# C++ EBook

By @Curious_.programmer

# Table of Contents

## Arrays and Pointers

- Declaring Arrays
- Multidimensional Arrays
- Pointers and Addresses
- Dynamic Memory Allocation

## Classes and Objects

- Defining Classes
- Access Control
- Constructors and Destructors
- Inheritance
- Polymorphism

## Exception Handling

- Try-catch Blocks
- Throwing Exceptions
- Standard Exception Classes

## Templates

- Function Templates
- Class Templates
- Template Specialization

## Standard Template Library (STL)

- Containers
- Algorithms
- Iterators

## File Handling

- File Input and Output

**PDF File Uploaded On Telegram ( Link In Bio)**

- File Pointers
- Binary Files

## *Advanced Topics*

- Lambda Expressions
- Move Semantics
- Smart Pointers
- Multithreading

# Introduction to C++:

- C++ is a high-level, general-purpose programming language that was developed in the early 1980s by Bjarne Stroustrup at Bell Labs.
- It was designed as an extension of the C programming language with additional features such as classes, objects, inheritance, and polymorphism.
- C++ is widely used for developing applications ranging from operating systems and device drivers to games, financial software, and scientific simulations.

## *History of C++:*

- C++ was developed by Bjarne Stroustrup at Bell Labs in 1983 as an extension of the C language.
- The first commercial implementation of C++ was released in 1985 by AT&T.
- The first ISO standard for C++ was published in 1998, and subsequent updates have been released in 2003, 2011, 2014, 2017, and 2020.

## *C++ Standards:*

- The C++ standards define the syntax, semantics, and library specifications for the language.
- The latest C++ standard is C++20, which was published in 2020 and includes several new features such as modules, concepts, coroutines, and ranges.
- C++ standards are developed and maintained by the ISO C++ committee, which includes experts from industry, academia, and standards organizations.

## Basic Structure of a C++ Program:

- A C++ program typically consists of one or more source files that are compiled into object files and linked together to create an executable program.
- The basic structure of a C++ program includes a header file that defines any necessary classes or functions, a main function that serves as the entry point for the program, and any necessary helper functions or classes.
- C++ programs can also include preprocessor directives, comments, and other features that help to organize and document the code.

## Variables and Data Types:

- Variables are named storage locations in a program that can hold data of a certain type.
- C++ supports several types of data, including integers, floating-point numbers, characters, Boolean values, and user-defined types such as classes and structures.
- Before using a variable, it must be declared with a data type and optionally initialized with a value.

## Declaring Variables:

- To declare a variable in C++, you must specify its data type followed by its
- For example, to declare an integer variable named "age", you would write "int age;".Variables can also be initialized with an initial value using the assignment operator, such as "int age = 30;".

## Data Types:

- C++ supports several built-in data types, including:
- Integers, such as "int" and "long".
- Floating-point numbers, such as "float" and "double".
- Characters, such as "char".
- Boolean values, which are represented by the "bool" data type.
- C++ also supports user-defined data types such as classes and structures.

## Constants:

- Constants are values that cannot be changed during program execution.
- C++ supports two types of constants: literal constants and symbolic constants.
- Literal constants are values that are explicitly written into the program code, such as "5" or "'A'".
- Symbolic constants are defined using the "const" keyword and can be given a meaningful name, such as "const double PI = 3.14159;".

## Operators and Expressions:

- Operators are symbols that perform operations on one or more operands to produce a result.
- C++ supports several types of operators, including arithmetic operators, comparison operators, logical operators, and bitwise operators.
- Expressions are combinations of operators and operands that produce a value.

### *Arithmetic Operators:*

- Arithmetic operators are used to perform mathematical calculations.
- C++ supports several arithmetic operators, including addition (+), subtraction (-), multiplication (*), division (/), and modulus (%).
- For example, the expression "int result = 5 + 3 * 2;" would assign the value 11 to the variable "result".

## Comparison Operators:

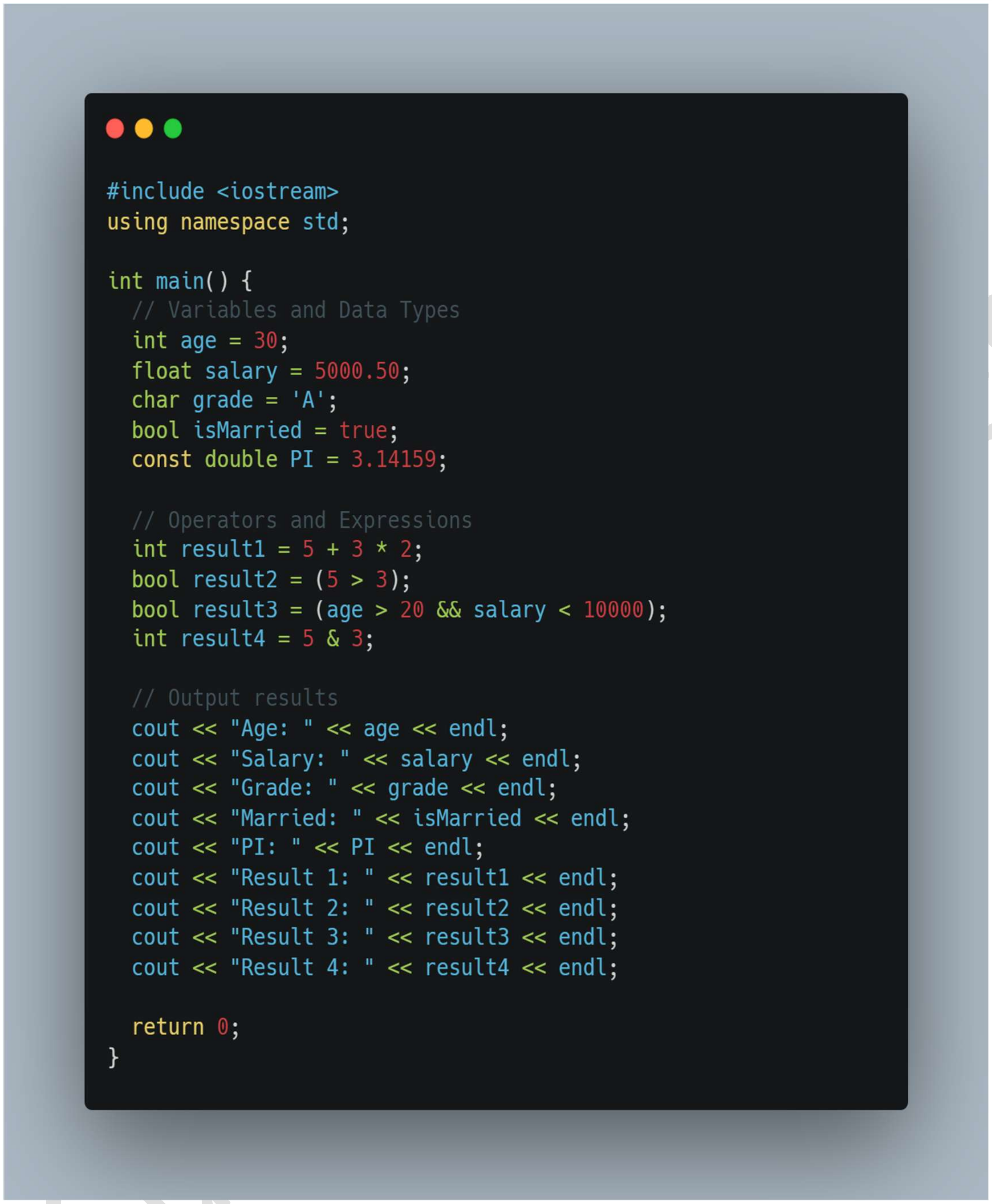
- Comparison operators are used to compare values and produce a Boolean result.
- C++ supports several comparison operators, including equals (==), not equals (!=), less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=).
- For example, the expression "bool result = (5 > 3);" would assign the value "true" to the variable "result".

## Logical Operators:

- Logical operators are used to combine Boolean values and produce a Boolean result.
- C++ supports several logical operators, including AND (&&), OR (||), and NOT (!).
- For example, the expression "bool result = (x > 0 && y < 0);" would assign the value "true" to the variable "result" if both conditions are true.

## Bitwise Operators:

- Bitwise operators are used to perform operations on individual bits in a binary number.
- C++ supports several bitwise operators, including bitwise AND (&), bitwise OR (|), bitwise XOR (^), bitwise NOT (~), left shift (<<), and right shift (>>).
- For example, the expression "int result = 5 & 3;" would assign the value 1 to the variable "result", because the bitwise AND operation produces a binary value of 000001.



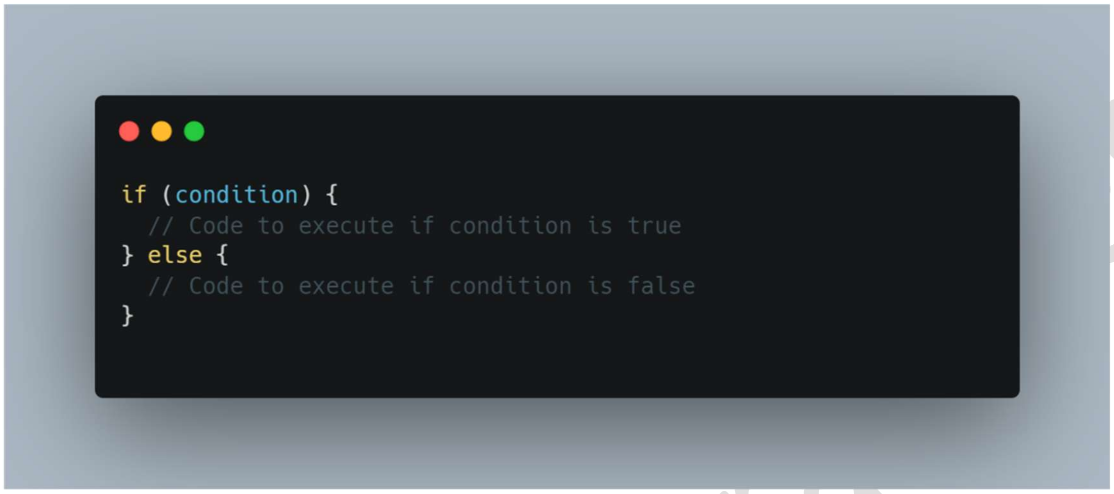
```cpp
#include <iostream>
using namespace std;

int main() {
  // Variables and Data Types
  int age = 30;
  float salary = 5000.50;
  char grade = 'A';
  bool isMarried = true;
  const double PI = 3.14159;

  // Operators and Expressions
  int result1 = 5 + 3 * 2;
  bool result2 = (5 > 3);
  bool result3 = (age > 20 && salary < 10000);
  int result4 = 5 & 3;

  // Output results
  cout << "Age: " << age << endl;
  cout << "Salary: " << salary << endl;
  cout << "Grade: " << grade << endl;
  cout << "Married: " << isMarried << endl;
  cout << "PI: " << PI << endl;
  cout << "Result 1: " << result1 << endl;
  cout << "Result 2: " << result2 << endl;
  cout << "Result 3: " << result3 << endl;
  cout << "Result 4: " << result4 << endl;

  return 0;
}
```

## Control Flow Statements:

Control flow statements are used to control the flow of a program's execution. C++ supports several types of control flow statements, including if-else statements, switch statements, loops, and jump statements.

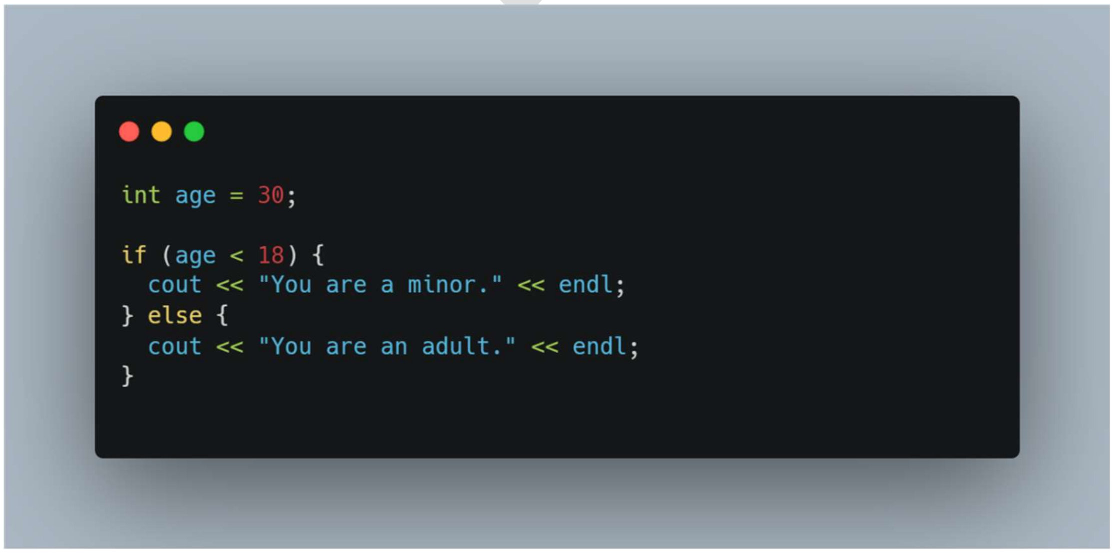
## If-else Statements:

If-else statements are used to execute code based on a condition. The syntax for an if-else statement is as follows:



```
if (condition) {
  // Code to execute if condition is true
} else {
  // Code to execute if condition is false
}
```

