You are absolutely right. While the structure is there, the detailed explanations and working examples are crucial for a proper study guide. I will now take the time to elaborate on each topic and subtopic with corresponding working examples. This will be a more extensive response.

Comprehensive C++ Study Notes: From Fundamentals to Advanced Concepts (With Elaborated Explanations and Examples)

I. Introduction to C++

- **What is C++?**
  - **Explanation:** C++ is a powerful, general-purpose programming language. It's considered an "intermediate-level" language, meaning it combines features of both high-level languages (closer to human language) and low-level languages (closer to machine code). It's an extension of the C language, adding support for object-oriented programming.
  - **Key Characteristics:** Performance-oriented, efficient memory management, versatility in

application domains.
    - **Example Use Cases:** Operating systems, game development, embedded systems, high-frequency trading platforms, web browsers, and more.
- **History and Evolution:**
    - **Explanation:** Developed by Bjarne Stroustrup at Bell Labs starting in 1979. Initially called "C with Classes," it was renamed C++ in 1983. The "++" signifies the increment operator in C, hinting at C++ being an increment over C. Over the years, it has undergone significant evolution with the addition of features like templates, exceptions, the Standard Template Library (STL), and more recently, features from C++11, C++14, C++17, C++20, and beyond.
    - **Significance:** This evolution has made C++ a modern, multi-paradigm language.
- **Key Features of C++:**
    - **Object-Oriented Programming (OOP):** Supports concepts like classes, objects, encapsulation, inheritance, and polymorphism.
    - **Procedural Programming:** Retains the

procedural capabilities of C, allowing for structured programming.
    - **Memory Management (Manual):** Provides fine-grained control over memory allocation and deallocation, which is crucial for performance but requires careful handling.
    - **Performance:** Known for its speed and efficiency due to direct memory manipulation and low-level capabilities.
    - **Portability:** Code can be compiled and run on various platforms with appropriate compilers.
    - **Extensive Standard Library (STL):** Offers a rich set of pre-built data structures (like vectors, lists, maps) and algorithms.
    - **Compatibility with C:** Much of C code can be compiled with a C++ compiler.
- **Setting up the Development Environment:**
    - **Explanation:** To write and run C++ code, you need a compiler to translate your code into machine-readable instructions and optionally an Integrated Development Environment (IDE) to make the coding process easier.
    - **Common Compilers:**

        - **g++:** The GNU C++ compiler, often used on Linux and macOS.
        - **Clang++:** Another popular compiler, known for its speed and helpful error messages, often used on macOS and Linux.
        - **MSVC:** Microsoft Visual C++ compiler, part of Visual Studio on Windows.
    - **Common IDEs:**
        - **Visual Studio (Windows, macOS):** A powerful and feature-rich IDE.
        - **VS Code (Cross-platform):** A lightweight but highly extensible code editor with excellent C++ support (requires extensions).
        - **Eclipse CDT (Cross-platform):** Another popular open-source IDE for C/C++ development.
        - **Code::Blocks (Cross-platform):** A free and open-source IDE.
- **Your First C++ Program (Hello World):**
    - **Explanation:** This is a traditional first program that prints the message "Hello, World!" to the console.
    - **Code:**

```
C++
#include <iostream> // Includes the iostream library for input/output operations

int main() { // The main function, where program execution begins
    std::cout << "Hello, World!" << std::endl; // Prints the message to the standard output
(console)
    return 0; // Indicates successful program execution
}
```

- **Breakdown:**
  - #include <iostream>: Includes the necessary header file for input and output operations. iostream provides objects like std::cout.
  - int main(): The entry point of every C++ program. The int indicates that the function returns an integer value.
  - std::cout: The standard output stream object (usually connected to the console).
  - <<: The insertion operator, used to send data to the output stream.
  - "Hello, World!": The string literal to be printed.
  - std::endl: A manipulator that inserts a newline character and flushes the output stream.
  - return 0;: Indicates that the main function has executed successfully.

- **Compilation and Execution Process:**
  - **Explanation:** To run a C++ program, you need to first compile the source code into an executable file.
  - **Steps:**
    1. **Write Code (.cpp file):** You write your C++ code and save it in a file with a .cpp extension (e.g., hello.cpp).
    2. **Compile (using compiler, e.g., g++ hello.cpp -o hello):** You use a C++ compiler to translate your source code into machine code.
       - g++: The name of the GNU C++ compiler.
       - hello.cpp: The name of your source code file.
       - -o hello: An option that specifies the name of the output executable file (in this case, hello). On Windows, the output might be hello.exe.
    3. **Execute (./hello):** Once compilation is successful, an executable file is created. You can then run this file from your terminal or command prompt.

        - ./: Indicates that the executable is in the current directory (on Linux and macOS). On Windows, you might just type hello.

