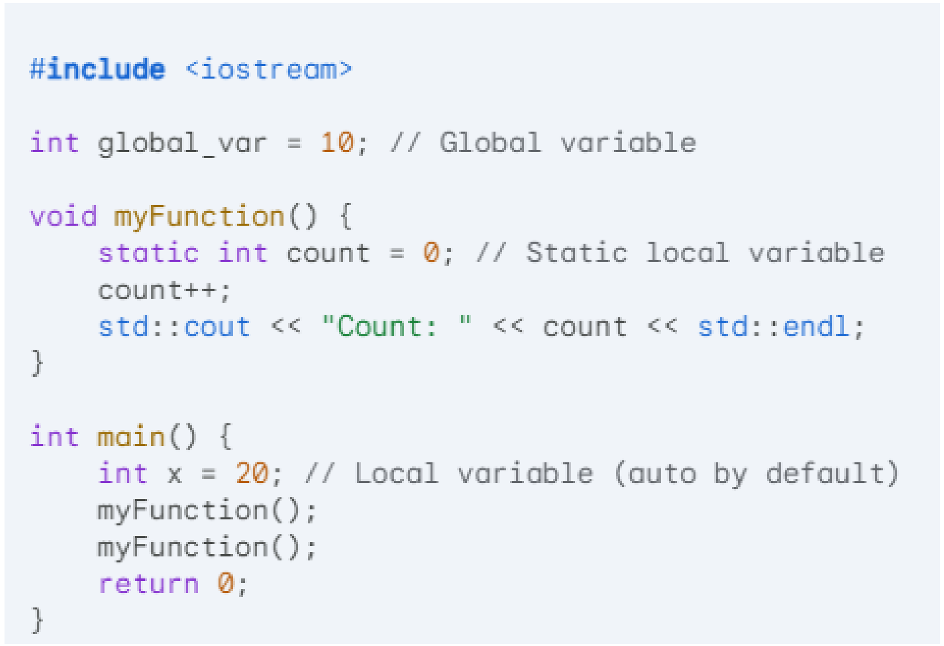
Declares a variable or function that has been defined elsewhere.

Used to access global variables from other files.

5. mutable Storage Class

Applies only to class member variables.

Allows modification of a const object's member.



```cpp
#include <iostream>

int global_var = 10; // Global variable

void myFunction() {
    static int count = 0; // Static local variable
    count++;
    std::cout << "Count: " << count << std::endl;
}

int main() {
    int x = 20; // Local variable (auto by default)
    myFunction();
    myFunction();
    return 0;
}
```

- Storage classes impact variable lifetime, scope, and linkage.
- Choose the appropriate storage class based on your program's requirements.
- Use static for variables that need to retain their value between function calls.
- Use extern to access global variables from other files.
- Use mutable cautiously for specific use cases.

LOOPS IN C++

Loops are control flow statements that allow you to execute a block of code repeatedly until a certain condition is met. They are essential for automating repetitive tasks and making your code more efficient

1 - for loop:

- Used when the number of iterations is known beforehand.

```
for (int i = 1; i <= 5; i++) {
    std::cout << i << " ";
}
```

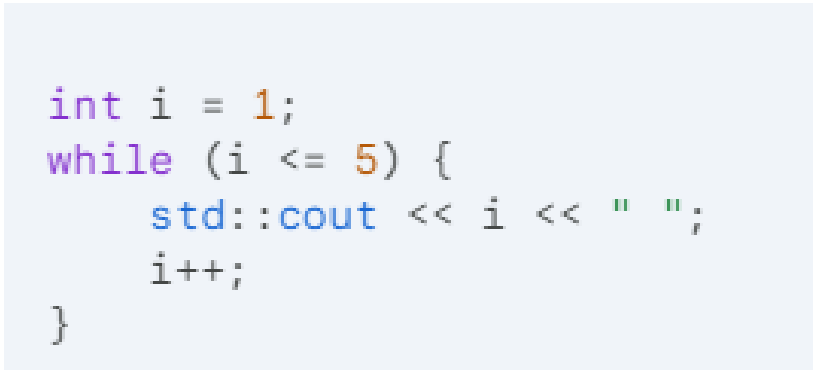
The for loop is a control flow statement that iterates a block of code a specified number of times. It's composed of three essential parts:
Initialization: This step runs once before the loop starts. Typically used to declare and initialize loop control variables.
Condition: This is checked at the beginning of each iteration. If true, the loop body executes; otherwise, the loop terminates.
Increment/Decrement: This step is executed after each loop iteration. It's commonly used to update the loop control variable.

- The initialization part runs once.
- The condition is checked, If true, the loop body executes.
- The increment/decrement part runs.
- The process repeats from step 2.
- When the condition becomes false, the loop terminates.

All three parts of the for loop are optional.The semicolon is mandatory to separate the three parts.The loop control variable can be declared and initialized within the loop itself.