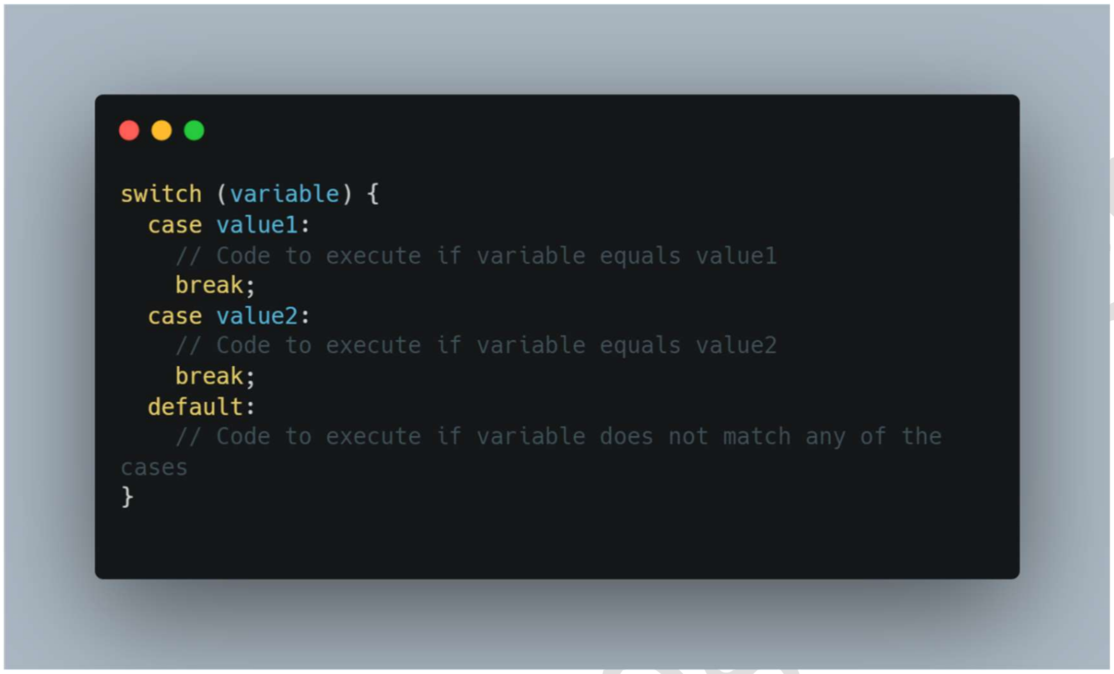
For example:



```
int age = 30;

if (age < 18) {
  cout << "You are a minor." << endl;
} else {
  cout << "You are an adult." << endl;
}
```

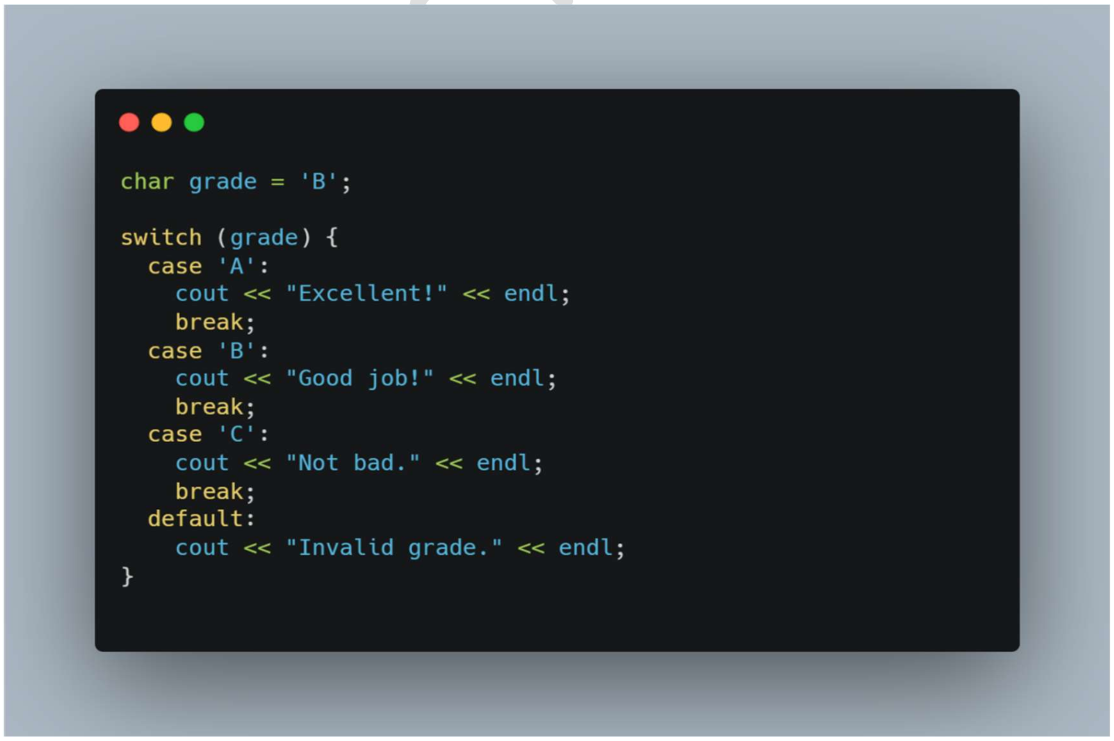
## Switch Statements:

Switch statements are used to execute different blocks of code based on the value of a variable. The syntax for a switch statement is as follows:



```
switch (variable) {
  case value1:
    // Code to execute if variable equals value1
    break;
  case value2:
    // Code to execute if variable equals value2
    break;
  default:
    // Code to execute if variable does not match any of the
cases
}
```

For example:

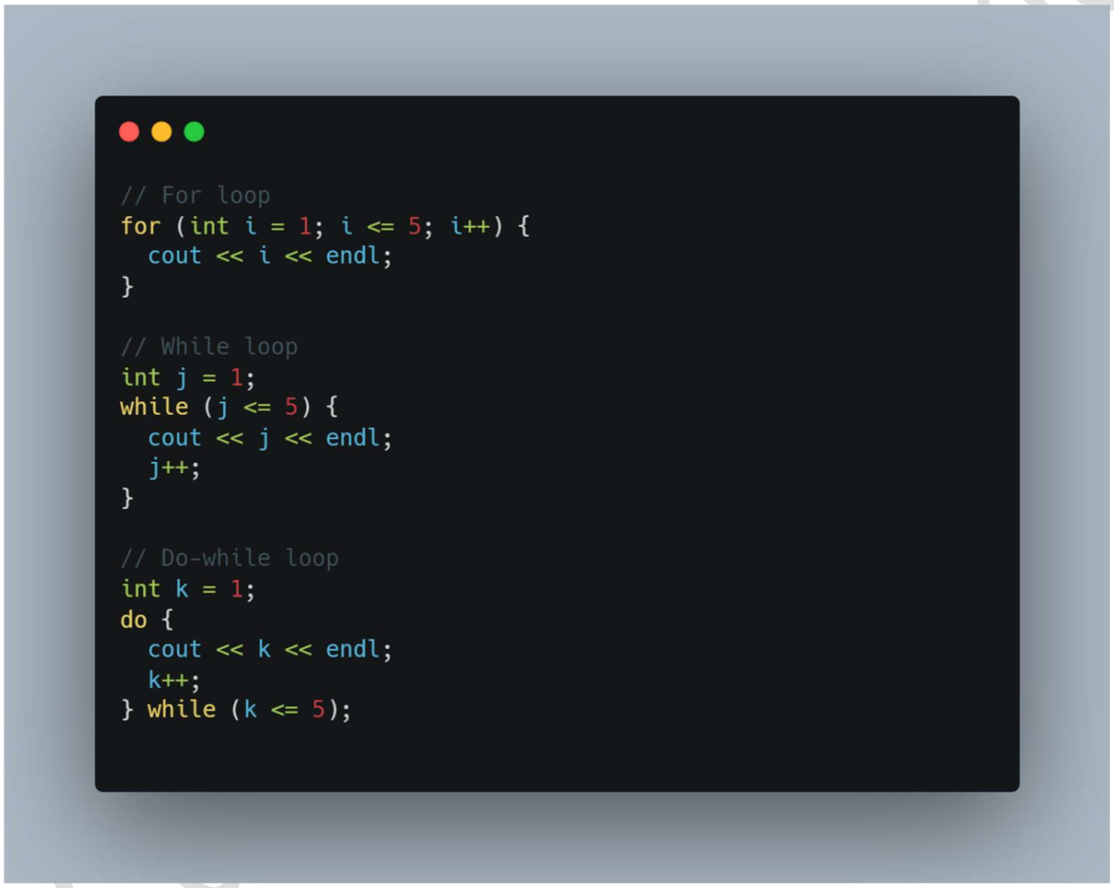


```
char grade = 'B';

switch (grade) {
  case 'A':
    cout << "Excellent!" << endl;
    break;
  case 'B':
    cout << "Good job!" << endl;
    break;
  case 'C':
    cout << "Not bad." << endl;
    break;
  default:
    cout << "Invalid grade." << endl;
}
```

## Loops:

Loops are used to execute a block of code repeatedly. C++ supports several types of loops, including for loops, while loops, and do-while loops.

For example:

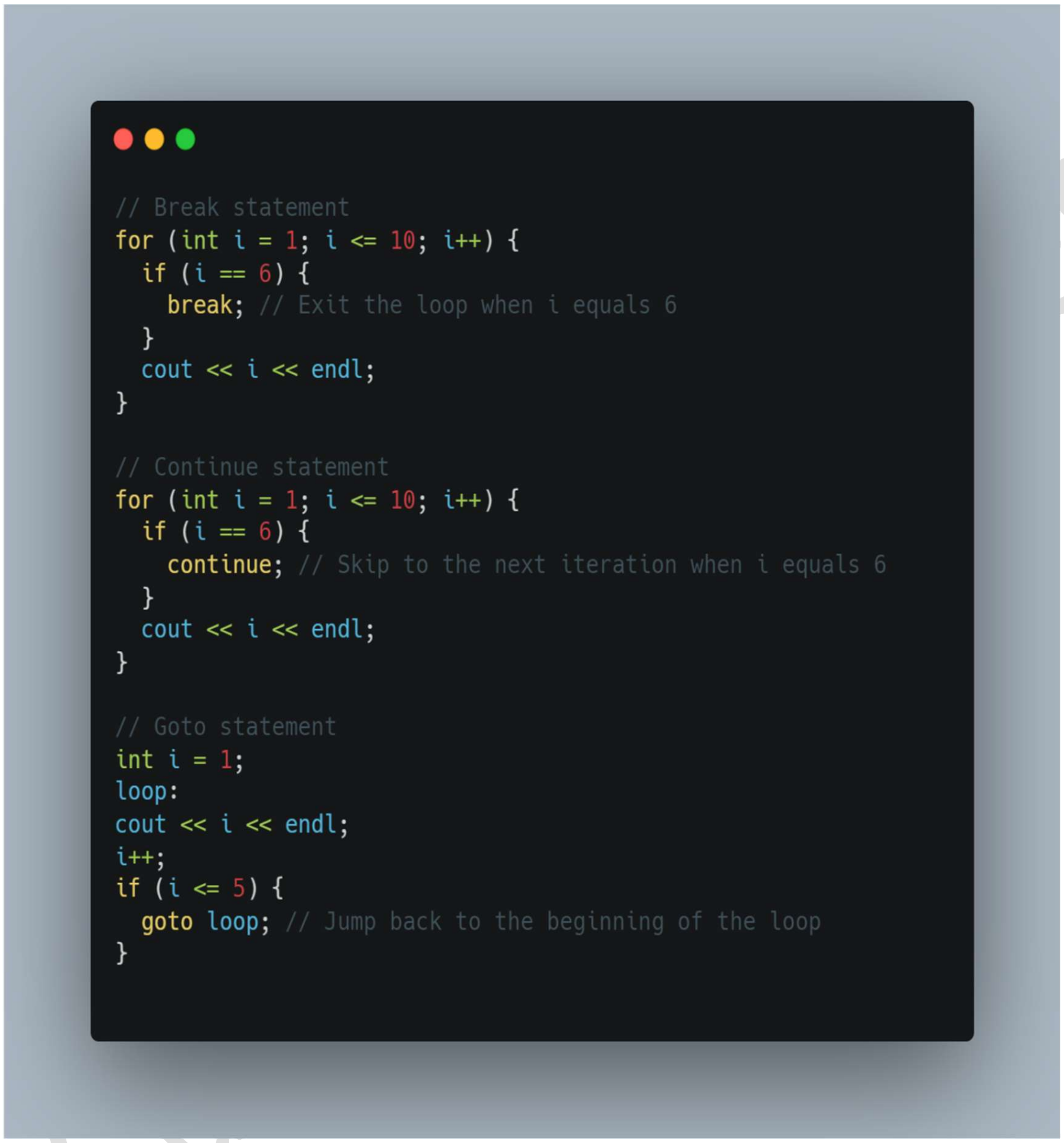


```
// For loop
for (int i = 1; i <= 5; i++) {
  cout << i << endl;
}

// While loop
int j = 1;
while (j <= 5) {
  cout << j << endl;
  j++;
}

// Do-while loop
int k = 1;
do {
  cout << k << endl;
  k++;
} while (k <= 5);
```

## *Jump Statements:*

Jump statements are used to transfer control to a different part of the program. C++ supports several types of jump statements, including break, continue, and goto.

For example:



```cpp
// Break statement
for (int i = 1; i <= 10; i++) {
  if (i == 6) {
    break; // Exit the loop when i equals 6
  }
  cout << i << endl;
}

// Continue statement
for (int i = 1; i <= 10; i++) {
  if (i == 6) {
    continue; // Skip to the next iteration when i equals 6
  }
  cout << i << endl;
}

// Goto statement
int i = 1;
loop:
cout << i << endl;
i++;
if (i <= 5) {
  goto loop; // Jump back to the beginning of the loop
}
```

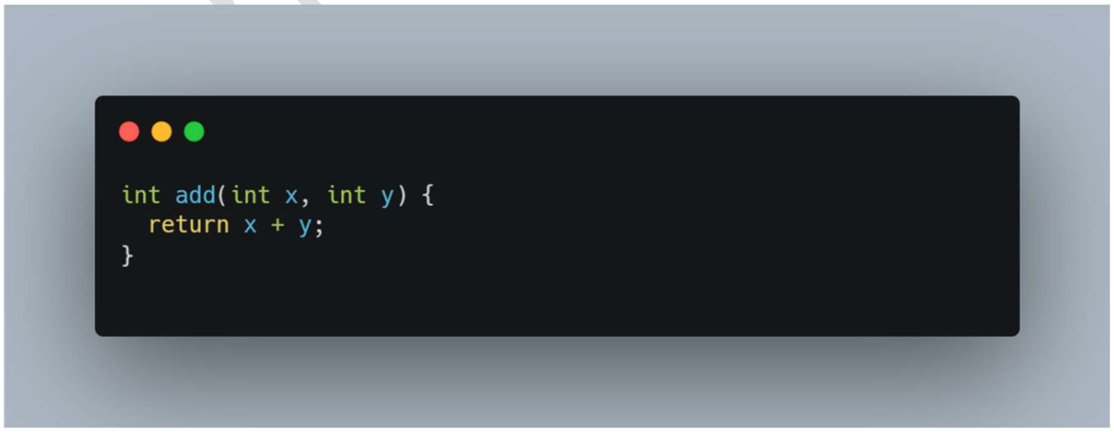
## Functions:

In C++, a function is a group of statements that together perform a task. Every C++ program has at least one function, which is the main() function.

## Defining Functions:

A function definition consists of a function header and a function body. The function header specifies the function's name, return type, and parameters. The function body contains the statements that perform the function's task.

