# How C++ Works

C++ code is written in a file called a source file with the extension .cpp. The collection of code in this file is known as the source code.

## Explanation of #include <iostream>

The # symbol indicates a preprocessor statement, meaning it’s executed before the actual compilation (which converts the source code into machine-readable code).

The #include directive is a preprocessor instruction that tells the compiler to include the contents of the file mentioned within < > into the source file. The file iostream is a standard library that contains predefined functions for input (like cin) and output (like cout).

The contents of iostream are copied into the current file before compilation.

### What is a function in programming?

**A function is a group of statements bundled together to perform a specific task. Functions allow code to be organized into reusable blocks, making programs more modular and easier to maintain.**

## The main() Function

The main() function is the entry point of any C++ program, and it is executed first when the program runs.

The keyword int means that this function returns an integer value, and return 0; indicates successful execution.

## Example Code

#include <iostream>  
  
using namespace std;  
  
int main() {  
 cout << "Hello World"; // Output message to console  
 return 0; // Return 0 to indicate success  
}

**Data & Information & Knowledge & Wisdom**

**Data and Information are important concepts in the world of computing and decision-making**

**Data : unstructured information, typically without inherent meaning.**

**Types of Data :**

**There are two types of Data:**

1. **Quantitative: Quantitative data refers to numerical information like weight, height, etc.**
2. **Qualitative: Qualitative data refers to non-numeric information like opinions, perceptions, etc.**

**Information : refers to processed, organized, and structured data. It gives context to the facts and facilitates decision-making or relevance to make it useful.**

Data → Processing → Information

***Processing***

**Raw Data → Structure it → Organize it → Analyze it → Give it meaning**

**Summary: The Cycle of Data to Information**

**Collect Data → Add Meaning → Generate Information**

**Difference between Information and Data**

**Data:**

* **Raw facts**
* **Unstructured/Unorganized**
* **Unanalyzed**
* **Unprocessed**
* **Has no meaning**

**Information:**

* **Driven from data**
* **Structured/Organized**
* **Analyzed**
* **Processed**
* **Has meaning**

***Knowledge & Wisdom***

**Knowledge : Over time, information accumulates and is understood, leading to knowledge.**

**Wisdom : When knowledge is applied effectively, it leads to wisdom.**

**Arrows indicate:**

* **"With time" (transition from Information to Knowledge).**
* **"Applied" (transition from Knowledge to Wisdom).**

***Data → Input → Processing → Output***

***Wisdom → Knowledge → Information***

***Applied → With Time***

**Compiler and an Interpreter**

**1. What is a Compiler?**

A compiler is a program that translates source code from a programming language (C++) into machine code that the computer can understand and execute. This is necessary because computers do not directly understand high-level programming languages.

The compiler doesn’t care about the file extension (such as .cpp for C++). It only processes the actual code inside the file. Interestingly, code can even exist inside files with extensions like .jpg or .png, though this is not common practice.

Compiler

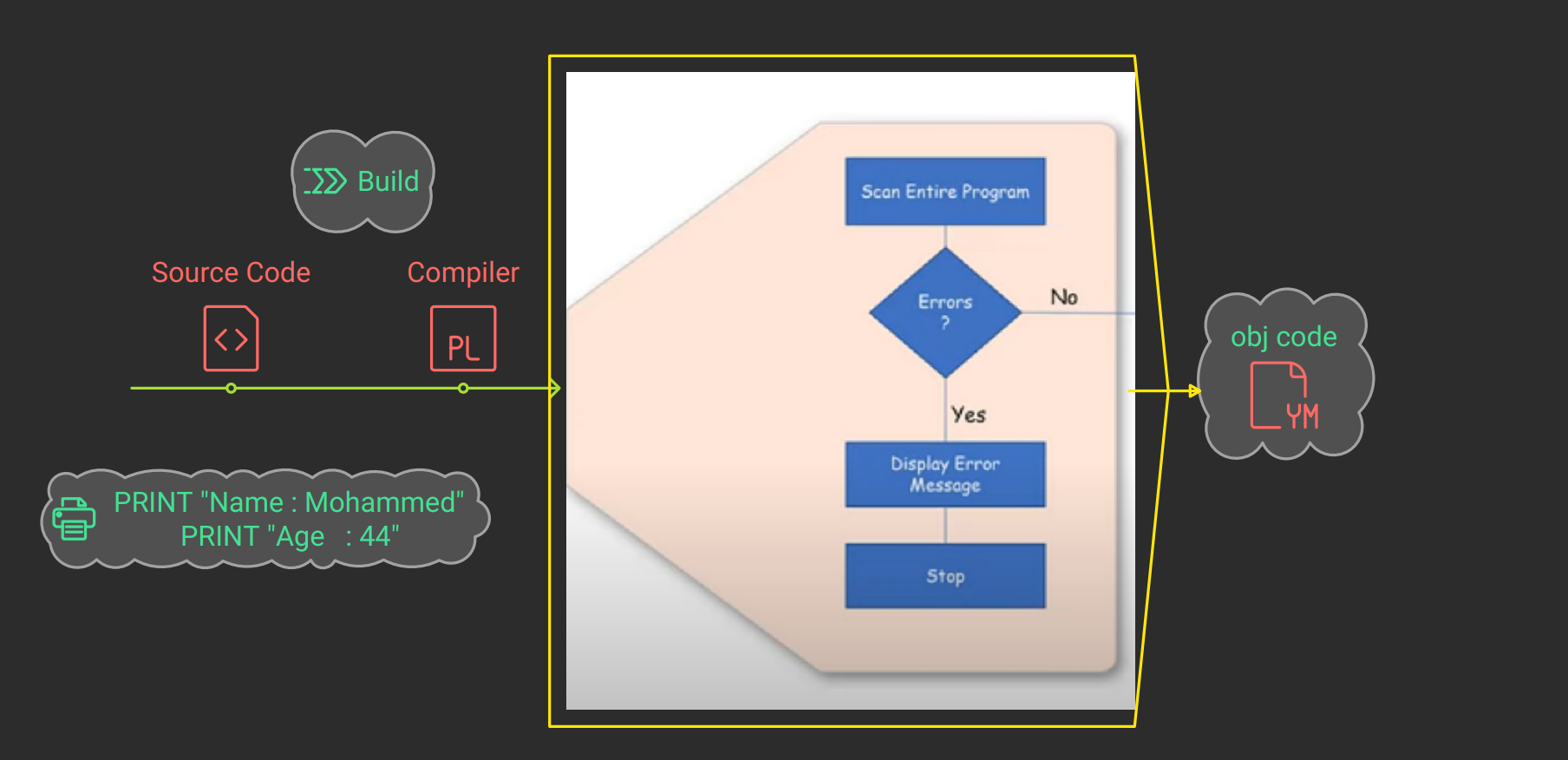
Here's a concise summary:

1. **Source Code** (C++): The original code written by the programmer.
2. **Compiler** (PL): Converts source code into object code.
3. **Linker**: Combines object code with libraries to create an executable.
4. **Execution**: The program runs.
5. **Loader**: Loads the program into memory.

**Output**:

Name: Mohammed

Age: 44



**2. Interpreter**

**A screenshot of a computer

Description automatically generated**

**2. Difference Between a Compiler and an Interpreter**

A compiler translates the entire program into machine code before execution, while an interpreter translates and executes the code line-by-line, without generating a full machine code file.

**What is a Header File?**

**A header file contains reusable functions and code, usually with a .h extension. It helps organize code and prevent repetition by using the #include directive.**

**Types of #include Syntax**

1. **Angle Brackets (< >):**
2. **#include <filename.h>**
   * **Used for system header files.**
   * **Example: #include <stdio.h>**
3. **Double Quotes (" "):**
4. **#include "filename.h"**
   * **Used for user-defined header files.**
   * **Example: #include "myheader.h"**

**Header File Syntax**

* **Standard Header File in System Directory**:
* #include <stdio.h>
  + Used for system header files that are pre-defined.
* **User-Defined Header File in Source Directory**:
* #include "Someheader.h"
  + Used for custom header files, typically in the same directory as the source file.

**Steps Involved in Compiling C++ Code**

**1. Preprocessing**

* **Role: Handles preprocessor directives (e.g., #include), macro expansion, and conditional compilation.**

**2. Creating a Translation Unit**

* **Role: The compiler checks the code, including header files, for syntax.**

**3. Compilation**

* **Role: Translates the source code into machine language and stores it as an object file (.obj).**

**Key Phases in Compilation:**

* **Lexical Analysis: Breaks the code into tokens (keywords, operators, identifiers).**
* **Syntax Analysis: Checks if the code follows C++ grammar.**
* **Semantic Analysis: Ensures the code is logically meaningful.**
* **Intermediate Code Generation: Converts source code into an intermediate representation (e.g., Postfix or Polish notation).**
* **Code Optimization: (Optional) Simplifies code to enhance performance and reduce memory usage.**
* **Output Code Generation: Generates machine code and stores it as an object file (.obj).**

**4. Linking**

* **Role: The linker combines object files into a single executable file, resolving references to external libraries or functions.**

**you need a translator**

A diagram of software

Description automatically generated

**Key Concepts in Programming**

1. **Programming Language**  
   A tool used to write a set of instructions for a computer to perform specific tasks or operations.
2. **Machine Language**  
   The language consisting of binary or hexadecimal instructions that a computer can directly understand and execute.
3. **Code**  
   A series of instructions written by the programmer to instruct the computer on what to do.
4. **Source Code**  
   A group of instructions written in a high-level programming language.
5. **Object Code**  
   A group of instructions written in machine language (binary or hexadecimal), generated from source code after compilation.
6. **Compiler/Interpreter**  
   Tools that translate high-level code into machine language. A compiler translates all code at once, while an interpreter does it line by line.
7. **Fast Language**  
   A language considered fast when it is close to the hardware (low-level), making execution more efficient.
8. **Slow Language**  
   A language considered slow when it is farther from the hardware (high-level), requiring more translation to machine code.
9. **High-Level Language**  
   A language that is more abstract, close to human languages, making it easier for programmers to understand and use (e.g., Python, Java).
10. **Low-Level Language**  
    A language that is closer to machine code, typically harder for humans to understand, and directly manipulates hardware (e.g., Assembly, C).
11. **Readable Language**  
    A language is considered readable when it is further from the hardware, using more human-readable syntax (high-level language).

**C++ Language Syntax**

**1. Syntax**

* **Definition: Syntax is a set of rules that must be followed when writing code according to the structure of the programming language.**
* **Syntax Highlighter: Some code editors feature a "syntax highlighter," which colors different types of code in specific colors to enhance readability.**

**2. Whitespace in C++**

* **Handling Whitespace**: C++ does not recognize spaces and line breaks between code as significant; it treats them as irrelevant.
* **Single Line Execution**: The compiler considers all lines of code as if they were written on a single line. You can write multiple separate lines of code or combine all commands into a single line.
* **Semicolon Requirement**: In either case, you must place a semicolon (;) after each line to indicate its termination.

### Example:

-There is no difference between executing the following code:

std::cout << " Line 1\n"; std::cout << "Line 2\n"; std::cout << "Line 3\n";

-And this code:

std::cout << " Line 1\n";

std::cout << "Line 2\n";

std::cout << "Line 3\n";

**Output:**

Line 1

Line 2

Line 3

* **Conclusion**: C++ treats spaces and line breaks as irrelevant characters unless they are inside double quotes.

**The Semicolon (;) in C++**

**1. Purpose of the Semicolon**

* **Termination of Statements**: The semicolon is used to terminate statements in C++. It signals to the compiler that the current statement has ended.

**2. Placement of the Semicolon**

* **Flexible Placement**: The semicolon can be placed anywhere after the line of code.
* **New Line Placement**: Because C++ treats spaces and line breaks as insignificant, the semicolon can also be placed at the beginning of a new line without changing its meaning.