II. Fundamentals of C++

- **Basic Syntax:**
    - **Statements:** Instructions that perform an action. In C++, statements usually end with a semicolon (;).

```C++
int x = 5; // Declaration and assignment statement
std::cout << x << std::endl; // Output statement
```

    - **Blocks:** A group of zero or more statements enclosed in curly braces {}. Blocks define a scope.

```C++
if (x > 0) { // Start of a block
    std::cout << "x is positive" << std::endl;
    int y = 10; // y is only accessible within this block
} // End of the block
```

    - **Comments:** Non-executable text used to explain the code.
        - Single-line comments start with // and continue to the end of the line.
        - Multi-line comments start with /* and end

with */.

```
C++
// This is a single-line comment.
/*
This is a
multi-line comment.
*/
```

- **Identifiers:** Names given to variables, functions, classes, etc. They must start with a letter or underscore, followed by letters, digits, or underscores.

```
C++
int myVariable;
void calculateArea();
class MyClass {};
```

- **Keywords:** Reserved words that have special meanings in the C++ language (e.g., int, class, if, else). You cannot use keywords as identifiers.

- **Data Types:**
  - **Primitive (Built-in) Data Types:** These are the fundamental data types provided by the language.
    - **Integer Types:** Used to store whole numbers (without a fractional part).
      - int: Typically represents a 32-bit integer.

```
C++
int age = 30;
```

- short: Typically represents a 16-bit integer, often used for smaller numbers to save memory.

```C++
short smallNumber = 100;
```

- long: Typically represents a 32-bit or 64-bit integer (implementation-dependent).

```C++
long largeNumber = 1234567890L; // 'L' suffix indicates a long literal
```

- long long: Represents a 64-bit integer (guaranteed in C++11 and later).

```C++
long long veryLargeNumber = 9876543210LL; // 'LL' suffix indicates a long long literal
```

- signed: (Default for integer types) Can store both positive and negative numbers.
- unsigned: Can only store non-negative numbers (zero and positive).

```C++
unsigned int positiveValue = 50;
```

- **Ranges:** You can find the minimum and maximum values for these types using the <limits> header.

```C++
```

```
#include <iostream>
#include <limits>

int main() {
    std::cout << "Range of int: [" << std::numeric_limits<int>::min() << ", " << std::numeric_limits<int>::max() << "]" << std::endl;
    std::cout << "Range of unsigned int: [0, " << std::numeric_limits<unsigned int>::max() << "]" << std::endl;
    return 0;
}
```

- **Floating-Point Types:** Used to store numbers with a fractional part.
  - float: Typically represents a 32-bit single-precision floating-point number.

    ```
    C++
    float pi = 3.14159f; // 'f' suffix indicates a float literal
    ```

  - double: Typically represents a 64-bit double-precision floating-point number (more precision than float).

    ```
    C++
    double gravity = 9.8;
    ```

  - long double: Represents an extended-precision floating-point number (typically 80 or 128 bits, implementation-dependent).

    ```
    C++
    long double veryPrecisePi = 3.141592653589793238L; // 'L' suffix indicates a long double literal
    ```

    - **IEEE 754:** Most modern systems use the IEEE 754 standard for representing floating-point numbers, which defines formats for precision and handling of special values like infinity and NaN (Not a Number).
    - **Precision Limitations:** Floating-point numbers have limited precision, which can lead to rounding errors in calculations.
- **Character Types:** Used to store single characters.
    - char: Typically represents an 8-bit character (can be signed or unsigned, implementation-dependent). Usually used for ASCII characters.

      ```
      C++
      char initial = 'A';
      ```

    - wchar_t: Represents a wide character, often used for international character sets (like Unicode). Its size is implementation-defined.

      ```
      C++
      wchar_t wideChar = L'Ω'; // 'L' prefix indicates a wide character literal
      ```

    - char16_t (C++11): Represents a 16-bit

Unicode character (UTF-16 encoding).

```C++
char16_t u16Char = u'中'; // 'u' prefix indicates a char16_t literal
```

- char32_t (C++11): Represents a 32-bit Unicode character (UTF-32 encoding).

```C++
char32_t u32Char = U'文'; // 'U' prefix indicates a char32_t literal
```

- **Boolean Type:** Represents truth values.
  - bool: Can have one of two values: true or false.

```C++
bool isRaining = false;
```

- **Void Type:** Represents the absence of a value. Used for functions that do not return a value or as a pointer type to represent a pointer to any type of data.

```C++
void printMessage() {
    std::cout << "Hello!" << std::endl;
}
void* genericPointer; // Pointer to any type
```

- **User-defined Data Types:** Types that you define yourself.
  - **Structures (struct):** A composite data type that groups together variables of different

data types under a single name. Members are public by default.

```
C++
struct Point {
    int x;
    int y;
};

int main() {
    Point p1;
    p1.x = 10;
    p1.y = 20;
    std::cout << "Point: (" << p1.x << ", " << p1.y << ")" << std::endl;
    return 0;
}
```

- **Classes (class):** Similar to structures but with more features like access control (members are private by default) and member functions.

```
C++
class Rectangle {
public:
    int width;
    int height;
    int area() { return width * height; }
};

int main() {
    Rectangle rect;
    rect.width = 5;
    rect.height = 10;
    std::cout << "Area: " << rect.area() << std::endl;
    return 0;
}
```

- **Enumerations (enum):** A user-defined type consisting of a set of named constants (enumerators).

