# C++ Programming For Absolute Beginners & Question Set I

Sahas Talasila

Contents

# A Short Overview of Syntax Differences between C and C++

C and C++ share much of their syntax, but there are notable differences. Below, we examine the main distinctions in the areas of variables, types, declarations, and functions. Where relevant, we show code snippets in both C and C++ to highlight differences.

## Variables and Types

### 1.1 Basic Types in C and C++

- Both C and C++ have fundamental types such as `char`, `int`, `float`, `double`, etc.
- C++ introduces additional built-in types such as `bool` (which is not available in C89 or C90). In C, boolean functionality is typically emulated via `#include <stdbool.h>` (C99) or using `int`.
- C++ allows the use of references (`int&`) which do not exist in C.

### 1.2 Variable Declarations

- In C89, all variable declarations must typically appear at the start of a block (though C99 relaxed this to allow mixed declarations).
- C++ allows declaring variables anywhere in a block, so you can declare them just before using them.
- C++ supports default initialization of certain types through constructors in user-defined classes. C does not have constructors or destructors.

#### 1.2.1 Example: Variable Declaration in C (C99 or later)

```c
#include <stdio.h>
#include <stdbool.h> // to have a bool type in C (C99)

int main(void) {
    bool flag = true; // Using C99's bool
    int i = 0;

    // ...
    return 0;
}
```

#### 1.2.2 Example: Variable Declaration in C++

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

int main() {
    bool flag = true; // Native C++ bool
    int i = 0;        // Declaration anywhere

    // ...
    return 0;
}
```

## Functions

### 2.1 Function Prototypes and Declarations

- In both C and C++, functions must be declared before they are used unless a default implicit declaration is allowed (in older C standards, calling an undeclared function was allowed, but it is considered bad practice).

- In C++, function declarations must specify the parameter types exactly. The older style of function declaration in C (K&R style) is valid C but is not valid C++.

## 2.2 Function Overloading

- C does not support function overloading: function names must be unique.
- C++ supports function overloading, meaning you can have multiple versions of the same function name with different parameter lists.

### 2.2.1 Example of Function Overloading in C++

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

void printValue(int x) {
    cout << "Integer: " << x << endl;
}

void printValue(double x) {
    cout << "Double: " << x << endl;
}

int main() {
    printValue(5);       // Calls printValue(int)
    printValue(5.0);     // Calls printValue(double)
    return 0;
}
```

## 2.3 Default Arguments

- C does not support default arguments in function parameters.
- C++ allows default parameters in function declarations.

### 2.3.1 Example of Default Arguments in C++

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

int add(int a, int b = 5) {
    return a + b;
}

int main() {
    cout << add(10) << endl;     // Uses default b=5 => prints 15
    cout << add(10, 20) << endl; // b=20 => prints 30
    return 0;
}
```

## References vs. Pointers

- C uses pointers for indirect referencing. C++ also uses pointers but introduces the concept of references (`&`), which act like aliases for existing variables.
- References must be initialized upon declaration in C++ and cannot be reseated (pointed to a different object) afterwards.

### 3.0.1 Example of Using References in C++

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

void increment(int &ref) {
    ref++;
}

int main() {
    int x = 10;
    increment(x); // x becomes 11
    cout << "x = " << x << endl;
    return 0;
}
```

## Summary of Main Differences

- **Variable Declarations:**
  - C (pre-C99) requires declarations at the start of a block, C++ does not.
  - C++ supports references and native `bool`.
- **Types:**
  - C++ provides `bool`, references, classes, and other built-in object-oriented features.
  - C typically uses `#include <stdbool.h>` for boolean or uses `int`.
- **Function Overloading:**
  - Only available in C++.
  - C requires unique function names.
- **Default Arguments:**
  - Only available in C++.
  - Not supported in C.