**3. Example Comparison**

* **Example 1**: All semicolons at the end of each line:

std::cout << "Line 1\n";

std::cout << "Line 2\n";

std::cout << "Line 3\n";

**Example 2**: Semicolons placed at the beginning of the new lines:

std::cout << "Line 1\n"

; std::cout << "Line 2\n"

;

std::cout << "Line 3\n";

**4. Output Confirmation**

* **Output of Both Examples**: Both code snippets produce the same output:

Line 1

Line 2

Line 3

### Curly Braces ({ })

* Definition: Curly braces are used to define a block of code, which is a group of programming statements.

### Strings in Programming

* Definition: In programming, sequences of characters are known as strings. Strings are distinguished by being enclosed in double quotes (""). This indicates to the programming language that the text is not part of the code or language components but is simply regular text.
* Code vs. String:

If text is written without double quotes, the language will interpret it as code.

If a programming statement is placed within double quotes, the language will treat it as a normal string and will not execute it.

### Backslash (\)

* Special Treatment: The backslash has special handling in strings. The character that follows the backslash is called a special character because it performs a unique function.
* Escape Sequences: The backslash combined with the subsequent special character is known as an escape sequence. Each escape sequence has its own distinct feature and usage.

**The cout Command**

* **Definition**: The cout command is short for "Character Output." In this context, "Character" refers to everything that can be written on a computer, including characters like letters, symbols, and even spaces.
* **Functionality**:
  + The semicolon (;) is also considered a character.
  + The cout function outputs a sequence of characters to the screen.

**Input vs. Output**

* **Output**: This refers to the results produced by a program, which is the opposite of "Input," the data received by the program from the user.

**Purpose of cout**

* Thus, cout stands for "Character Output," which clarifies its function: it prints (outputs) a collection of characters as output on a display screen (often referred to as a black screen or terminal).

**Stream Insertion Operator (>>)**

* **Definition**: The >> symbols are known as the stream insertion operators.
* **Functionality**: As indicated by the name, these operators insert the data that follows them into the function that precedes them. In this case, they send the sequence of characters to the cout function to be displayed on the screen.

### ****What are comments in C++?****

Comments in C++ are used to summarize an algorithm, identify a variable’s purpose, or clarify a code segment that appears unclear. Comments are also used for:

* Comments are used for easier debugging.
* It makes a program more readable and gives an overall description of the code.
* Comments are helpful in skipping the execution of some parts of the code.
* Every time a program or code is reused after long periods of time, the comment recaps all the information of the code quickly.

**Types of Comments in C++**

* **Single-line comment**
* **Multi-line comment**

**Creating Single-Line Comments in C++**

**How are single-line comments created?**  
In C++, single-line comments are initiated with // (double forward slashes). The compiler ignores everything on that line after the slashes.

**Example:**

// This is a comment

std::cout << "Hello, World!\n"; // This is another comment

**Creating Multi-Line Comments in C++**

**How are multi-line comments created?**  
Multi-line comments in C++ are enclosed between /\* and \*/. The compiler disregards everything between these markers.

**Example:**

/\* This is a

multi-line comment \*/

std::cout << "Hello, World!\n"; /\* Another comment \*/

**Step 4: Common Mistakes Regarding the Use of Comments**

**What are common mistakes?**  
A frequent mistake is adding comments to code that is already self-explanatory. If the purpose of a code segment can be easily understood from the code itself, additional comments may be unnecessary and could clutter the code.

Debugger

Namespace

### Step 1: What is the purpose of using namespace std?

**Answer:**  
The statement using namespace std is used to import all the elements (functions, objects, and types) from the std namespace into the current scope of a program. This allows the programmer to use elements like cout or cin without having to prefix them with std::

**Example without using namespace std:**

std::cout << "Hello, World!";

**Example with using namespace std:**

using namespace std;

cout << "Hello, World!";

### Step 2: Why is using namespace std considered bad practice?

**Answer:**  
The main reason using namespace std is considered bad practice is that the std namespace is vast and importing everything from it can lead to name conflicts. When you bring the entire std namespace into the current scope, there’s a higher chance of conflicts with other identifiers (e.g., if you define a variable or function with the same name as something in std).

### Step 3: What is the recommended alternative to using namespace std?

**Answer:**  
Instead of using using namespace std, it is recommended to explicitly specify the namespace with the **scope resolution operator** (::) each time you reference a standard library object or function. This ensures clarity and avoids potential naming conflicts.

std::cout << "Hello, World!"; // Instead of cout, use std::cout

std::vector<int> myVector; // Instead of vector, use std::vector

### Step 4: Why is the alternative considered better?

**Answer:**  
By explicitly using std::, you make it clear that the identifier belongs to the standard library, avoiding ambiguity. It also minimizes the risk of accidentally overriding or conflicting with identifiers from the standard library or other libraries that might be included in the program.

**Preprocessing Directives**

**Step 1: Definition of a Variable**

**Answer: A variable is a data container used to store values in computer memory**

### Step 2: What are the mandatory naming rules for identifiers?

**Answer:**

1. **Must Be Unique:  
   Variable names must be unique; two variables cannot have the same name unless one is within a namespace or both are in different namespaces.**
2. **Case Sensitive:  
   Variable names are case-sensitive, meaning that name and Name are considered different variables.**
3. **Cannot Start With Numbers:  
   The first character of a variable name cannot be a number, but numbers are allowed in other positions.**
4. **Numbers, Letters, or Underscore:  
   Variable names can only contain numbers, letters, or the underscore (\_) character. Names can begin with an underscore.**
5. **No White Space or Special Characters:  
   Variable names cannot contain special characters (like #) or spaces, as spaces would mislead the compiler into interpreting them as separate words.**
6. **Reserved Keywords:  
   Variable names cannot be any reserved keywords in the language (e.g., int, class).**

### Step 3: What are the optional best practices for naming identifiers?

**Answer:**

1. **Related Names:**  
   Identifier names should be logical and descriptive of what the variable will store. This helps improve code readability.
2. **Writing Style:**  
   Adopting globally recognized coding styles can enhance the clarity and consistency of the code.

**Variables Advanced**

### Step 1: Can a variable be declared without being assigned a value immediately?

**Answer:**  
Yes, a variable can be declared without assigning a value at the time of declaration. The value can be assigned later in the code.

**Example:**

int a;

a = 100;

### Step 2: Can multiple variables of the same type be declared together, and can they all be assigned values?

**Answer:**  
Yes, multiple variables of the same type can be declared together by separating them with commas. You can assign values to all of them, some of them, or none at the time of declaration.

**Example 1 (without initial values):**

int b, c, d;

b = 10;

c = 20;

d = 30;

**Example 2 (with mixed initial values):**

int e, f = 40, g;

e = 20;

g = 60;

### Step 3: Can multiple variables be assigned the same value in a single statement?

**Answer:**  
Yes, you can assign the same value to multiple variables in a single statement using the chaining assignment method.

Example

int h, i, j;

h = i = j = 10;

### Variable Scope?

* two types of variable scopes:

1. Local Variables
2. Global Variables

1. Local Variables

Variables defined within a function or block are said to be local to those functions.

* Anything between ‘{‘ and ‘}’ is said to inside a block.
* Local variables do not exist outside the block in which they are declared, i.e. they **can not** be accessed or used outside that block.
* **Declaring local variables**: Local variables are declared inside a block.

2.Global Variables

**As the name suggests, Global Variables can be accessed from any part of the program.**

* **They are available through out the life time of a program.**
* **They are declared at the top of the program outside all of the functions or blocks.**
* **Declaring global variables: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.**

#include <iostream>

using namespace std;

// Global variable

int globalVar = 10;

void display() {

// Local variable

int localVar = 5;

cout << "Local Variable: " << localVar << endl; // Accessing local variable

cout << "Global Variable: " << globalVar << endl; // Accessing global variable

}

int main() {

cout << "Global Variable accessed in main: " << globalVar << endl; // Access global variable

// cout << "Local Variable in main: " << localVar << endl;

return 0;

}

### ****Constants const Keyword****

Constants in C++ refer to variables with fixed values that cannot be changed

**How to Define Constants in C++?**

We can define the constants in C++ using three ways:

1. **Using const Keyword**
2. **Using constexpr Keyword**
3. **Using #define Preprocessor**

### ****Using const keyword****

The syntax is simple:

const data\_type variable\_name = value;

Example:

const int maxLimit = 100;

### ****Constants using**** constexpr ****Keyword****

The syntax is similar to const:

constexpr data\_type variable\_name = value;

Example:

constexpr int arraySize = 100;

### ****Constants using**** #define ****Preprocessor****

The syntax uses the preprocessor directive:

#define MACRO\_NAME replacement\_value

Example:

#define PI 3.14159

### Final Answer: ****Comparison and Best Practices****

 **Use const** : when you need an immutable value that might be initialized at runtime or compile time.

 **Use constexpr** : for values that need to be evaluated at compile time, optimizing performance and ensuring constant expressions.

 **Avoid #define :**  for constants if you need type safety, as #define simply performs text substitution and can introduce subtle bugs.

**Constants using const**

* **Syntax**: const type name = value;
* **Evaluation Time**: Runtime/Compile
* **Type Safety**: Yes
* **Scope Support**: Yes
* **Usage Scenario**: For variables that won’t change after initialization.

**Constants using constexpr**

* **Syntax**: constexpr type name = value;
* **Evaluation Time**: Compile time
* **Type Safety**: Yes
* **Scope Support**: Yes
* **Usage Scenario**: Compile-time constants and optimizations, when values are known before program execution.

**Constants using #define**

* **Syntax**: #define NAME value
* **Evaluation Time**: Preprocessing
* **Type Safety**: No
* **Scope Support**: No
* **Usage Scenario**: Avoid for constants, use for macros. Typically useful in cases requiring text substitution during preprocessing.

Escape Sequences Character

**C++ Data Types**

C++ supports the following types of data:

1. **Primary/Built-in/Fundamental Data Types**
2. **Derived Data Types**
3. **Abstract/User-defined Data Types**

**Main Categories of Data Types in C++**

C++ data types can be broadly classified into three categories:

**1. Primitive Data Types**

These are built-in or predefined data types that can be directly used by the user to declare variables.

**Primitive data types available in C++ include:**

* Integer
* Character
* Boolean
* Floating Point
* Double Floating Point
* Valueless or Void
* Wide Character

#include <iostream>

#include <string>

using namespace std;

int main() {

// Integer data types

int a = 10;

short b = 20;

long c = 30;

long long d = 40;

cout << "Integer data types: " << endl;

cout << "int: " << a << endl;

cout << "short: " << b << endl;

cout << "long: " << c << endl;

cout << "long long: " << d << endl;

// Floating-point data types

float e = 3.14f;

double f = 3.141592;

long double g = 3.14159265358979L;

cout << "Floating-point data types: " << endl;

cout << "float: " << e << endl;

cout << "double: " << f << endl;

cout << "long double: " << g << endl;

// Character data types

char h = 'a';

wchar\_t i = L'b';

char16\_t j = u'c';

char32\_t k = U'd';

cout << "Character data types: " << endl;

cout << "char: " << h << endl;

wcout << "wchar\_t: " << i << endl;

cout << "char16\_t: " << j << endl;

cout << "char32\_t: " << k << endl;

// Boolean data type

bool l = true;

bool m = false;

cout << "Boolean data type: " << endl;

cout << "true: " << l << endl;

cout << "false: " << m << endl;

// String data type

string n = "Hello, world!";

cout << "String data type: " << endl;

cout << n << endl;

return

0;

}

**2. Derived Data Types**

Derived data types are created from primitive or built-in data types.

**Types of Derived Data Types:**

* Function
* Array
* Pointer
* Reference

**3. Abstract or User-Defined Data Types**

These data types are defined by the user, allowing greater flexibility and abstraction.