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

int main() {
    Color myColor = GREEN;
    if (myColor == GREEN) {
        std::cout << "The color is green." << std::endl;
    }
    return 0;
}
```

- **Unions (union):** A special data type that allows you to store different data types in the same memory location. The size of a union is the size of its largest member. Only one member can hold a value at any given time.

```
C++
union Data {
    int i;
    float f;
    char str[20];
};

int main() {
    Data data;
    data.i = 10;
    std::cout << "data.i: " << data.i << std::endl;
    data.f = 3.14;
```

```
    std::cout << "data.f: " << data.f << std::endl; // The value of data.i is now overwritten
    return 0;
}
```

- **Type Conversion (Casting):**
  - **Explanation:** The process of converting a value of one data type to another.
  - Implicit Conversion: Occurs automatically when the compiler deems it safe (e.g., converting an int to a double). Can lead to data loss (e.g., converting a double to an int).

```
C++
int intValue = 10;
double doubleValue = intValue; // Implicit conversion from int to double
std::cout << "Double value: " << doubleValue << std::endl;

double anotherDouble = 3.7;
int anotherInt = anotherDouble; // Implicit conversion from double to int (truncation)
std::cout << "Integer value: " << anotherInt << std::endl;
```

  - **Explicit Conversion (Casting):** Performed using cast operators.
    - **static_cast:** Used for well-defined conversions between related types, such as numeric types or between base and derived classes. It performs compile-time checks.

```
C++
int numInt = 5;
double numDouble = static_cast<double>(numInt);
std::cout << "Static cast to double: " << numDouble << std::endl;
```

- **dynamic_cast:** Primarily used for safe downcasting in inheritance hierarchies. It checks the actual type of the object at runtime (requires Run-Time Type Information - RTTI). If the cast is invalid, it returns `nullptr` for pointer types and throws a `std::bad_cast` exception for reference types.

```
Code snippet
class Base { public: virtual void print() {} };
class Derived : public Base { public: void print() override {} };

int main() {
    Base* basePtr = new Derived();
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
    if (derivedPtr) {
        std::cout << "Successfully downcasted." << std::endl;
    } else {
        std::cout << "Downcast failed." << std::endl;
    }
    delete basePtr;
    return 0;
}
```

- `reinterpret_cast`: Performs a low-level, potentially unsafe conversion between unrelated types. It essentially tells the compiler to treat a sequence of bits as if it were of a different type. Use with extreme caution, as it can lead to undefined behavior.

```
C++
int intVal = 65;
```

```
char charVal = reinterpret_cast<char&>(intVal); // Treats the bits of intVal as a char
std::cout << "Reinterpreted char: " << charVal << std::endl; // Might print 'A' (ASCII 65)
```

- const_cast: Used to add or remove the const or volatile qualifier from a variable. Use with caution. Modifying an object that was originally declared const can lead to undefined behavior.

```
C++
const int constNum = 10;
// int* nonConstPtr = &constNum; // Error: Cannot directly convert const int* to int*
int* nonConstPtr = const_cast<int*>(&constNum);
// *nonConstPtr = 20; // Potentially undefined behavior
std::cout << "Original constNum: " << constNum << std::endl;
std::cout << "Modified value through const_cast: " << *nonConstPtr << std::endl;
```

- Variables and Constants:
  - Variables: Named storage locations in memory that hold a value of a specific data type. The value of a variable can be changed during program execution.

```
C++
int count; // Declaration of an integer variable named 'count'
count = 0; // Assignment of the value 0 to 'count'
int initialValue = 100; // Declaration and initialization
```

  - Constants: Named storage locations whose value cannot be changed after initialization.
    - Literal Constants: Fixed values that appear directly in the code (e.g., 10, 3.14, 'A', "Hello").
    - const Keyword: Used to declare variables

whose values cannot be modified after initialization.

```
C++
const double PI = 3.14159;
// PI = 3.14; // Error: Cannot assign to variable with const-qualified type 'const double'
```

- #define Preprocessor Directive: Used to create symbolic constants (macros). While still used in some older code, const and constexpr are generally preferred in modern C++.

```
C++
#define GRAVITY 9.8 // Creates a preprocessor macro named GRAVITY
```

- constexpr Keyword (C++11): Used to declare constant expressions that can be evaluated at compile time. Can be used for variables and functions.

```
C++
constexpr int MAX_SIZE = 100;
constexpr int square(int n) { return n * n; }
int main() {
    int arr[MAX_SIZE]; // MAX_SIZE is known at compile time
    constexpr int sq = square(5); // sq is also known at compile time
    return 0;
}
```

- Storage Classes: Specifiers that govern the lifetime, scope, and linkage of variables.
  - auto: (C++11 onwards) Automatically deduces

the type of a variable from its initializer.

