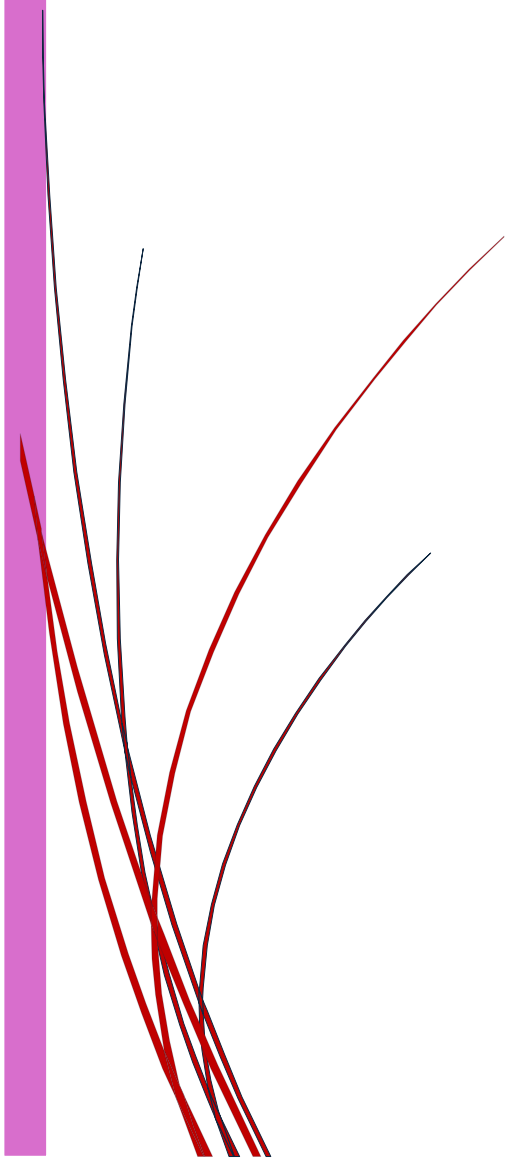
Programming in C++

GATE Computer Education



PROGRAMMING IN C++

Table of Contents

1. Introduction to C++

Turbo C++ is an older integrated development environment (IDE) and compiler for the C++ programming language, created by Borland. It was popular in the late 1980s and 1990s, especially in educational settings. Even though it's considered outdated compared to modern IDEs, Turbo C++ is still used in some educational institutions for teaching the basics of C++ programming.

What is C++?
C++ is a programming language created by Bjarne Stroustrup in 1985. It's used for creating software that can range from small programs to large applications like video games, operating systems, and complex simulations.

Getting Started with Turbo C++
Installation

1. **Download Turbo C++:**
    - Search for "Turbo C++ download" online and find a reliable source. Borland no longer maintains it, so you might have to use third-party sites.
2. **Install Turbo C++:**
    - Run the installer and follow the on-screen instructions. Turbo C++ was designed for DOS, so you might need a DOSBox emulator if you are using a modern Windows OS.

Interface Overview

- **Editor Window:** Where you write your C++ code.
- **Menu Bar:** Provides options to manage files, compile and run programs, and access other tools.
- **Output Window:** Displays the output of your program and any compilation errors.

Writing Your First Program in Turbo C++
Here's a step-by-step guide to writing and running a simple "Hello, World!" program:

1. **Open Turbo C++:**
    - Launch Turbo C++ from your desktop or start menu.
2. **Create a New File:**
    - Go to File > New to open a new file for writing code.

Write Your Code:

- Type the following code into the editor

```
#include <iostream.h>

void main() {
   cout << "Hello, World!";
}
```

Save Your File:

- Go to File > Save and save your file with a .cpp extension (e.g., hello.cpp).

Compile Your Program:

- Go to Compile > Compile (or press Alt + F9). This step translates your code into an executable file. If there are errors, they will appear in the output window.

Run Your Program:

- Go to Run > Run (or press Ctrl + F9). This will execute your compiled program, and you should see "Hello, World!" in the output window.

Understanding the Code

- #include <iostream.h>: This line includes the input-output stream library, necessary for using cout to print to the screen.
- void main(): This is the main function where the execution of your program begins.
- cout << "Hello, World!"; This line outputs the string "Hello, World!" to the console.

Basic Features of Turbo C++

1. **Syntax Highlighting:**
   The editor highlights different parts of your code (keywords, strings, comments) in different colors to make it easier to read.
2. **Debugger:**
   Allows you to run your program step-by-step to find and fix errors.
3. **Project Management:**
   Turbo C++ allows you to manage multiple files and projects within the IDE.

Common Errors and Troubleshooting

1. **Syntax Errors:**
   Make sure every statement ends with a semicolon (;).
   Ensure all braces ({}) are properly paired.
2. **Linker Errors:**
   Occur when the compiler can't find a reference to a function or variable. Check your includes and function declarations.
3. **Runtime Errors:**
   Errors that occur when your program is running, often due to invalid operations (e.g., dividing by zero).

2. Basics of C++ Programming

Basic Structure of a C++ Program

A simple C++ program typically includes headers, a main function, and statements. Here's the basic structure:

```
#include <iostream.h>  // Include the input-output stream library

int main() {  // Main function where program execution begins
   std::cout << "Hello, World!";  // Output "Hello, World!" to the console
   return 0;  // Return 0 to indicate that the program ended successfully
}
```

1. Directives
C++ programs typically start with include preprocessor directives to bring in necessary header files:

```
#include <iostream.h>  // Standard I/O stream
```

These lines are instructions to the preprocessor. #include <iostream> tells the compiler to include the standard input-output stream library.

2. Main Function

```
int main() {
  // Statements
  return 0;
}
```

- ✓ int main(): This declares the main function. The int specifies the return type of the function (typically int for main), and main() indicates that this function takes no arguments.
- ✓ {}: Curly braces define the beginning and end of the function's body.
- ✓ return 0;: Indicates the exit status of the program (0 typically means successful execution).

This is the entry point of a C++ program. The main function is where the execution starts. int indicates that the function returns an integer value.

3. Statements
Inside the main function, you can write statements to perform operations:

```
cout << "Hello, World!"; // Example statement
```

- ✓ cout: A function used to print output to the console.
- ✓ ";": Statements in C++ must end with a semicolon (;).

