PROJECT REPORT

C AND CPP REPORT

DEEP DAS

U23AI052

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to all the seniors at Phoenix Aero for their invaluable contributions and for creating such a conducive environment for learning programming languages. Their guidance and support have been instrumental in the completion of this report.

I am especially thankful for the unwavering support, approachability, and willingness to clear doubts exhibited by all of our seniors. Their mentorship has significantly contributed to my learning and growth.

I also thank the rest of the senior team for their guidance and mentorship throughout this journey. Their insights and advice have been truly invaluable.

Once again, thank you all for your contributions and for making this learning experience so enriching.

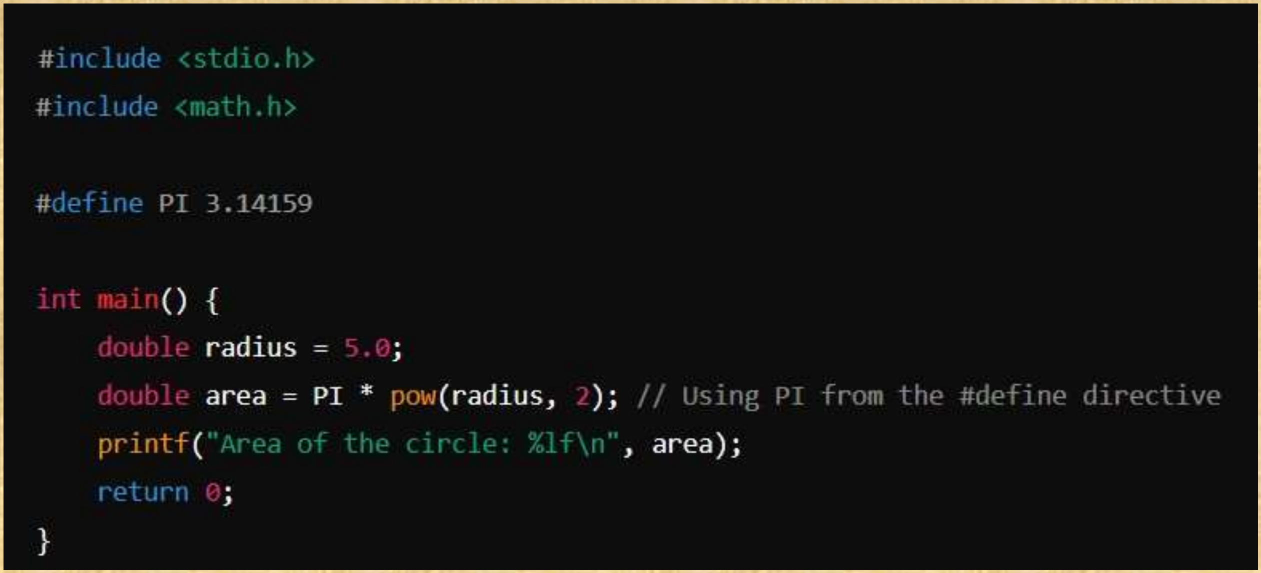
INDEX

In this modern age, we are privileged to embrace the pursuit of greatness, knowing that every remarkable achievement has humble beginnings.
This report signifies a significant milestone in our collective journey with Team Phoenix Aero, where we have diligently honed our skills in C++ and revisited the fundamentals

of C. It serves as a testament to our commitment to learning and growth. The report provides a comprehensive overview of our exploration, encompassing key concepts such as Decision Control Instructions, Case Control Instructions, Loop Control Instructions, Functions, Various Container Types, Exception Handling, and more. Each section represents a crucial step forward in our quest for mastery in these programming languages, showcasing our dedication to excellence.

01 | General concepts of C programming

C is a widely-used programming language known for its simplicity and efficiency. At the core of every C program is the main function, which serves as the entry point for the program. It is where the program begins its execution, and it must be present in every C program. Here's a basic example of a main function:

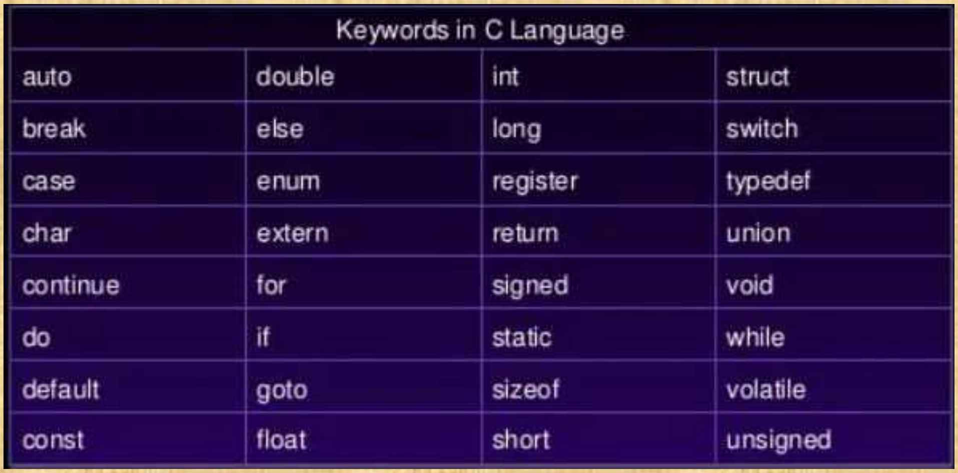


```c
#include <stdio.h>
#include <math.h>

#define PI 3.14159

int main() {
    double radius = 5.0;
    double area = PI * pow(radius, 2); // Using PI from the #define directive
    printf("Area of the circle: %lf\n", area);
    return 0;
}
```

In this example, #include <stdio.h> is a preprocessor directive that tells the compiler to include the standard input-output library, which provides functions like printf,scanf,etc for printing to the console. The

#include<math.h> directive includes the math library, which provides mathematical functions like pow,sqrt,log,etc for mathematical calculations. The #define PI 3.14159 directive defines a constant PI with the value 3.14159, which is used in the calculation of the area of a circle.