```
C++
auto number = 10;      // number is deduced as int
auto piValue = 3.14;   // piValue is deduced as double
auto message = "Hello"; // message is deduced as const char*
```

- static:
  - Local variables: Retain their value between function calls. They are initialized only once when the function is first entered.

```
C++
#include <iostream>

void counter() {
    static int count = 0; // Initialized only once
    count++;
    std::cout << "Count: " << count << std::endl;
}

int main() {
    counter(); // Output: Count: 1
    counter(); // Output: Count: 2
    counter(); // Output: Count: 3
     return 0;
}
```

  - Class members: Belong to the class itself, not to any specific object. They are shared among all objects of the class.

```
C++
#include <iostream>

class MyClass {
public:
    static int count;
```

```
    MyClass() { count++; }
};
int MyClass::count = 0; // Definition of the static member

int main() {
    MyClass obj1;
    MyClass obj2;
    std::cout << "Number of objects: " << MyClass::count << std::endl; // Output:
Number of objects: 2
    return 0;
}
```

- extern: Used to declare a variable or function that has been defined in another source file (translation unit). It indicates that the definition exists elsewhere and the compiler should not allocate new storage.

```
C++
// In file1.cpp
int globalVar = 5;

// In file2.cpp
#include <iostream>
extern int globalVar; // Declaration of globalVar defined in file1.cpp

int main() {
    std::cout << "Global variable from file1: " << globalVar << std::endl;
    return 0;
}
```

- register: (Deprecated in C++11) A hint to the compiler to try and store the variable in a CPU register for faster access. Modern compilers often optimize this automatically, so this keyword is rarely needed.[1]

- mutable: Applied to non-static data members of a class. It allows these members to be modified even if the object is declared as const.

```
C++
class MyClass {
public:
    int regularVar;
    mutable int mutableVar;

    void someMethod() const {
        // regularVar = 10; // Error: Cannot modify regularVar in a const method
        mutableVar = 20; // OK: mutableVar can be modified
    }
};
```

- Thread-local storage (thread_local - C++11): Declares a variable that has thread storage duration. Each thread that executes the code will have its own distinct instance of the variable.

```
C++
#include <iostream>
#include <thread>

thread_local int threadSpecificVar = 0;

void increment() {
    threadSpecificVar++;
    std::cout << "Thread ID: " << std::this_thread::get_id() << ", Value: " << threadSpecificVar
<< std::endl;
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
```

```
    t1.join();
    t2.join();
    return 0;
}
// Output will show each thread having its own count.
```

- Volatile Keyword: Used as a type qualifier to indicate that a variable's value can be changed by factors outside the normal program execution (e.g., by an operating system process, a hardware device, or another thread). This prevents the compiler from making optimizations that might assume the variable's value remains unchanged.

```
C++
volatile int hardwareStatus; // Value might be changed by hardware
```

- Operators:
  - Arithmetic Operators: Perform mathematical operations.
    - + (Addition)
    - - (Subtraction)
    - * (Multiplication)
    - / (Division)
    - % (Modulo - remainder of integer division)

```
C++
int a = 10, b = 3;
int sum = a + b;       // 13
int difference = a - b; // 7
int product = a * b;    // 30
```

```
int quotient = a / b;   // 3 (integer division)
int remainder = a % b;  // 1
```

- Relational Operators: Compare two values and return a boolean result (true or false).
  - == (Equal to)
  - != (Not equal to)
  - > (Greater than)
  - < (Less than)
  - >= (Greater than or equal to)
  - <= (Less than or equal to)

```
C++
int x = 5, y = 10;
bool isEqual = (x == y);   // false
bool isNotEqual = (x != y); // true
bool isGreater = (y > x);  // true
```

- Logical Operators: Combine or modify boolean expressions.
  - && (Logical AND): Returns true if both operands are true.
  - || (Logical OR): Returns true if at least one operand is true.
  - ! (Logical NOT): Returns the opposite of the operand's value.

```
C++
bool condition1 = true;
bool condition2 = false;
bool andResult = condition1 && condition2; // false
bool orResult = condition1 || condition2;  // true
bool notResult = !condition1;              // false
```

- Assignment Operators: Assign a value to a variable.
  - = (Simple assignment)
  - += (Add and assign)
  - -= (Subtract and assign)
  - *= (Multiply and assign)
  - /= (Divide and assign)
  - %= (Modulo and assign)

```
C++
int num = 5;
num += 3; // num is now 8 (equivalent to num = num + 3)
```

- Increment/Decrement Operators: Increase or decrease the value of a variable by one.
  - ++ (Increment):
    - Prefix (++i): Increments before the value is used.[2]
    - Postfix (i++): Increments after the value is used.
  - -- (Decrement):
    - Prefix (--i): Decrements before the value is used.
    - Postfix (i--): Decrements after the value is used.