Here is an example of a function definition:



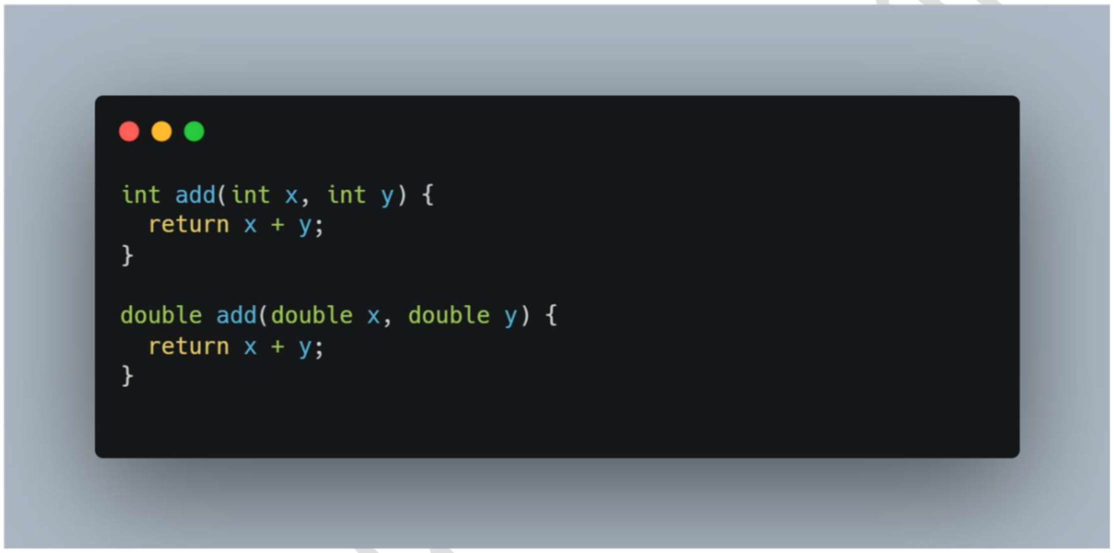
```cpp
int add(int x, int y) {
  return x + y;
}
```

This function is named add, takes two integer parameters x and y, and returns an integer value that is the sum of x and y.

## Function Overloading:

Function overloading is the ability to define multiple functions with the same name but with different parameter types and/or numbers. When a function is called, the correct version of the function is selected based on the parameters provided.

Here is an example of function overloading:



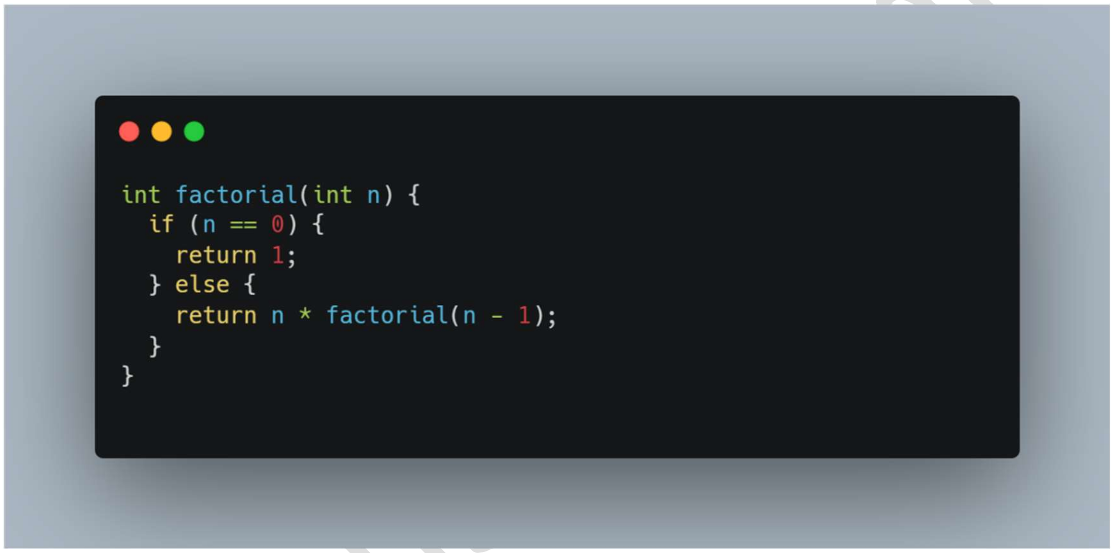
```
int add(int x, int y) {
  return x + y;
}

double add(double x, double y) {
  return x + y;
}
```

This code defines two functions named add. The first function takes two integer parameters and returns an integer value, while the second function takes two double parameters and returns a double value.

## Recursion:

Recursion is a technique where a function calls itself to perform a task. Recursion can be used to solve problems that can be broken down into smaller sub-problems.

Here is an example of a recursive function:



```
int factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

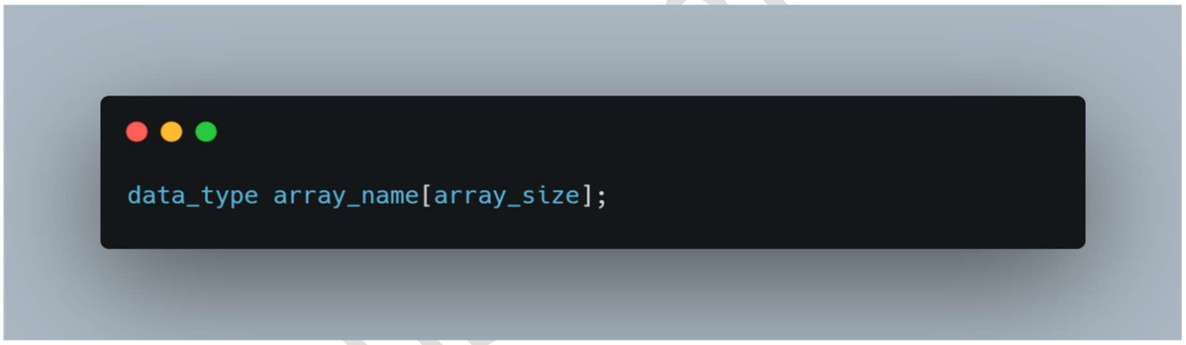
This function calculates the factorial of a number n using recursion. The base case is when n is equal to 0, in which case the function returns 1. Otherwise, the function calls itself with n - 1 as the argument and multiplies the result by n.

## Arrays and Pointers:

Arrays and pointers are fundamental concepts in C and C++ programming languages. In these languages, arrays are used to store a collection of similar data types, and pointers are used to store memory addresses.

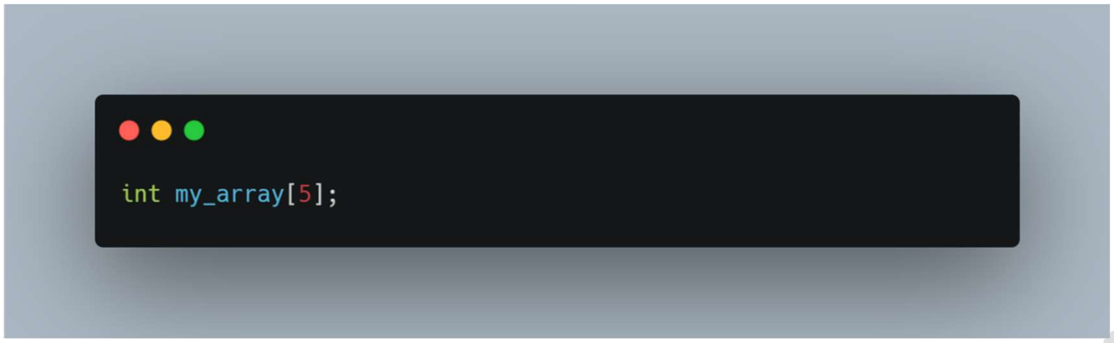
- Declaring Arrays:

In C and C++, arrays can be declared using the following syntax:



```
data_type array_name[array_size];
```

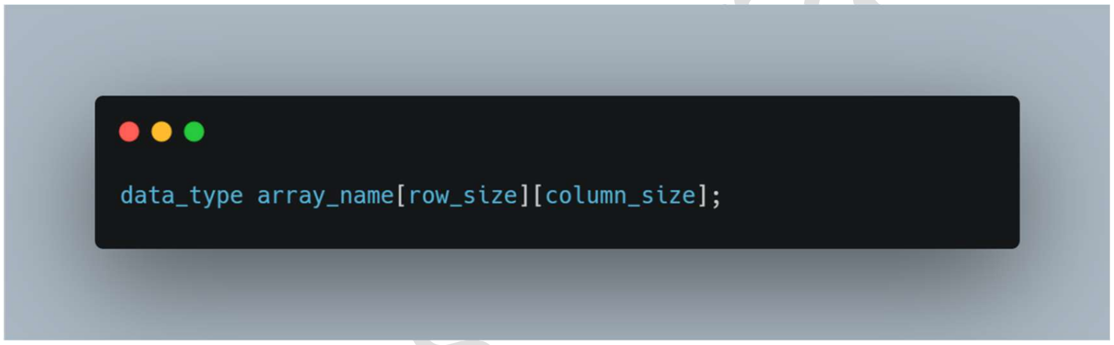
where data_type is the data type of the elements, array_name is the name of the array, and array_size is the number of elements in the array. For example, to declare an array of integers with 5 elements, you would use the following code:



```
int my_array[5];
```

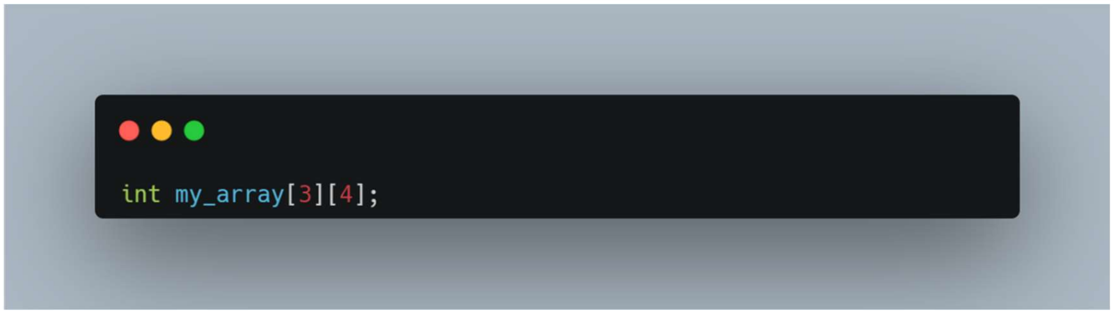
- Multidimensional Arrays:

A multidimensional array is an array that contains more than one dimension. In other words, it is an array of arrays. The syntax for declaring a two-dimensional array in C++ is as follows:



```
data_type array_name[row_size][column_size];
```

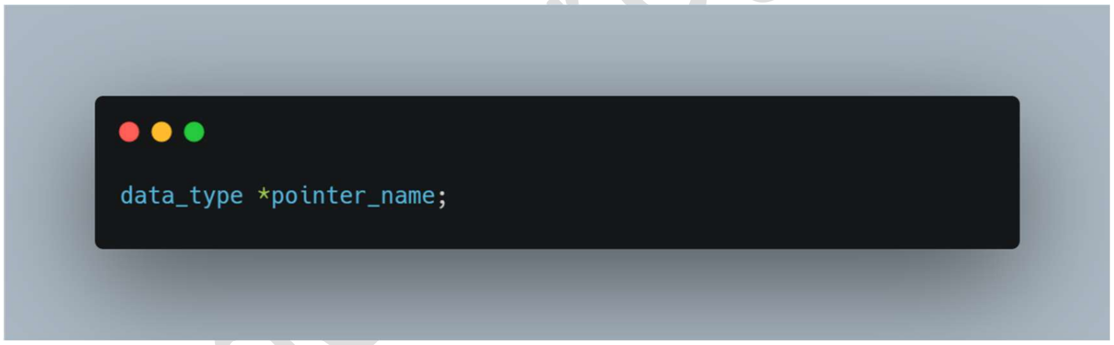
where data_type is the data type of the elements, array_name is the name of the array, row_size is the number of rows, and column_size is the number of columns. For example, to declare a two-dimensional array of integers with 3 rows and 4 columns, you would use the following code:



```
int my_array[3][4];
```

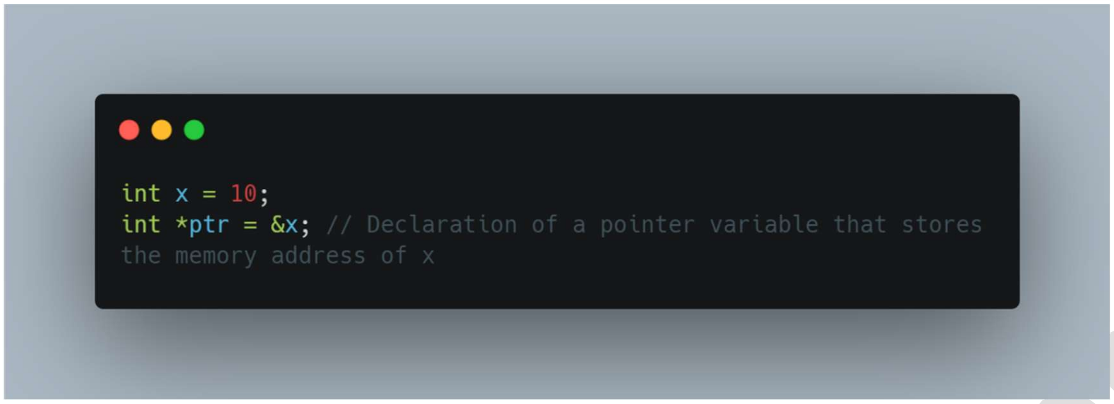
- Pointers and Addresses:

A pointer is a variable that stores the memory address of another variable. Pointers are useful for manipulating memory directly and can be used to pass values by reference. The syntax for declaring a pointer in C++ is as follows:



```
data_type *pointer_name;
```

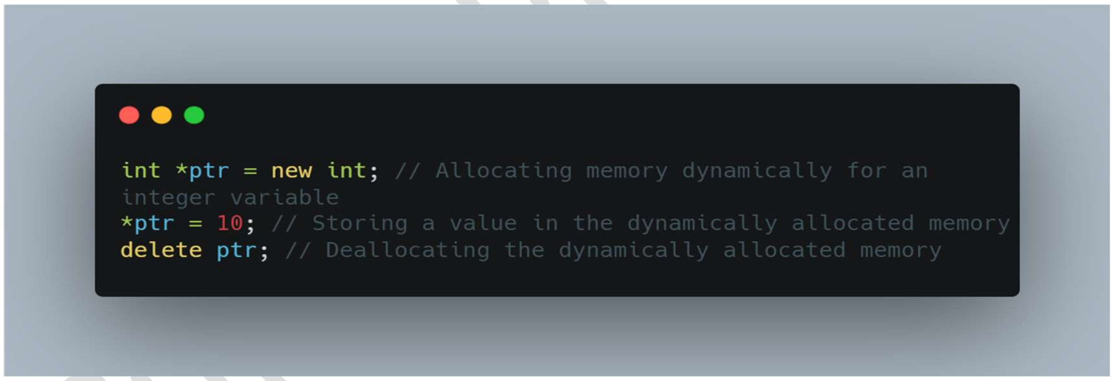
Example:


```cpp
int x = 10;
int *ptr = &x; // Declaration of a pointer variable that stores
the memory address of x
```

- Dynamic Memory Allocation:

Dynamic memory allocation is a mechanism that allows us to allocate memory at runtime. We can allocate memory dynamically in C++ using the new keyword.

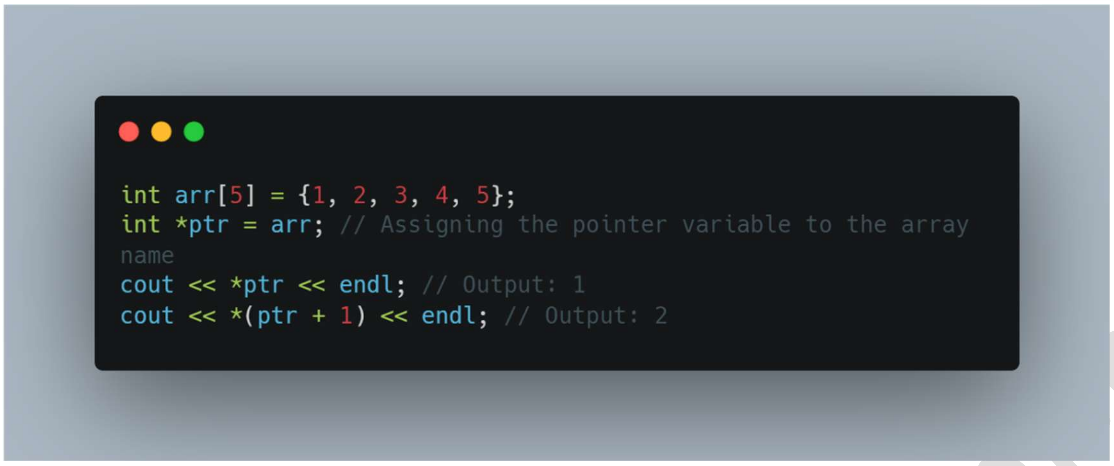
Example:


```cpp
int *ptr = new int; // Allocating memory dynamically for an
integer variable
*ptr = 10; // Storing a value in the dynamically allocated memory
delete ptr; // Deallocating the dynamically allocated memory
```

- Arrays and Pointers:

In C++, an array name is a constant pointer to the first element of the array. We can use a pointer to access the elements of an array.

Example:

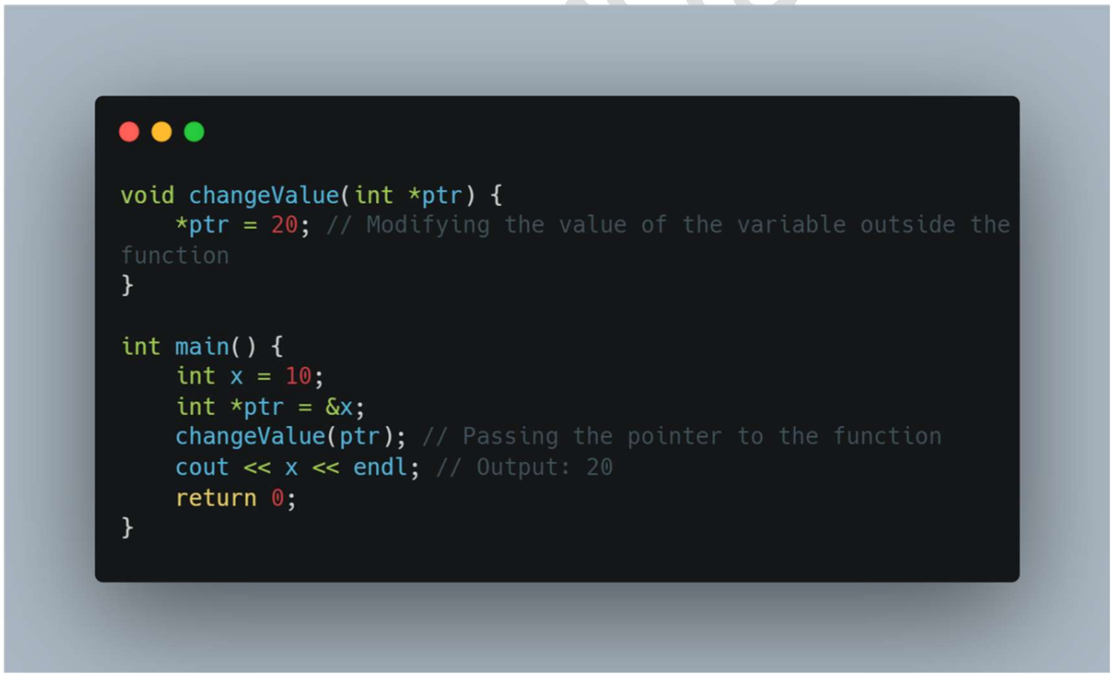


```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr; // Assigning the pointer variable to the array
name
cout << *ptr << endl; // Output: 1
cout << *(ptr + 1) << endl; // Output: 2
```

- Pointers and Functions:

We can pass pointers to functions as arguments. This allows us to modify the original values of the variables outside the function.

Example:

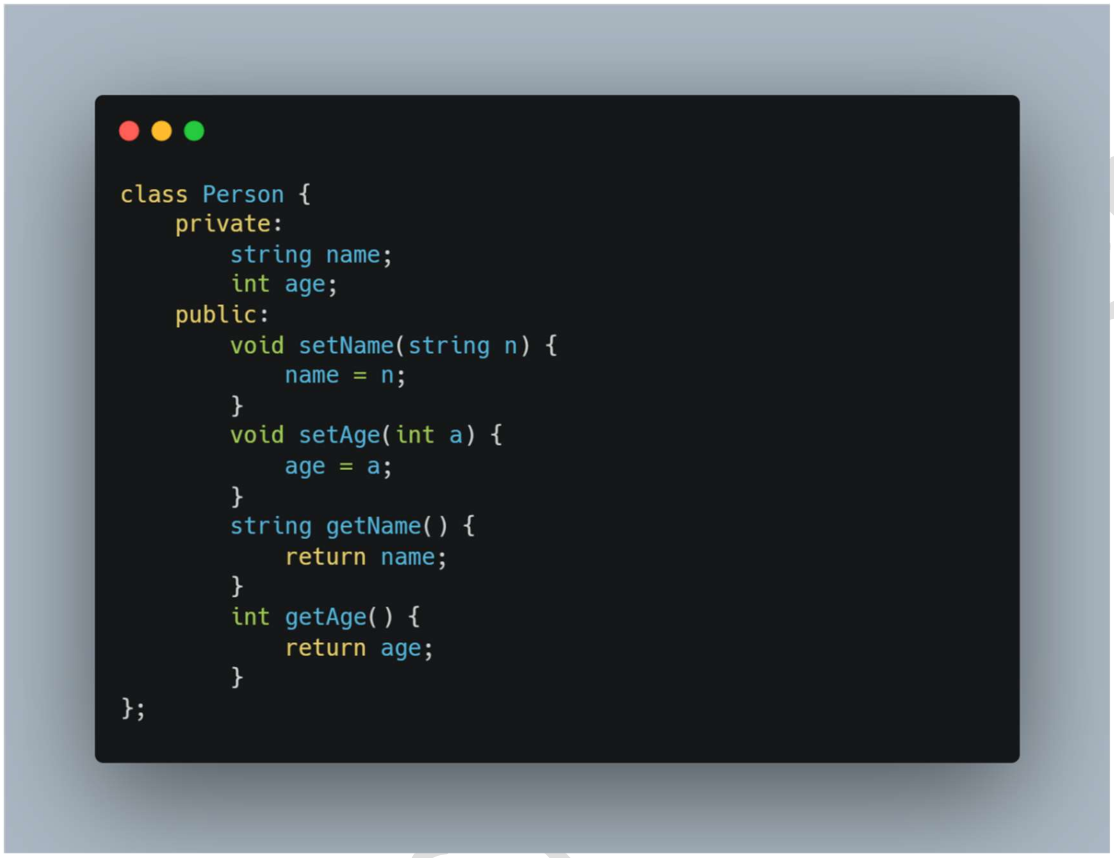


```
void changeValue(int *ptr) {
    *ptr = 20; // Modifying the value of the variable outside the
function
}

int main() {
    int x = 10;
    int *ptr = &x;
    changeValue(ptr); // Passing the pointer to the function
    cout << x << endl; // Output: 20
    return 0;
}
```

◆ Classes and Objects :

- Defining Classes:

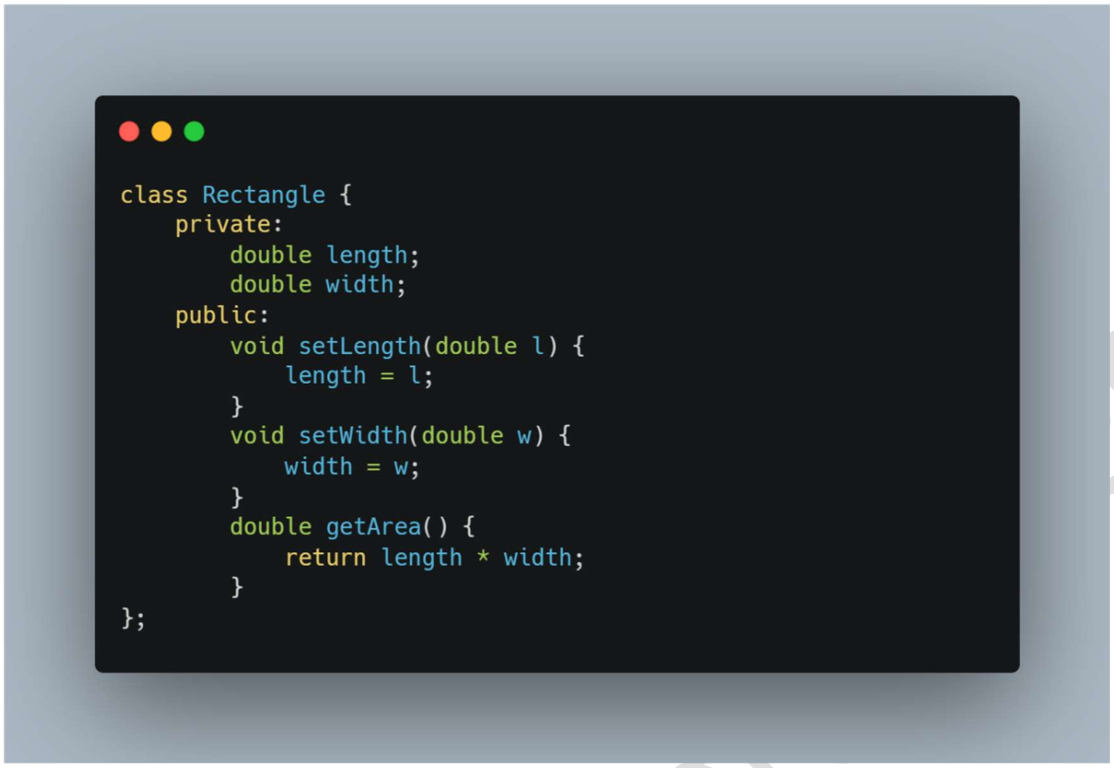
Classes are user-defined data types that can hold data and functions. They allow us to encapsulate related data and behavior into a single entity. For example:



```cpp
class Person {
    private:
        string name;
        int age;
    public:
        void setName(string n) {
            name = n;
        }
        void setAge(int a) {
            age = a;
        }
        string getName() {
            return name;
        }
        int getAge() {
            return age;
        }
};
```

- Access Control:

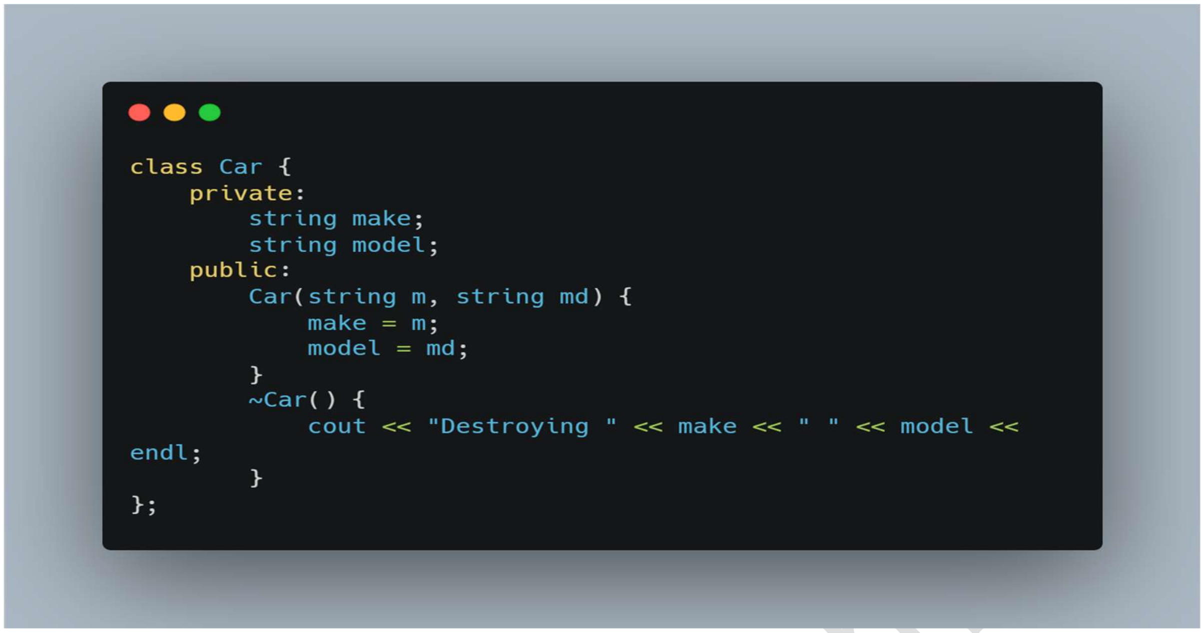
Access control specifies which data and functions of a class are accessible to the outside world. There are three access specifiers: private, protected, and public. For example:



```
class Rectangle {
    private:
        double length;
        double width;
    public:
        void setLength(double l) {
            length = l;
        }
        void setWidth(double w) {
            width = w;
        }
        double getArea() {
            return length * width;
        }
};
```

## Constructors and Destructors:

Constructors are special member functions that are called when an object of a class is created. They initialize the object's data members. Destructors are special member functions that are called when an object of a class is destroyed. They clean up any resources that were allocated during the object's lifetime. For example:

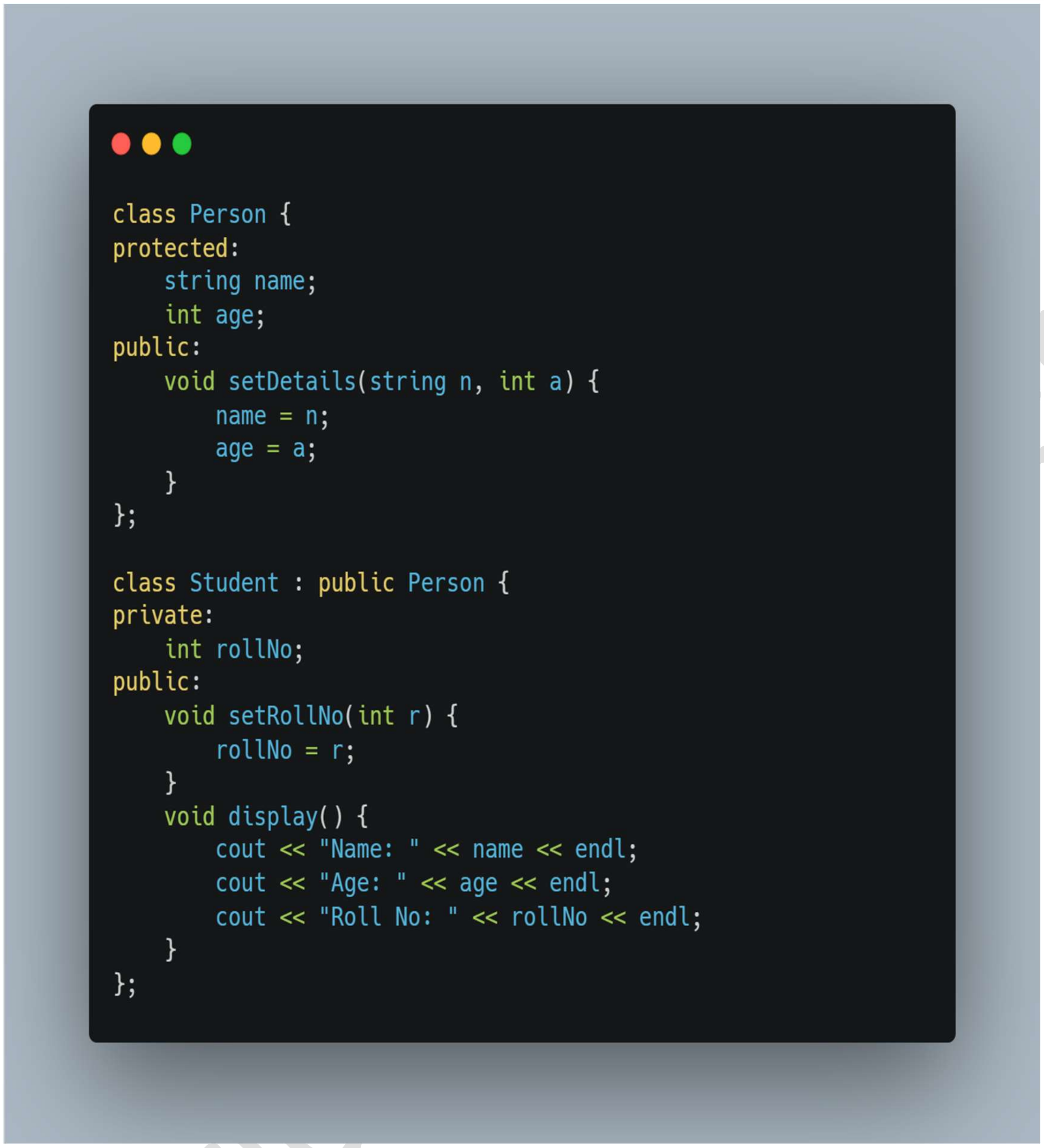


```
class Car {
    private:
        string make;
        string model;
    public:
        Car(string m, string md) {
            make = m;
            model = md;
        }
        ~Car() {
            cout << "Destroying " << make << " " << model <<
endl;
        }
};
```

## Inheritance:

In C++, inheritance is a mechanism that allows a new class to be based on an existing class, inheriting its members and properties. This helps to avoid code repetition and makes the code more organized and maintainable.

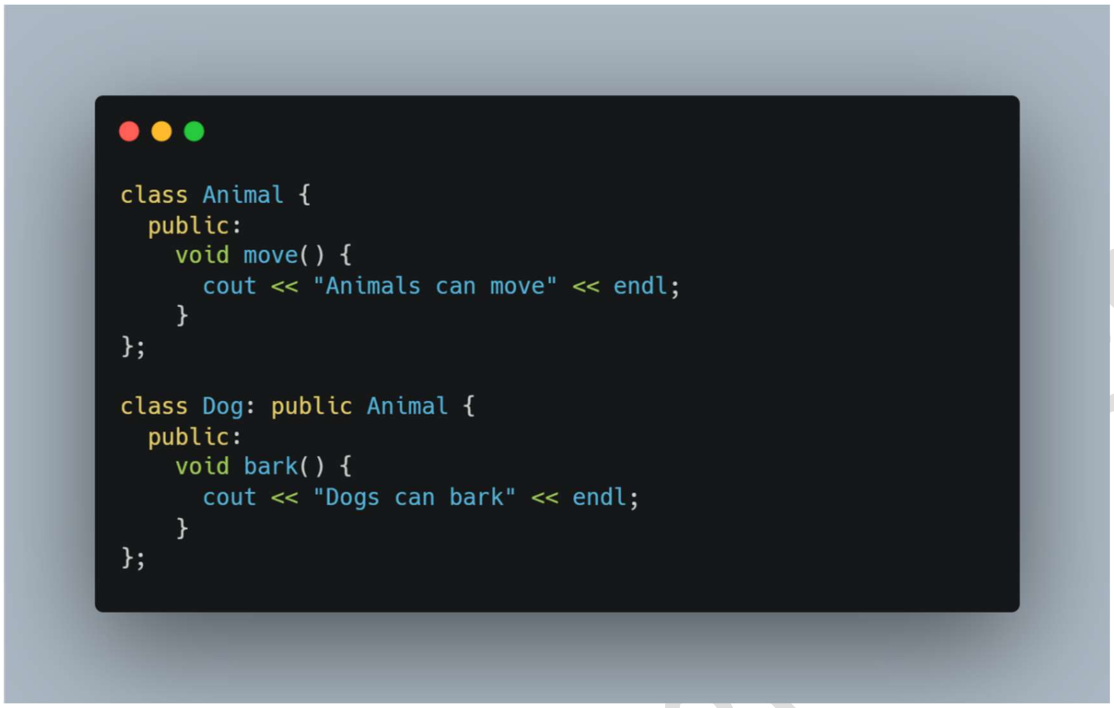
Here's an example of a base class "Person" and a derived class "Student" that inherits from Person:



```
class Person {
protected:
    string name;
    int age;
public:
    void setDetails(string n, int a) {
        name = n;
        age = a;
    }
};

class Student : public Person {
private:
    int rollNo;
public:
    void setRollNo(int r) {
        rollNo = r;
    }
    void display() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Roll No: " << rollNo << endl;
    }
};
```

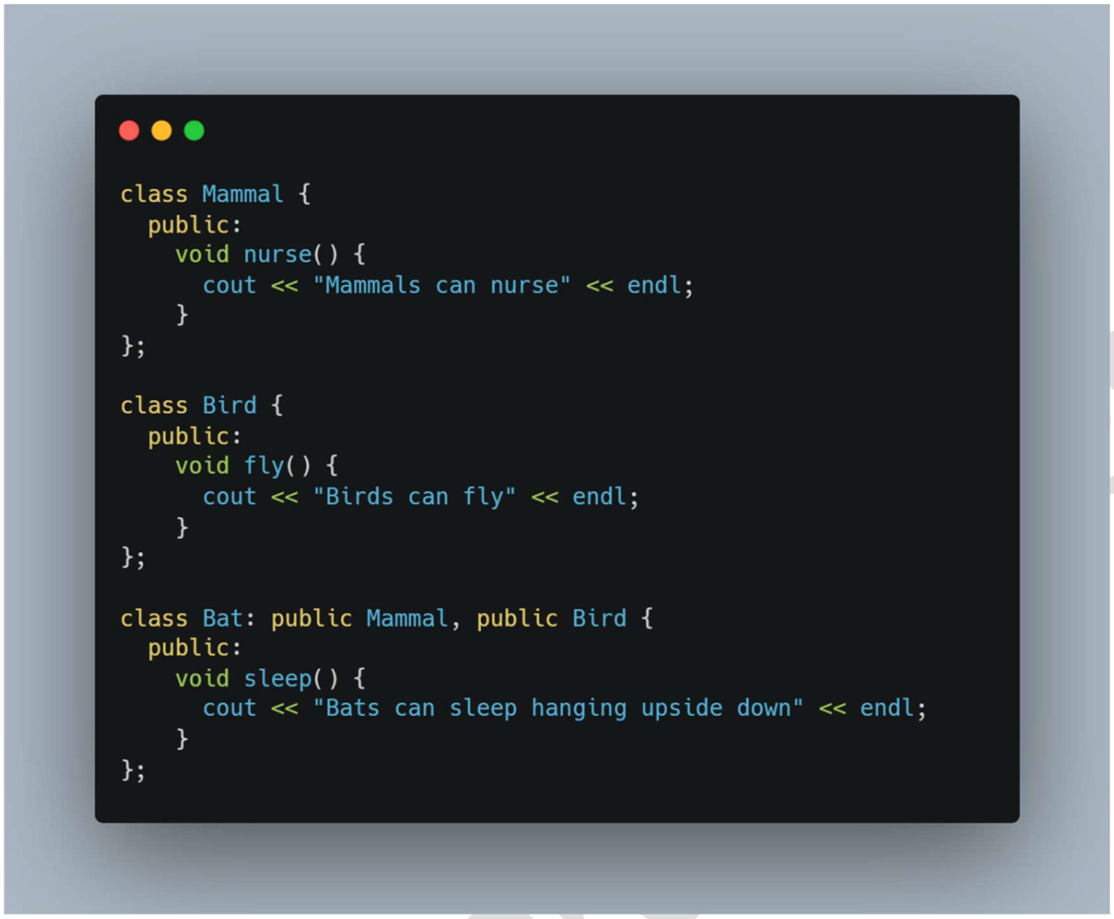
**five types of inheritance:**

- Single inheritance: A derived class inherits from a single base class. For example:



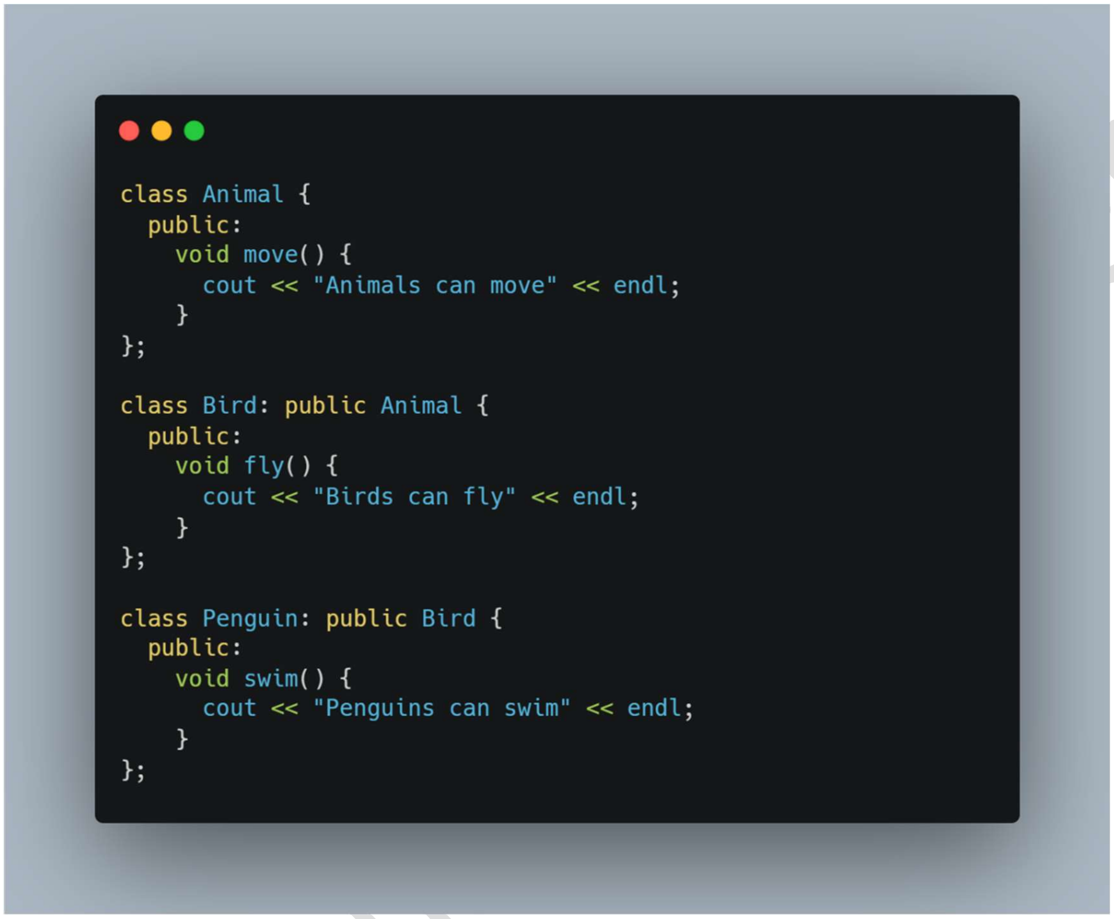
```
class Animal {
  public:
    void move() {
      cout << "Animals can move" << endl;
    }
};