4. Comments
Comments are used to annotate the code and are ignored by the compiler:

```
// This is a single-line comment
```

```
/*
  This is a
  multi-line comment
*/
```

- ✓ //: Begins a single-line comment.
- ✓ /* */: Encloses multi-line comments.

5. Variables and Data Types
Variables are used to store data, and data types define the type of data a variable can hold. Common data types include:

int: Integer numbers
float: Floating-point numbers
double: Double-precision floating-point numbers
char: Single characters
bool: Boolean values (true or false)

Example program:

```
int age = 21;
float height = 5.9;
char grade = 'A';
bool isStudent = true;
```

- ✓ int, float, char: Data types for integers, floating-point numbers, and characters, respectively.
- ✓ = Assignment operator used to initialize variables.

6. Input and Output
C++ uses cin for input and cout for output.

Example:

```
#include <iostream>

int main() {
   int age;
   std::cout << "Enter your age: ";
   std::cin >> age;  // Take input from the user
   std::cout << "You are " << age << " years old.";
   return 0;
}
```

7. Control Structures
Control structures are used to control the flow of a program. Common control structures include:

If-Else Statement:

```
if (condition) {
   // Code block executed if condition is true
} else {
   // Code block executed if condition is false
}
```

```
int number = 10;
if (number > 0) {
   std::cout << "Positive number";
} else {
   std::cout << "Negative number";
```

```
}
```

For Loop

```
for (int i = 0; i < N; i++) {
  // Code block executed repeatedly
}
```

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

While Loop

```
while (condition) {
  // Code block executed while condition is true
}
```

```
int i = 0;
while (i < 5) {
  std::cout << i << " ";
  i++;
}
```

- ✓ if-else: Executes code based on a condition.
- ✓ for: Executes a code block iteratively a specified number of times.
- ✓ while: Executes a code block repeatedly as long as a condition is true.

8. Functions

Functions are blocks of code that perform specific tasks and can be reused.

Example:

```
#include <iostream>

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

int main() {
  int sum = add(3, 4);
  std::cout << "Sum: " << sum;
  return 0;
}
```

9. Arrays

Arrays are collections of elements of the same data type stored at contiguous memory locations.

Example:

```
#include <iostream>

int main() {
```

```
  int numbers[5] = {1, 2, 3, 4, 5};
  for (int i = 0; i < 5; i++) {
    std::cout << numbers[i] << " ";
  }
  return 0;
}
```

10. Pointers

Pointers are variables that store memory addresses of other variables.
Example:

```
#include <iostream>

int main() {
  int num = 10;
  int* ptr = &num;  // Pointer ptr stores the address of num
  std::cout << "Value of num: " << num << "\n";
  std::cout << "Address of num: " << &num << "\n";
  std::cout << "Value pointed to by ptr: " << *ptr << "\n";  // Dereferencing ptr
  return 0;
}
```

8. Classes and Objects
C++ supports object-oriented programming through classes and objects.

Example:

```
#include <iostream.h>
#include <string.h>

class Car {
public:
  std::string brand;
  int year;

  void honk() {
    std::cout << "Beep beep!";
  }
};

int main() {
  Car myCar;
  myCar.brand = "Toyota";
  myCar.year = 2020;
  std::cout << "Car brand: " << myCar.brand << "\n";
  std::cout << "Car year: " << myCar.year << "\n";
  myCar.honk();
  return 0;
}
```

3. Data Types in C++

C++ supports a variety of data types, which can be categorized into fundamental data types, derived data types, and user-defined data types. Here is an overview of the most commonly used data types in C++.

1. Fundamental Data Types

a. Integer Types:

- int: Basic integer type.
- short: Short integer, usually 16 bits.
- long: Long integer, usually 32 bits.
- long long: Very long integer, usually 64 bits.
- unsigned: Modifier to represent only non-negative values.

Example

```
int a = 10;
short b = 20;
long c = 30000;
long long d = 1000000000;
unsigned int e = 50;
```

b. Floating-Point Types:

- float: Single-precision floating-point number, usually 32 bits.
- double: Double-precision floating-point number, usually 64 bits.
- long double: Extended-precision floating-point number, varies in size.

Example:

```
float f = 3.14f;
double g = 2.718281828;
long double h = 1.41421356237309504880L;
```

c. Character Type:

- char: Character type, usually 8 bits.

Example:

```
char ch = 'A';
```

d. Boolean Type:

- bool: Boolean type, represents true or false.

Example:

```
bool isReady = true;
```

2. Derived Data Types

a. Arrays:

- Collection of elements of the same type.

Example:

```
int arr[5] = {1, 2, 3, 4, 5};
```

b. Pointers:

- Variable that stores the memory address of another variable.

Example:

```
int x = 10;
int* ptr = &x;  // Pointer to integer x
```

Programs:

Program for practising data types:

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

int main() {
  // Integer data type
  int integerVar = 100;
  cout << "Integer: " << integerVar << endl;

  // Floating-point data types
  float floatVar = 10.5f;
  double doubleVar = 20.123456789;
  long double longDoubleVar = 30.123456789123456L;
  cout << "Float: " << floatVar << endl;
  cout << "Double: " << doubleVar << endl;
  cout << "Long Double: " << longDoubleVar << endl;

  // Character data type
  char charVar = 'A';
  cout << "Character: " << charVar << endl;

  // Boolean data type
  bool boolVar = true;
  cout << "Boolean: " << boolVar << endl;

  // Short integer data type
  short shortVar = 32000;
  cout << "Short Integer: " << shortVar << endl;

  // Long integer data type
  long longVar = 100000L;
  cout << "Long Integer: " << longVar << endl;

  // Long long integer data type
  long long longLongVar = 1000000000LL;
  cout << "Long Long Integer: " << longLongVar << endl;
```

```
    return 0;
}
```

```
#include <iostream.h>
#include <string.h>

using namespace std;

// Define a class named Car
class Car {
public:
    // Attributes
    string brand;
    string model;
    int year;

    // Method to display car details
    void displayDetails() {
        cout << "Brand: " << brand << endl;
        cout << "Model: " << model << endl;
        cout << "Year: " << year << endl;
    }