**User-defined data types in C++ include:**

* Class
* Structure
* Union
* Enumeration
* typedef-defined Data Type

**Macro Constants**

**Character Limits**

* **CHAR\_MIN**: The minimum value for an object of type char.
* **CHAR\_MAX**: The maximum value for an object of type char.
* **SCHAR\_MIN**: The minimum value for an object of type signed char.
* **SCHAR\_MAX**: The maximum value for an object of type signed char.
* **UCHAR\_MAX**: The maximum value for an object of type unsigned char.
* **CHAR\_BIT**: The number of bits in a char object.
* **MB\_LEN\_MAX**: The maximum number of bytes in a multi-byte character.

**Short Integer Limits**

* **SHRT\_MIN**: The minimum value for an object of type short int.
* **SHRT\_MAX**: The maximum value for an object of type short int.
* **USHRT\_MAX**: The maximum value for an object of type unsigned short int.

**Integer Limits**

* **INT\_MIN**: The minimum value for an object of type int.
* **INT\_MAX**: The maximum value for an object of type int.
* **UINT\_MAX**: The maximum value for an object of type unsigned int.

**Long Integer Limits**

* **LONG\_MIN**: The minimum value for an object of type long int.
* **LONG\_MAX**: The maximum value for an object of type long int.
* **ULONG\_MAX**: The maximum value for an object of type unsigned long int.

**Long Long Integer Limits**

* **LLONG\_MIN**: The minimum value for an object of type long long int.
* **LLONG\_MAX**: The maximum value for an object of type long long int.
* **ULLONG\_MAX**: The maximum value for an object of type unsigned long long int.

Notes

These examples show how to use the limits macros defined in <climits> to check and display the range of different data types in C++.

**Type Conversion in C++**

**1. Implicit Type Conversion (Automatic Type Conversion)**

* **Definition:**  
  Implicit type conversion is automatically performed by the compiler when expressions contain variables of different data types. This ensures there is no loss of data by promoting smaller data types to the larger data type in the expression.
* **When It Happens:**  
  Implicit conversion occurs when you mix variables of different data types in an expression. The smaller data types are promoted to the larger data type in order to prevent data loss.
* **Example Data Type Hierarchy:**
* bool -> char -> short int -> int -> unsigned int -> long -> unsigned long -> long long -> float -> double -> long double
* **Notes on Data Loss:**
  + Loss of signs when a signed type is implicitly converted to unsigned.
  + Overflow may occur when converting large types like long long to float.

**2. Explicit Type Conversion (using Cast Operator)**

Explicit type conversion (also known as **type casting**) is when the programmer explicitly tells the compiler to convert one type to another, typically using a cast operator.

There are **four types** of casting in C++:

**Summary of Casting Types**

1. **Static Cast**
   * **Use Case:** Used for most conversions, especially between related types like int to float.
   * **Syntax:**
   * static\_cast<target\_type>(expression);
2. **Dynamic Cast**
   * **Use Case:** Used for safe casting in class hierarchies, especially with polymorphism (e.g., casting between base and derived class pointers).
   * **Syntax:**
   * dynamic\_cast<target\_type>(expression);
3. **Const Cast**
   * **Use Case:** Adds or removes the const qualifier from a variable, allowing modification of const variables.
   * **Syntax:**
   * const\_cast<target\_type>(expression);
4. **Reinterpret Cast**
   * **Use Case:** Used for low-level operations that reinterpret the bits of an object as a different type, such as casting between raw pointers and other types.
   * **Syntax:**
   * reinterpret\_cast<target\_type>(expression);

Code

#include <iostream>

using namespace std;

int main() {

// Static Cast

int x = 10;

float y = static\_cast<float>(x); // Convert int to float

cout << "Static Cast: " << y << endl;

// Dynamic Cast

class Base {

public:

virtual void show() { cout << "Base class\n"; }

};

class Derived : public Base {

public:

void show() override { cout << "Derived class\n"; }

};

Base\* basePtr = new Derived();

Derived\* derivedPtr = dynamic\_cast<Derived\*>(basePtr); // Safe cast

if (derivedPtr) {

cout << "Dynamic Cast: Successful\n";

derivedPtr->show();

}

// Const Cast

const int a = 50;

int\* b = const\_cast<int\*>(&a); // Remove const

\*b = 100; // Modify value

cout << "Const Cast: " << \*b << endl;

// Reinterpret Cast

long long num = 1234567890;

char\* ch = reinterpret\_cast<char\*>(&num); // Treat the address as a char pointer

cout << "Reinterpret Cast: " << \*ch << endl; // Print the first byte

delete basePtr; // Clean up

return 0;

}

**1. Arithmetic Operators**

These operators perform basic arithmetic operations.

* **Operators:**
  + + : Addition
  + - : Subtraction
  + \* : Multiplication
  + / : Division
  + % : Modulus (remainder)

**Example:**

int a = 10, b = 5;

int sum = a + b; // Addition

int diff = a - b; // Subtraction

int product = a \* b; // Multiplication

int quotient = a / b; // Division

int remainder = a % b; // Modulus

**2. Relational Operators**

These operators are used to compare two values.

* **Operators:**
  + == : Equal to
  + != : Not equal to
  + > : Greater than
  + < : Less than
  + >= : Greater than or equal to
  + <= : Less than or equal to

**Example:**

int a = 10, b = 5;

bool result1 = a == b; // false

bool result2 = a != b; // true

bool result3 = a > b; // true

bool result4 = a < b; // false

**3. Logical Operators**

These operators are used to perform logical operations, typically with boolean values.

* **Operators:**
  + && : Logical AND
  + || : Logical OR
  + ! : Logical NOT

**Example:**

bool x = true, y = false;

bool result1 = x && y; // Logical AND (false)

bool result2 = x || y; // Logical OR (true)

bool result3 = !x; // Logical NOT (false)

**4. Bitwise Operators**

These operators perform bit-level operations on integer types.

* **Operators:**
  + & : Bitwise AND
  + | : Bitwise OR
  + ^ : Bitwise XOR
  + ~ : Bitwise NOT
  + << : Left shift
  + >> : Right shift

**Example:**

int a = 5, b = 3; // 5 = 0101, 3 = 0011 (in binary)

int and\_result = a & b; // Bitwise AND (1)

int or\_result = a | b; // Bitwise OR (7)

int xor\_result = a ^ b; // Bitwise XOR (6)

int not\_result = ~a; // Bitwise NOT (-6)

int left\_shift = a << 1; // Left shift (10)

int right\_shift = a >> 1; // Right shift (2)

**5. Assignment Operators**

These operators are used to assign values to variables.

* **Operators:**
  + = : Simple assignment
  + += : Add and assign
  + -= : Subtract and assign
  + \*= : Multiply and assign
  + /= : Divide and assign
  + %= : Modulus and assign

**Example:**

int a = 10;

a += 5; // a = a + 5 -> 15

a -= 3; // a = a - 3 -> 12

a \*= 2; // a = a \* 2 -> 24

a /= 4; // a = a / 4 -> 6

a %= 3; // a = a % 3 -> 0

**6. Ternary or Conditional Operators**

The ternary operator is a shorthand for an if-else statement. It evaluates a condition and returns one of two values based on the condition.

* **Syntax:**
* condition ? expression1 : expression2;
* **Example:**

int a = 10, b = 5;

int max = (a > b) ? a : b; // If a > b, max = a, otherwise max = b

cout << "Max value is: " << max; // Output: 10

**Summary of Operators**

1. **Arithmetic Operators:**  
   +, -, \*, /, %
2. **Relational Operators:**  
   ==, !=, >, <, >=, <=
3. **Logical Operators:**  
   &&, ||, !
4. **Bitwise Operators:**  
   &, |, ^, ~, <<, >>
5. **Assignment Operators:**  
   =, +=, -=, \*=, /=, %=
6. **Ternary/Conditional Operator:**  
   condition ? expr1 : expr2

**C++ sizeof Operator**

Here’s the code for all the examples of the sizeof operator in C++:

**1. Number of Bytes Taken by Data Types**

#include <iostream>

using namespace std;

int main()

{

cout << "No of Bytes taken up by char is " << sizeof(char) << endl;

cout << "No of Bytes taken up by int is " << sizeof(int) << endl;

cout << "No of Bytes taken up by float is " << sizeof(float) << endl;

cout << "No of Bytes taken up by double is " << sizeof(double) << endl;

cout << "No of Bytes taken up by long is " << sizeof(long) << endl;

return 0;

}

**2. Number of Bytes Taken by Variables**

#include <iostream>

using namespace std;

int main()

{

int a;

float b;

char g;

cout << "No of Bytes taken up by a is " << sizeof(a) << endl;

cout << "No of Bytes taken up by b is " << sizeof(b) << endl;

cout << "No of Bytes taken up by g is " << sizeof(g) << endl;

return 0;

}

**3. Number of Bytes Taken by an Expression**

#include <iostream>

using namespace std;

int main()

{

int a = 5;

long x = 9;

double p = 10.2;

float g = 2.5;

cout << "No of Bytes taken up by (a+g) is " << sizeof(a + g) << endl;

cout << "No of Bytes taken up by (a+x) is " << sizeof(a + x) << endl;

cout << "No of Bytes taken up by (a+p) is " << sizeof(a + p) << endl;

cout << "No of Bytes taken up by (x+p) is " << sizeof(x + p) << endl;

return 0;

}

**4. Find the Size of an Array**

#include <iostream>

using namespace std;

int main()

{

int x[] = {1, 2, 3, 5, 6, 7, 8, 9};

int length = sizeof(x) / sizeof(x[0]);

cout << "Length of the array is " << length << endl;

return 0;

}

**5. Find the Size of a Class**

#include <iostream>

using namespace std;

class GFG

{

int x;

};

int main()

{

GFG g;

cout << "Size of class GFG is in bytes : " << sizeof(g) << endl;

return 0;

}

**6. Find the Size of Pointers**

#include <iostream>

using namespace std;

int main()

{

int \*a = new int(10);

char \*g = new char('g');

double \*d = new double(7.5);

cout << "Size of pointer a is " << sizeof(a) << endl;

cout << "Size of pointer \*a is " << sizeof(\*a) << endl;

cout << "Size of pointer g is " << sizeof(g) << endl;

cout << "Size of pointer \*g is " << sizeof(\*g) << endl;

cout << "Size of pointer d is " << sizeof(d) << endl;

cout << "Size of pointer \*d is " << sizeof(\*d) << endl;

return 0;

}

**7. Nesting of sizeof() Operator**

#include <iostream>

using namespace std;

int main()

{

int x;

double y;

cout << "Nesting of sizeof operator is implemented as sizeof(x\*sizeof(y)) :" << sizeof(x \* sizeof(y)) << endl;

return 0;

}

**8. Find the Size of a Structure**

#include <iostream>

using namespace std;

struct gfg

{

int z;

float d;

char s[20];

} g;

int main()

{

cout << "Size of structure is " << sizeof(g) << endl;

return 0;

}

**9. Find the Size of a Union**

#include <iostream>

using namespace std;

union gfg

{

int z;

double d;

} g;

int main()

{

cout << "Size of union is " << sizeof(g) << endl;

return 0;

}

Here’s a detailed breakdown of the examples of the **Scope Resolution Operator (::)** in C++:

**Example 1: Accessing a Global Variable When a Local Variable Has the Same Name**

#include<iostream>

using namespace std;

int x; // Global x

int main() {

int x = 10; // Local x

cout << "Value of global x is " << ::x; // Access global x

cout << "\nValue of local x is " << x; // Access local x

return 0;

}

**Output:**

Value of global x is 0

Value of local x is 10

**Example 2: Defining a Function Outside a Class**

#include <iostream>

using namespace std;

class A {

public:

void fun(); // Declaration

};

void A::fun() {

cout << "fun() called"; // Definition outside class

}

int main() {

A a;

a.fun(); // Function call

return 0;