2 - while loop:

- Used when the number of iterations is not known beforehand.The condition is checked before the loop body executes.



```
int i = 1;
while (i <= 5) {
    std::cout << i << " ";
    i++;
}
```

- The condition is evaluated.
- If the condition is true, the code within the loop body is executed.
- The condition is evaluated again.
- Steps 2 and 3 repeat as long as the condition remains true.
- Once the condition becomes false, the loop terminates.

▪ The condition is checked *before* each iteration.

▪ It's essential to ensure that the condition will eventually become false to avoid infinite loops.

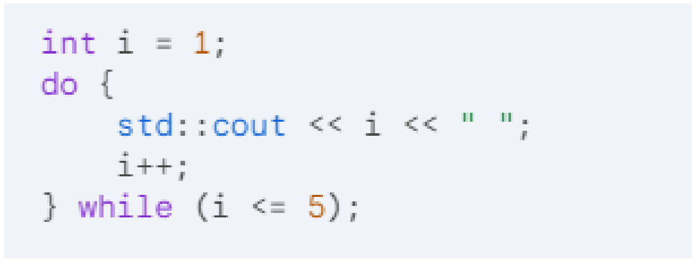
▪ The loop body can contain any valid C++ statements.

When to use a while loop:

- When the number of iterations is unknown or depends on a runtime condition.
- When the loop condition needs to be checked at the beginning of each iteration

3 - do-while loop:

Similar to the while loop, but the condition is checked after the loop body executes. Guarantees at least one execution of the loop body.



```cpp
int i = 1;
do {
    std::cout << i << " ";
    i++;
} while (i <= 5);
```

When to use a do-while loop:

When you need to execute a block of code at least once, regardless of the condition.
When the loop condition depends on the result of the loop body.

while loop: Checks the condition before executing the loop body.
do-while loop: Executes the loop body at least once, then checks the condition.

Choosing the Right Loop

- Use a for loop when the number of iterations is known in advance.
- Use a while loop when the number of iterations is unknown or depends on a condition.
- Use a do-while loop when you need to execute the loop body at least once, regardless of the condition.

Loop Control Statements

Loop control statements allow you to modify the normal execution flow of a loop. These statements are used to alter the loop's iteration process based on certain conditions.

1 - break:
Immediately terminates the entire loop, regardless of the loop condition.
Often used in conjunction with conditional statements.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }
    std::cout << i << " ";
}
```

2 - continue:
Skips the rest of the current iteration and jumps to the beginning of the next iteration.
Useful for skipping specific iterations based on a condition.

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    std::cout << i << " ";
}
```

FUNCTIONS

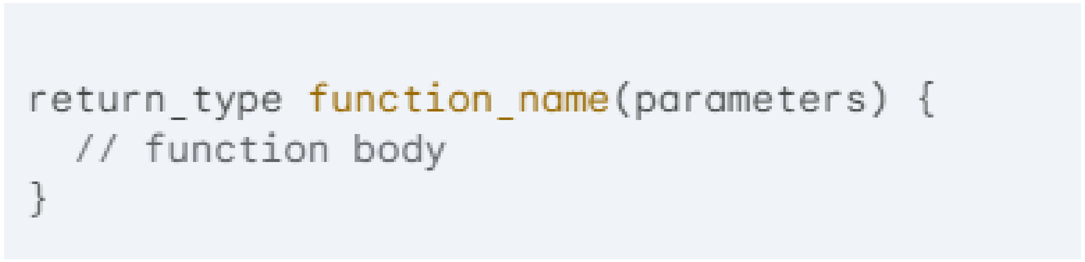
A function in C++ is a block of code that performs a specific task. It's designed to encapsulate a set of instructions, making code more organized, reusable, and easier to understand.

Key components of a function:

- **Function declaration:** This specifies the function's name, return type, and parameters.
- **Function definition:** This contains the actual code that the function executes.
- **Function call:** This invokes the function to perform its task.

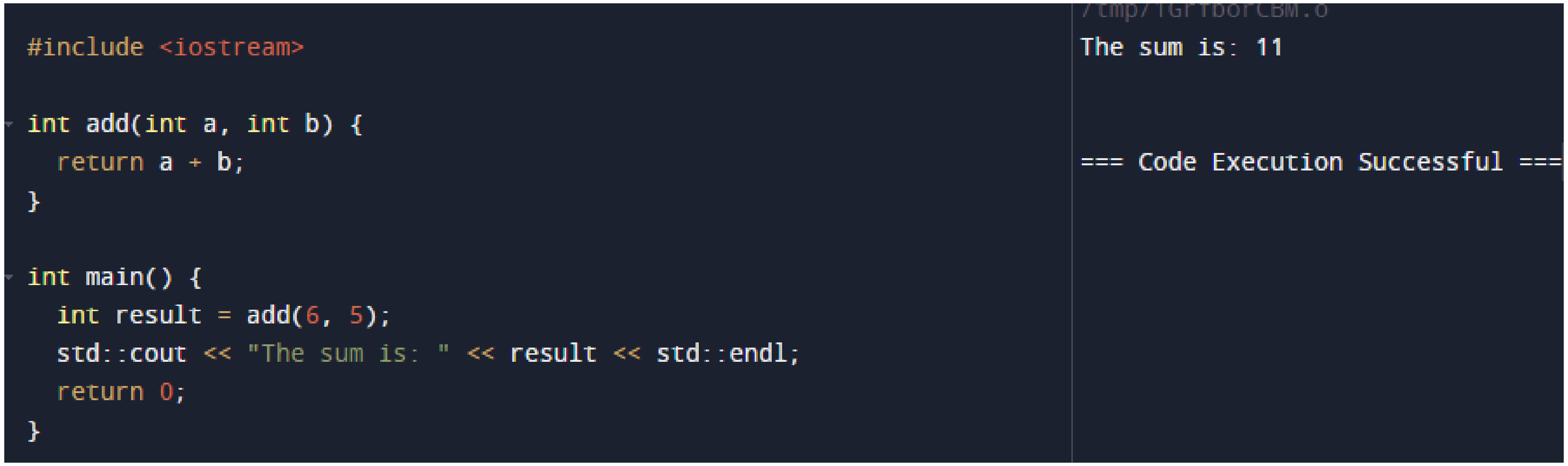
Benefits of using functions:

- **Modularity:** Breaks down code into smaller, manageable units.
- **Reusability:** Can be called multiple times from different parts of the program.
- **Abstraction:** Hides implementation details, making code easier to understand.
- **Efficiency:** Can improve performance by avoiding code duplication.



```
return_type function_name(parameters) {
  // function body
}
```

- **return_type:** The data type of the value returned by the function.
- **function_name:** The name used to identify the function.
- **parameters:** Optional variables passed to the function.
- **function body:** The code that performs the function's task.



```cpp
#include <iostream>

int add(int a, int b) {
  return a + b;
}

int main() {
  int result = add(6, 5);
  std::cout << "The sum is: " << result << std::endl;
  return 0;
}
```

```
/tmp/TGrfborCBM.o
The sum is: 11


=== Code Execution Successful ===
```

Call Types of a Function in C++

There are primarily two call types for functions in C++:

1. Call by Value

Mechanism: When a function is called by value, a copy of the actual argument is passed to the function.
Impact: Changes made to the parameter within the function do not affect the original argument in the calling function.

2. Call by Reference

Mechanism: When a function is called by reference, the address of the actual argument is passed to the function.
Impact: Changes made to the parameter within the function affect the original argument in the calling function.

- ❑ The & symbol before the parameter type in the function declaration indicates call by reference.
- ❑ Call by reference can be more efficient for large data structures as it avoids copying.
- ❑ However, it can also lead to unexpected side effects if not used carefully.

Beyond these basic types, there are also concepts like:

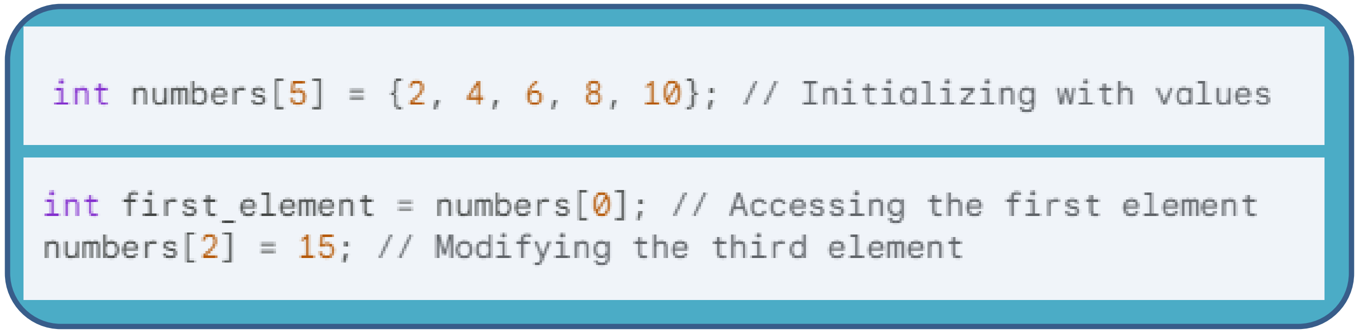
- **Call by Pointer:** Similar to call by reference, but using pointers explicitly.
- **Default Arguments:** Providing default values for function parameters.
- **Variable-length Argument Lists:** Using ... to handle a variable number of arguments (e.g., printf).

Arrays in C++

An array is a collection of similar data types stored in contiguous memory locations. This means that elements of the same type are placed next to each other in memory, and you can access them using an index.

data_type array_name[size];

- data_type: The type of elements the array will hold (e.g., int, float, char).
- array_name: The name of the
- array.size: The number of elements the array can hold.
- The size of an array is fixed once declared. You cannot change it during program execution.



```
int numbers[5] = {2, 4, 6, 8, 10}; // Initializing with values
```

```
int first_element = numbers[0]; // Accessing the first element
numbers[2] = 15; // Modifying the third element
```