There are around 32 keywords in C language which are reserved by the language and you can’t use them in variable declaration,statements,etc.



Keywords in C Language

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
continue	for	signed	void
do	if	static	while
default	goto	sizeof	volatile
const	float	short	unsigned

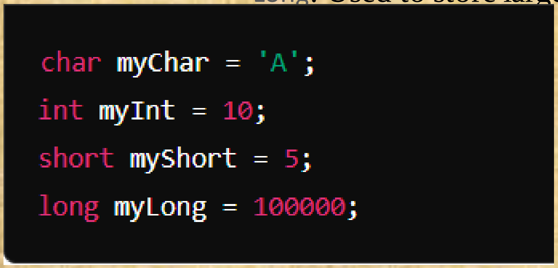
02 | Data types,operators in C

In both C and C++, data types are used to define the type of data that a variable can hold. These languages provide a variety of data types to accommodate different kinds of data and optimize memory usage. Here's an overview of the basic data types in C and C++, along with examples:

Primary data types:

Integer Types:

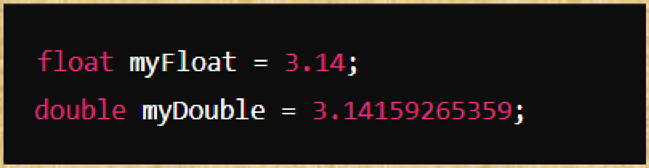
•char: Used to store characters. Size is 1 byte.
•int: Used to store integers. Size is typically 4 bytes.
•short: Used to store small integers. Size is typically 2 bytes.
•Long: Used to store large integers. Size is typically 4 bytes or 8 bytes.



```c
char myChar = 'A';
int myInt = 10;
short myShort = 5;
long myLong = 100000;
```

Floating-Point Types:

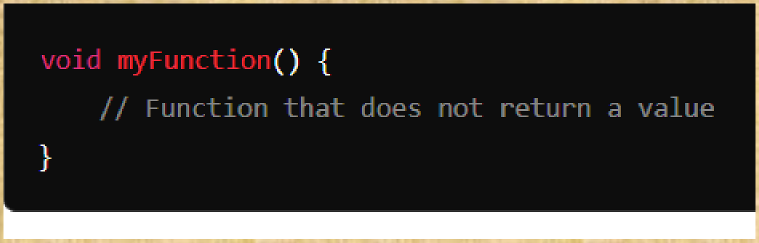
- `float`: Used to store single-precision floating-point numbers. Size is typically 4 bytes.
- `double`: Used to store double-precision floating-point numbers. Size is typically 8 bytes.



```c
float myFloat = 3.14;
double myDouble = 3.14159265359;
```

Void Type:

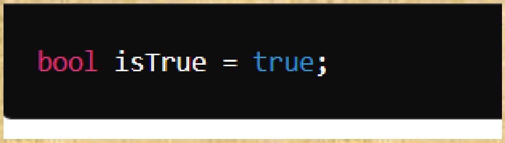
•`void`: Represents the absence of type. It is commonly used as the return type of functions that do not return a value.



```c
void myFunction() {
    // Function that does not return a value
}
```

Boolean Type:

•C does not have a built-in boolean type, but it is often represented using integers (0 for false, non-zero for true).
•C++ introduced a boolean type `bool` with the values `true` and `false`.

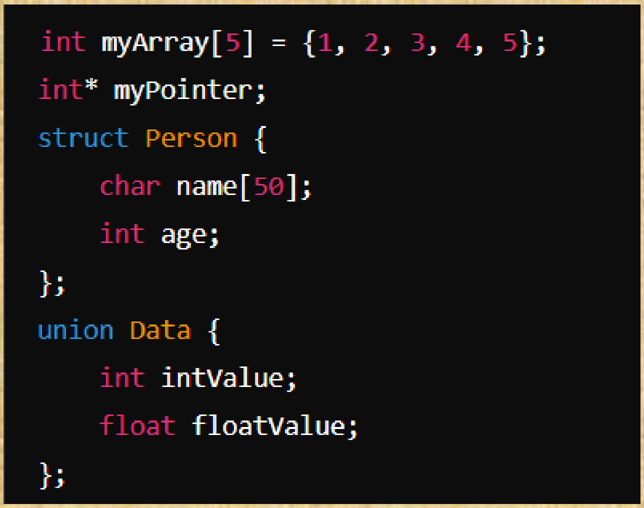


```
bool isTrue = true;
```

Secondary/Derived data types:

1> **Derived Types:**

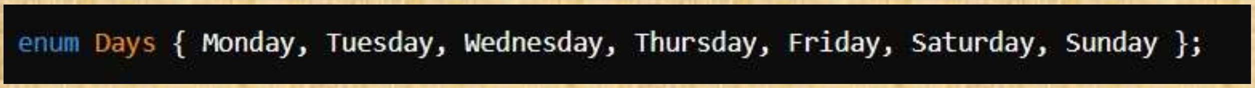
- `arrays`: Used to store a collection of elements of the same data type.
- `pointers`: Used to store the memory address of another variable.
- `structures`: Used to group together variables of different types under a single name.
- `unions`: Similar to structures but can store only one value at a time.



```
int myArray[5] = {1, 2, 3, 4, 5};
int* myPointer;
struct Person {
    char name[50];
    int age;
};
union Data {
    int intValue;
    float floatValue;
};
```