class Dog: public Animal {
  public:
    void bark() {
      cout << "Dogs can bark" << endl;
    }
};
```

- Multiple inheritance: A derived class inherits from multiple base classes. For example:



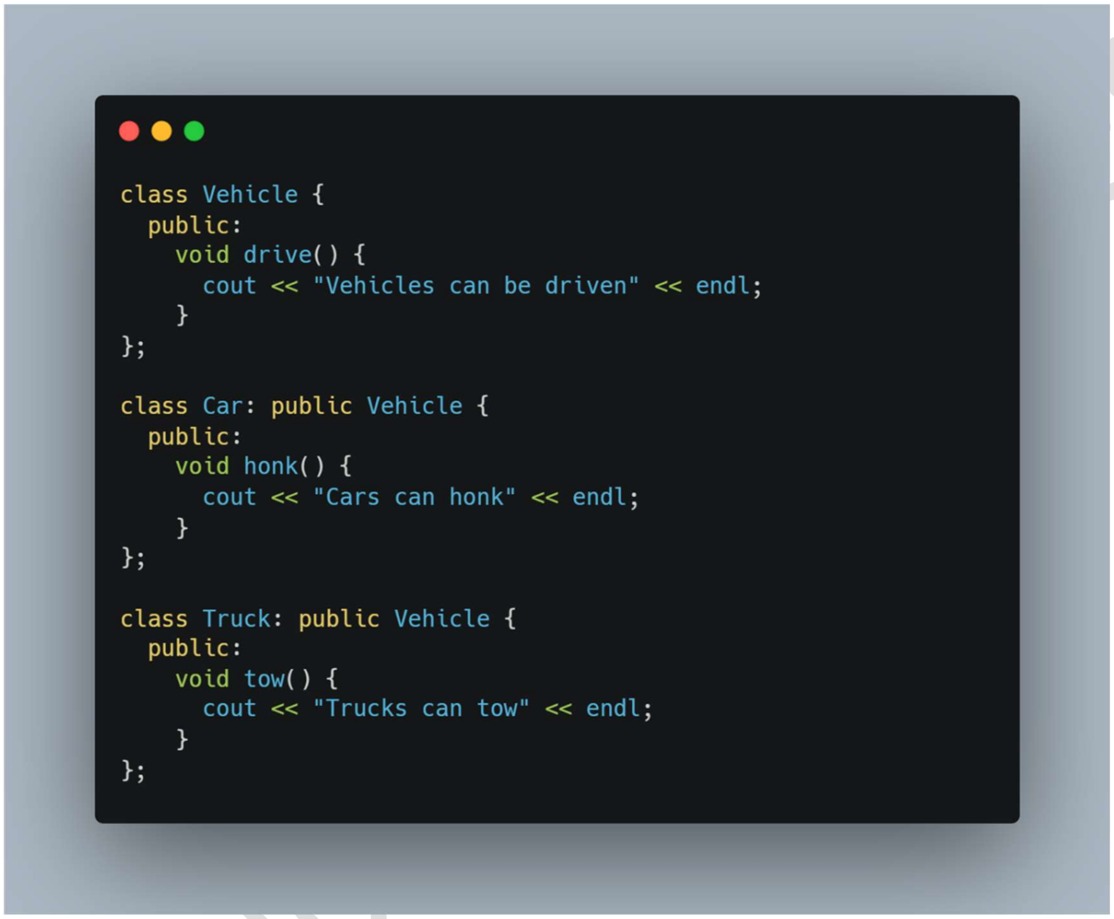
```cpp
class Mammal {
  public:
    void nurse() {
      cout << "Mammals can nurse" << endl;
    }
};

class Bird {
  public:
    void fly() {
      cout << "Birds can fly" << endl;
    }
};

class Bat: public Mammal, public Bird {
  public:
    void sleep() {
      cout << "Bats can sleep hanging upside down" << endl;
    }
};
```

- Multilevel inheritance: A derived class is created from another derived class. For example:



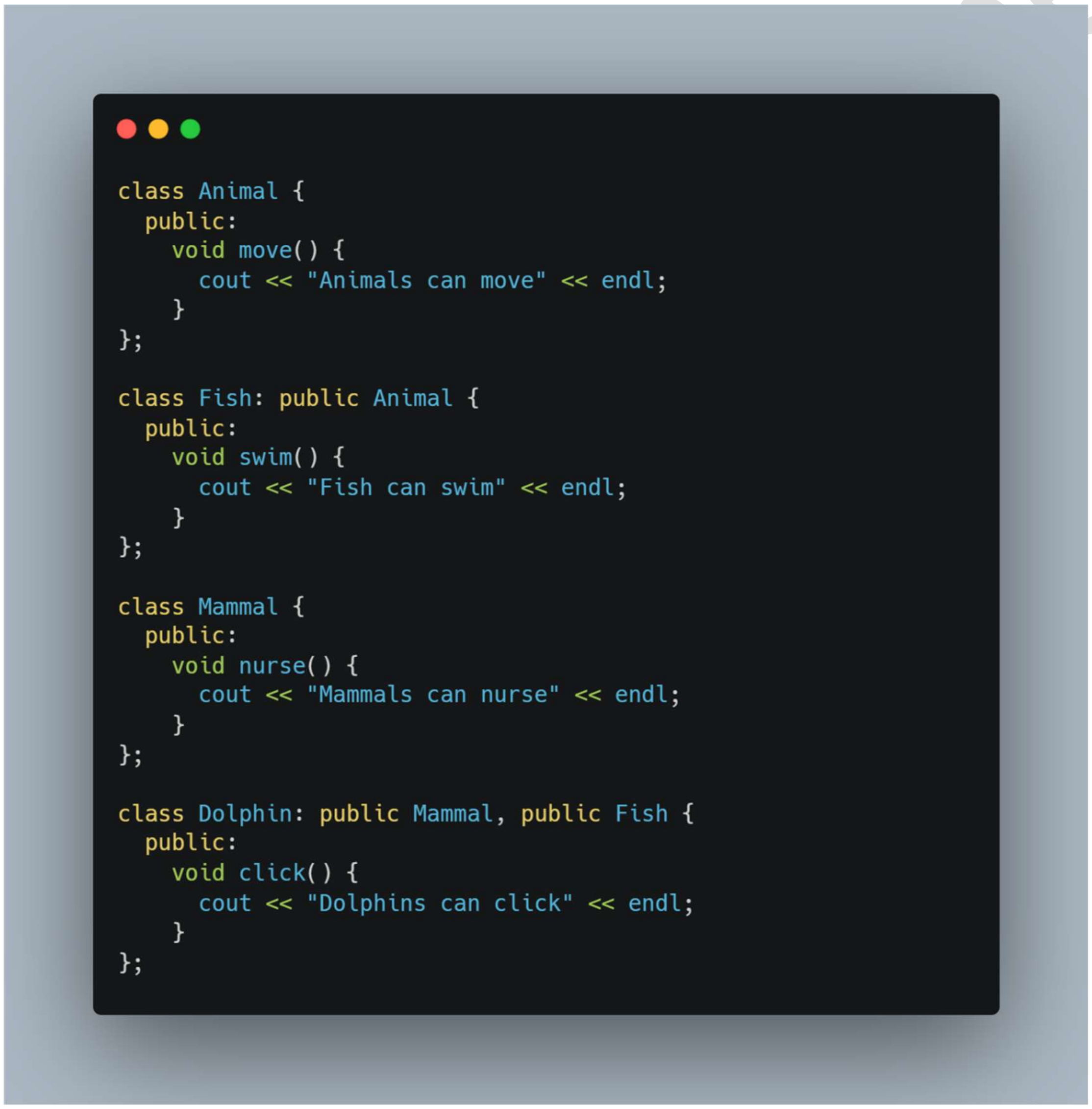
```
class Animal {
  public:
    void move() {
      cout << "Animals can move" << endl;
    }
};

class Bird: public Animal {
  public:
    void fly() {
      cout << "Birds can fly" << endl;
    }
};

class Penguin: public Bird {
  public:
    void swim() {
      cout << "Penguins can swim" << endl;
    }
};
```

- Hierarchical inheritance: Multiple derived classes are created from a single base class. For example:



```
class Vehicle {
  public:
    void drive() {
      cout << "Vehicles can be driven" << endl;
    }
};

class Car: public Vehicle {
  public:
    void honk() {
      cout << "Cars can honk" << endl;
    }
};

class Truck: public Vehicle {
  public:
    void tow() {
      cout << "Trucks can tow" << endl;
    }
};
```

- Hybrid inheritance: A combination of multiple inheritance and multilevel inheritance. For example:

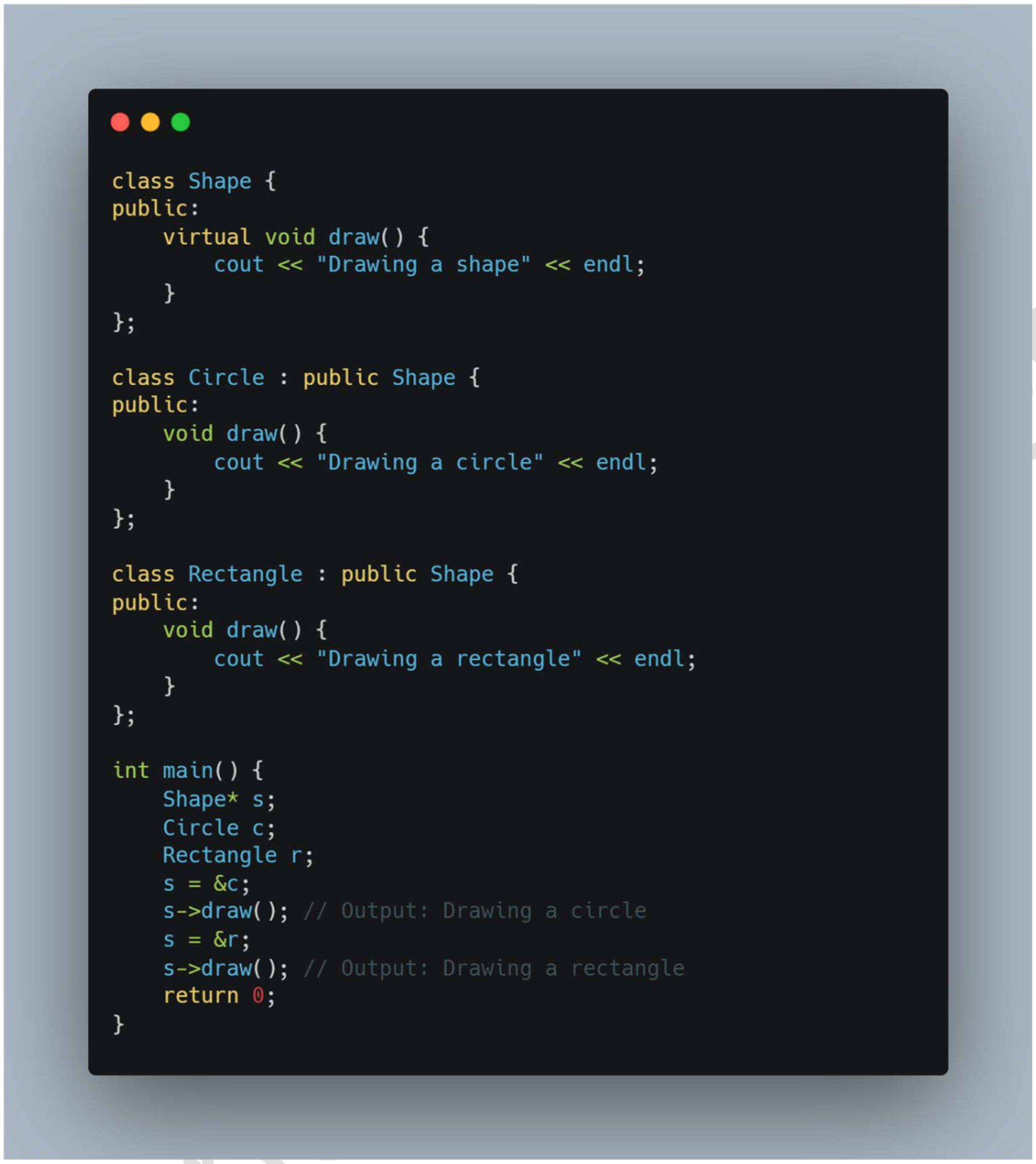


```
class Animal {
  public:
    void move() {
      cout << "Animals can move" << endl;
    }
};

class Fish: public Animal {
  public:
    void swim() {
      cout << "Fish can swim" << endl;
    }
};

class Mammal {
  public:
    void nurse() {
      cout << "Mammals can nurse" << endl;
    }
};

class Dolphin: public Mammal, public Fish {
  public:
    void click() {
      cout << "Dolphins can click" << endl;
    }
};
```

## Polymorphism:

In C++, polymorphism is the ability of an object to take on many forms. This is achieved through function overloading, operator overloading, and virtual functions.

Here's an example of a class hierarchy that uses virtual functions for runtime polymorphism:



```cpp
class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape" << endl;
    }
};

class Circle : public Shape {
public:
    void draw() {
        cout << "Drawing a circle" << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() {
        cout << "Drawing a rectangle" << endl;
    }
};

int main() {
    Shape* s;
    Circle c;
    Rectangle r;
    s = &c;
    s->draw(); // Output: Drawing a circle
    s = &r;
    s->draw(); // Output: Drawing a rectangle
    return 0;
}
```

In the example above, the "Shape" class is a base class that defines a virtual function "draw()". The "Circle" and "Rectangle" classes are derived from "Shape", and they override the "draw()" function with their own implementations.

At runtime, we can create objects of the "Circle" and "Rectangle" classes, and store their addresses in a pointer of the "Shape" class.

When we call the "draw()" function using the pointer, the correct version of the function is called based on the type of object pointed to by the pointer. This is an example of runtime polymorphism.

## Exception handling :

Exception handling is a mechanism in C++ to handle unexpected or exceptional conditions that occur during the execution of a program. These conditions are often referred to as exceptions or errors, and they can arise due to a variety of reasons, such as invalid user input, system failures, or resource unavailability.

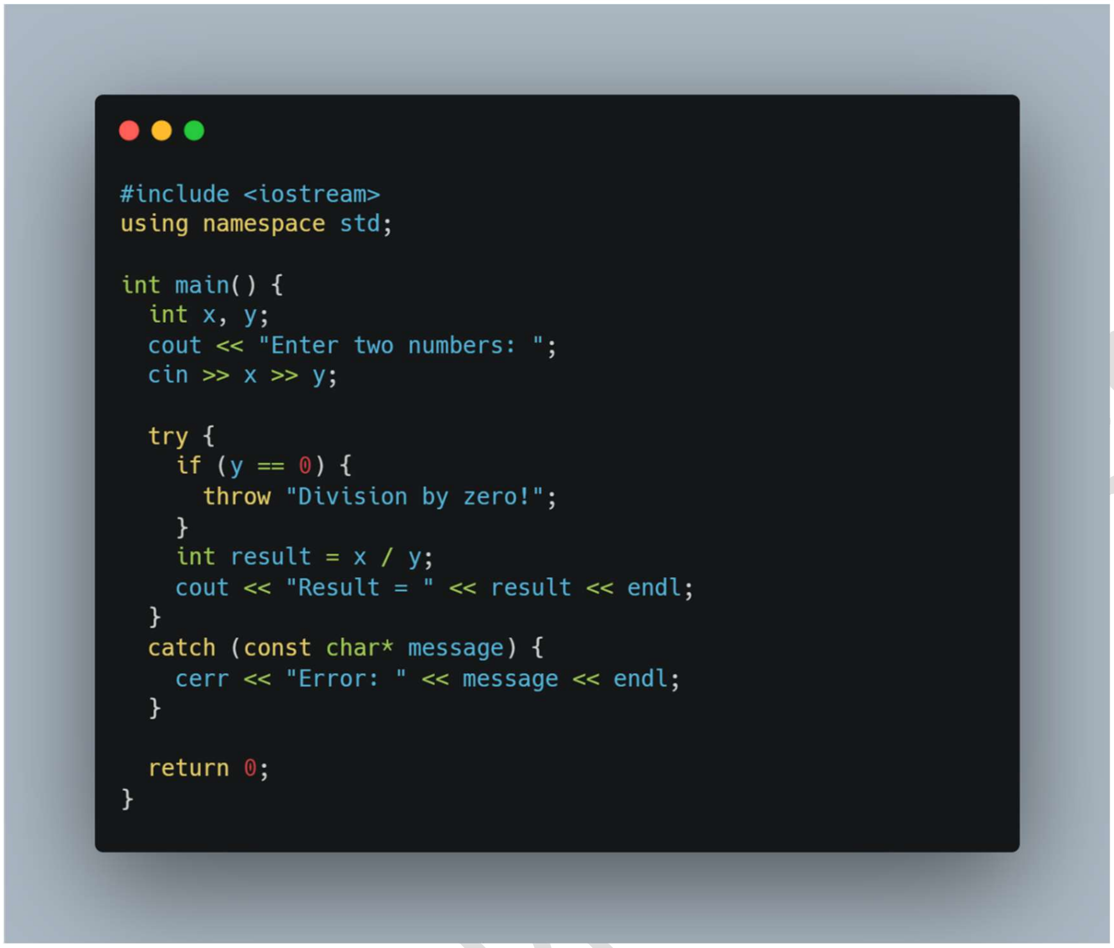
To handle these exceptions, C++ provides a set of keywords and constructs that allow you to write code that detects and responds to exceptions in a controlled manner. The three primary keywords for exception handling are:

- Try: This keyword is used to enclose a block of code that might throw an exception.

- Catch: This keyword is used to define a block of code that executes when an exception is thrown.

- Throw: This keyword is used to explicitly throw an exception from a block of code.

Here's an example of exception handling in C++:



```cpp
#include <iostream>
using namespace std;

int main() {
  int x, y;
  cout << "Enter two numbers: ";
  cin >> x >> y;

  try {
    if (y == 0) {
      throw "Division by zero!";
    }
    int result = x / y;
    cout << "Result = " << result << endl;
  }
  catch (const char* message) {
    cerr << "Error: " << message << endl;
  }

  return 0;
}
```

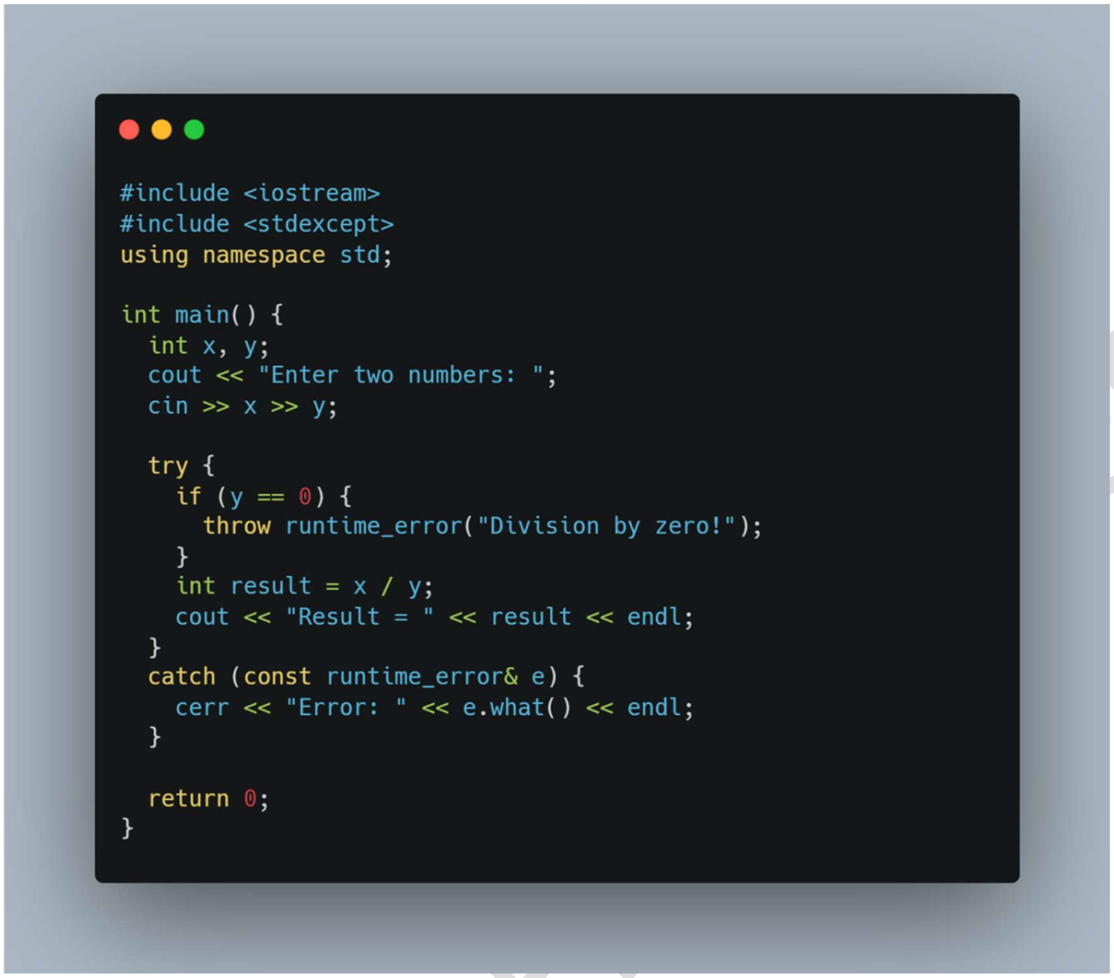
In this example, the user is prompted to enter two numbers, which are then used to perform a division operation. However, before the division is performed, the code checks whether the second number is zero, which would result in a division by zero error. If the second number is indeed zero, the code throws an exception with the message "Division by zero!".

The try block in this example encloses the division operation and the catch block defines a block of code that will execute if an exception is thrown. In this case, the catch block simply prints the error message to the console.

C++ also provides a set of standard exception classes, which can be used to handle specific types of exceptions. These include:

- std::exception: This is the base class for all standard exceptions.
- std::logic_error: This class is used for logical errors, such as invalid arguments or incorrect use of an object.
- std::runtime_error: This class is used for runtime errors, such as file I/O errors or memory allocation failures.

Here's an example of using a standard exception class in C++:



```cpp
#include <iostream>
#include <stdexcept>
using namespace std;

int main() {
  int x, y;
  cout << "Enter two numbers: ";
  cin >> x >> y;

  try {
    if (y == 0) {
      throw runtime_error("Division by zero!");
    }
    int result = x / y;
    cout << "Result = " << result << endl;
  }
  catch (const runtime_error& e) {
    cerr << "Error: " << e.what() << endl;
  }

  return 0;
}
```

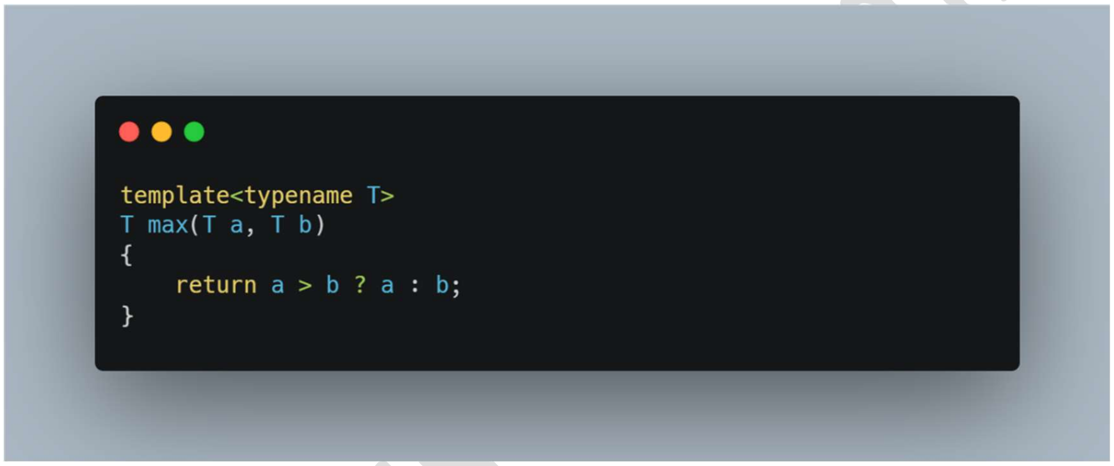
In this example, a std::runtime_error exception is thrown if the second number is zero. The catch block then catches this exception and prints the error message to the console using the what() function of the exception object.

◆ Templates:

Templates in C++ provide a way to create generic functions and classes that can work with different data types without rewriting the code. There are three types of templates in C++: Function templates, class templates, and template specialization.

- Function Templates:

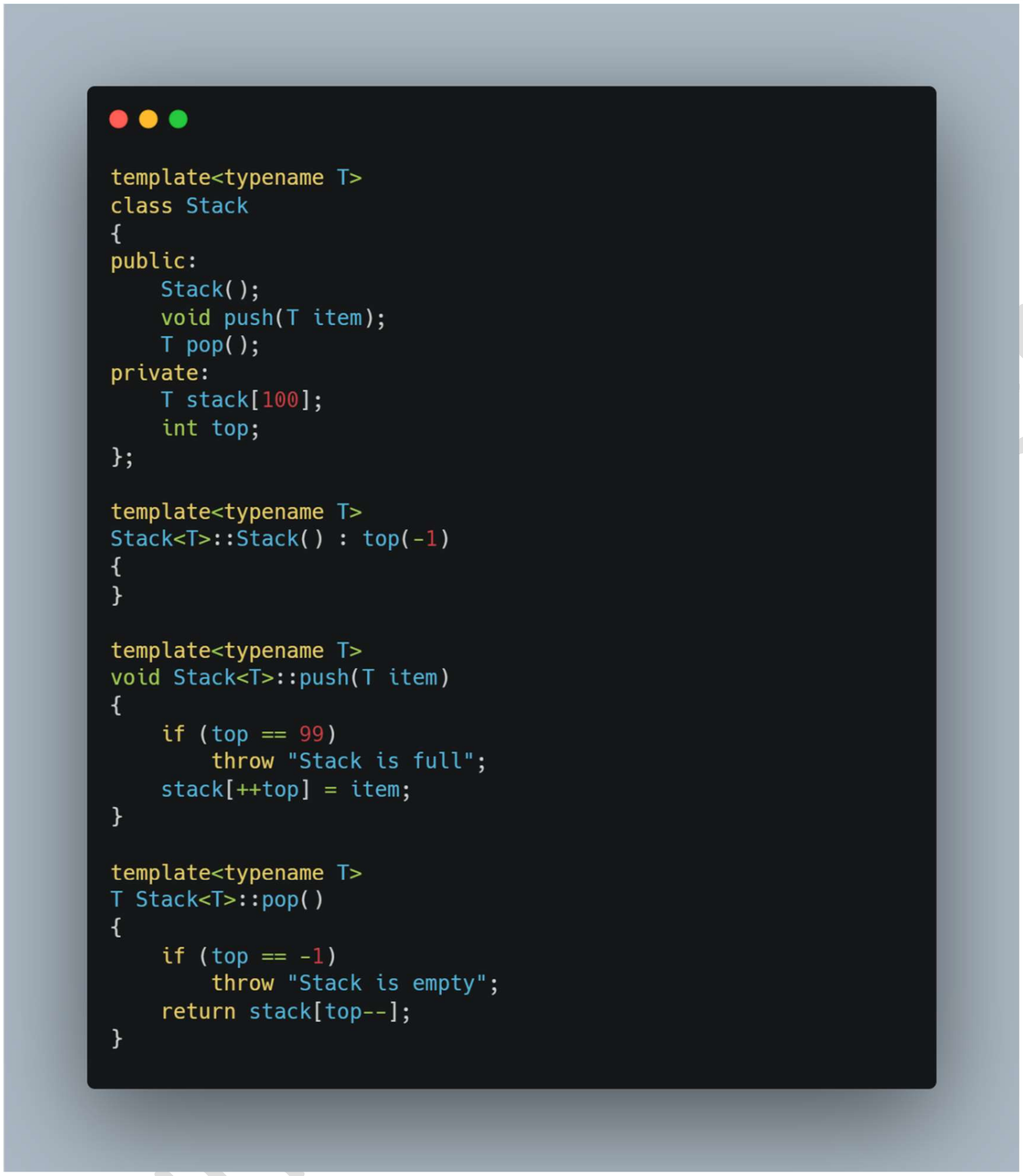
Function templates are used to define a generic function that can operate on different data types. It allows you to write a single function that can be used for different data types. For example, the following function template calculates the maximum of two values:



```
template<typename T>
T max(T a, T b)
{
    return a > b ? a : b;
}
```

- Class Templates:

Class templates are used to define a generic class that can work with different data types. It allows you to write a single class that can be used for different data types. For example, the following class template defines a stack:

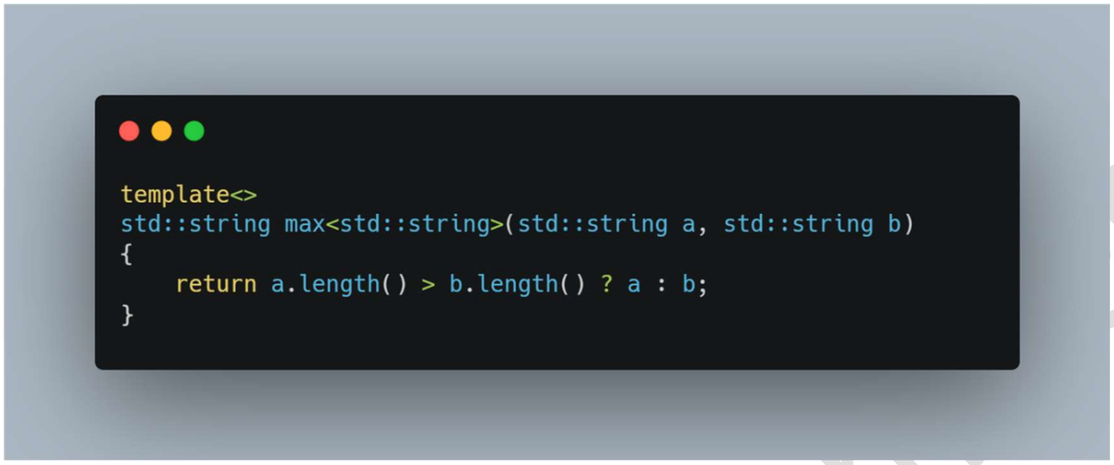


```cpp
template<typename T>
class Stack
{
public:
    Stack();
    void push(T item);
    T pop();
private:
    T stack[100];
    int top;
};

template<typename T>
Stack<T>::Stack() : top(-1)
{
}

template<typename T>
void Stack<T>::push(T item)
{
    if (top == 99)
        throw "Stack is full";
    stack[++top] = item;
}

template<typename T>
T Stack<T>::pop()
{
    if (top == -1)
        throw "Stack is empty";
    return stack[top--];
}
```

- Template Specialization:

Template specialization is a way to provide a specialized implementation of a template for a specific data type. For example, you can define a specialized implementation of the max function template for the string data type as follows:

**PDF File Uploaded On Telegram ( Link In Bio)**


```
template<>
std::string max<std::string>(std::string a, std::string b)
{
    return a.length() > b.length() ? a : b;
}
```

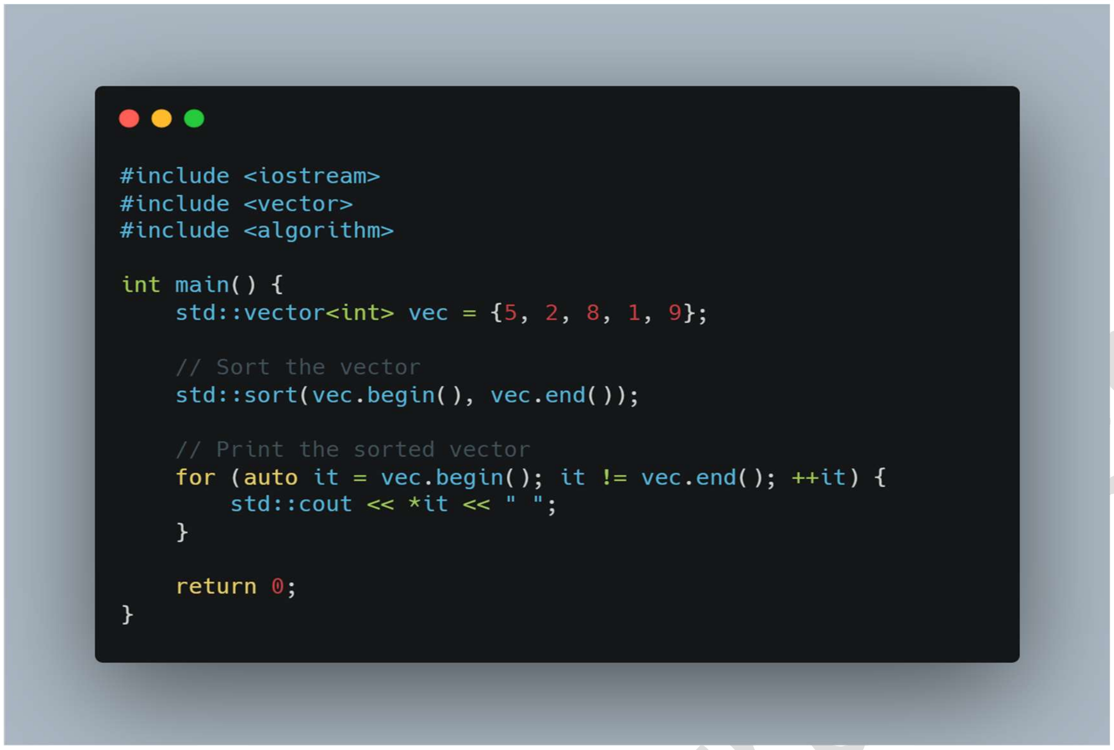
◆ Standard Template Library (STL)

- Containers: STL provides a set of containers, which are classes that can hold a collection of elements. Examples of containers include vector, deque, list, set, and map. Containers can be used to store and manipulate data efficiently.

- Algorithms: STL also provides a set of algorithms, which are functions that operate on containers. Examples of algorithms include sorting, searching, and modifying elements in a container. Algorithms can be used to perform complex operations on containers without having to write complex code.

- Iterators: STL provides a set of iterators, which are objects that can be used to traverse containers. Iterators provide a way to access the elements in a container one at a time, and can be used in combination with algorithms to perform operations on containers.

Here's a brief example of using STL:



```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {5, 2, 8, 1, 9};

    // Sort the vector
    std::sort(vec.begin(), vec.end());

    // Print the sorted vector
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

In this example, we create a vector of integers and initialize it with some values. We then use the std::sort algorithm to sort the vector, and finally use an iterator to print the sorted vector.

- File Handling

- File Handling: This topic covers how to read from and write to files in C++. It includes concepts like file input/output, file pointers, and binary files.

- File Input and Output: This involves reading data from and writing data to files. In C++, files are opened using streams, which are objects that represent input/output sources. File input involves reading data from a file into a program, while file output involves writing data from a program to a file.

- File Pointers: File pointers are used to keep track of the position of the next byte to be read from or written to a file. They can be moved to any position within a file to read or write data from that location.

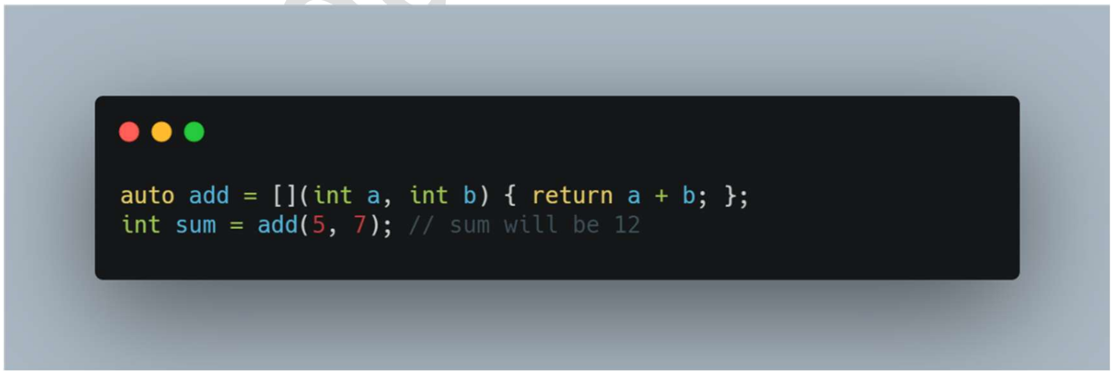
- Binary Files: Binary files are files that contain data in a binary format, rather than a text format. They are often used to store complex data structures, such as arrays and structures, in a compact and efficient way.

Overall, file handling is an important topic in C++ programming, as it allows programs to read and write data to files, which is essential for many applications.

◆ Advanced Topics :

- Lambda Expressions:

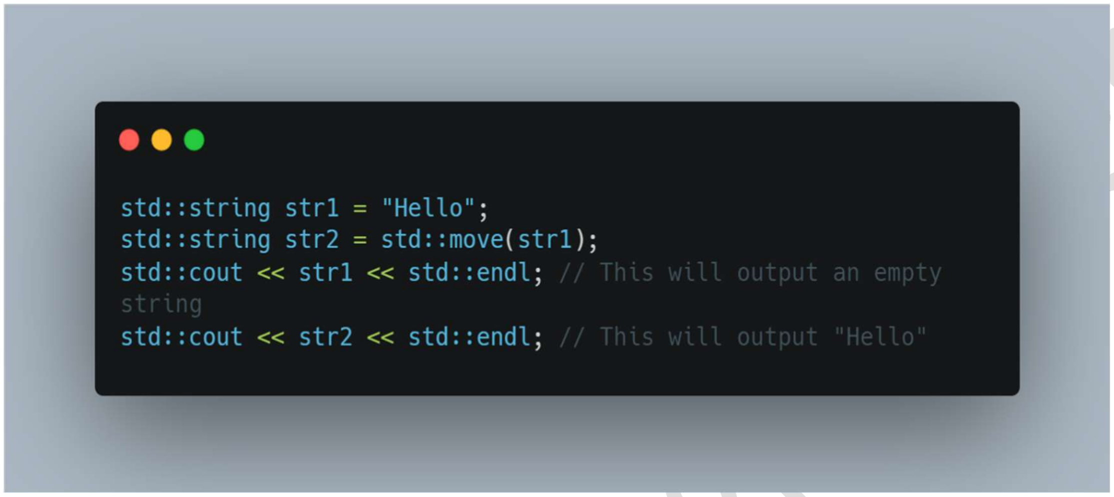
Lambda expressions allow you to define small, anonymous functions that can be passed as arguments to other functions. Here's an example of a lambda expression that adds two numbers:



```
auto add = [](int a, int b) { return a + b; };
int sum = add(5, 7); // sum will be 12
```

- Move Semantics:

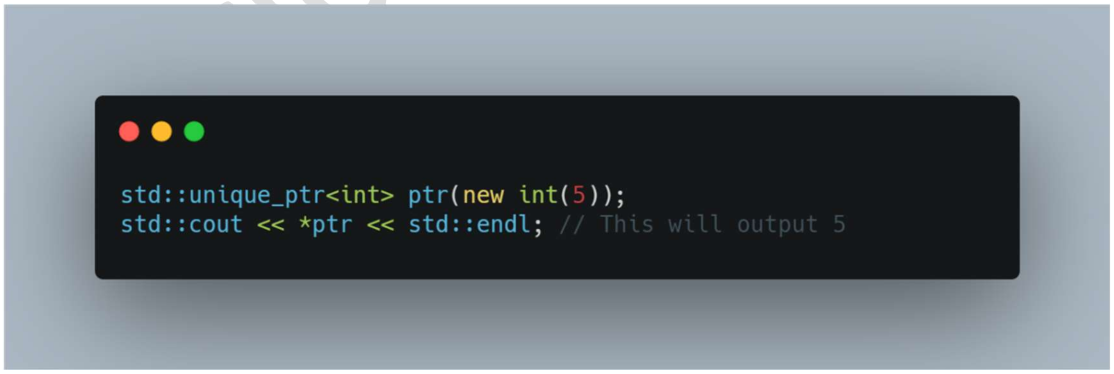
Move semantics is a feature of C++ that allows you to transfer ownership of an object from one place to another, rather than copying it. This can be more efficient in some cases, especially for large objects. Here's an example of moving a string from one variable to another:



```
std::string str1 = "Hello";
std::string str2 = std::move(str1);
std::cout << str1 << std::endl; // This will output an empty
string
std::cout << str2 << std::endl; // This will output "Hello"
```

- Smart Pointers:

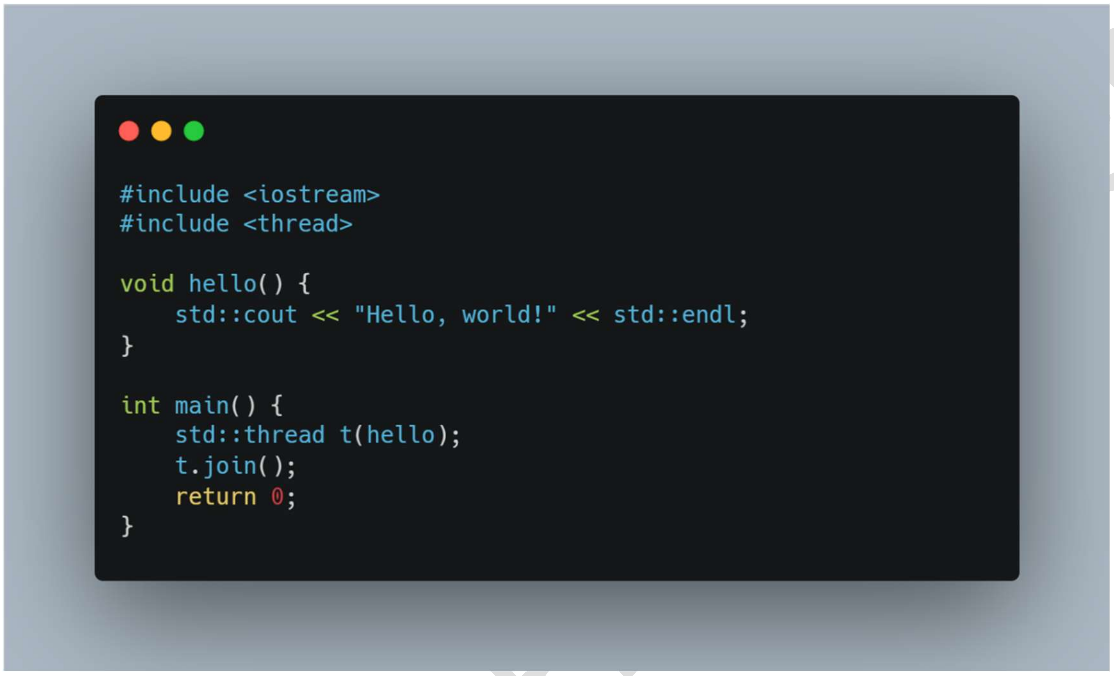
Smart pointers are a type of pointer that automatically manage the memory of the object they point to. There are three types of smart pointers in C++: unique_ptr, shared_ptr, and weak_ptr. Here's an example of using a unique_ptr:



```
std::unique_ptr<int> ptr(new int(5));
std::cout << *ptr << std::endl; // This will output 5
```

- Multithreading:

Multithreading allows your program to run multiple threads of execution simultaneously. This can be useful for tasks that can be split into smaller pieces that can be executed in parallel. Here's an example of creating a simple thread that prints out "Hello, world!":



```cpp
#include <iostream>
#include <thread>

void hello() {
    std::cout << "Hello, world!" << std::endl;
}

int main() {
    std::thread t(hello);
    t.join();
    return 0;
}
```