    // Method to honk
    void honk() {
        cout << "Beep beep!" << endl;
    }
};

int main() {
    // Create an object of Car
    Car car1;
    car1.brand = "Toyota";
    car1.model = "Corolla";
    car1.year = 2020;

    // Create another object of Car
    Car car2;
    car2.brand = "Honda";
    car2.model = "Civic";
    car2.year = 2019;

    // Display details and honk for car1
    cout << "Car 1 details:" << endl;
    car1.displayDetails();
    car1.honk();
    cout << endl;

    // Display details and honk for car2
    cout << "Car 2 details:" << endl;
    car2.displayDetails();
```

```
    car2.honk();

    return 0;
}
```

4. Variables and Constants

Variables
Variables in C++ are used to store data that can be modified during program execution. They have a specific type that determines the kind of data they can hold, such as integers, floating-point numbers, characters, etc.

Declaring and Initializing Variables

- **Declaration**: Declares a variable with a specific type but does not initialize it.
- **Initialization**: Assigns an initial value to a variable at the time of declaration.

```
int age;          // Declaration
age = 25;         // Initialization

int year = 2024;    // Declaration and Initialization
```

Variable Naming Rules

1. Must start with a letter (uppercase or lowercase) or an underscore (_).
2. Can contain letters, digits, and underscores.
3. Cannot be a reserved keyword (like int, float, return, etc.).

```
int number;    // Valid
float _value;  // Valid
char char1;    // Valid
int 1number;   // Invalid (starts with a digit)
```

Constants

Constants are used to store data that should not be modified once initialized. In C++, constants can be defined using the const keyword or #define preprocessor directive.

Using const Keyword

The const keyword makes a variable's value immutable after initialization.

```
const int DAYS_IN_WEEK = 7;
const float PI = 3.14159;
```

Example Program Demonstrating Variables and Constants

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

```
int main() {
   // Variable declarations and initializations
   int age = 25;
   float height = 5.9;
   char grade = 'A';
   bool isStudent = true;

   // Constant declaration using const keyword
   const int DAYS_IN_WEEK = 7;
   const float PI = 3.14159;

   // Constant declaration using #define directive
   #define MAX_SCORE 100
   #define PASSING_GRADE 50

   // Output variable values
   cout << "Age: " << age << endl;
   cout << "Height: " << height << " feet" << endl;
   cout << "Grade: " << grade << endl;
   cout << "Is student: " << boolalpha << isStudent << endl; // boolalpha prints boolean as
true/false

   // Output constant values
   cout << "Days in a week: " << DAYS_IN_WEEK << endl;
   cout << "Value of PI: " << PI << endl;
   cout << "Max score: " << MAX_SCORE << endl;
   cout << "Passing grade: " << PASSING_GRADE << endl;

   // Uncommenting the next line will cause a compilation error
   // DAYS_IN_WEEK = 8; // Error: cannot modify a const variable

   return 0;
}
```

5. Operators

Operators in C++ are symbols that tell the compiler to perform specific mathematical, logical, or relational operations. Understanding operators is crucial because they are used to manipulate data and control the flow of a program. Here's a breakdown of the most commonly used operators in C++.

1. Arithmetic Operators
These operators are used to perform basic mathematical operations.

Addition (+): Adds two operands.

```
int sum = 5 + 3; // sum is 8
```

Subtraction (-): Subtracts the second operand from the first.

```
int difference = 5 - 3;  // difference is 2
```

Multiplication (*): Multiplies two operands.

```
int product = 5 * 3;  // product is 15
```

Division (/): Divides the first operand by the second. Note that if both operands are integers, the result will be an integer.

```
int quotient = 6 / 3;  // quotient is 2
```

Modulus (%): Returns the remainder of the division of the first operand by the second

```
int remainder = 5 % 3;  // remainder is 2
```

2. Relational Operators

These operators are used to compare two values. The result of a relational operation is either true or false.

Equal to (==): Checks if two operands are equal.

```
if (a == b) { /* true if a is equal to b */ }
```

Not equal to (!=): Checks if two operands are not equal.

```
if (a != b) { /* true if a is not equal to b */ }
```

Greater than (>): Checks if the first operand is greater than the second.

```
if (a > b) { /* true if a is greater than b */ }
```

Less than (<): Checks if the first operand is less than the second.

```
if (a < b) { /* true if a is less than b */ }
```

Greater than or equal to (>=): Checks if the first operand is greater than or equal to the second.

```
if (a >= b) { /* true if a is greater than or equal to b */ }
```

Less than or equal to (<=): Checks if the first operand is less than or equal to the second.

```
if (a <= b) { /* true if a is less than or equal to b */ }
```

3. Logical Operators

Logical operators are used to combine two or more conditions.

Logical AND (&&): Returns true if both operands are true.

```
if (a > 0 && b > 0) { /* true if both a and b are positive */ }
```

Logical OR (||): Returns true if at least one of the operands is true.

```
if (a > 0 || b > 0) { /* true if either a or b is positive */ }
```

Logical NOT (!): Reverses the logical state of its operand.

```
if (!a) { /* true if a is false (or 0) */ }
```

4. Assignment Operators

Assignment operators are used to assign values to variables.

Assignment (=): Assigns the value on the right to the variable on the left.

```
int a = 5; // a is assigned the value 5
```

Add and assign (+=): Adds the right operand to the left operand and assigns the result to the left operand.

```
a += 3; // equivalent to a = a + 3;
```

Subtract and assign (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.

```
a -= 2; // equivalent to a = a - 2;
```

Multiply and assign (*=): Multiplies the left operand by the right operand and assigns the result to the left operand.

```
a *= 4; // equivalent to a = a * 4;
```

Divide and assign (/=): Divides the left operand by the right operand and assigns the result to the left operand.

```
a /= 2; // equivalent to a = a / 2;
```

Modulus and assign (%=): Takes the modulus of the left operand with the right operand and assigns the result to the left operand.

```
a %= 3; // equivalent to a = a % 3;
```

5. Increment and Decrement Operators
These operators are used to increase or decrease the value of a variable by one.

Increment (++): Increases the value of a variable by one.

```
a++;  // equivalent to a = a + 1;
```

Decrement (--): Decreases the value of a variable by one.