}

**Output:**

fun() called

**Example 3: Accessing a Class’s Static Variables**

#include<iostream>

using namespace std;

class Test {

static int x;

public:

static int y;

void func(int x) {

cout << "Value of static x is " << Test::x; // Access static variable

cout << "\nValue of local x is " << x; // Access local variable

}

};

int Test::x = 1;

int Test::y = 2;

int main() {

Test obj;

int x = 3;

obj.func(x); // Calling function

cout << "\nTest::y = " << Test::y; // Access static variable

return 0;

}

**Output:**

Value of static x is 1

Value of local x is 3

Test::y = 2

**Example 4: Using :: in Multiple Inheritance**

#include<iostream>

using namespace std;

class A {

protected:

int x;

public:

A() { x = 10; }

};

class B {

protected:

int x;

public:

B() { x = 20; }

};

class C: public A, public B {

public:

void fun() {

cout << "A's x is " << A::x; // Access A's x

cout << "\nB's x is " << B::x; // Access B's x

}

};

int main() {

C c;

c.fun(); // Calling function

return 0;

}

**Output:**

A's x is 10

B's x is 20

**Example 5: Using :: with Namespaces**

#include <bits/stdc++.h>

#include <iostream>

using namespace std;

string name1 = "GFG";

string favlang = "python";

string companyName = "GFG\_2.0";

class Developer {

public:

string name = "krishna";

string favLang = "c++";

string company = "GFG";

Developer(string favlang, string company)

: favLang(favlang), company(companyName) {}

};

int main() {

Developer obj = Developer("python", "GFG");

cout << "favourite Language : " << obj.favLang << endl;

cout << "company Name : " << obj.company << endl;

return 0;

}

**Output:**

favourite Language : python

company Name : GFG\_2.0

**Example 6: Using :: to Refer to a Class Inside Another Class**

#include <iostream>

using namespace std;

class outside {

public:

int x;

class inside {

public:

int x;

static int y;

int foo() {

return 0;

}

};

};

int outside::inside::y = 5;

int main() {

outside A;

outside::inside B;

return 0;

}

**Example 7: Referencing a Member of the Base Class in the Derived Object**

#include <iostream>

using namespace std;

class Base {

public:

void func() {

cout << "This is Base class" << endl;

}

};

class Derived : public Base {

public:

void func() {

cout << "This is Derived class" << endl;

}

};

int main() {

Derived obj;

obj.Base::func(); // Call Base class function

obj.func(); // Call Derived class function

return 0;

}

**Output:**

This is Base class

This is Derived class

**Summary of Examples:**

1. **Access Global Variable:** Use :: to access a global variable when a local variable has the same name.
2. **Define Functions Outside Class:** Use :: to define class functions outside the class.
3. **Access Static Variables:** Use :: to access a static variable in a class.
4. **Multiple Inheritance:** Resolve ambiguity in multiple inheritance using ::.
5. **Namespaces:** Use :: to reference a class in a specific namespace.
6. **Nested Classes:** Use :: to refer to a class defined inside another class.
7. **Base Class Member:** Use :: to call a method from a base class in the derived class.

These examples illustrate the versatility of the scope resolution operator in C++. Let me know if you need more details!

**C++ Input/Output**

**C++ Control Statements**

**Here’s a detailed breakdown of Decision-Making Statements in C++:**

**1. if Statement**

**The if statement is used to execute a block of code only if a specified condition is true.**

**Syntax:**

**if (condition) {**

**// Code to be executed if condition is true**

**}**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int num = 10;**

**if (num > 0) {**

**cout << "The number is positive." << endl;**

**}**

**return 0;**

**}**

**Output:**

**The number is positive.**

**2. if-else Statement**

**The if-else statement is used to execute one block of code if the condition is true, and another block if the condition is false.**

**Syntax:**

**if (condition) {**

**// Code to be executed if condition is true**

**} else {**

**// Code to be executed if condition is false**

**}**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int num = -5;**

**if (num >= 0) {**

**cout << "The number is positive." << endl;**

**} else {**

**cout << "The number is negative." << endl;**

**}**

**return 0;**

**}**

**Output:**

**The number is negative.**

**3. if-else-if Ladder**

**The if-else-if ladder is used to check multiple conditions. If one condition is true, it executes the corresponding block. If no conditions are true, the else block is executed.**

**Syntax:**

**if (condition1) {**

**// Code to be executed if condition1 is true**

**} else if (condition2) {**

**// Code to be executed if condition2 is true**

**} else {**

**// Code to be executed if none of the above conditions are true**

**}**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int num = 75;**

**if (num >= 90) {**

**cout << "Grade A" << endl;**

**} else if (num >= 75) {**

**cout << "Grade B" << endl;**

**} else if (num >= 50) {**

**cout << "Grade C" << endl;**

**} else {**

**cout << "Fail" << endl;**

**}**

**return 0;**

**}**

**Output:**

**Grade B**

**4. Nested if Statement**

**A nested if statement is an if statement inside another if statement. It is used when you want to test multiple conditions in a hierarchy.**

**Syntax:**

**if (condition1) {**

**if (condition2) {**

**// Code to be executed if both condition1 and condition2 are true**

**}**

**}**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int num = 25;**

**if (num > 0) {**

**if (num % 2 == 0) {**

**cout << "The number is positive and even." << endl;**

**} else {**

**cout << "The number is positive and odd." << endl;**

**}**

**}**

**return 0;**

**}**

**Output:**

**The number is positive and odd.**

**5. switch Statement**

**The switch statement is used to test a variable against multiple values. It is a more efficient alternative to using multiple if-else statements when checking the same variable against different conditions.**

**Syntax:**

**switch (expression) {**

**case value1:**

**// Code to be executed if expression == value1**

**break;**

**case value2:**

**// Code to be executed if expression == value2**

**break;**

**default:**

**// Code to be executed if no case matches**

**}**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int day = 3;**

**switch (day) {**

**case 1: cout << "Monday"; break;**

**case 2: cout << "Tuesday"; break;**

**case 3: cout << "Wednesday"; break;**

**case 4: cout << "Thursday"; break;**

**case 5: cout << "Friday"; break;**

**case 6: cout << "Saturday"; break;**

**case 7: cout << "Sunday"; break;**

**default: cout << "Invalid day"; break;**

**}**

**return 0;**

**}**

**Output:**

**Wednesday**

**6. Conditional Operator (?:)**

**The conditional operator is a shorthand for an if-else statement. It evaluates a condition and returns one of two values based on whether the condition is true or false.**

**Syntax:**

**condition ? expression1 : expression2;**

**If the condition is true, expression1 is evaluated; otherwise, expression2 is evaluated.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int num = 10;**

**string result = (num > 0) ? "Positive" : "Negative";**

**cout << result << endl;**

**return 0;**

**}**

**Output:**

**Positive**

**7. Jump Statements: break, continue, goto, return**

**- break: The break statement is used to exit from a loop or switch statement prematurely.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**for (int i = 0; i < 5; i++) {**

**if (i == 3) {**

**break; // Exit the loop when i == 3**

**}**

**cout << i << " ";**

**}**

**return 0;**

**}**

**Output:**

**0 1 2**

**- continue: The continue statement is used to skip the current iteration of a loop and continue with the next iteration.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**for (int i = 0; i < 5; i++) {**

**if (i == 2) {**

**continue; // Skip when i == 2**

**}**

**cout << i << " ";**

**}**

**return 0;**

**}**

**Output:**

**0 1 3 4**

**- goto: The goto statement transfers control to another part of the program. It should be used cautiously to avoid confusing code flow.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int num = 5;**

**if (num == 5) {**

**goto label; // Jump to label if num is 5**

**}**

**cout << "This will not be printed." << endl;**

**label:**

**cout << "Jumped to label." << endl;**

**return 0;**

**}**

**Output:**

**Jumped to label.**

**- return: The return statement is used to exit a function and optionally return a value.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int add(int a, int b) {**

**return a + b; // Return the sum of a and b**

**}**

**int main() {**

**cout << "Sum: " << add(5, 3) << endl;**

**return 0;**

**}**

**Output:**

**Sum: 8**

**Summary of Decision-Making Statements:**

1. **if Statement: Executes code if a condition is true.**
2. **if-else Statement: Executes code if the condition is true, and another block if false.**
3. **if-else-if Ladder: Used to check multiple conditions.**
4. **Nested if Statement: One if statement inside another if.**
5. **switch Statement: Used to compare a variable against multiple values.**
6. **Conditional Operator (?:): A shorthand for if-else.**
7. **Jump Statements: Includes break, continue, goto, and return to control the flow of the program.**

**These decision-making structures are essential for writing logical, control-driven programs in C++. Let me know if you need more details!**

**Types of Loops in C++**

**C++ provides three main types of loops for repetitive execution of code: while, for, and do-while. Each loop has its unique structure and behavior in terms of when the condition is checked and how the loop body is executed.**

**1. while Loop**

**The while loop checks the condition first before executing the loop body. If the condition is false initially, the body of the loop will not be executed at all.**

**Syntax:**

**while (condition) {**

**// Code to be executed while condition is true**

**}**

**Explanation:**

* **The condition is evaluated.**
* **If it is true, the loop body executes.**
* **After executing the body, the condition is checked again.**
* **The loop continues until the condition becomes false.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int i = 0;**

**while (i < 5) {**

**cout << i << " ";**

**i++; // Incrementing i**

**}**

**return 0;**

**}**

**Output:**

**0 1 2 3 4**

**2. for Loop**

**The for loop is used when you know beforehand how many times you want to execute a statement or a block of statements. It provides initialization, condition-checking, and updating of the loop variable all in one line.**

**Syntax:**

**for (initialization; condition; update) {**

**// Code to be executed while condition is true**

**}**

**Explanation:**

* **Initialization: Runs once before the loop starts.**
* **Condition: Checked before each iteration; if true, the loop body executes.**
* **Update: Runs after each iteration of the loop.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**for (int i = 0; i < 5; i++) {**

**cout << i << " ";**

**}**

**return 0;**

**}**

**Output:**

**0 1 2 3 4**

**3. Range-Based for Loop**

**The range-based for loop introduced in C++11 provides a simpler and more concise way to iterate over containers like arrays, vectors, and maps without needing an index or iterator.**

**Syntax:**

**for (element : container) {**

**// Code to be executed for each element in the container**

**}**

**Example with vector:**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**int main() {**

**vector<int> v = { 0, 1, 2, 3, 4, 5 };**

**for (auto i : v) {**

**cout << i << ' ';**

**}**

**return 0;**

**}**

**Output:**

**0 1 2 3 4 5**

**Example with an array:**

**#include <iostream>**

**using namespace std;**

**int main() {**

**int a[] = {0, 1, 2, 3, 4, 5};**

**for (int n : a) {**

**cout << n << ' ';**

**}**

**return 0;**

**}**

**Output:**

**0 1 2 3 4 5**

**Example with map:**

**#include <iostream>**

**#include <map>**

**using namespace std;**

**int main() {**

**map<int, int> MAP({ {1, 10}, {2, 20}, {3, 30} });**

**for (auto i : MAP) {**

**cout << "Key: " << i.first << ", Value: " << i.second << '\n';**

**}**

**return 0;**

**}**

**Output:**

**Key: 1, Value: 10**

**Key: 2, Value: 20**

**Key: 3, Value: 30**

**Range-Based for Loop (C++17 and Higher)**

**In C++17 and above, you can also use structured bindings to directly unpack a map entry (which is a pair object) into two separate variables, making the code more readable.**

**Syntax:**

**for (auto& [key, value] : container) {**

**// Code to use key and value**

**}**

**Example with map:**

**#include <iostream>**

**#include <map>**

**using namespace std;**

**int main() {**

**map<int, int> MAP({ {1, 10}, {2, 20}, {3, 30} });**

**for (auto& [key, value] : MAP) {**

**cout << key << " has value " << value << endl;**

**}**

**return 0;**

**}**

**Output:**

**1 has value 10**

**2 has value 20**

**3 has value 30**

**4. do-while Loop**

**The do-while loop ensures that the body of the loop is executed at least once, as the condition is checked after executing the loop body.**