2> **Enumerated Types:**

- `enum`: Used to define a set of named integer constants.



```
enum Days { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

03 | Operators in C,C++

Operators in programming languages are symbols that represent computations or actions to be performed on data. In C and C++, operators can be classified into several categories:

1. **Arithmetic Operators:** These operators perform arithmetic operations on numerical values. Examples include + (addition), - (subtraction), * (multiplication), / (division), and % (modulo).

2. **Relational Operators:** Relational operators are used to compare two values. Examples include == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), and <= (less than or equal to).

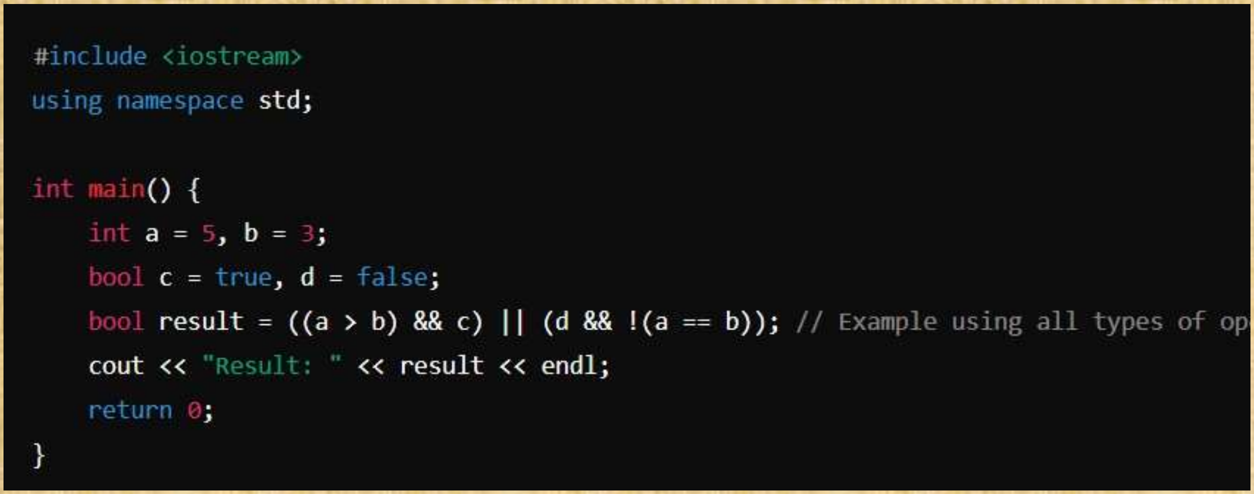
3. **Logical Operators:** Logical operators perform logical operations on boolean values. Examples include && (logical AND), || (logical OR), and ! (logical NOT).

4. **Bitwise Operators:** Bitwise operators perform operations at the bit level. Examples include & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), << (left shift), and >> (right shift).

5. **Assignment Operators:** Assignment operators are used to assign values to variables. Examples include = (simple assignment), += (addition assignment), -= (subtraction assignment), *= (multiplication assignment), /= (division assignment), and %= (modulo assignment).

6. **Unary Operators:** Unary operators operate on a single operand. Examples include ++ (increment), -- (decrement), and sizeof (returns the size of a data type).

7. **Ternary Operator:** The ternary operator ?: is a conditional operator that takes three operands and evaluates to a value based on a condition.



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

int main() {
    int a = 5, b = 3;
    bool c = true, d = false;
    bool result = ((a > b) && c) || (d && !(a == b)); // Example using all types of op
    cout << "Result: " << result << endl;
    return 0;
}
```

04 | Logic gates

Logic gates are the basic building blocks of digital circuits. They are electronic devices that perform logical operations on one or more binary inputs and produce a single binary output based on that operation. There are several types of logic gates, each with its own truth table that defines the output for all possible combinations of inputs.

The main types of logic gates are :

AND Gate:

The output of an AND gate is high (1) only when all of its inputs are high (1), otherwise, it's low (0). Truth table:

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate:

The output of an OR gate is high (1) when at least one of its inputs is high (1), otherwise, it's low

(0). Truth table:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate (Inverter):

The output of a NOT gate is the inverse of its input. If the input is high (1), the output is low (0), and vice

versa. Truth table:

A	Y
0	1
1	0

NAND Gate (NOT-AND):

The output of a NAND gate is the opposite of an AND gate. It's low (0) only when all of its inputs are high (1), otherwise, it's high (1).

Truth table:

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate (NOT-OR):

The output of a NOR gate is the opposite of an OR gate. It's high (1) only when all of its inputs are low (0), otherwise, it's low (0).

Truth table:

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

XOR Gate (Exclusive OR):

The output of an XOR gate is high (1) when the number of high inputs is odd, otherwise, it's low (0).

Truth table:

A	B	Y
0	0	0
0	1	1
1	0	1

1	1	0

XNOR Gate (Exclusive NOR):

The output of an XNOR gate is high (1) when the number of high inputs is even, otherwise, it's low (0). Truth table:

A	**B**	**Y**
0	0	1
0	1	0
1	0	0
1	1	1

05 | Introduction to C++

C++ is a general-purpose programming language that was developed as an extension of the C programming language. It was created by Bjarne Stroustrup in 1979 at Bell Labs as an enhancement to the C language with features such as classes and objects, which support object-oriented programming (OOP) principles. C++ retains the efficiency and flexibility of C while adding features for OOP, making it a powerful language for developing complex software systems.