```
a--;  // equivalent to a = a - 1;
```

6. Conditional (Ternary) Operator

This is a shorthand way of writing an if-else statement. It takes three operands.

```
int result = (a > b) ? a : b;  // If a is greater than b, result is a; otherwise, result is b.
```

7. Bitwise Operators

These operators work on the binary representation of integers.

Bitwise AND (&): Compares each bit of the two operands and returns a new number whose bits are set to 1 only if both bits are 1.

```
int result = a & b;
```

Bitwise OR (|): Compares each bit of the two operands and returns a new number whose bits are set to 1 if at least one of the bits is 1.

```
int result = a | b;
```

Bitwise XOR (^): Compares each bit of the two operands and returns a new number whose bits are set to 1 if the bits are different.

```
int result = a ^ b;
```

Bitwise NOT (~): Flips all the bits of the operand.

```
int result = ~a;
```

Left Shift (<<): Shifts the bits of the first operand to the left by the number of positions specified by the second operand.

```
int result = a << 1;
```

Right Shift (>>): Shifts the bits of the first operand to the right by the number of positions specified by the second operand.

```
int result = a >> 1;
```

8. Sizeof Operator

The sizeof operator returns the size (in bytes) of a data type or variable.

```
int size = sizeof(int);  // size will hold the value 4 (on most systems)
```

Summary

- **Arithmetic operators** allow you to perform basic math operations.
- **Relational and logical operators** help you compare values and make decisions.
- **Assignment operators** let you assign values to variables.
- **Increment and decrement operators** modify a variable's value by one.
- **Bitwise operators** manipulate individual bits in a number.
- The **sizeof operator** tells you how much memory a data type or variable takes up.

These operators are fundamental tools in C++ programming and are used extensively in writing code. Understanding them will help you perform various tasks in your programs, from simple calculations to complex logic operations.

Example program:

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

int main() {
  // Declare variables
  int a = 10;
  int b = 3;
  int result;

  // Arithmetic Operators
  result = a + b;  // Addition
  cout << "a + b = " << result << endl;

  result = a - b;  // Subtraction
  cout << "a - b = " << result << endl;

  result = a * b;  // Multiplication
  cout << "a * b = " << result << endl;

  result = a / b;  // Division
  cout << "a / b = " << result << endl;

  result = a % b;  // Modulus
  cout << "a % b = " << result << endl;

  // Relational Operators
  cout << "a == b: " << (a == b) << endl;  // Equal to
  cout << "a != b: " << (a != b) << endl;  // Not equal to
  cout << "a > b: " << (a > b) << endl;    // Greater than
  cout << "a < b: " << (a < b) << endl;    // Less than
  cout << "a >= b: " << (a >= b) << endl;  // Greater than or equal to
  cout << "a <= b: " << (a <= b) << endl;  // Less than or equal to

  // Logical Operators
```

```
    cout << "(a > 5) && (b < 5): " << ((a > 5) && (b < 5)) << endl;  // Logical AND
    cout << "(a > 5) || (b > 5): " << ((a > 5) || (b > 5)) << endl;  // Logical OR
    cout << "!(a == b): " << !(a == b) << endl;                 // Logical NOT

    // Assignment Operators
    result = a;  // Assignment
    cout << "result = a: " << result << endl;

    result += b;  // Add and assign
    cout << "result += b: " << result << endl;

    result -= b;  // Subtract and assign
    cout << "result -= b: " << result << endl;

    result *= b;  // Multiply and assign
    cout << "result *= b: " << result << endl;

    result /= b;  // Divide and assign
    cout << "result /= b: " << result << endl;

    result %= b;  // Modulus and assign
    cout << "result %= b: " << result << endl;

    return 0;
}
```

6. Control Structures

Control structures in C++ are fundamental building blocks that allow you to control the flow of your program. They enable you to make decisions, repeat tasks, and choose between different paths based on conditions. Here's an introduction to the most common control structures in C++.

1. Decision-Making Structures

if Statement
The if statement allows you to execute a block of code only if a specified condition is true.

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

Example:

```
int age = 18;
if (age >= 18) {
    cout << "You are eligible to vote.";
}
```

if-else Statement

The if-else statement provides an alternative block of code to execute if the condition is false.

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

Example:

```
int age = 16;
if (age >= 18) {
   cout << "You are eligible to vote.";
} else {
   cout << "You are not eligible to vote.";
}
```

else if Ladder

The else if ladder allows you to check multiple conditions.

```
if (condition1) {
   // Code to execute if condition1 is true
} else if (condition2) {
   // Code to execute if condition2 is true
} else {
   // Code to execute if none of the above conditions are true
}
```

Example:

```
int marks = 85;
if (marks >= 90) {
   cout << "Grade: A";
} else if (marks >= 80) {
   cout << "Grade: B";
} else {
   cout << "Grade: C";
}
```

switch Statement

The switch statement allows you to select one of many code blocks to be executed, based on the value of an expression.

```
switch (expression) {
   case value1:
      // Code to execute if expression equals value1
      break;
   case value2:
      // Code to execute if expression equals value2
      break;
   // You can have any number of case statements
   default:
```

```
    // Code to execute if expression doesn't match any case
}
```

Example:

```
int day = 3;
switch (day) {
  case 1:
    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  default:
    cout << "Invalid day";
}
```

2. Looping Structures

for Loop

The for loop is used to repeat a block of code a specific number of times.

```
for (initialization; condition; update) {
  // Code to execute in each iteration
}
```

Example:

```
for (int i = 1; i <= 5; i++) {
  cout << i << " ";
}
// Output: 1 2 3 4 5
```

while Loop

The while loop repeats a block of code as long as the condition is true.

```
while (condition) {
  // Code to execute as long as the condition is true
}
```

Example:

```
int i = 1;
while (i <= 5) {
  cout << i << " ";
  i++;
}
// Output: 1 2 3 4 5
```

do-while Loop
The do-while loop is similar to the while loop, but it guarantees that the code block is executed at least once because the condition is checked after the loop body.

```
do {
  // Code to execute
} while (condition);
```

Example:

```
int i = 1;
do {
  cout << i << " ";
  i++;
} while (i <= 5);
// Output: 1 2 3 4 5
```

3. Jump Statements

break Statement
The break statement is used to exit a loop or switch statement prematurely.

```
for (int i = 1; i <= 5; i++) {
  if (i == 3) {
    break;
  }
  cout << i << " ";
}
// Output: 1 2
```

continue Statement
The continue statement skips the rest of the loop iteration and continues with the next iteration.

```
for (int i = 1; i <= 5; i++) {
  if (i == 3) {
    continue;
  }
  cout << i << " ";
}
// Output: 1 2 4 5
```

return Statement
The return statement is used to exit a function and optionally return a value to the calling function.

```
int add(int a, int b) {
  return a + b;  // returns the sum of a and b
}

int result = add(3, 4);  // result is 7
```

Summary

- Decision-making structures (if, else if, else, switch) allow you to execute code based on conditions.
- Looping structures (for, while, do-while) let you repeat code multiple times.
- Jump statements (break, continue, return) control the flow of loops and functions.