**Syntax:**

**do {**

**// Code to be executed**

**} while (condition);**

**Explanation:**

* **The loop body is executed once.**
* **After executing the body, the condition is checked.**
* **If the condition is true, the loop runs again; otherwise, the loop terminates.**

**Example:**

**#include<iostream>**

**using namespace std;**

**int main() {**

**int i = 0;**

**do {**

**cout << i << " ";**

**i++; // Incrementing i**

**} while (i < 5);**

**return 0;**

**}**

**Output:**

**0 1 2 3 4**

**Summary of Loops:**

* **while loop: First checks the condition, then executes the body.**
* **for loop: Initializes, checks the condition, executes the body, and updates the variable.**
* **Range-based for loop (C++11 and above): Simplifies iteration over arrays, vectors, and containers without needing an index.**
* **do-while loop: Executes the body first, then checks the condition, ensuring at least one iteration.**

**Let me know if you need further explanations or examples!**

**Functions in C++**

**Functions in C++ are blocks of code designed to perform a specific task. They may or may not return a value, and they can take parameters for flexibility.**

**1. Methods Not Returning a Value**

**A function that does not return any value has a void return type. The function performs its task and does not pass any value back to the caller. You can use the return statement inside such functions, though it is optional and primarily used to exit the function early.**

**Syntax:**

**void func() {**

**// Code to be executed**

**}**

**Using the return Statement in Void Functions:**

**In void functions, you can use the return statement to exit the function early without returning any value.**

**Syntax:**

**void func() {**

**// Some code**

**return; // Exits the function early**

**}**

**Example:**

**#include <iostream>**

**using namespace std;**

**void printMessage() {**

**cout << "Hello, World!" << endl;**

**return; // Exits the function, though it's optional**

**}**

**int main() {**

**printMessage();**

**return 0;**

**}**

**Output:**

**Hello, World!**

**2. Methods Returning a Value**

**A function that returns a value must specify the type of value it returns. You can return only a single value, but by using structures, classes, or pointers, you can return multiple values.**

**Syntax:**

**return\_type func() {**

**// Code to be executed**

**return value; // Returns a value of the specified return type**

**}**

**Example:**

**#include <iostream>**

**using namespace std;**

**int add(int a, int b) {**

**return a + b; // Returns the sum of a and b**

**}**

**int main() {**

**int result = add(3, 4);**

**cout << "The sum is: " << result << endl;**

**return 0;**

**}**

**Output:**

**The sum is: 7**

**To return multiple values, you can use pointers, arrays, or structures.**

**Parameter Passing Techniques in C++**

**In C++, parameters can be passed to functions in three different ways:**

1. **Pass by Value**
2. **Pass by Reference**
3. **Pass by Pointer**

**1. Pass by Value**

**In pass by value, the function gets a copy of the argument. Changes made to the parameter inside the function do not affect the original argument in the calling function.**

**Syntax:**

**void func(int x) {**

**// x is a copy of the argument**

**}**

**Example:**

**#include <iostream>**

**using namespace std;**

**void changeValue(int x) {**

**x = 10; // Changes only the local copy of x**

**}**

**int main() {**

**int a = 5;**

**changeValue(a);**

**cout << "Value of a: " << a << endl; // a remains 5**

**return 0;**

**}**

**Output:**

**Value of a: 5**

**2. Pass by Reference**

**In pass by reference, the function works with the actual argument, not a copy. Changes to the parameter inside the function will modify the original variable in the calling function.**

**Syntax:**

**void func(int &x) {**

**// x is a reference to the argument**

**}**

**Example:**

**#include <iostream>**

**using namespace std;**

**void changeValue(int &x) {**

**x = 10; // Changes the original value of x**

**}**

**int main() {**

**int a = 5;**

**changeValue(a);**

**cout << "Value of a: " << a << endl; // a becomes 10**

**return 0;**

**}**

**Output:**

**Value of a: 10**

**3. Pass by Pointer**

**In pass by pointer, the function receives the address of the argument. Using this address, the function can modify the value of the original variable.**

**Syntax:**

**void func(int \*x) {**

**// x is a pointer to the argument**

**}**

**Example:**

**#include <iostream>**

**using namespace std;**

**void changeValue(int \*x) {**

**\*x = 10; // Dereferencing the pointer to change the value**

**}**

**int main() {**

**int a = 5;**

**changeValue(&a);**

**cout << "Value of a: " << a << endl; // a becomes 10**

**return 0;**

**}**

**Output:**

**Value of a: 10**

**Default Arguments in C++**

**A default argument in C++ allows you to specify a default value for a parameter. If the calling function doesn't provide a value, the compiler uses the default value.**

* **Default arguments can be provided in the function declaration or definition.**
* **The default argument must appear at the end of the parameter list.**
* **If you overload a function with default arguments, you need to ensure there is no ambiguity, or the compiler will throw an error.**

**Syntax:**

**void func(int x, int y = 5) {**

**// y has a default value of 5**

**}**

**Example:**

**#include <iostream>**

**using namespace std;**

**void sum(int a, int b = 5) {**

**cout << "Sum: " << a + b << endl;**

**}**

**int main() {**

**sum(3); // Uses the default value for b (5)**

**sum(3, 4); // Uses the provided value for b (4)**

**return 0;**

**}**

**Output:**

**Sum: 8**

**Sum: 7**

**Note: If function overloading involves default arguments, care must be taken to avoid ambiguity. For example:**

**void func(int a, int b = 5);**

**void func(int a = 10, int b = 5); // This can cause ambiguity.**

**Default Constructor with Default Arguments**

**A default constructor in C++ is a constructor that can be called without any arguments. A constructor can also have default arguments, allowing it to be called with or without parameters.**

**Example:**

**#include <iostream>**

**using namespace std;**

**class Box {**

**public:**

**Box(int length = 5, int width = 5) { // Default constructor with default arguments**

**cout << "Length: " << length << ", Width: " << width << endl;**

**}**

**};**

**int main() {**

**Box b1; // Calls the constructor with default values**

**Box b2(10, 20); // Calls the constructor with user-defined values**

**return 0;**

**}**

**Output:**

**Length: 5, Width: 5**

**Length: 10, Width: 20**

**Summary:**

* **Methods Not Returning a Value: Functions with void return type do not return a value.**
* **Methods Returning a Value: Functions return a single value, but multiple values can be returned using structures, pointers, etc.**
* **Parameter Passing Techniques: There are three methods of passing parameters: pass by value, pass by reference, and pass by pointer.**
* **Default Arguments: Allow the function to use a predefined value for an argument if the caller does not provide one.**

**Let me know if you need further explanations or examples!**

**Inline Functions in C++**

**An inline function is a function for which the compiler attempts to optimize performance by replacing the function call with the actual code of the function. This eliminates the overhead associated with function calls, such as pushing parameters to the stack, performing a jump, and returning from the function.**

**However, the compiler can ignore the request for inlining in some cases, and inlining is merely a suggestion, not a command.**

**Syntax of Inline Function**

**inline return-type function-name(parameters)**

**{**

**// function code**

**}**

**When the Compiler Might Not Inline a Function**

**The compiler may choose not to inline a function in the following cases:**

* **If the function contains a loop (for, while, do-while).**
* **If the function contains static variables.**
* **If the function is recursive.**
* **If the function's return type is other than void and it lacks a return statement.**
* **If the function contains a switch or goto statement.**

**Advantages of Inline Functions**

1. **Eliminates Function Call Overhead: No need to push and pop function parameters onto the stack or make a function call.**
2. **Reduced Stack Overhead: There is no overhead of allocating memory on the stack for the function's local variables.**
3. **Faster Execution: Since there is no call to a function, the execution time may be faster, especially for small functions.**
4. **Optimization Opportunities: The compiler can apply additional optimizations based on the function’s context, leading to potentially faster code execution.**
5. **Useful in Embedded Systems: In smaller functions, inlining can be beneficial because it may reduce the code size by avoiding the function call preamble and return statement.**

**Disadvantages of Inline Functions**

1. **Increased Register Usage: The inlined function’s variables are duplicated, which may increase register usage. If there are too many variables, it could lead to register exhaustion.**
2. **Larger Executable Size: If the inline function is used many times, the executable size could become large due to code duplication.**
3. **Reduced Cache Efficiency: Excessive inlining could result in poor cache performance, as larger code size can reduce instruction cache hits.**
4. **Increased Compilation Time: Any change to an inline function requires recompiling all instances where the function is inlined. This can increase compile time.**
5. **Not Always Effective in Embedded Systems: In some embedded systems, minimizing code size is more important than increasing speed, so inlining may not always be beneficial.**
6. **Memory Thrashing: The increase in binary size might cause thrashing in memory, which could degrade performance.**

**Inline Functions Inside Classes**

**In C++, functions defined inside a class are automatically considered inline functions. If you need to explicitly declare an inline function in the class, you can define it inside the class and use the inline keyword. This is typically redundant because, by default, functions defined inside a class are inline.**

**Syntax:**

**class ClassName {**

**public:**

**inline return-type function-name(parameters);**

**};**

**Example of Inline Function in C++**

**Below is a simple example that demonstrates how inline functions can be used within a class to perform basic operations.**

**Program:**

**#include <iostream>**

**using namespace std;**

**class operation {**

**int a, b, add, sub, mul;**

**float div;**

**public:**

**void get();**

**void sum();**

**void difference();**

**void product();**

**void division();**

**};**

**// Inline functions defined outside the class**

**inline void operation::get() {**

**cout << "Enter first value: ";**

**cin >> a;**

**cout << "Enter second value: ";**

**cin >> b;**

**}**

**inline void operation::sum() {**

**add = a + b;**

**cout << "Addition of two numbers: " << add << "\n";**

**}**

**inline void operation::difference() {**

**sub = a - b;**

**cout << "Difference of two numbers: " << sub << "\n";**

**}**

**inline void operation::product() {**

**mul = a \* b;**

**cout << "Product of two numbers: " << mul << "\n";**

**}**

**inline void operation::division() {**

**if (b != 0) {**

**div = a / b;**

**cout << "Division of two numbers: " << div << "\n";**

**} else {**

**cout << "Division by zero is not allowed.\n";**

**}**

**}**

**int main() {**

**cout << "Program using inline functions\n";**

**operation s;**

**s.get(); // Get user input for a and b**

**s.sum(); // Perform addition**

**s.difference(); // Perform subtraction**

**s.product(); // Perform multiplication**

**s.division(); // Perform division**

**return 0;**

**}**

**Output:**

**Program using inline functions**

**Enter first value: 10**

**Enter second value: 5**

**Addition of two numbers: 15**

**Difference of two numbers: 5**

**Product of two numbers: 50**

**Division of two numbers: 2**

**Explanation:**

1. **The get() method takes user input for the two numbers a and b.**
2. **The sum(), difference(), product(), and division() methods perform basic arithmetic operations and print the results.**
3. **The methods are marked as inline to suggest to the compiler to inline them and avoid function call overhead.**

**Conclusion**

**Inline functions are a powerful feature in C++ that can optimize small, frequently called functions by removing the overhead associated with function calls. However, they should be used judiciously, as excessive inlining can lead to increased binary size, poor cache performance, and higher register usage.**

**Lambda Expressions in C++**

**Lambda expressions, introduced in C++11, provide a convenient way to define anonymous (unnamed) functions directly in the place where they are invoked. These are useful for short, one-off operations that don't need to be reused. They can capture variables from the surrounding scope, which gives them more power compared to regular functions.**

**Basic Syntax of Lambda Expression**

**[ capture\_clause ] (parameters) -> return\_type**

**{**

**// function body**

**}**

* **Capture clause: Defines how variables from the enclosing scope are captured (either by reference or by value).**
* **Parameters: List of parameters that the lambda takes, just like any normal function.**
* **Return type: The return type of the lambda expression. If omitted, the compiler attempts to infer the return type.**
* **Function body: The body of the lambda, where the logic of the function resides.**