---

## Why C++ is Unique Compared to C

C++ extends C with features that facilitate more complex and flexible programming paradigms. While it retains much of C's syntax and low-level control, it introduces several key concepts that make it unique:

1. **Object-Oriented Programming (OOP):**
   - C++ supports classes, inheritance, and polymorphism, enabling developers to encapsulate data and functions within objects.
   - OOP mechanisms help structure large codebases and promote code reuse, which is not an inherent feature of C.
2. **Encapsulation Through Access Specifiers:**
   - C++ classes allow developers to control access to data and methods with `public`, `protected`, and `private` sections, enabling safer data management.
   - C requires manual conventions (e.g., function naming) to mimic encapsulation.
3. **Function Overloading and Templates:**
   - **Overloading** allows multiple functions with the same name but different signatures.
   - **Templates** introduce generic programming, letting you write code that works with any data type while avoiding redundancy.
   - These features simplify code reuse and readability, whereas C does not natively support such generic constructs or overloading.
4. **Operator Overloading:**
   - C++ allows redefining the meaning of operators (e.g., `+`, `==`) for user-defined types.
   - In C, operators cannot be overloaded, so custom behavior must be emulated through functions.
5. **RAII (Resource Acquisition Is Initialization):**
   - In C++, resources (memory, file handles, etc.) are typically acquired and released by constructing and destroying objects.
   - This leads to safer and more intuitive resource management as objects automatically release resources when they go out of scope.
   - In C, the programmer often manually handles `malloc` / `free`, and errors can lead to memory leaks.
6. **Standard Template Library (STL):**
   - C++ comes with a rich set of template-based containers (e.g., `std::vector`, `std::map`, `std::list`) and algorithms.
   - The STL reduces the need to write custom data structures and algorithms from scratch.
   - C does not provide a comparable standard library for higher-level containers and algorithms.
7. **Exception Handling:**
   - C++ provides structured exceptions with `try`, `throw`, and `catch`, enabling error handling without constant status code checks.
   - C typically relies on return values and global error variables like `errno`.

## C++ Examples Showcasing Core Features

Below are several small C++ programs that demonstrate different aspects of the language, including variable types, operators, loops, and conditional flow. Finally, we provide an example that combines all of these aspects into one program. This should be familiar to you.

### 4.1 Different Variable Types

```cpp
#include <iostream>

int main() {
    int wholeNumber = 42;           // Integer
    double pi = 3.14159;            // Floating-point
    char letter = 'A';              // Character
    bool isCplusplusFun = true;     // Boolean
    float percentage = 99.9f;       // Single-precision float

    std::cout << "Integer: " << wholeNumber << std::endl;
    std::cout << "Double: " << pi << std::endl;
    std::cout << "Char: " << letter << std::endl;
    std::cout << "Bool: " << isCplusplusFun << std::endl;
    std::cout << "Float: " << percentage << '%' << std::endl;

    return 0;
}
```

Listing 1: Example of different variable types

## Operators

```cpp
#include <iostream>

int main() {
    int a = 10;
    int b = 3;

    // Arithmetic Operators
    std::cout << "a + b = " << (a + b) << std::endl;
    std::cout << "a - b = " << (a - b) << std::endl;
    std::cout << "a * b = " << (a * b) << std::endl;
    std::cout << "a / b = " << (a / b) << std::endl; // integer division
    std::cout << "a % b = " << (a % b) << std::endl; // modulus

    // Increment / Decrement
    a++; // Post-increment
    std::cout << "After a++: " << a << std::endl;
    ++b; // Pre-increment
    std::cout << "After ++b: " << b << std::endl;

    // Relational and Logical Operators
    std::cout << "Is a > b? " << (a > b) << std::endl;
    std::cout << "Is a == b? " << (a == b) << std::endl;
    std::cout << "Is (a > b) && (b > 5)? " << ((a > b) && (b > 5)) << std::endl;

    return 0;
}
```

Listing 2: Example showcasing basic operators

## Loops

```cpp
#include <iostream>

int main() {
    // For loop
    std::cout << "For loop example:" << std::endl;
    for (int i = 0; i < 5; i++) {
        std::cout << "i = " << i << std::endl;
    }

    // While loop
    std::cout << "\nWhile loop example:" << std::endl;
    int count = 0;
    while (count < 3) {
        std::cout << "count = " << count << std::endl;
        count++;
    }

    // Do-while loop
    std::cout << "\nDo-while loop example:" << std::endl;
    int num = 0;
    do {
        std::cout << "num = " << num << std::endl;
        num++;
    } while (num < 2);

    return 0;
}
```

Listing 3: Examples of different loop constructs

## Conditional Flow Statements

```cpp
#include <iostream>

int main() {
    int x = 10;

    if (x > 0) {     // if-else statement
        std::cout << "x is positive" << std::endl;
    } else if (x < 0) {
        std::cout << "x is negative" << std::endl;
    } else {
        std::cout << "x is zero" << std::endl;
    }

    char grade = 'B';     // switch statement
    switch (grade) {
        case 'A':
            std::cout << "Excellent!" << std::endl;
            break;
        case 'B':
            std::cout << "Well done" << std::endl;
            break;
        case 'C':
            std::cout << "You passed" << std::endl;
            break;
        default:
            std::cout << "Invalid grade" << std::endl;
    }

    return 0;
```

```
}
```

Listing 4: Examples of if/else and switch statements

## Combined Example

In this final example, we combine variables of different types, operators, loops, and conditionals in a simple interactive program.