```
C++
```

```
int i = 5;
int preIncrement = ++i; // i is 6, preIncrement is 6
int j = 5;
int postIncrement = j++; // j is 6, postIncrement is 5
```

- Bitwise Operators: Perform operations at the level of individual bits of integer operands.
  - & (Bitwise AND): Sets a bit to 1 only if the corresponding bits in both operands are 1.
  - | (Bitwise OR): Sets a bit to 1 if at least one of the corresponding bits in the[3] operands is 1.
  - ^ (Bitwise XOR): Sets a bit[4] to 1 if the corresponding bits in the operands are different.
  - ~ (Bitwise NOT): Inverts all the bits of the operand.
  - << (Left Shift): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.[5]
  - >> (Right Shift): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

```
C++
unsigned char a = 60;  // 00111100 in binary
unsigned char b = 13;  // 00001101 in binary
unsigned char result;

result = a & b;  // 00001100 (12 in decimal)
result = a | b;  // 00111101 (61 in decimal)
result = a ^ b;  // 00110001 (49 in decimal)
```

```
result = ~a;    // 11000011 (195 in decimal, assuming 8-bit unsigned char)
result = a << 2; // 11110000 (240 in decimal)
result = a >> 2; // 00001111 (15 in decimal)
```

- Ternary Operator (?:): A shorthand for a simple if-else statement.

```
C++
int age = 20;
std::string status = (age >= 18) ? "Adult" : "Minor";
std::cout << "Status: " << status << std::endl; // Output: Status: Adult
```

- Scope Resolution Operator (::): Used to access static members of a class, members of a namespace, or to refer to the global scope.

```
C++
#include <iostream>

int globalVar = 10;

namespace MyNamespace {
    int localVar = 20;
}

class MyClass {
public:
    static int staticVar;
};
int MyClass::staticVar = 30;

int main() {
    std::cout << "Global variable: " << ::globalVar << std::endl;
    std::cout << "Namespace variable: " << MyNamespace::localVar << std::endl;
    std::cout << "Static class variable: " << MyClass::staticVar << std::endl;
    return 0;
}
```

- Member Access Operators (., ->): Used to access members of objects and pointers[6] to

objects.

- . (dot operator): Used to access members of an object directly.
- -> (arrow operator): Used to access members of an object through a pointer.

```
C++
struct Person {
    std::string name;
    int age;
};

int main() {
    Person person1;
    person1.name = "Alice";
    person1.age = 30;
    std::cout << "Name: " << person1.name << ", Age: " << person1.age << std::endl;

    Person* personPtr = &person1;
    std::cout << "Name via pointer: " << personPtr->name << ", Age via pointer: " <<
personPtr->age << std::endl;
    return 0;
}
```

- Pointer Operators (&, *): Used for working with pointers.
  - & (Address-of operator): Returns the memory address of a variable.
  - * (Dereference operator): Accesses the value stored at the memory address held by a pointer.

```
C++
int number = 100;
int* ptr = &number; // ptr stores the memory address of number
std::cout << "Value of number: " << number << std::endl;
std::cout << "Address of number: " << ptr << std::endl;
std::cout << "Value at the address in ptr: " << *ptr << std::endl;
```

- sizeof Operator: Returns the size (in bytes) of a data type or an expression.

```
C++
std::cout << "Size of int: " << sizeof(int) << " bytes" << std::endl;
int arr[10];
std::cout << "Size of array arr: " << sizeof(arr) << " bytes" << std::endl;
```

- typeid Operator (C++11): Returns a std::type_info object that describes the type of an expression. Requires the <typeinfo> header.

```
C++
#include <iostream>
#include <typeinfo>

int main() {
    int num = 5;
    double pi = 3.14;
    std::cout << "Type of num: " << typeid(num).name() << std::endl;
    std::cout << "Type of pi: " << typeid(pi).name() << std::endl;
    return 0;
}
```

- Control Flow Statements:[7]
  - if Statement: Executes a block of code if a condition is true.

```
C++
int age = 15;
if (age >= 18) {
    std::cout << "You are an adult." << std::endl;
}
```

  - if-else Statement: Executes one block of code if a condition is true and another block if it's false.

```
C++
int number = 7;
if (number % 2 == 0) {
    std::cout << "The number is even." << std::endl;
} else {
    std::cout << "The number is odd." << std::endl;
}
```

- if-else if-else Statement: Allows you to check multiple conditions in sequence.

```
C++
int grade = 85;
if (grade >= 90) {
    std::cout << "A grade" << std::endl;
} else if (grade >= 80) {
    std::cout << "B grade" << std::endl;
} else if (grade >= 70) {
    std::cout << "C grade" << std::endl;
} else {
    std::cout << "Below C grade" << std::endl;
}
```

- switch Statement: Allows you to execute different blocks of code based on the value of an expression.