These control structures form the foundation of C++ programming, enabling you to write flexible and efficient code.

7. Arrays and Strings

In C++, arrays and strings are fundamental data structures that allow you to store and manipulate collections of data. Here's an introduction to arrays and strings in C++ for beginners.

1. Arrays
An array is a collection of elements of the same data type stored in contiguous memory locations. Each element in an array is identified by an index, starting from 0.

Declaring and Initializing Arrays
You can declare an array by specifying the data type, array name, and the number of elements (size).

```
int numbers[5];  // Declares an array of 5 integers
```

You can also initialize an array at the time of declaration:

```
int numbers[5] = {10, 20, 30, 40, 50};  // Initializes the array with values
```

If you don't specify the size, it will be determined automatically based on the number of initializers:

```
int numbers[] = {10, 20, 30, 40, 50};  // Size is automatically set to 5
```

Accessing Array Elements
You can access individual elements of an array using their index:

```
cout << "First element: " << numbers[0] << endl;
cout << "Second element: " << numbers[1] << endl;
```

Here's a simple program that demonstrates how to work with arrays:

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

int main() {
  int numbers[5] = {10, 20, 30, 40, 50};

  // Loop through the array and print each element
  for (int i = 0; i < 5; i++) {
    cout << "Element at index " << i << " is: " << numbers[i] << endl;
  }
```

```
    return 0;
}
```

2. Strings

In C++, a string is a sequence of characters. Strings are often used to store text and are supported by the `string` class in the C++ Standard Library.

Declaring and Initializing Strings

You can declare and initialize strings in various ways:

```
#include <iostream.h>
#include <string.h>
using namespace std;

int main() {
    string greeting = "Hello, World!";  // Initializes a string
    cout << greeting << endl;

    string name;  // Declares an empty string
    cout << "Enter your name: ";
    cin >> name;  // Input a string from the user
    cout << "Hello, " << name << "!" << endl;

    return 0;
}
```

String Operations

You can perform various operations on strings, such as concatenation, finding the length, and accessing characters.

- Concatenation: You can concatenate strings using the `+` operator.

```
 string firstName = "John";
 string lastName = "Doe";
 string fullName = firstName + " " + lastName;
 cout << fullName << endl;  // Output: John Doe
```

- Length: The `length()` or `size()` method returns the number of characters in the string.

```
 cout << "Length of fullName: " << fullName.length() << endl;
```

- Accessing Characters: You can access individual characters using the `[]` operator.

```
 char firstChar = fullName[0];  // First character of the string
 cout << "First character: " << firstChar << endl;
```

Here's a simple program that shows how to work with strings:

```
#include <iostream.h>
#include <string.h>
using namespace std;
```

```
int main() {
   string name;
   cout << "Enter your name: ";
   getline(cin, name);  // Input a string with spaces
   cout << "Hello, " << name << "!" << endl;

   string greeting = "Welcome to C++ programming.";
   cout << "Greeting: " << greeting << endl;

   // Concatenate strings
   string fullName = name + " " + "Smith";
   cout << "Full name: " << fullName << endl;

   // Find the length of the string
   cout << "Length of your name: " << name.length() << endl;

   return 0;
}
```

Summary

- Arrays allow you to store and access a fixed-size sequence of elements of the same type.
- Strings in C++ are sequences of characters used to store and manipulate text data.
- Both arrays and strings can be accessed and manipulated using loops, operators, and functions provided by C++.

8. Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design software. It is one of the most popular approaches to writing code, especially in C++. OOP makes it easier to manage and organize complex programs by focusing on objects rather than just functions and logic.

Key Concepts of OOP:

1. **Classes and Objects**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**
5. **Abstraction**

Let's break these down with simple explanations.

1. Classes and Objects
Class: A class is like a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that an object can have. For example, consider a class Car that describes what a car is.

```
class Car {
```

```
public:
  string brand;
  string model;
  int year;

  void start() {
    cout << "Car started!" << endl;
  }
};
```

Object: An object is an instance of a class. If a class is a blueprint, then an object is the actual car built from that blueprint. Each object can have its own values for the attributes defined in the class.

```
Car myCar;  // Create an object of class Car
myCar.brand = "Toyota";
myCar.model = "Corolla";
myCar.year = 2020;

myCar.start();  // Call the start method
```

2. Encapsulation

Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit or class. It also involves restricting direct access to some of the object's components, which is done using access specifiers (public, private, protected).

Public: Members declared as public are accessible from outside the class.
Private: Members declared as private are not accessible from outside the class; they can only be accessed from within the class.

Example:

```
class Car {
private:
  int speed;  // Private attribute

public:
  void setSpeed(int s) {
    speed = s;  // Public method to set speed
  }

  int getSpeed() {
    return speed;  // Public method to get speed
  }
};

int main() {
  Car myCar;
  myCar.setSpeed(120);  // Set speed using public method
```

```
  cout << "Speed: " << myCar.getSpeed() << endl;  // Access speed using public method
  return 0;
}
```

3. Inheritance

Inheritance allows a new class to inherit the properties and behaviors of an existing class. The new class is called the **derived class** (or child class), and the existing class is the **base class** (or parent class).

Example:

```
class Vehicle {
public:
  string brand = "Ford";
  void honk() {
    cout << "Beep beep!" << endl;
  }
};

class Car : public Vehicle {
public:
  string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();  // Inherited method
  cout << myCar.brand + " " + myCar.model << endl;  // Inherited and new properties
  return 0;
}
```

4. Polymorphism

Polymorphism means "many forms." In OOP, it allows objects to be treated as instances of their parent class, even though they might actually be instances of a derived class. This is often achieved using method overriding, where a derived class provides its own version of a method that is already defined in its base class.

Example:

```
class Animal {
public:
  void sound() {
    cout << "Some sound" << endl;
  }
};

class Dog : public Animal {
public:
```

```
  void sound() override {
    cout << "Woof!" << endl;
  }
};

class Cat : public Animal {
public:
  void sound() override {
    cout << "Meow!" << endl;
  }
};

int main() {
  Animal* myAnimal = new Animal();
  Animal* myDog = new Dog();
  Animal* myCat = new Cat();

  myAnimal->sound();  // Outputs: Some sound
  myDog->sound();     // Outputs: Woof!
  myCat->sound();     // Outputs: Meow!

  delete myAnimal;
  delete myDog;
  delete myCat;

  return 0;
}
```

5. Abstraction

Abstraction is the concept of hiding complex implementation details and showing only the essential features of an object. This is usually achieved through abstract classes and interfaces. In C++, abstraction can be implemented using abstract classes that have pure virtual functions.

Example:

```
class Shape {
public:
  virtual void draw() = 0;  // Pure virtual function
};

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

class Square : public Shape {
public:
  void draw() override {
```

```
    cout << "Drawing Square" << endl;
  }
};

int main() {
  Shape* shape1 = new Circle();
  Shape* shape2 = new Square();

  shape1->draw();  // Outputs: Drawing Circle
  shape2->draw();  // Outputs: Drawing Square

  delete shape1;
  delete shape2;

  return 0;
}
```

Summary

- **Classes and Objects**: Classes define the blueprint for objects. Objects are instances of classes.
- **Encapsulation**: Encapsulation bundles data and methods together and restricts access to some of the object's components.
- **Inheritance**: Inheritance allows a class to inherit properties and methods from another class.
- **Polymorphism**: Polymorphism allows objects to be treated as instances of their parent class, and it supports method overriding.
- **Abstraction**: Abstraction hides complex details and shows only the necessary features.

OOP helps to create organized, reusable, and maintainable code. It models real-world entities, making it easier to design and implement software systems.

9. Access Modifiers

Access modifiers in C++ are keywords used to define the accessibility or scope of class members (attributes and methods). They control how the data and methods within a class can be accessed from outside the class.

C++ has three primary access modifiers:

1. public
2. private
3. protected

1. Public

- Syntax: 'public:'
- Members declared as 'public' can be accessed from anywhere in the program. This means any code, whether inside or outside the class, can access these members.

Example:

```
class MyClass {
public:
  int x;  // Public attribute

  void display() {  // Public method
    cout << "Value of x: " << x << endl;
  }
};

int main() {
  MyClass obj;
  obj.x = 10;  // Accessible from outside the class
  obj.display();  // Accessible from outside the class
  return 0;
}
```

2. Private

- Syntax: 'private:'
- Members declared as 'private' can only be accessed from within the class. They are not accessible from outside the class, ensuring that the internal implementation details are hidden and protected.

Example:

```
class MyClass {
private:
  int x;  // Private attribute

public:
  void setX(int val) {  // Public method to set private attribute
    x = val;
  }

  void display() {  // Public method to access private attribute
    cout << "Value of x: " << x << endl;
  }
};

int main() {
  MyClass obj;
  // obj.x = 10;  // Error! Cannot access private member directly
  obj.setX(10);  // Set value through public method
  obj.display();  // Display value through public method
  return 0;
}
```

3. Protected

- Syntax: 'protected:'

- Members declared as 'protected' are similar to 'private' members, but they can also be accessed by derived classes (child classes). This is useful in inheritance, where a base class wants to allow derived classes to access certain members but still keep them hidden from outside the hierarchy.

Example:

```
class Base {
protected:
  int x;  // Protected attribute

public:
  void setX(int val) {
    x = val;
  }
};

class Derived : public Base {
public:
  void display() {
    cout << "Value of x: " << x << endl;  // Accessible in derived class
  }
};

int main() {
  Derived obj;
  obj.setX(10);  // Set value using public method
  obj.display();  // Display value, accessing protected member from derived class
  return 0;
}
```

Summary

- Public:
  - Accessible from anywhere (inside and outside the class).
  - Suitable for interface methods and data meant to be exposed to users of the class.

- Private:
  - Accessible only from within the class.
  - Used to hide implementation details and protect data from external interference.

- Protected:
  - Accessible within the class and by derived classes.
  - Used when you want to share data or methods with derived classes while keeping them hidden from the rest of the program.

10. Constructors and Destructors

Constructors and destructors are special member functions in C++ that are used to initialize and clean up objects when they are created and destroyed, respectively. Understanding these concepts is fundamental to mastering object-oriented programming in C++.

1. Constructors

A constructor is a special function that is automatically called when an object of a class is created. Constructors are used to initialize the object's attributes and perform any setup necessary for the object.

Key Points about Constructors:

- Name: The constructor has the same name as the class.
- No return type: Constructors do not have a return type, not even 'void'.
- Automatic invocation: They are called automatically when an object is created.
- Overloading: Multiple constructors can be defined in a class, each with different parameters (this is called constructor overloading).

Types of Constructors:

1. Default Constructor:
- A constructor that takes no arguments.
- If you don't define any constructor, C++ provides a default constructor automatically.

```
class MyClass {
public:
    MyClass() {  // Default constructor
        cout << "Default Constructor called!" << endl;
    }
};

int main() {
    MyClass obj;  // Calls the default constructor
    return 0;
}
```

2. Parameterized Constructor:
- A constructor that takes one or more parameters, allowing you to initialize an object with specific values at the time of creation.

```
class MyClass {
private:
    int x;

public:
    MyClass(int val) {  // Parameterized constructor
        x = val;
        cout << "Parameterized Constructor called with value: " << x << endl;
```

```
    }
};

int main() {
    MyClass obj(10);  // Calls the parameterized constructor
    return 0;
}
```

3. Copy Constructor:

- A constructor that creates a new object as a copy of an existing object.
- It is invoked when an object is passed by value, returned from a function, or explicitly copied.