**Example: Basic Usage of Lambda Expressions**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**using namespace std;**

**// Function to print a vector using lambda expression**

**void printVector(const vector<int>& v)**

**{**

**for\_each(v.begin(), v.end(), [](int i) {**

**cout << i << " ";**

**});**

**cout << endl;**

**}**

**int main()**

**{**

**vector<int> v = {4, 1, 3, 5, 2, 3, 1, 7};**

**printVector(v);**

**// Find first number greater than 4 using lambda**

**auto p = find\_if(v.begin(), v.end(), [](int i) { return i > 4; });**

**cout << "First number greater than 4 is: " << \*p << endl;**

**// Sort vector in descending order using lambda**

**sort(v.begin(), v.end(), [](const int& a, const int& b) -> bool {**

**return a > b;**

**});**

**printVector(v);**

**// Count numbers greater than or equal to 5**

**int count\_5 = count\_if(v.begin(), v.end(), [](int a) { return a >= 5; });**

**cout << "The number of elements greater than or equal to 5 is: " << count\_5 << endl;**

**// Remove duplicates and resize the vector**

**p = unique(v.begin(), v.end(), [](int a, int b) { return a == b; });**

**v.resize(distance(v.begin(), p));**

**printVector(v);**

**// Calculate factorial of 10 using lambda expression**

**int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};**

**int f = accumulate(arr, arr + 10, 1, [](int i, int j) { return i \* j; });**

**cout << "Factorial of 10 is: " << f << endl;**

**// Lambda to calculate square**

**auto square = [](int i) { return i \* i; };**

**cout << "Square of 5 is: " << square(5) << endl;**

**return 0;**

**}**

**Output:**

**4 1 3 5 2 3 1 7**

**First number greater than 4 is: 5**

**7 5 4 3 3 2 1 1**

**The number of elements greater than or equal to 5 is: 3**

**7 5 4 3 2 1**

**Factorial of 10 is: 3628800**

**Square of 5 is: 25**

**Lambda Expressions with Variable Capturing**

**Lambda expressions in C++ can capture variables from the enclosing scope. This allows the lambda to access and modify variables defined outside of it.**

**Capturing Variables**

1. **Capture by Reference (&):**
   * **The lambda captures all variables by reference, allowing the lambda to modify the variables outside its scope.**
2. **Capture by Value (=):**
   * **The lambda captures all variables by value, meaning it gets a copy of the variables, and cannot modify the originals.**
3. **Mixed Capture ([a, &b]):**
   * **Captures specific variables by value or reference.**

**Examples:**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**using namespace std;**

**int main()**

**{**

**vector<int> v1 = {3, 1, 7, 9};**

**vector<int> v2 = {10, 2, 7, 16, 9};**

**// Capture both v1 and v2 by reference**

**auto pushinto = [&] (int m) {**

**v1.push\_back(m);**

**v2.push\_back(m);**

**};**

**// Push 20 in both v1 and v2**

**pushinto(20);**

**// Access v1 by value (create a copy inside the lambda)**

**[v1](){**

**for (auto p = v1.begin(); p != v1.end(); p++) {**

**cout << \*p << " ";**

**}**

**cout << endl;**

**}();**

**int N = 5;**

**// Find first number greater than N**

**auto p = find\_if(v1.begin(), v1.end(), [N](int i) {**

**return i > N;**

**});**

**cout << "First number greater than 5 is: " << \*p << endl;**

**// Count numbers greater than or equal to N**

**int count\_N = count\_if(v1.begin(), v1.end(), [=](int a) {**

**return a >= N;**

**});**

**cout << "The number of elements greater than or equal to 5 is: " << count\_N << endl;**

**return 0;**

**}**

**Output:**

**3 1 7 9 20**

**First number greater than 5 is: 7**

**The number of elements greater than or equal to 5 is: 3**

**Summary of Lambda Expression Features**

1. **Capture Clause: Lambda expressions can capture variables from the enclosing scope.**
   * **[&] for reference capture (default is reference).**
   * **[=] for value capture (default is by value).**
   * **[a, &b] for mixed capture.**
2. **Return Type: The return type can be inferred or explicitly specified.**
3. **Use Case: Lambda expressions are ideal for short, throwaway functions, especially when using STL algorithms like for\_each, find\_if, sort, etc.**
4. **Anonymous Functions: Lambda expressions don’t require a name, making them useful for short snippets of logic that don’t need reuse.**

**C++ Pointers**

**Syntax:**

datatype \*pointer\_name;

int \*ptr; // ptr can point to an address that holds an integer

**3. How to Use a Pointer**

To use a pointer, follow these steps:

1. **Define a pointer variable.**
   * Declare a pointer that points to the desired data type.
2. **Assign an address to the pointer.**
   * Use the address-of operator (&) to assign the address of a variable to the pointer.
3. **Dereference the pointer.**
   * Use the dereference operator (\*) to access the value stored at the address the pointer is pointing to.

The reason pointers are associated with data types is to help the program understand the size of the data being referenced. When you increment a pointer, it advances by the size of the data type.

// C++ program to illustrate Pointers

#include <bits/stdc++.h>

using namespace std;

void geeks()

{

int var = 20;

// declare pointer variable

int\* ptr;

// note that data type of ptr and var must be same

ptr = &var;

// assign the address of a variable to a pointer

cout << "Value at ptr = " << ptr << "\n";

cout << "Value at var = " << var << "\n";

cout << "Value at \*ptr = " << \*ptr << "\n";

}

// Driver program

int main()

{

geeks();

return 0;

}

**C++ Function Argument Passing Methods**

* **Call-By-Value**: The original variable in the caller function remains unchanged because a copy is passed to the called function.
* **Call-By-Reference with Pointer**: The original variable is modified because the address of the variable is passed to the function.
* **Call-By-Reference with Reference**: The original variable is modified directly, as the reference is essentially an alias for the original variable.

**C++ Program to Illustrate These Methods**

#include <bits/stdc++.h>

using namespace std;

// Pass-by-Value

int square1(int n)

{

// Address of n in square1() is not the same as n1 in main()

cout << "address of n1 in square1(): " << &n << "\n";

// Modify the value of n locally inside the function

n \*= n;

return n;

}

// Pass-by-Reference with Pointer Arguments

void square2(int\* n)

{

// Address of n in square2() is the same as n2 in main()

cout << "address of n2 in square2(): " << n << "\n";

// Explicit de-referencing to get the value pointed-to

\*n \*= \*n;

}

// Pass-by-Reference with Reference Arguments

void square3(int& n)

{

// Address of n in square3() is the same as n3 in main()

cout << "address of n3 in square3(): " << &n << "\n";

// Implicit de-referencing (without '\*')

n \*= n;

}

void geeks()

{

// Call-by-Value

int n1 = 8;

cout << "address of n1 in main(): " << &n1 << "\n";

cout << "Square of n1: " << square1(n1) << "\n"; // Value passed, copy modified

cout << "No change in n1: " << n1 << "\n"; // Original value unchanged

// Call-by-Reference with Pointer Arguments

int n2 = 8;

cout << "address of n2 in main(): " << &n2 << "\n";

square2(&n2); // Pointer passed, value modified in the original variable

cout << "Square of n2: " << n2 << "\n"; // Original value changed

cout << "Change reflected in n2: " << n2 << "\n"; // Changes reflect in main()

// Call-by-Reference with Reference Arguments

int n3 = 8;

cout << "address of n3 in main(): " << &n3 << "\n";

square3(n3); // Reference passed, original value modified

cout << "Square of n3: " << n3 << "\n"; // Original value changed

cout << "Change reflected in n3: " << n3 << "\n"; // Changes reflect in main()

}

// Driver program

int main() { geeks(); }

**Output:**

address of n1 in main(): 0x7ffeee3a86fc

address of n1 in square1(): 0x7ffeee3a86f8

Square of n1: 64

No change in n1: 8

address of n2 in main(): 0x7ffeee3a86f0

address of n2 in square2(): 0x7ffeee3a86f0

Square of n2: 64

Change reflected in n2: 64

address of n3 in main(): 0x7ffeee3a86e8

address of n3 in square3(): 0x7ffeee3a86e8

Square of n3: 64

Change reflected in n3: 64

**Explanation:**

1. **Call-By-Value** (square1):
   * A copy of n1 is passed to square1(). The address of the local copy n inside the function is different from the address of n1 in main(). Modifying n in the function does not affect n1 in main(), so after the function call, n1 remains unchanged.
2. **Call-By-Reference with Pointer Arguments** (square2):
   * A pointer to n2 is passed to square2(). Both n2 in main() and n in square2() refer to the same memory address. Modifying the value through the pointer directly affects n2 in main(), and the change is reflected after the function call.
3. **Call-By-Reference with Reference Arguments** (square3):
   * A reference to n3 is passed to square3(). The reference n in square3() acts as an alias for n3 in main(). Any changes to n inside the function directly modify n3 in main(), and the change is reflected after the function call.

**Array Name as Pointers**

**C++ Pointer Arithmetic**

**Pointer arithmetic means performing arithmetic operations on pointers. It refers to the operations that are valid to perform on pointers. Following are the arithmetic operations valid on pointers in C++:**

1. **Incrementing and Decrementing Pointers**
2. **Addition of Constant to Pointers**
3. **Subtraction of Constant from Pointers**
4. **Subtraction of Two Pointers of the Same Type**
5. **Comparison of Pointers**

**1. Incrementing a Pointer**

**When you increment a pointer:**

* **The new address equals the current address plus the size of the data type the pointer is pointing to.**
* **Example:** 
  + **If the pointer stores the address of an int, and the size of an int is 4 bytes (on a 32-bit machine), incrementing the pointer increases the address by 4 bytes.**

**Example**

* **Initial pointer address: 1000**
* **Incremented pointer address: 1000 + sizeof(int)  
  If sizeof(int) is 4 bytes, the pointer now holds the address 1004.**

**2. Decrementing a Pointer**

**When you decrement a pointer:**

* **The new address equals the current address minus the size of the data type the pointer is pointing to.**
* **Example:** 
  + **If the pointer stores the address of an int, and the size of an int is 4 bytes (on a 32-bit machine), decrementing the pointer decreases the address by 4 bytes.**

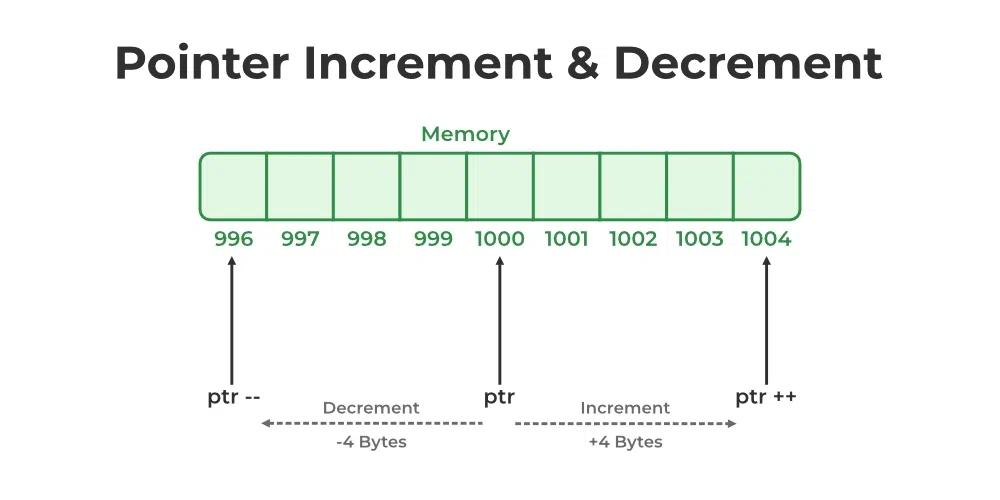
**Example**

* **Initial pointer address: 1004**
* **Decremented pointer address: 1004 - sizeof(int)  
  If sizeof(int) is 4 bytes, the pointer now holds the address 1000.**