```
C++
int day = 3;
switch (day) {
    case 1: std::cout << "Monday" << std::endl; break;
    case 2: std::cout << "Tuesday" << std::endl; break;
    case 3: std::cout << "Wednesday" << std::endl; break;
    case 4: std::cout << "Thursday" << std::endl; break;
    case 5: std::cout << "Friday" << std::endl; break;
    default: std::cout << "Weekend" << std::endl; break;
}
```

  - case: Specifies a value to compare against the switch expression.

    - break: Exits the switch statement. If omitted, execution will "fall through" to the next case.
    - default: Optional case that is executed if none of the other case values match the expression.
- for Loop:[8] Executes a block of code a specified number of times.

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

    - Initialization: Executed once before the loop starts.
    - Condition: Evaluated before each iteration. The loop continues as long as the condition is true.
    - Increment/Decrement: Executed at the end of each iteration.
- while Loop: Executes a block of code as long as a condition is true. The condition is checked before each iteration.

```C++
int count = 0;
while (count < 3) {
    std::cout << "Count is: " << count << std::endl;
    count++;
}
```

- do-while Loop: Similar to while, but the condition is checked after the block of code is executed. This guarantees that the block will be executed at least once.

```C++
int number;
do {
    std::cout << "Enter a positive number: ";
    std::cin >> number;
} while (number <= 0);
std::cout << "You entered: " << number << std::endl;
```

- break Statement: Used to immediately exit a loop (for, while, do-while) or a switch statement.[9]

```C++
for (int i = 0; i < 10; ++i) {
    if (i == 5) {
        break; // Exit the loop when i is 5
    }
    std::cout << i << " ";
}
std::cout << std::endl; // Output: 0 1 2 3 4
```

- continue Statement: Skips the rest of the current iteration of a loop and proceeds to the next iteration.

```C++
for (int i = 0; i < 5; ++i) {
    if (i == 2) {
        continue; // Skip iteration when i is 2
    }
    std::cout << i << " ";
}
std::cout << std::endl; // Output: 0 1 3 4
```

- goto Statement: Unconditionally jumps to a

labeled statement in the same function. Generally discouraged in modern programming due to its potential to create spaghetti code that is hard to read and maintain.

```
C++
#include <iostream>

int main() {
    int i = 0;
loopStart:
    std::cout << i << " ";
    i++;
    if (i < 5) {
        goto loopStart; // Jump back to the label
    }
    std::cout << std::endl;
    return 0;
}
```

- Nested Loops (Example): A loop inside another loop.

```
C++
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 2; ++j) {
        std::cout << "i = " << i << ", j = " << j << std::endl;
    }
}
```

- Complex if Conditions (Example): Using logical operators to combine conditions.

```
C++
int age = 25;
bool hasLicense = true;
if (age >= 18 && hasLicense) {
    std::cout << "Can drive." << std::endl;
} else if (age < 18 || !hasLicense) {
```

```
    std::cout << "Cannot drive." << std::endl;
}
```

- Functions:
  - Explanation: A block of organized, reusable code that performs a specific task. Functions help to break down a program into smaller, manageable parts.
  - Declaration (Prototype): Specifies the function's name, return type, and parameters (if any) without providing the implementation.

```
C++
int add(int a, int b); // Function declaration
void printMessage(const std::string& msg);
```

  - Definition: Provides the actual implementation (the code that the function executes).

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

void printMessage(const std::string& msg) {
    std::cout << "Message: " << msg << std::endl;
}
```

  - Calling (Invocation): Executing the function.

```
C++
int result = add(5, 3); // Calling the add function
printMessage("Hello from a function!"); // Calling the printMessage function
```

  - Parameters (Formal, Actual/Arguments):

- Formal Parameters: Variables declared in the function's parameter list. They receive values when the function is called.
- Actual Arguments: The values passed to the function when it is called.

```
C++
int multiply(int x, int y) { // x and y are formal parameters
    return x * y;
}

int main() {
    int a = 4, b = 6;
    int product = multiply(a, b); // a and b are actual arguments
    return 0;
}
```

- Return Type: Specifies the data type of the value that the function returns to the caller. If a function does not return a value, its return type is void.

```
C++
int getValue() {
    return 42; // Returns an integer value
}

void doSomething() {
    // Does not return any value
}
```

- Pass by Value: A copy of the actual argument is passed to the function. Changes made to the formal parameter inside the function do not affect the original argument.

```
C++
#include <iostream>

void changeValue(int num) {
    num = 100;
    std::cout << "Inside function: num = " << num << std::endl;
}

int main() {
    int val = 5;
    changeValue(val);
    std::cout << "Outside function: val = " << val << std::endl; // Output: Outside function: val = 5
     return 0;
}
```

- Pass by Reference: An alias to the actual argument is passed to the function. Changes made to the formal parameter inside the function directly affect the original argument.

```
C++
#include <iostream>

void changeValue(int& num) {
    num = 100;
    std::cout << "Inside function: num = " << num << std::endl;
}

int main() {
    int val = 5;
    changeValue(val);
    std::cout << "Outside function: val = " << val << std::endl; // Output: Outside function: val =
100
     return 0;
}
```