```
class MyClass {
private:
    int x;

public:
    MyClass(int val) {  // Parameterized constructor
        x = val;
    }

    MyClass(const MyClass &obj) {  // Copy constructor
        x = obj.x;
        cout << "Copy Constructor called!" << endl;
    }
};

int main() {
    MyClass obj1(10);  // Calls the parameterized constructor
    MyClass obj2 = obj1;  // Calls the copy constructor
    return 0;
}
```

2. Destructors

A destructor is a special member function that is automatically called when an object goes out of scope or is explicitly deleted. Destructors are used to release resources that the object may have acquired during its lifetime, such as memory or file handles.

Key Points about Destructors:

- Name: The destructor has the same name as the class, but with a tilde ('~') prefix.
- No arguments: Destructors do not take any arguments and cannot be overloaded.
- No return type: Destructors do not have a return type.
- Automatic invocation: They are called automatically when the object is destroyed.

Example of a Destructor:

```
class MyClass {
public:
    MyClass() {  // Constructor
```

```
    cout << "Constructor called!" << endl;
  }

  ~MyClass() {  // Destructor
    cout << "Destructor called!" << endl;
  }
};

int main() {
  MyClass obj;  // Constructor is called
  return 0;  // Destructor is called automatically when the object goes out of scope
}
```

- The constructor is called when the object 'obj' is created.
- The destructor is called automatically when 'obj' goes out of scope at the end of the 'main()' function.

3. Constructor and Destructor in Dynamic Memory Allocation

When you use dynamic memory allocation (e.g., using 'new'), constructors and destructors play a critical role in managing memory.

Example:

```
class MyClass {
public:
  MyClass() {
    cout << "Constructor called!" << endl;
  }

  ~MyClass() {
    cout << "Destructor called!" << endl;
  }
};

int main() {
  MyClass *obj = new MyClass();  // Constructor is called

  delete obj;  // Destructor is called when the object is deleted
  return 0;
}
```

- 'new MyClass();' dynamically allocates memory for a 'MyClass' object, and the constructor is called.
- 'delete obj;' releases the allocated memory, and the destructor is called.

Summary

- Constructors are special member functions used to initialize objects. They are called automatically when an object is created.
  - Default Constructor: Takes no arguments and is provided automatically if no other constructor is defined.

- Parameterized Constructor: Allows the initialization of objects with specific values.
- Copy Constructor: Creates a new object as a copy of an existing object.

- Destructors are special member functions used to clean up when an object is destroyed. They are automatically called when an object goes out of scope or is deleted.

11. Polymorphism

Polymorphism is one of the core concepts of Object-Oriented Programming (OOP) in C++. The term "polymorphism" is derived from Greek, meaning "many shapes" or "many forms." In programming, polymorphism allows one interface to be used for a general class of actions. This means that the same operation can behave differently on different classes.

Types of Polymorphism in C++

There are two main types of polymorphism in C++:

1. Compile-time Polymorphism (also known as Static Polymorphism)
2. Run-time Polymorphism (also known as Dynamic Polymorphism)

1. Compile-time Polymorphism

Compile-time polymorphism is achieved through function overloading and operator overloading. In this type of polymorphism, the function to be invoked is determined at compile time.

Function Overloading
Function overloading allows you to define multiple functions with the same name but different parameter lists. The correct function to be called is determined by the number and type of arguments passed.

Example:

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

class Print {
public:
  void show(int x) {  // Function to print an integer
    cout << "Integer: " << x << endl;
  }

  void show(double x) {  // Function to print a double
    cout << "Double: " << x << endl;
  }

  void show(string x) {  // Function to print a string
    cout << "String: " << x << endl;
  }
};
```

```
int main() {
   Print p;
   p.show(5);       // Calls show(int)
   p.show(3.14);    // Calls show(double)
   p.show("Hello"); // Calls show(string)
   return 0;
}
```

- The 'show()' function is overloaded to handle different data types: 'int', 'double', and 'string'.
- The appropriate version of 'show()' is called based on the argument passed.

Operator Overloading

Operator overloading allows you to redefine the behavior of operators for user-defined types (such as objects). This enables operators to work with objects in a way that is intuitive and consistent with their typical use.

Example:

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

class Complex {
private:
   int real, imag;

public:
   Complex(int r = 0, int i = 0) : real(r), imag(i) {}

   // Operator overloading for '+'
   Complex operator + (const Complex &obj) {
      Complex result;
      result.real = real + obj.real;
      result.imag = imag + obj.imag;
      return result;
   }

   void print() {
      cout << real << " + " << imag << "i" << endl;
   }
};

int main() {
   Complex c1(3, 2), c2(1, 7);
   Complex c3 = c1 + c2;  // Uses overloaded '+' operator
   c3.print();  // Outputs: 4 + 9i
   return 0;
}
```

- The '+' operator is overloaded to add two 'Complex' numbers.
- When 'c1 + c2' is executed, the overloaded '+' operator is invoked.

2. Run-time Polymorphism

Run-time polymorphism is achieved through inheritance and virtual functions. In this type of polymorphism, the function to be invoked is determined at runtime.

Inheritance and Virtual Functions

In C++, a base class pointer or reference can point to an object of a derived class. If a function is marked as 'virtual' in the base class, the function to be executed is determined at runtime based on the actual object type being pointed to.

Example:

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

class Animal {
public:
  virtual void sound() {  // Virtual function
    cout << "Animal makes a sound" << endl;
  }
};

class Dog : public Animal {
public:
  void sound() override {  // Override the base class method
    cout << "Dog barks" << endl;
  }
};

class Cat : public Animal {
public:
  void sound() override {  // Override the base class method
    cout << "Cat meows" << endl;
  }
};

int main() {
  Animal* a1 = new Dog();
  Animal* a2 = new Cat();

  a1->sound();  // Outputs: Dog barks
  a2->sound();  // Outputs: Cat meows

  delete a1;
  delete a2;

  return 0;
}
```

- The 'sound()' function in the 'Animal' class is marked as 'virtual', which means it can be overridden by derived classes ('Dog' and 'Cat').
- Even though the pointer 'a1' is of type 'Animal*', the 'sound()' function of the 'Dog' class is called, demonstrating run-time polymorphism.