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

int main() {
    // Variable declarations
    string name;
    int age = 0;
    double sum = 0.0;

    std::cout << "Enter your name: ";
    std::cin >> name;

    std::cout << "Enter your age: ";
    std::cin >> age;

    // Conditional statement
    if (age < 18) {
        std::cout << "Sorry, " << name << ", you are under 18.\n";
    } else {
        std::cout << "Welcome, " << name << "! You are old enough.\n";
    }

    // Loop to add numbers
    std::cout << "\nLet's add 5 numbers:\n";
    for (int i = 1; i <= 5; i++) {
        double value;
        std::cout << "Enter number " << i << ": ";
        std::cin >> value;
        sum += value; // using the + operator
    }

    // Output results using operators
    std::cout << "\nThe total sum of your 5 numbers is: " << sum << std::endl;
    std::cout << "Average of the 5 numbers is: " << (sum / 5) << std::endl;

    // Another conditional section
    if (sum > 50) {
        std::cout << "That's quite a large sum!" << std::endl;
    } else {
        std::cout << "That's a modest sum." << std::endl;
    }

    return 0;
}
```

Listing 5: Combined example

## Comparative Example in C and C++

Below are two short programs that perform similar tasks (printing a boolean variable and calling a function that prints a value). The C program uses `stdbool.h` for boolean support and only a single function, while the C++ program uses native `bool` and demonstrates function overloading.

### C Version

```c
#include <stdio.h>
#include <stdbool.h> // Required for bool in C (C99 and later)

void printValueInt(int val) {
    printf("Integer: %d\n", val);
}

int main(void) {
    bool flag = true;  // C99 bool
    printf("Flag is: %d\n", flag); // prints 1 or 0

    // Since C does not support function overloading,
    // we define separate functions for each data type if needed.
    printValueInt(42);

    return 0;
}
```

Listing 6: C example using stdbool.h and printf

### C++ Version

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

// Overloaded functions: same name, different parameter types
void printValue(int val) {
    cout << "Integer: " << val << endl;
}

void printValue(double val) {
    cout << "Double: " << val << endl;
}

int main() {
    bool flag = true;       // Native C++ bool
    cout << "Flag is: " << flag << endl; // prints 1 or 0 by default

    // In C++, we can overload functions, so the same name works for different types.
    printValue(42);
    printValue(3.14);

    return 0;
}
```

Listing 7: C++ example using native bool, function overloading, and iostream

## Key Differences Highlighted

- **Boolean Support:**
  - In C, `#include <stdbool.h>` is needed to use `bool`, `true`, and `false`.
  - In C++, `bool` is a built-in type and requires no extra header.
- **Input/Output:**
  - The C example uses `printf` from `stdio.h`.
  - The C++ example uses `cout` from `iomanip` (actually provided by `iostream`) for output.
- **Function Overloading:**
  - C does not allow functions to have the same name with different parameter lists (no overloading).
  - C++ allows multiple functions with the same name, as long as the parameter signatures differ.

# Assessment: Variables, Data Types, and Functions

## Part 1 A: Multiple Choice Questions (10 questions)

1. Which of the following data types is used to store a single character in C/C++?
   (a) `int`
   (b) `char`
   (c) `bool`
   (d) `double`
2. In C++ (and in C99 with `stdbool.h`), which keyword represents a boolean type?
   (a) `bit`
   (b) `bool`
   (c) `logical`
   (d) `boolean`
3. Which of the following statements regarding variable scope is true?
   (a) Global variables can only be accessed inside one function.
   (b) Local variables can be accessed anywhere in the program.
   (c) Global variables are accessible by all functions by default.
   (d) Static variables cannot be declared inside functions.
4. Which operator is used to obtain the memory address of a variable in both C and C++?
   (a) `%`
   (b) `&`
   (c) `*`
   (d) `^`
5. In C++, which of the following allows you to write multiple functions with the same name but different parameter types?