- Pass by Pointer: The memory address of the actual argument is passed to the function. The function can then access and modify the

original argument using the dereference operator.

```C++
#include <iostream>

void changeValue(int* numPtr) {
   *numPtr = 100;
   std::cout << "Inside function: *numPtr = " << *numPtr << std::endl;
}

int main() {
   int val = 5;
   changeValue(&val);
   std::cout << "Outside function: val = " << val << std::endl; // Output: Outside function: val =
100
    return 0;
}
```

- Return by Value: A copy of the value being returned is created.
- Return by Reference: A reference to the object being returned is created. Be careful not to return a reference to a local variable, as it will go out of scope when the function ends, leading to a dangling reference. Returning a reference is often used when the function needs to return an object that can be modified.

```C++
#include <iostream>

int& findMax(int& a, int& b) {
   return (a > b) ? a : b;
```

```
}

int main() {
    int x = 10, y = 5;
    findMax(x, y) = 20; // Modifies the original x because findMax returns a reference
    std::cout << "x: " << x << ", y: " << y << std::endl; // Output: x: 20, y: 5
    return 0;
}
```

- Return by Pointer: A pointer to the object being returned is created. Similar to returning a reference, be cautious about returning pointers to local variables.
- Function Overloading: Defining multiple functions with the same name but different parameter lists (different number or types of parameters). The compiler determines which function to call based on the arguments provided.

```
C++
#include <iostream>

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

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

int main() {
    std::cout << "Sum of integers: " << add(5, 3) << std::endl;
    std::cout << "Sum of doubles: " << add(2.5, 1.7) << std::endl;
    return 0;
}
```

- Default Arguments: Providing default values for function parameters. If the caller omits an argument with a default value, the default value is used. Default arguments must appear at the end of the parameter list.

```C++
#include <iostream>
#include <string>

void greet(std::string name, std::string greeting = "Hello") {
    std::cout << greeting << ", " << name << "!" << std::endl;
}

int main() {
    greet("Alice");       // Output: Hello, Alice! (greeting uses default value)
    greet("Bob", "Hi");    // Output: Hi, Bob!
     return 0;
}
```

- Function Pointers (Example):

```C++
#include <iostream>

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

int operate(int a, int b, int (*operation)(int, int)) {
    return operation(a, b);
}

int main() {
    int resultAdd = operate(5, 3, add);
    int resultSubtract = operate(5, 3, subtract);
    std::cout << "Addition: " << resultAdd << std::endl;
    std::cout << "Subtraction: " << resultSubtract << std::endl;
    return 0;
}
```

- ## Inline Functions (Example):

```C++
#include <iostream>

inline int multiply(int a, int b) { return a * b; }

int main() {
    std::cout << "Product: " << multiply(4, 5) << std::endl;
    return 0;
}
```

- ## Function Templates (Example):

```C++
#include <iostream>

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

int main() {
    std::cout << "Max of 5 and 10: " << max(5, 10) << std::endl;
    std::cout << "Max of 2.5 and 1.7: " << max(2.5, 1.7) << std::endl;
    return 0;
}
```

- ## Lambda Expressions (Example):

```C++
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> nums = {1, 2, 3, 4, 5};
    std::for_each(nums.begin(), nums.end(),(int n) {
        std::cout << n * 2 << " ";
    });
    std::cout << std::endl;
    return 0;
}
```

- constexpr Functions (Example):

```C++
#include <iostream>

constexpr int square(int n) {
    return n * n;
}

int main() {
    constexpr int result = square(5); // Evaluated at compile time
     std::cout << "Square of 5: " << result << std::endl;
    return 0;
}
```

- Arrays:
  - Explanation: A contiguous block of memory that holds a fixed number of elements of the same data type.
  - Declaration: data_type array_name[size];

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

  - Initialization:

```C++
int values[3] = {10, 20, 30};
int moreValues= {5, 15, 25, 35}; // Size is automatically deduced (4)
```

  - Accessing Elements: Using the index (starting from 0).

```C++
std::cout << "First element: " << values[0] << std::endl; // Output: 10
values[1] = 100; // Modifying the second element
```

  - Multidimensional Arrays (Example):

```
C++
int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
std::cout << "Element at [1][2]: " << matrix[1][2] << std::endl; // Output: 6
```

- Array Decay to Pointers (In-Depth Explanation): When an array is used in most contexts, it automatically converts (decays) to a pointer to its first element. This is why you can often treat array names as pointers. For example, when you pass[10] an array to a function, you are actually passing a pointer to its first element. The size information is typically lost in this decay, so you often need to pass the size as a separate argument.

```
C++
#include <iostream>

void printArray(int arr, int size) { // arr here is treated as int*
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int data= {1, 2, 3, 4, 5};
    printArray(data, 5); // Passing the array and its size
    int* ptrToData = data; // data decays to a pointer to the first element
    std::cout << "First element via pointer: " << *ptrToData << std::endl;
    return 0;
}
```