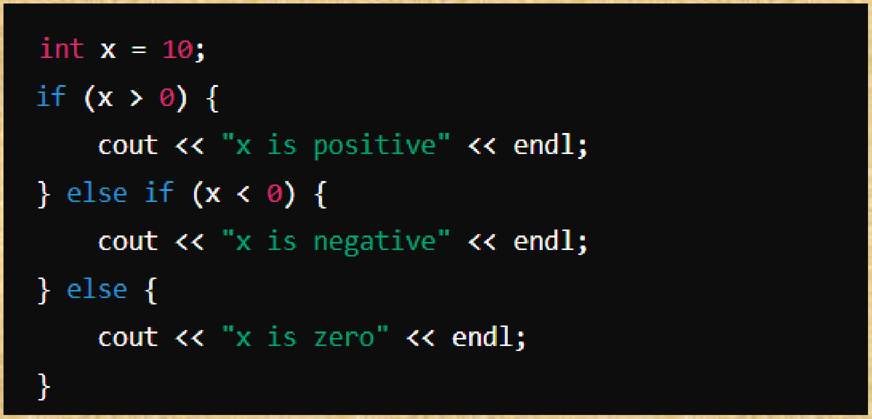
One of the main reasons for the shift from C to C++ was the need for better code organization and reusability. C++ introduced the concept of classes and objects, which allow developers to create modular, reusable code. This approach to programming promotes better code organization, as related data and functions are grouped together in classes. Additionally, C++ provides features such as inheritance, polymorphism, and encapsulation, which help in building more robust and maintainable code.

Another reason for the popularity of C++ is its compatibility with C. C++ is largely backward compatible with C, meaning that most C code can be compiled and run in a C++ environment. This compatibility allows developers to leverage existing C code while gradually transitioning to the more advanced features of C++, making the migration process smoother for projects already written in C.

In C, the printf function is used for output, and scanf for input. To move to the next line when printing output, the newline character \n is used. In contrast, C++ uses the iostream library, where cout is used for output and cin for input. The endl manipulator is used with cout to move to the next line and flush the output buffer.

06 | Loops and conditional statements

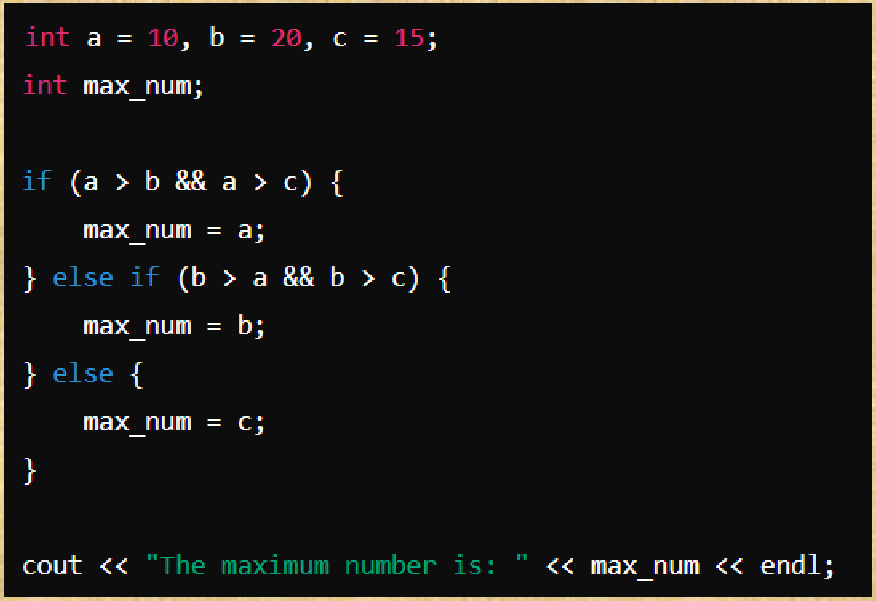
Conditional statements in C++ allow you to control the flow of your program based on certain conditions. The `if`, `else if`, and `else` statements are used to execute different blocks of code depending on the evaluation of one or more conditions. For example:



```cpp
int x = 10;
if (x > 0) {
    cout << "x is positive" << endl;
} else if (x < 0) {
    cout << "x is negative" << endl;
} else {
    cout << "x is zero" << endl;
}
```

In this example, the `if` statement checks if `x` is greater than 0, and if so, it prints "x is positive". If not, it checks if `x` is less than 0, and if so, it prints "x is negative". If neither condition is true, it prints "x is zero".

Conditional statements are essential for problem-solving and algorithmic tasks. For instance, consider a problem where you need to find the maximum of three numbers.



```cpp
int a = 10, b = 20, c = 15;
int max_num;

if (a > b && a > c) {
    max_num = a;
} else if (b > a && b > c) {
    max_num = b;
} else {
    max_num = c;
}

cout << "The maximum number is: " << max_num << endl;
```

In this example, the program uses conditional statements to compare the values of a, b, and c to find the maximum number among them. The result is then stored in the max_num variable and displayed to the user.

Loops are used to repeat a certain task without writing the code again and again. It is also known as

flow control statements in the programming world.

A while loop in C++ is a fundamental control flow statement that allows you to repeatedly execute a block of code as long as a specified condition remains true. The syntax of a while loop consists of the ‘while’ keyword followed by a condition in parentheses, and then a block of code enclosed in curly braces. The condition is evaluated before each iteration of the loop, and if it evaluates to true, the code inside the loop is executed. If the condition evaluates to false, the loop is exited, and the program continues with the code after the loop.