Summary

- Compile-time Polymorphism:
  - Achieved through function overloading and operator overloading.
  - The function or operator to be invoked is determined at compile time.

- Run-time Polymorphism:
  - Achieved through inheritance and virtual functions.
  - The function to be invoked is determined at runtime based on the object type.

12. Friend Classes and Functions

In C++, the concept of friend classes and friend functions allows certain functions or classes to access the private and protected members of another class. This provides a way to break the usual access controls, but it should be used cautiously to maintain encapsulation.

1. Friend Function

A friend function is a function that is not a member of a class but has access to the class's private and protected members. You can declare a function as a friend by using the 'friend' keyword inside the class definition.

Example

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

class Box {
private:
   double width;

public:
   // Constructor
   Box(double w) : width(w) {}

   // Friend function declaration
   friend void printWidth(Box box);
};

// Definition of friend function
void printWidth(Box box) {
   // Accessing private member of the Box class
   cout << "Width of the box: " << box.width << endl;
}

int main() {
   Box box(10.5);
   printWidth(box);  // Friend function can access private members
   return 0;
```

```
}
```

- The 'printWidth' function is declared as a friend of the 'Box' class.
- Even though 'printWidth' is not a member of the 'Box' class, it can access the private member 'width'.

2. Friend Class

A friend class is a class that is allowed to access the private and protected members of another class. If class 'B' is a friend of class 'A', then all member functions of class 'B' can access the private and protected members of class 'A'.

Example

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

class Engine {
private:
  int horsepower;

public:
  Engine(int hp) : horsepower(hp) {}

  // Friend class declaration
  friend class Car;
};

class Car {
public:
  void showEnginePower(Engine engine) {
    // Accessing private member of the Engine class
    cout << "Engine horsepower: " << engine.horsepower << endl;
  }
};

int main() {
  Engine engine(400);
  Car car;
  car.showEnginePower(engine);  // Car class can access private members of Engine
  return 0;
}
```

- The 'Car' class is declared as a friend of the 'Engine' class.
- As a result, the 'Car' class can access the private member 'horsepower' of the 'Engine' class.

Summary

- Friend Function: A non-member function that has access to the private and protected members of a class.
- Friend Class: A class that can access the private and protected members of another class.

- Usage: Friend functions and classes should be used sparingly and only when necessary, as they expose the internal details of a class.

13. Standard Library Functions

The C++ Standard Library provides a rich set of functions that are designed to perform various tasks such as input/output operations, string manipulation, mathematical computations, and more. These functions are grouped into different libraries, each serving a specific purpose.

Categories of Standard Library Functions in C++

1. Input/Output Functions
2. String Functions
3. Mathematical Functions
4. Algorithm Functions
5. Utility Functions
6. Date and Time Functions

1. Input/Output Functions

These functions handle the input and output operations in C++. They are mostly found in the '<iostream>' header.

- 'cin': Standard input stream (used to take input from the user).
- 'cout': Standard output stream (used to print output to the console).
- 'cerr': Standard error stream (used to display error messages).
- 'getline': Reads a line of text, including spaces.
- 'fstream', 'ifstream', 'ofstream': Handle file input and output.

Example:

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

int main() {
   string name;
   cout << "Enter your name: ";
   getline(cin, name);
   cout << "Hello, " << name << "!" << endl;
   return 0;
}
```

2. String Functions

These functions perform operations on strings and are found in the '<string>' header.

- 'length()': Returns the length of the string.
- 'substr()': Returns a substring from a string.
- 'find()': Searches for a substring within a string.

- 'append()': Appends one string to another.
- 'compare()': Compares two strings.

Example:

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

int main() {
   string str = "Hello, World!";
   cout << "Length of string: " << str.length() << endl;
   cout << "Substring: " << str.substr(7, 5) << endl;
   return 0;
}
```

3. Mathematical Functions

These functions are used for performing mathematical operations and are found in the '<cmath>' header.

- 'sqrt()': Calculates the square root.
- 'pow()': Raises a number to a power.
- 'abs()': Returns the absolute value.
- 'sin()', 'cos()', 'tan()': Trigonometric functions.
- 'log()': Returns the natural logarithm.

Example:

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

int main() {
   cout << "Square root of 16: " << sqrt(16) << endl;
   cout << "2 raised to the power 3: " << pow(2, 3) << endl;
   return 0;
}
```

4. Algorithm Functions

These functions are used for various algorithms and are found in the '<algorithm>' header.

- 'sort()': Sorts elements in a range.
- 'reverse()': Reverses the order of elements in a range.
- 'find()': Finds an element in a range.
- 'max()' and 'min()': Returns the maximum and minimum of two values.

Example:

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

```
int main() {
   vector<int> vec = {3, 1, 4, 1, 5, 9};
   sort(vec.begin(), vec.end());
   cout << "Sorted vector: ";
   for (int i : vec) {
      cout << i << " ";
   }
   cout << endl;
   return 0;
}
```

5. Utility Functions

Utility functions provide general-purpose features and are found in headers like '<utility>' and '<functional>'.

- 'pair': Combines two values.
- 'make_pair()': Creates a pair.
- 'swap()': Swaps two values.
- 'move()': Moves an object.

Example:

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

int main() {
   pair<int, string> p = make_pair(1, "one");
   cout << "Pair: " << p.first << ", " << p.second << endl;
   return 0;
}
```

6. Date and Time Functions

These functions handle date and time operations and are found in the '<ctime>' header.

- 'time()': Returns the current time.
- 'localtime()': Converts time to local time.
- 'difftime()': Calculates the difference between two times.
- 'strftime()': Formats date and time.

Example:

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

int main() {
   time_t now = time(0);
```

```
    tm *ltm = localtime(&now);
    cout << "Current year: " << 1900 + ltm->tm_year << endl;
    return 0;
}
```

Summary

C++ Standard Library functions are extensive and provide essential tools for working with various aspects of programming, including input/output operations, string manipulation, mathematical computations, and more. Understanding and utilizing these functions effectively is crucial for writing efficient and powerful C++ programs.