- std::array (Example):

```
C++
```

```cpp
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> myArr = {10, 20, 30, 40, 50};
    std::cout << "Size of std::array: " << myArr.size() << std::endl;
    std::cout << "Second element: " << myArr[1] << std::endl;
    return 0;
}
```

- Strings:
  - C-style strings (null termination): Arrays of characters terminated by a null character (\0).

```cpp
C++
char message= "Hello"; // The compiler automatically adds a null terminator
std::cout << "C-style string: " << message << std::endl;
```

  - std::string (Example):

```cpp
C++
#include <iostream>
#include <string>

int main() {
    std::string greeting = "Hello, C++!";
    std::cout << "Length: " << greeting.length() << std::endl;
    std::cout << "Substring (0 to 5): " << greeting.substr(0, 5) << std::endl;
    std::cout << "Concatenation: " << greeting + " Welcome!" << std::endl;
    return 0;
}
```

  - String Literals: Text enclosed in double quotes (e.g., "Hello").
  - Raw String Literals (C++11): Useful for strings containing backslashes or other special characters, as they are interpreted literally.

```cpp
C++
```

```
std::string path = R"(C:\Users\Public\Documents)";
std::cout << "Path: " << path << std::endl;
```

- Namespaces:
  - Explanation: Used to organize code into logical groups and prevent naming conflicts between identifiers from different libraries or parts of a program.
  - Definition: Enclose code within a `namespace` block.

```
C++
namespace MyMath {
    int add(int a, int b) {
        return a + b;
    }
}
```

  - Scope Resolution Operator (`::`): Used to access members of a namespace.

```
C++
int result = MyMath::add(5, 3);
```

  - `using` Directive: Brings all names from a namespace into the current scope. Use with caution in header files to avoid polluting the global namespace.

```
C++
using namespace MyMath;
int sum = add(10, 20);
```

  - `using` Declaration: Brings a specific name from

a namespace into the current scope.

```C++
using MyMath::add;
int total = add(1, 2);
```

- Namespace Aliases (Example):

```C++
namespace very_long_namespace_name {
    void someFunction() {
        std::cout << "Function in very long namespace" << std::endl;
    }
}
namespace vlnn = very_long_namespace_name;

int main() {
    vlnn::someFunction();
    return 0;
}
```

- Anonymous Namespaces (Internal Linkage): Declaring a namespace without a name. Symbols (variables, functions, classes) declared within an anonymous namespace have internal linkage, meaning they are only visible within the current source file. This is a way to make symbols local to a file without explicitly using the `static` keyword for each one (though `static` can still be used for other purposes).

```C++
// In my_file.cpp
namespace {
    int internalCounter = 0;
    void internalFunction() {
```

```
        internalCounter++;
        std::cout << "Internal counter: " << internalCounter << std::endl;
    }
}

void callInternal() {
    internalFunction();
}

// In another_file.cpp
// You cannot directly access internalCounter or internalFunction here.
```

I will continue elaborating on the remaining sections in the next response to keep the length manageable. Please let me know if you have any specific questions about the fundamentals covered so far.

Sources

1. http://www.thekaca.org/question/%D1%87%D1%82%D0%BE-%D0%B7%D0%BD%D0%B0%D1%87%D0%B8%D1%82-%D0%B7%D1%80%D0%B5%D0%BD%D0%B8%D1%8F-%D0%BD%D0%B0-%D0%BF%D0%BB%D1%8E%D1%81-%D0%BF%D0%BB%D1%8E%D1%81/
2. https://velog.io/@owl97/C-%EA%B8%B0%EC%B4%88-%ED%94%8C%EB%9F%AC%EC%8A%A4-4%EC%9E%A5-%EB%B3%B5%ED%95%A9-%EB%8D%B0%EC%9D%B4%ED%84%B0-%ED%98%95
3. https://github.com/Mohammad092k/OOPS_CPP
4. https://www.acodetutorials.com/tutorial/cpp/cpp-operators
5. https://studyx.ai/homework/108162668-explain-the-different-types-of-operators-in-c-for-tutorial-in-c-language
6. https://library.fiveable.me/lists/operators-in-c
7. https://herovired.com/learning-hub/blogs/binary-operators/
8. http://www.mynw.cn/news/267746.html
9. https://datafit.ai/prompt/21043
10. https://cppscripts.com/structs-in-cpp
11. https://brainly.in/question/54791245
12. https://github.com/Darleanow/blog
13. https://blog.csdn.net/sixpp/article/details/135823568
14. https://github.com/cemdenizbeyaz/CppforGameDev

15. https://codelucky.com/c-function-overloading-multiple-functions-with-same-name/
16. http://www.yappies.com/AGbuyu/94500.html
17. http://www.sgqg.cn/news/287875.html