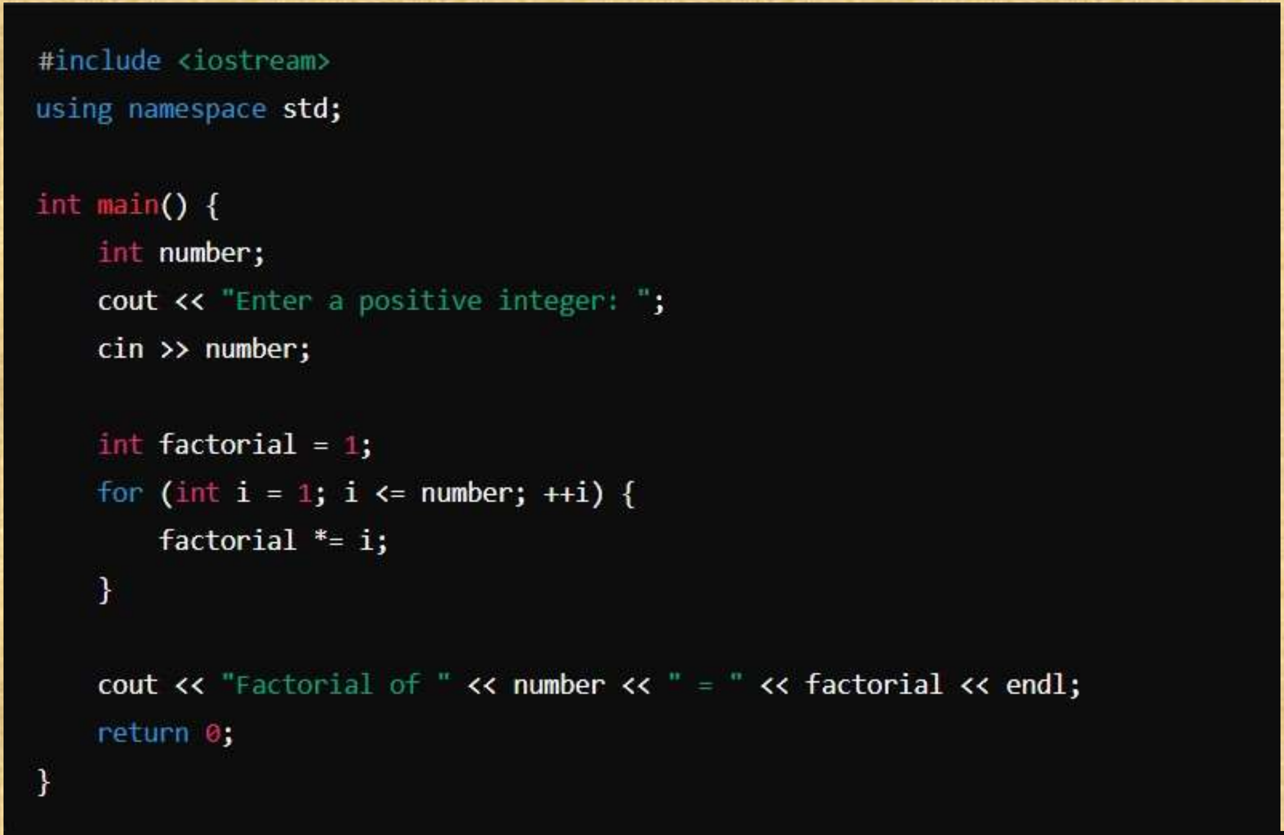
While loops are useful for situations where you want to repeat a block of code a certain number of times or until a specific condition is met. For example, you might use a while loop to read data from a file until the end of the file is reached, or to process user input until a specific command is entered. One important thing to remember when using a while loop is to ensure that the condition will eventually become false; otherwise, you may end up with an infinite loop, which can cause your program to hang or become unresponsive.

A 'for' loop in C++ is a powerful and versatile construct used for iterating over a sequence of values or executing a block of code a specific number of times. It is often preferred when the number of iterations is known before the loop starts. The syntax of a 'for' loop consists of three parts: initialization, condition, and iteration, separated by semicolons and enclosed in parentheses. Here's the syntax:

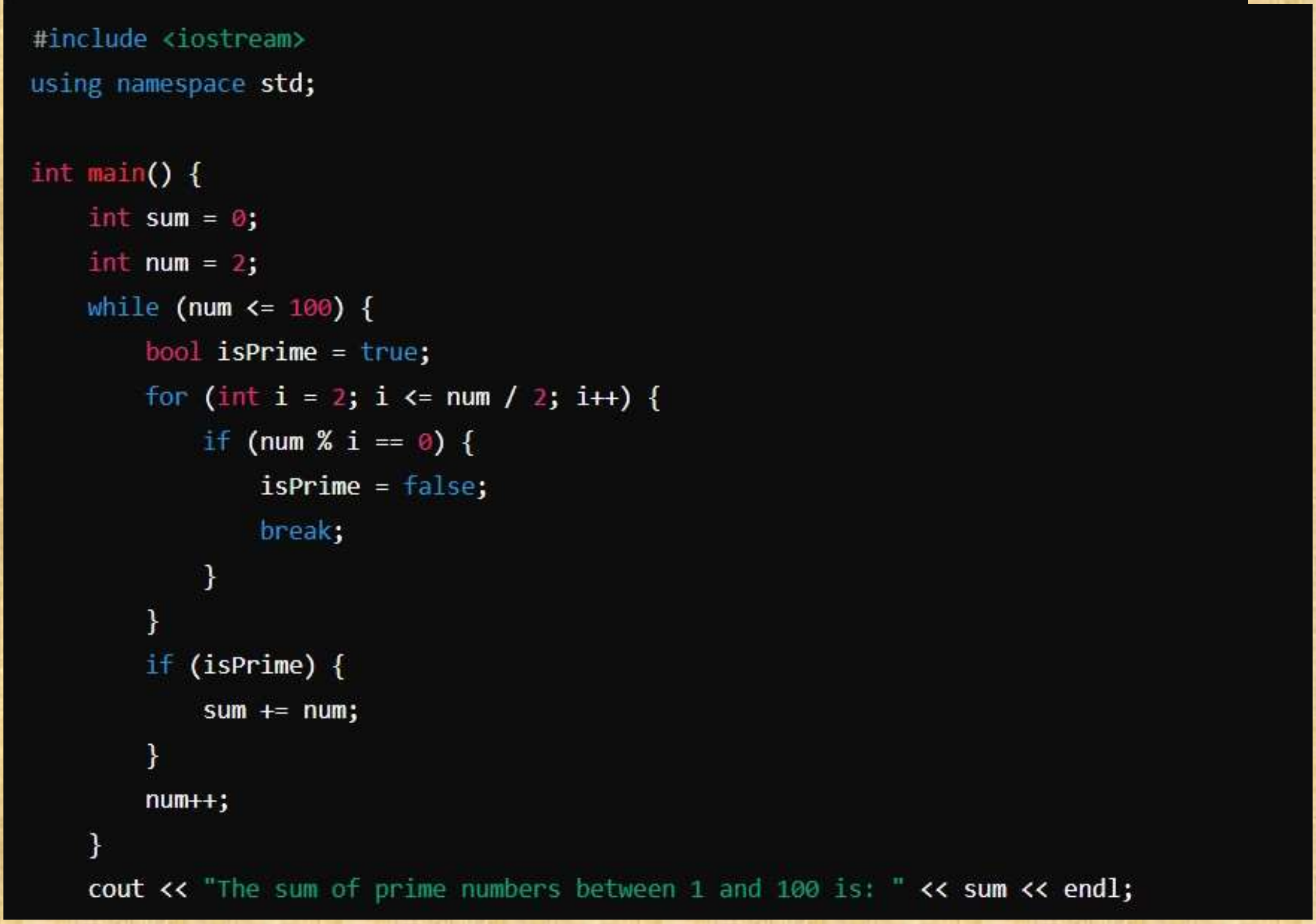
```
For(initialisation ; iteration ; condition){

}
```

The 'initialization' part is executed once at the beginning of the loop and is typically used to initialize a loop control variable. The 'condition' part is evaluated before each iteration, and if it evaluates to true, the code inside the loop is executed. The 'iteration' part is executed at the end of each iteration and is typically used to update the loop control variable.

Examples of code with both the type of loops are given below :



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

int main() {
    int number;
    cout << "Enter a positive integer: ";
    cin >> number;

    int factorial = 1;
    for (int i = 1; i <= number; ++i) {
        factorial *= i;
    }

    cout << "Factorial of " << number << " = " << factorial << endl;
    return 0;
}
```



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

int main() {
    int sum = 0;
    int num = 2;
    while (num <= 100) {
        bool isPrime = true;
        for (int i = 2; i <= num / 2; i++) {
            if (num % i == 0) {
                isPrime = false;
                break;
            }
        }
        if (isPrime) {
            sum += num;
        }
        num++;
    }
    cout << "The sum of prime numbers between 1 and 100 is: " << sum << endl;
```

07 | Functions

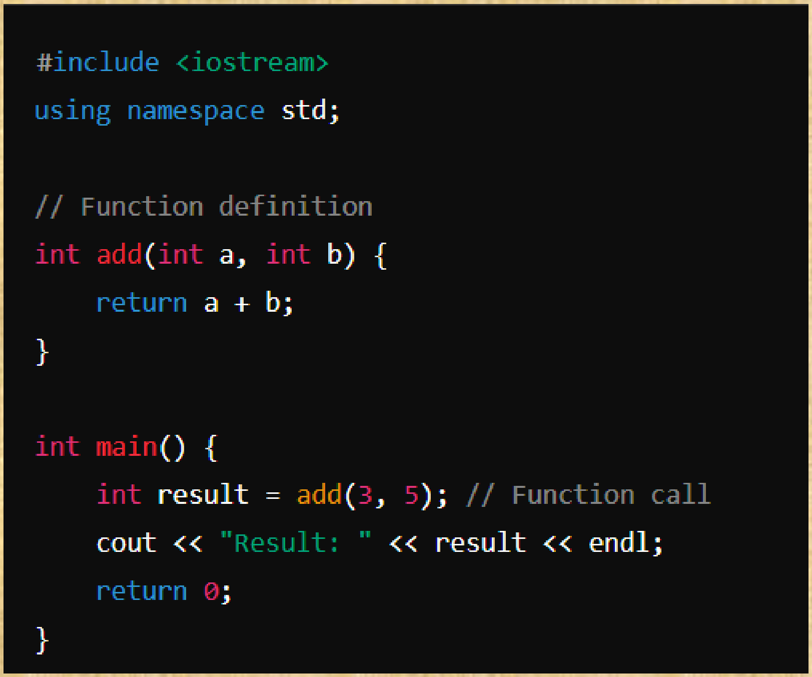
Functions in C++ are reusable blocks of code that perform a specific task. They help in organizing code, making it more readable, maintainable, and easier to debug. Functions are essential in programming as they allow you to break down complex problems into smaller, manageable parts. They also promote code reusability, as a function can be called multiple times from different parts of a program.

To define a function in C++, you use the following syntax: Return_type function_name(parameters){

}

Here, Return_type specifies the type of value the function returns, function_name is the name of the function, and parameters is a list of parameters that the function takes (if any). The function body contains the code that defines the behavior of the function.