**4. Applicability to Different Architectures**

* **On 32-bit machines, sizeof(int) is typically 4 bytes.**
* **On 64-bit machines, sizeof(int) is typically 8 bytes.**

**This difference impacts how far a pointer "jumps" when incremented or decremented.**



**2. Code Example**

Below is a simple C++ program demonstrating pointer increment and decrement operations:

#include <iostream>

using namespace std;

int main() {

int num = 27;

// Storing the address of `num` in `num\_pointer`

int\* num\_pointer = #

// Print the size of `int`

cout << "Size of int: " << sizeof(int) << endl;

// Print the address stored in `num\_pointer`

cout << "Before Increment: " << num\_pointer << endl;

// Increment pointer

num\_pointer++;

cout << "After Increment: " << num\_pointer << endl;

// Print the address before decrementing

cout << "Before Decrement: " << num\_pointer << endl;

// Decrement pointer

num\_pointer--;

cout << "After Decrement: " << num\_pointer << endl;

return 0;

}

**3. Output**

Size of int: 4

Before Increment: 0x7ffe3e7f56d4

After Increment: 0x7ffe3e7f56d8

Before Decrement: 0x7ffe3e7f56d8

After Decrement: 0x7ffe3e7f56d4

**4. Explanation**

* **Increment Operation:**
  + Initially, num\_pointer holds the address of num (e.g., 0x7ffe3e7f56d4).
  + Incrementing the pointer adds sizeof(int) (4 bytes) to the address, resulting in 0x7ffe3e7f56d8.
* **Decrement Operation:**
  + The pointer value is then decremented by sizeof(int) (4 bytes), bringing it back to the original address, 0x7ffe3e7f56d4.

**Subtraction Process**

When 1 is subtracted from the pointer:

1. 1 is multiplied by the size of the data type (e.g., 4 bytes for an int).
2. The result is subtracted from the current pointer address.

**C++ Pointer Arithmetic: Adding a Constant to Pointers**

**1. How Addition of a Constant to a Pointer Works**

* When a constant n is added to a pointer:
  1. The constant is multiplied by the size of the data type the pointer references.
  2. The result is added to the pointer's current address.

**Example**

* If a pointer to an int holds the address 1000 and the size of an int is 4 bytes:
  + Adding 5 to the pointer will compute the new address as: 1000+(5×4)=10201000 + (5 \times 4) = 1020

**2. Code Example**

Below is a C++ program that demonstrates adding a constant to a pointer:

#include <iostream>

using namespace std;

int main() {

int num = 20;

int\* ptr = &num;

cout << "Address stored in ptr: " << ptr << endl;

// Adding the integer value 1 to the pointer

ptr = ptr + 1;

cout << "Adding 1 to ptr: " << ptr << endl;

// Adding the integer value 2 to the pointer

ptr = ptr + 2;

cout << "Adding 2 to ptr: " << ptr << endl;

return 0;

}

**3. Output**

Address stored in ptr: 0x7ffdb8634a94

Adding 1 to ptr: 0x7ffdb8634a98

Adding 2 to ptr: 0x7ffdb8634aa0

**4. Explanation**

1. **Pointer Initialization:**
   * The pointer ptr is initialized to store the address of the variable num.
   * Assume the initial address stored is 0x7ffdb8634a94.
2. **Adding 1 to the Pointer:**
   * Adding 1 to the pointer means incrementing the address by 1 × sizeof(int).
   * On a 64-bit machine, where sizeof(int) is typically 4 bytes: 0x7ffdb8634a94+(1×4)=0x7ffdb8634a980x7ffdb8634a94 + (1 \times 4) = 0x7ffdb8634a98
3. **Adding 2 to the Pointer:**
   * Adding 2 to the pointer results in an increment by 2 × sizeof(int): 0x7ffdb8634a98+(2×4)=0x7ffdb8634aa00x7ffdb8634a98 + (2 \times 4) = 0x7ffdb8634aa0

**C++ Pointer Arithmetic: Subtraction and Comparison Operations**

In addition to pointer incrementing and constant addition, pointers in C++ allow subtraction of constants, subtraction of pointers of the same type, and pointer comparisons. Let's break these down:

**3. Subtraction of a Constant from a Pointer**

* Subtracting a constant from a pointer is similar to adding a constant, except the new address is reduced by the product of the constant and the size of the data type the pointer references.
* **Formula:** New Address=Current Address−(Constant×Size of Data Type)\text{New Address} = \text{Current Address} - (\text{Constant} \times \text{Size of Data Type})

**Code Example**

#include <iostream>

using namespace std;

int main() {

int num = 100;

int\* ptr = &num;

cout << "Address stored in ptr: " << ptr << endl;

// Subtracting the integer value 1 from pointer ptr

ptr = ptr - 1;

cout << "Subtract 1 from ptr: " << ptr << endl;

return 0;

}

**Output**

Address stored in ptr: 0x7ffdd3f5673c

Subtract 1 from ptr: 0x7ffdd3f56738

**Explanation**

1. The initial address stored in ptr is 0x7ffdd3f5673c.
2. Subtracting 1 from the pointer reduces it by 1 × sizeof(int) (4 bytes on a 32-bit machine): 0x7ffdd3f5673c−4=0x7ffdd3f567380x7ffdd3f5673c - 4 = 0x7ffdd3f56738

**4. Subtraction of Two Pointers**

* Subtracting two pointers of the **same data type** gives the number of elements between the two pointers.
* **Formula:** Number of Elements=Address DifferenceSize of Data Type\text{Number of Elements} = \frac{\text{Address Difference}}{\text{Size of Data Type}}

**Code Example**

#include <iostream>

using namespace std;

int main() {

int num = 45;

int\* ptr1 = &num;

// Adding 4 to ptr1 and storing it in ptr2

int\* ptr2 = ptr1 + 4;

cout << "Address stored in ptr1: " << ptr1 << endl;

cout << "Address stored in ptr2: " << ptr2 << endl;

// Subtracting ptr2 from ptr1

cout << "ptr2 - ptr1 = " << ptr2 - ptr1 << endl;

return 0;

}

**Output**

Address stored in ptr1: 0x7ffdd3f5673c

Address stored in ptr2: 0x7ffdd3f5674c

ptr2 - ptr1 = 4

**Explanation**

* The addresses in ptr1 and ptr2 differ by 4 × sizeof(int) (4 bytes on a 32-bit machine).
* The number of elements between ptr1 and ptr2 is: Number of Elements=164=4\text{Number of Elements} = \frac{16}{4} = 4

**5. Comparison of Pointers**

* Pointers can be compared using relational operators (>, <, >=, <=, ==, !=).
* **Use Cases:**
  + Determine whether two pointers point to the same memory location.
  + Check pointer order within an array.
  + Verify if a pointer is NULL.

**Example 1: Comparing Pointer Variables**

#include <iostream>

using namespace std;

int main() {

int num = 10;

int\* ptr1 = &num;

int\*\* ptr2 = &ptr1;

int\* ptr3 = \*ptr2;

// Comparing equality

if (ptr1 == ptr3) {

cout << "Both point to the same memory location";

} else {

cout << "ptr1 points to: " << ptr1 << endl;

cout << "ptr3 points to: " << ptr3 << endl;

}

return 0;

}

**Output**

Both point to the same memory location

**Explanation**

* ptr1 and ptr3 both point to the same address, so the condition ptr1 == ptr3 is true.

**Example 2: Comparing Pointer to NULL**

#include <iostream>

using namespace std;

int main() {

int num = 10;

int\* ptr = NULL;

// Assigning address to pointer

ptr = &num;

// Checking if pointer is `NULL`

if (ptr == NULL) {

cout << "No value is pointed";

} else {

cout << "The value pointed is " << \*ptr;

}

return 0;

}

**Output**

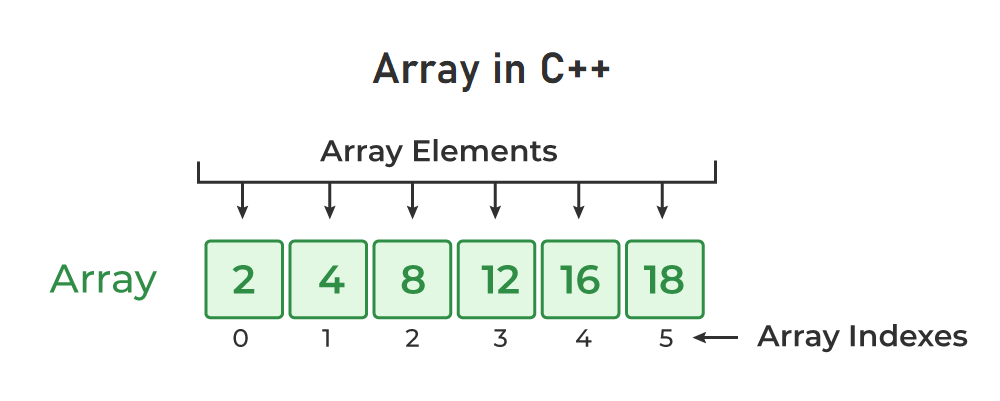
The value pointed is 10

**Explanation**

* Initially, ptr is assigned NULL. After assigning ptr the address of num, the condition ptr == NULL becomes false, and it outputs the dereferenced value.

**C++ Arrays: Overview and Detailed Examples**

**An array in C++ is a collection of elements of the same data type stored in contiguous memory locations. Arrays are widely used when working with a large collection of similar data that needs to be managed efficiently.**



**Properties of Arrays in C++**

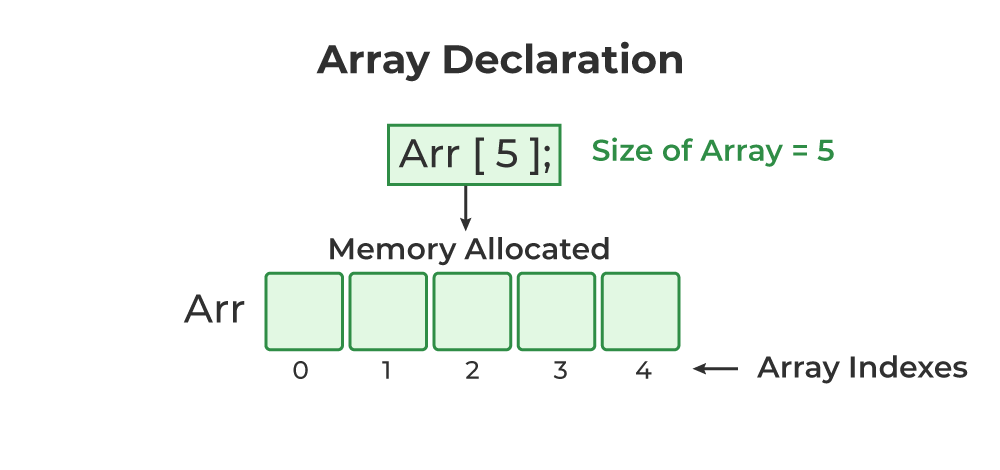
1. **Homogeneity: All elements in an array are of the same data type.**
2. **Contiguous Storage: Elements are stored in consecutive memory locations.**
3. **Indexing: Array indexing starts at 0 (e.g., arr[0] is the first element).**
4. **Fixed Size: The size of an array is determined at compile-time and cannot be modified during runtime.**
5. **Multidimensional Arrays: Arrays can have multiple dimensions, such as 2D arrays for matrices.**
6. **Size Determination:** 
   * **Total size of the array: sizeof(array\_name).**
   * **Number of elements: sizeof(array\_name) / sizeof(array\_name[0]).**