(a) Function pointers

(b) Function overloading

(c) Function templates only

(d) Macros

6. Which data type is most appropriate for storing a value like 3.14159 in C/C++?

   (a) `int`

   (b) `char`

   (c) `float`

   (d) `double`

7. Which statement about functions in C/C++ is correct?

   (a) A function must always return an `int`.

   (b) A function cannot be declared without parameters.

   (c) Functions can return any valid data type or `void`.

   (d) Only C++ supports functions with parameters.

8. In C, how do you create a constant variable that cannot be modified?

   (a) Use `#define` macro

   (b) Use `static`

   (c) Use `const` keyword before the data type

   (d) You cannot create constant variables in C

9. Which of the following is the correct way to declare a pointer to an integer in C/C++?

   (a) `int ptr*;`

   (b) `int *ptr;`

   (c) `int &ptr;`

   (d) `pointer int ptr;`

10. Which looping construct checks the condition *after* each iteration in C/C++?

    (a) `for`

    (b) `while`

    (c) `do-while`

    (d) None of the above

## Part 1 B: Short Form Questions (10 questions)

1. Define the term *variable declaration* and give one example in C/C++.
2. What is the primary difference between `float` and `double` in terms of precision?
3. Explain how you would declare and initialize a `bool` variable in C++.
4. How do you return a value from a function in C/C++? Provide a brief example.
5. What keyword is used in C++ to prevent a variable from being modified after initialization?
6. In C++, what is the purpose of a function prototype?
7. How would you write a function in C/C++ that does not return any value nor take any parameters?
8. State one reason why you might use a `typedef` in C.
9. Provide an example of a function call with two arguments in C/C++.
10. Explain what happens if you call a function in C/C++ but do not include its declaration or definition before the call.

## Part C: Long Form / Open-Ended Questions (5 questions)

1. Compare and contrast the way C and C++ handle boolean types, including any relevant headers or keywords required in C.
2. Discuss how local and global variables are stored in memory, highlighting the implications for program design and potential issues (e.g., naming conflicts, memory usage).
3. Explain the concept of *function overloading* in C++. How is it implemented, and why can it be advantageous compared to having multiple function names?
4. Describe in detail how function prototypes and definitions work together in a C/C++ program. Include information about header files, forward declarations, and linking.
5. Suppose you have a function that receives a large array or object as a parameter. Compare the implications of passing it by value, by pointer, and by reference (in C++). Which method would you typically choose for performance or safety reasons, and why?

1. **Comparison of Boolean Types in C and C++**

   **C Boolean Handling:** Standard C (prior to C99) lacks a native boolean type, so programmers use `int`, with 0 for false and non-zero (typically 1) for true. Since C99, the `_Bool` type, which holds only 0 or 1, is available. Including `<stdbool.h>` defines `bool` as a macro for `_Bool`, `true` as 1, and `false` as 0.

```c
#include <stdbool.h>
bool is_valid = true;
if (is_valid) { /* do something */ }
```

   Without `<stdbool.h>`, custom boolean definitions (e.g., `typedef int bool;`) are needed, reducing type safety as `_Bool` accepts integer assignments but coerces them to 0 or 1.

   **C++ Boolean Handling:** C++ has a native `bool` type that holds `true` or `false`, requiring no header. It enforces stricter type checking, converting non-zero integers to `true` and zero to `false`, but `bool` is distinct from `int`.

```cpp
bool is_valid = true;
if (is_valid) { /* do something */ }
```

   **Comparison:** C requires `<stdbool.h>` for `bool`, while C++ includes it natively. C++'s `bool` is more type-safe than C's `_Bool`. C code without `<stdbool.h>` may use inconsistent boolean definitions, affecting portability. Both types typically use 1 byte, though this is implementation-dependent.

   **Implications:** C's reliance on `<stdbool.h>` can lead to inconsistencies, while C++'s native `bool` enhances readability. C's integer-based booleans are flexible for low-level tasks like bit manipulation.