To use a function, you simply call it by its name and provide any required arguments. For example:



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

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

int main() {
    int result = add(3, 5); // Function call
    cout << "Result: " << result << endl;
    return 0;
}
```

In C++, parameters are variables declared in a function's declaration or definition, while arguments are the actual values or variables passed to the function when it is called. Parameters act as placeholders for the values that will be passed to the function, and arguments are the actual values that are assigned to those placeholders when the function is called.

For example, in the function declaration int add(int a, int b), a and b are parameters. When the function is called with add(3,5) here 3 and 5 are the arguments that are passed to the function, and they are assigned to the parameters a and b respectively.

08 | Object-oriented programming

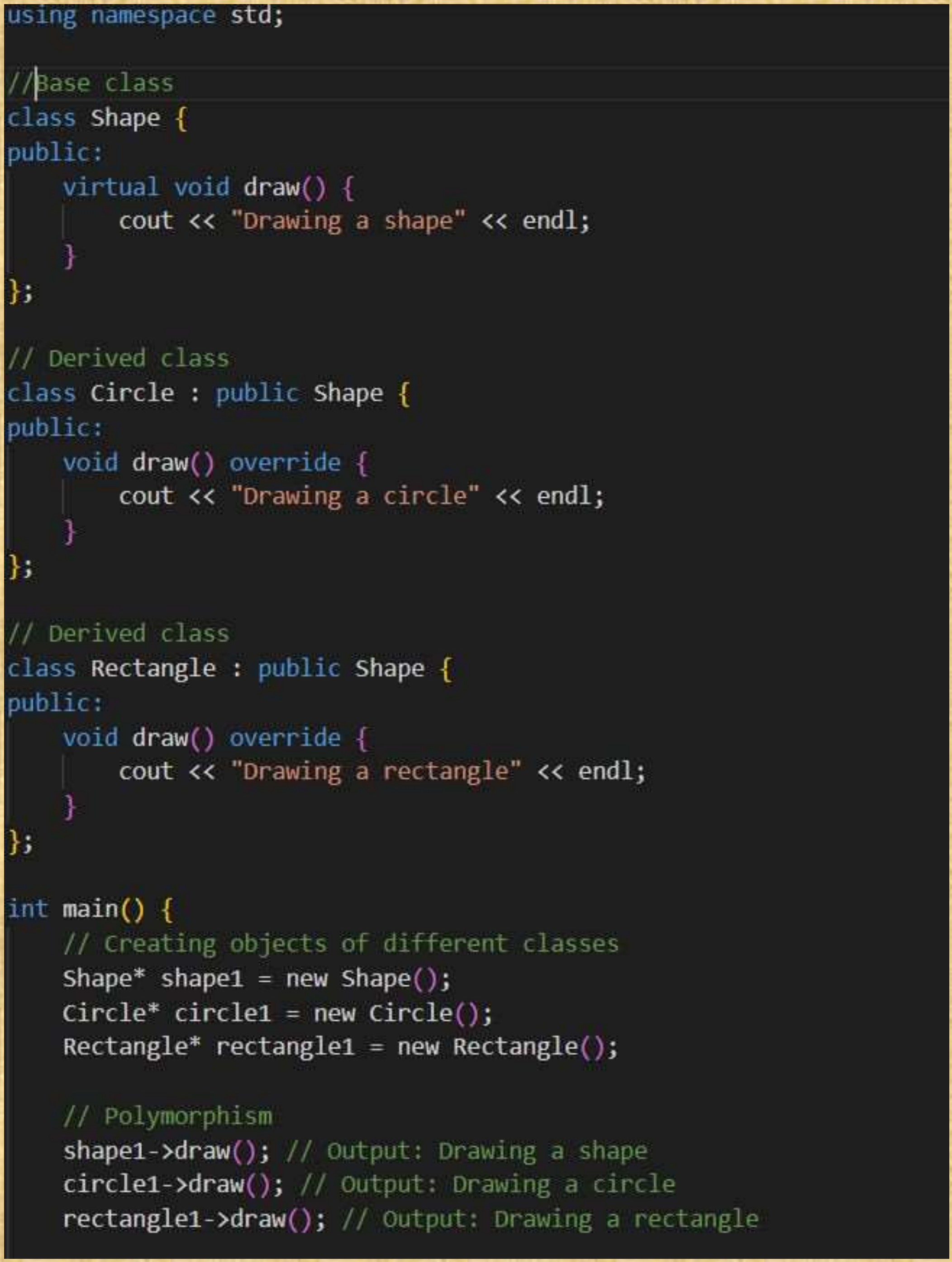
Object-oriented programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. It is based on the concept of "objects," which can contain data in the form of fields (often known as attributes or properties), and code in the form of procedures (often known as methods). OOP focuses on the creation of objects that interact with each other to solve a problem.

One of the key principles of OOP is encapsulation, which refers to the bundling of data and the methods that operate on that data into a single unit or class. This means that the internal workings of an object can be hidden from the outside world, and only a specific set of methods can interact with the object's data. This helps in making the code more modular, maintainable, and secure.

Another important concept in OOP is inheritance, which allows a class to inherit the properties and methods of another class. This promotes code reusability, as common functionality can be defined in a base class and then inherited by other classes. For example, you might have a base class shape with methods for calculating area and perimeter, and then have subclasses like circles and rectangle that inherit from shape.

Polymorphism is also a key concept in OOP, which allows objects of different classes to be treated as objects of a common superclass. This means that you can have a single method that can operate on different types of objects. For example, you might have a draw method in a superclass shape, and then override this method in subclasses like circles and rectangle to provide specific implementations.

Here's a simple example in C++ that demonstrates the concepts of classes, objects, encapsulation, inheritance, and polymorphism:



```cpp
using namespace std;

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

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

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

int main() {
    // Creating objects of different classes
    Shape* shape1 = new Shape();
    Circle* circle1 = new Circle();
    Rectangle* rectangle1 = new Rectangle();

    // Polymorphism
    shape1->draw(); // Output: Drawing a shape
    circle1->draw(); // Output: Drawing a circle
    rectangle1->draw(); // Output: Drawing a rectangle
```

In this example, we have a base class 'shape' with a virtual 'draw' method. We then have two derived classes 'circle' and 'rectangle' that inherit from 'shape' and override the 'draw' method to provide specific implementations. In the main function, we create objects of these classes and demonstrate polymorphism by calling the 'draw' method on objects of type shape,circles and rectangle which results in different messages being printed based on the actual object type.

09 | Exception handling

Exception handling in C++ allows you to handle runtime errors and exceptional situations gracefully. It provides a way to separate error-handling code from normal code, improving code readability and maintainability. The basic syntax for exception handling in C++ includes the try ,throw and catch keywords.

Here's a brief explanation of each keyword:

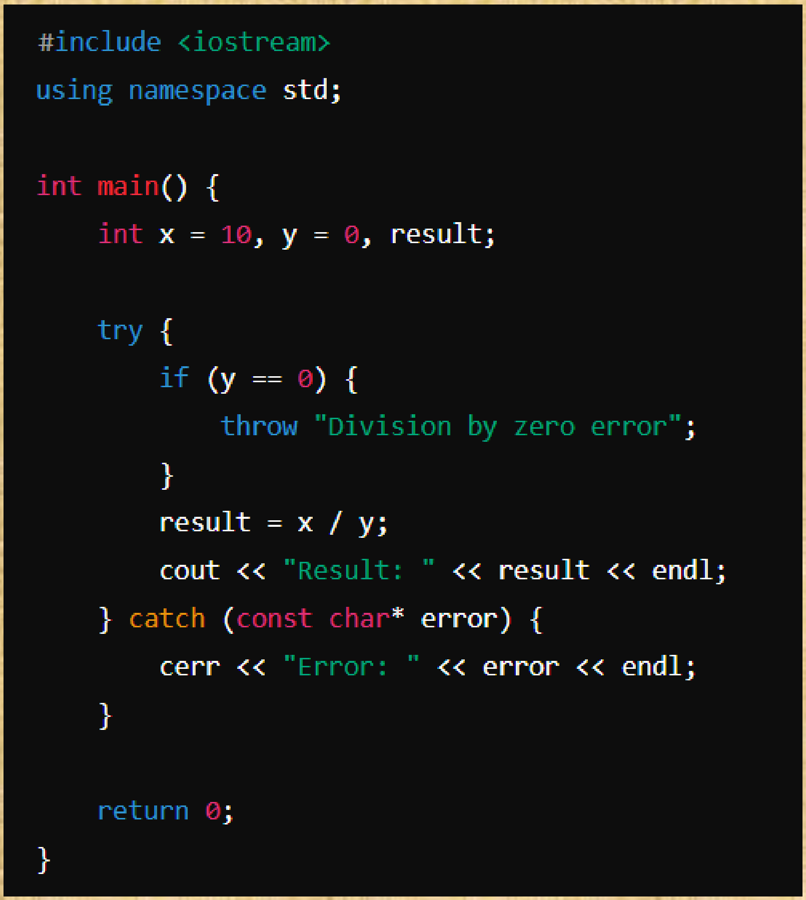
•try: The try block is used to enclose the code that might throw an exception. If an exception is thrown within the try block, the control is transferred to the nearest catch block that can handle the exception.

- throw: The throw keyword is used to explicitly throw an exception. You can throw any data type, including primitive types, objects, or even pointers.

- catch: The catch block is used to catch and handle exceptions thrown by the try block. It specifies the type of exception it can handle, and if the type matches the thrown exception, the code inside the catch block is executed.

Exception handling is a powerful tool in programming that allows you to gracefully handle errors and exceptional situations. It is particularly useful in situations where errors are expected to occur occasionally, but you still want your program to continue executing or provide meaningful feedback to the user. Exception handling helps in separating the error-handling logic from the normal flow of the program, making the code more readable and maintainable.

Exception handling is especially valuable in situations where recovering from an error is possible or when you want to provide alternative behavior in case of an error. For example, in a file processing application, you can use exception handling to catch and handle file-related errors, such as file not found or permission denied, without crashing the entire program. Additionally, exception handling can be used to enforce certain conditions or constraints, such as ensuring that a function receives valid input, and if not, throwing an exception to indicate the error.



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

int main() {
    int x = 10, y = 0, result;

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

    return 0;
}
```

Conclusion

In conclusion, our exploration of C and C++ has revealed the foundational aspects of these languages and their profound impact on programming. We've learned how to utilize functions, pointers, and references in C/C++, which are essential tools for building efficient and modular code. Additionally, the concept of classes and objects in C++ provides a structured approach to program design, enhancing code organization and readability.

Throughout our journey, we've discovered the power of loops, conditional statements, and switch-case statements in simplifying complex logic and making programming more manageable. As we continue to delve deeper into the world of C and C++, we are equipped with the knowledge and skills to tackle diverse programming challenges and create innovative solutions.

As we look ahead, let us embrace the opportunities that C and C++ offer us to push the boundaries of software development and create impactful applications. Let's strive to leverage these languages to their fullest potential, embracing the creativity and ingenuity they inspire. Together, we can continue to advance technology and shape the future of programming.