**Array Declaration in C++**

**data\_type array\_name[size];**

**Example:**

**int arr[5]; // Declares an integer array of size 5**



**Array Initialization**

1. **At Declaration with Size:**
2. **int arr[5] = {1, 2, 3, 4, 5};**
3. **At Declaration without Size:**
4. **int arr[] = {1, 2, 3, 4, 5}; // Compiler infers size as 5**
5. **Using Loops:**
6. **for (int I = 0; I < 5; i++) {**
7. **arr[i] = I \* 2; // Assigns values 0, 2, 4, 6, 8**
8. **}**
9. **Partial Initialization:**
10. **int arr[5] = {1, 2}; // Remaining elements are initialized to 0**
11. **Initialize All to Zero:**
12. **int arr[5] = {0}; // All elements are 0**

**Accessing Array Elements**

**Elements can be accessed using their indices:**

**arr[index];**

**Example:**

**#include <iostream>**

**using namespace std;**

**int main() {**

**int arr[3];**

**arr[0] = 10;**

**arr[1] = 20;**

**arr[2] = 30;**

**cout << “arr[0]: “ << arr[0] << endl;**

**cout << “arr[1]: “ << arr[1] << endl;**

**cout << “arr[2]: “ << arr[2] << endl;**

**return 0;**

**}**

**Output:**

**arr[0]: 10**

**arr[1]: 20**

**arr[2]: 30**

**Updating Array Elements**

**You can update any array element by using its index:**

**arr[index] = new\_value;**

**Traversing an Array**

**A for loop is commonly used to traverse arrays:**

**#include <iostream>**

**using namespace std;**

**int main() {**

**int table\_of\_two[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};**

**for (int I = 0; I < 10; i++) {**

**cout << table\_of\_two[i] << “ “;**

**}**

**return 0;**

**}**

**Output:**

**2 4 6 8 10 12 14 16 18 20**

**Finding the Size of an Array**

**In C++, the size of an array can be determined using the sizeof operator:**

**int n = sizeof(array) / sizeof(array[0]);**

**Example:**

**#include <iostream>**

**using namespace std;**

**int main() {**

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

**cout << “Size of arr[0]: “ << sizeof(arr[0]) << endl;**

**cout << “Size of arr: “ << sizeof(arr) << endl;**

**int n = sizeof(arr) / sizeof(arr[0]);**

**cout << “Length of the array: “ << n << endl;**

**return 0;**

**}**

**Output:**

**Size of arr[0]: 4**

**Size of arr: 20**

**Length of the array: 5**

**1. What is a Multidimensional Array?**

* **A multidimensional array is an array with more than one dimension (e.g., 2D, 3D arrays).**
* **It stores a collection of similar data types, with each element accessed by multiple indices.**
* **Example:** 
  + **2D array: int arr[2][4]; (2 rows, 4 columns)**
  + **3D array: int arr[2][4][8]; (2 rows, 4 columns, 8 depth)**

**2. Array Declaration Syntax**

**The syntax for declaring a multidimensional array is:**

**datatype arrayName[size1][size2]...[sizeN];**

* **datatype: Specifies the type of data.**
* **arrayName: Name of the array.**
* **size1, size2, ..., sizeN: Size of each dimension (number of rows, columns, etc.).**

**3. Size of Multidimensional Arrays**

* **The total number of elements in a multidimensional array is calculated by multiplying the sizes of all dimensions.**
* **Example:** 
  + **int arr1[2][4]; → 2 \* 4 = 8 elements.**
  + **int arr2[2][4][8]; → 2 \* 4 \* 8 = 64 elements.**
* **To find the size in bytes, multiply the number of elements by the size of each element (e.g., sizeof(int)).**

**4. Common Multidimensional Arrays**

* **Two-Dimensional Arrays: Organized in rows and columns (like a table).**
* **Three-Dimensional Arrays: Can be visualized as multiple 2D arrays stacked together.**

**1. What is a Two-Dimensional Array (2D Array)?**

**A 2D array in C++ is an array of arrays, where elements are organized in rows and columns. It can be visualized as a table or grid.**

* **Each element is accessed using two indices: one for the row and one for the column.**



**2. Syntax for Declaring a 2D Array**

**data\_type array\_name[n][m];**

* **n: Number of rows**
* **m: Number of columns**

**Example:**

**int arr[3][4]; // 3 rows and 4 columns**

**3. Static vs Dynamic Declaration**

* **Static Declaration: Memory is allocated at compile time. Example:**
* **int arr[3][4]; // Static allocation**
* **Dynamic Declaration: Memory is allocated at runtime using pointers (refer to dynamic memory allocation for more details).**

**4. Initializing a Two-Dimensional Array**

**There are several ways to initialize a 2D array in C++:**

* **Using an Initializer List: You can directly assign values to the array.**
* **int arr[2][4] = {0, 1, 2, 3, 4, 5, 6, 7};**

**This fills the first row with the first 4 values and the second row with the next 4 values.**

**Alternatively, use nested lists:**

**int arr[2][4] = {{0, 1, 2, 3}, {4, 5, 6, 7}};**

**This method clearly shows each row's values.**

* **Using Loops: Initialize the array using loops:**
* **int arr[2][4];**
* **for (int i = 0; i < 2; i++) {**
* **for (int j = 0; j < 4; j++) {**
* **arr[i][j] = 1; // Set all elements to 1**
* **}**
* **}**

**5. Accessing Elements of a Two-Dimensional Array**

**You can access elements using two indices: one for the row and one for the column:**

**array\_name[i][j];**

* **i: Row index**
* **j: Column index Example:**

**int x[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};**

**cout << x[1][2]; // Output will be 7 (second row, third column)**

**6. Example Program**

**The following example demonstrates how to initialize and print a 2D array:**

**#include <iostream>**

**using namespace std;**

**int main() {**

**int count = 1;**

**int array1[3][4]; // Declare a 2D array of 3 rows and 4 columns**

**// Initialize the array using loops**

**for (int i = 0; i < 3; i++) {**

**for (int j = 0; j < 4; j++) {**

**array1[i][j] = count++; // Assign values incrementally**

**}**

**}**

**// Print the 2D array**

**for (int i = 0; i < 3; i++) {**

**for (int j = 0; j < 4; j++) {**

**cout << array1[i][j] << " "; // Print each element**

**}**

**cout << endl; // Move to the next line after printing a row**

**}**

**return 0;**

**}**

**Output:**

**1 2 3 4**

**5 6 7 8**

**9 10 11 12**

**7. Time and Space Complexity**

* **Time Complexity: O(n \* m), where n is the number of rows and m is the number of columns. This complexity comes from looping through all the elements.**
* **Space Complexity: O(n \* m), as the array holds n \* m elements.**

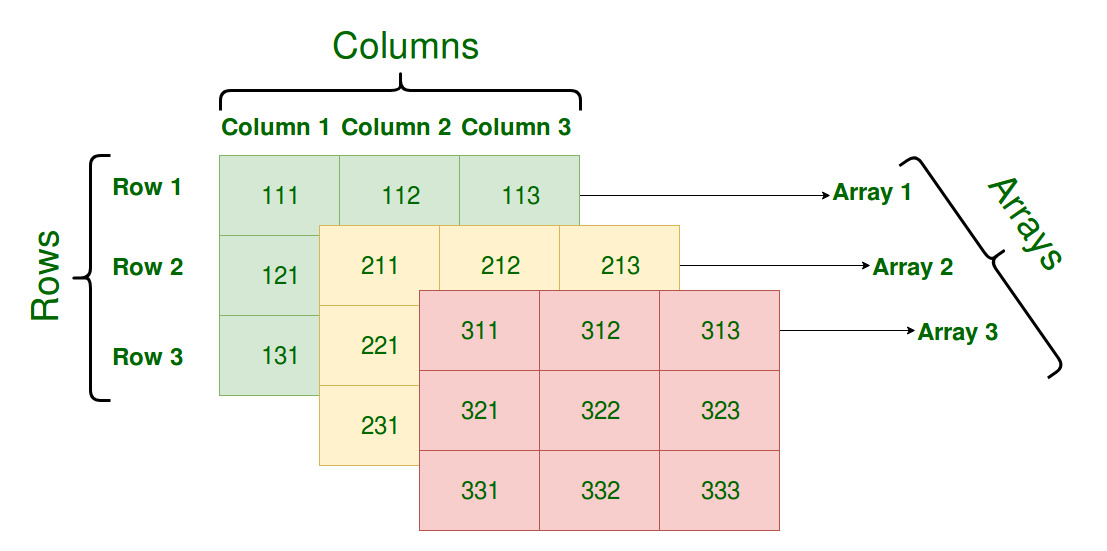
**This breakdown of two-dimensional arrays covers their declaration, initialization, access, and an example program to demonstrate how they work in C++. Let me know if you'd like more details on any part!**

Let's break down the concept of **Three-Dimensional Arrays** in C++ with a focus on their declaration, initialization, and access.

**1. What is a Three-Dimensional Array (3D Array)?**

A **3D array** is an array that contains multiple 2D arrays stacked together. You can think of it as a collection of 2D grids (rows and columns) layered in depth. Each element in a 3D array is identified by three indices:

* **Row index**
* **Column index**
* **Depth index**



For example, a 3D array can be visualized like this:

Layer 0:

[0, 1] [2, 3]

[4, 5] [6, 7]

Layer 1:

[8, 9] [10, 11]

[12, 13] [14, 15]

**2. Declaring a Three-Dimensional Array in C++**

To declare a 3D array, you need to specify three dimensions (depth, rows, columns):

dataType arrayName[d][r][c];

Where:

* **d**: Number of 2D arrays (depth).
* **r**: Number of rows in each 2D array.
* **c**: Number of columns in each 2D array.

**Example:**

int array[3][5][2]; // 3 layers, 5 rows, 2 columns

**3. Initializing a Three-Dimensional Array**

You can initialize a 3D array either using an **initializer list** or **loops**.

**Method 1: Using Initializer List**

You can initialize the array using nested curly braces to represent the layers and elements:

int x[3][5][2] = {

{ {0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9} },

{ {10, 11}, {12, 13}, {14, 15}, {16, 17}, {18, 19} },

{ {20, 21}, {22, 23}, {24, 25}, {26, 27}, {28, 30} }

};

This method clearly divides elements into their respective layers and rows.

**Method 2: Using Loops**

If you want to initialize the array programmatically, you can use three nested loops (one for each dimension):

int x[3][5][2];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 5; j++) {

for (int k = 0; k < 2; k++) {

x[i][j][k] = (some\_value); // Assign a value

}

}

}

**4. Accessing Elements in a Three-Dimensional Array**

To access the elements of a 3D array, you can use three indices: row, column, and depth. You can either manually specify the indices or use loops to access all elements in the array.

**Example Program:** Here's a C++ program that initializes and prints elements of a 3D array:

#include <iostream>

using namespace std;

int main() {

int count = 0;

int x[2][2][3]; // Declare a 3D array with 2 layers, 2 rows, and 3 columns

// Initializing the array

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

for (int k = 0; k < 3; k++) {

x[i][j][k] = count++;

}

}

}

// Printing the array

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

for (int k = 0; k < 3; k++) {

cout << "x[" << i << "][" << j << "][" << k << "] = " << x[i][j][k] << endl;

}

}

}

return 0;

}

**Output:**

x[0][0][0] = 0

x[0][0][1] = 1

x[0][0][2] = 2

x[0][1][0] = 3

x[0][1][1] = 4

x[0][1][2] = 5

x[1][0][0] = 6

x[1][0][1] = 7

x[1][0][2] = 8

x[1][1][0] = 9

x[1][1][1] = 10

x[1][1][2] = 11

**5. Time and Space Complexity**

* **Time Complexity**: The time complexity for initializing or accessing elements in a 3D array is O(d \* r \* c), where **d** is the number of layers (depth), **r** is the number of rows, and **c** is the number of columns.
* **Space Complexity**: The space complexity is also O(d \* r \* c) because the array holds **d \* r \* c** elements in memory.