2. **Local and Global Variables: Memory Storage and Implications**

   **Memory Storage:**

   *Local Variables:* Allocated on the stack when a function is called, each call creating a new stack frame. They exist only during the function's execution and are deallocated upon return. Their scope is limited to the declaring block or function.

```c
void func() {
    int local = 10; // Stored on stack, destroyed when func returns
}
```

   *Global Variables:* Stored in the data segment (static storage) at program startup, persisting for the program's entire duration. They are accessible throughout the program unless restricted by `static` (file scope).

```c
int global = 10; // Stored in data segment, persists until program ends
```

   **Implications for Program Design:**

   *Global Variables:* Simplify data sharing across functions and maintain persistent state. However, they risk naming conflicts, are hard to debug due to widespread access, and occupy memory throughout execution. Best practice is to minimize their use and employ `static` for file-scope globals.

   *Local Variables:* Enhance encapsulation, reduce conflicts, and are automatically managed. They require parameter passing for data sharing, increasing complexity, and large local arrays or recursion can cause stack overflow. Use them for temporary, function-specific data.

   **Potential Issues:** Global variables with common names may clash across files without `static` or namespaces (C++). Excessive globals waste memory, while large local variables risk stack overflow.

In multithreaded programs, globals need synchronization to avoid race conditions, unlike thread-safe local variables.

3. **Function Overloading in C++**

   **Concept:** Function overloading allows multiple C++ functions to share the same name but differ in parameter lists (number, types, or order). The compiler selects the correct function based on arguments. C does not support this, requiring unique function names.

   **Implementation:** Define functions with the same name but different signatures. Return type alone cannot differentiate them. The compiler uses name mangling to create unique internal names based on parameter types.

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

   Calling example:

```
std::cout << add(1, 2) << std::endl;      // Calls int add(int, int)
std::cout << add(1.5, 2.5) << std::endl;  // Calls double add(double, double)
std::cout << add(1, 2, 3) << std::endl;   // Calls int add(int, int, int)
```

   **Advantages:** Overloading improves code readability by using intuitive names (e.g., `add` for different types) instead of multiple names (e.g., `addInt`, `addDouble`). It reduces naming complexity, enhances maintainability, and allows consistent interfaces for related operations, making code more user-friendly.

4. **Function Prototypes and Definitions in C/C++**

   **Function Prototypes:** A prototype declares a function's name, return type, and parameter list without its body, informing the compiler of its signature. It allows functions to be called before their definition.

```
int sum(int a, int b); // Prototype
```

   **Function Definitions:** Provide the actual implementation, including the body. They must match the prototype's signature.

```
int sum(int a, int b) {
    return a + b;
}
```

   **Working Together:** Prototypes enable forward declarations, allowing functions to be defined later or in separate files. The compiler checks calls against prototypes for type safety. During linking, the linker resolves calls to definitions.

   **Header Files:** Prototypes are typically placed in header files (`.h`) to share declarations across multiple source files. The header is included in source files, and definitions are provided in one source file to avoid multiple definition errors.

```
// sum.h
#ifndef SUM_H
#define SUM_H
int sum(int a, int b);
#endif

```

```
// sum.c
#include "sum.h"
int sum(int a, int b) {
    return a + b;
}

// main.c
#include "sum.h"
int main() {
    printf("%d\n", sum(3, 4));
    return 0;
}
```

**Forward Declarations:** Prototypes allow functions to call each other regardless of definition order within a file.

**Linking:** The linker combines object files, resolving function calls to their definitions. Missing definitions cause linker errors.

5. **Parameter Passing for Large Arrays or Objects**

**Passing by Value:** Creates a copy of the entire array or object, stored on the stack. For large data, this is slow due to copying overhead and increases stack memory usage. Changes to the copy do not affect the original.

```
void process(std::vector<int> data) { /* Copy made */ }
```

**Passing by Pointer (C/C++):** Passes the memory address, avoiding copying. Only the pointer (typically 4–8 bytes) is copied. Modifications to the data via the pointer affect the original. Null pointers or invalid addresses risk crashes.

```
void process(int *arr, int size) { arr[0] = 10; } // Modifies original
```

**Passing by Reference (C++):** Passes an alias to the original data, avoiding copying. Like pointers, modifications affect the original, but references cannot be null and are syntactically cleaner. Size information is needed for arrays unless using C++ containers.

```
void process(std::vector<int> &data) { data[0] = 10; } // Modifies original
```

**Choice for Performance/Safety:** Passing by reference (C++) is typically preferred for large objects or arrays. It avoids copying (fast), is safer than pointers (no null or invalid address issues), and is readable. For C, passing by pointer is the only option to avoid copying. Use `const` (e.g., `const std::vector<int> data`) to prevent unintended modifications, balancing performance and safety.