Introduction to C++

C++ is a versatile and powerful programming language created by Bjarne Stroustrup in 1979 as an enhancement of the C language. It combines the low-level memory control and performance of C with high-level features such as object-oriented programming, generic programming (templates), and a rich standard library.

C++ is widely used in system programming (operating systems, drivers), game development, real-time simulations, financial software, embedded systems, and high-performance applications. Its key strength is efficiency: programs written in C++ can run very close to the speed of hand-written assembly code while still being readable and maintainable.

A C++ program must have a function called main(), which is the entry point where execution begins. The language is case-sensitive, compiled (not interpreted), and requires explicit declaration of variables before use.

Understanding the core concepts—data types, control flow (loops), arrays, strings, and pointers—provides the foundation for everything else in C++.

Page 1: Data Types in C++

Data types specify what kind of data a variable can store and how much memory it occupies. Choosing the correct type improves performance, prevents bugs, and makes code clearer.

C++ categorizes data types as follows:

Fundamental (Built-in) Types

- **Integer types**: Used for whole numbers.
    - int: Typically 4 bytes, range roughly -2 billion to +2 billion.
    - short: Usually 2 bytes, smaller range.
    - long: At least 4 bytes, often larger range.
    - long long: 8 bytes, for very large integers.
    - Modifiers: signed (default, allows negative values) and unsigned (only non-negative, doubles positive range).
- **Floating-point types**: For numbers with decimal parts.
    - float: Single precision (about 7 decimal digits accuracy).
    - double: Double precision (about 15 decimal digits, default for floating-point literals).

    - long double: Extended precision on some platforms.
- **Character type**:
    - char: 1 byte, stores a single character (e.g., 'A', '7', '$') or small integers.
- **Boolean type**:
    - bool: Stores true or false.
- **Void type**:
    - void: Represents the absence of a value (used for functions that return nothing).

Other Categories

- Derived types: Arrays, pointers, references (explained later).
- User-defined types: struct, class, enum, union.

Variables are declared by specifying the type followed by the name, e.g., double temperature = 23.5;. The const keyword creates constants that cannot be modified after initialization: const int MAX_STUDENTS = 100;.

Proper use of data types prevents overflow errors and optimizes memory usage.

Page 2: Loops in C++

Loops enable a program to repeat a block of code multiple times, which is essential for processing collections of data, performing calculations repeatedly, or waiting for certain conditions.

C++ provides several loop constructs:

for Loop Best when the number of repetitions is known in advance. It consists of three parts: initialization, condition, and increment/decrement. The loop continues as long as the condition is true.

while Loop Checks the condition before executing the loop body. If the condition is false from the start, the body never runs. Ideal when the number of iterations depends on a runtime condition (e.g., reading input until end-of-file).

do-while Loop Similar to while, but the condition is checked after the body executes. This guarantees the body runs at least once. Commonly used for menus or input validation where the user must see the prompt first.

Range-based for Loop (introduced in C++11) Simplifies iteration over all elements in an array or container without needing an index. It is clean and less error-prone.

Inside any loop, you can use:

- break: Immediately exits the loop.
- continue: Skips the rest of the current iteration and proceeds to the next.

Nested loops (a loop inside another loop) are common for working with multidimensional data or generating patterns.

Avoiding infinite loops (where the condition never becomes false) is critical; always ensure there is a way for the loop to terminate.

Page 3: Arrays in C++

An array is a fixed-size collection of elements of the same data type stored in consecutive memory locations. This contiguous storage allows fast access using an index (starting at 0).

Key characteristics:

- Size must be known at compile time (for traditional arrays).
- All elements share the same type.
- Access time is constant regardless of index due to direct memory addressing.

Arrays are useful for storing lists of values such as scores, temperatures, or names when the maximum number is known.

Multidimensional arrays create grid-like structures (e.g., 2D for matrices, 3D for volumes). Internally, they are stored in row-major order in C++.

Important pitfalls:

- No built-in bounds checking—accessing an invalid index leads to undefined behavior (crashes or corrupted data).
- The array name acts as a constant pointer to the first element (connection to pointers).

For cases where size is determined at runtime or needs to change, modern C++ recommends std::vector from the Standard Library, but understanding raw arrays remains essential for performance-critical code and interoperability with C.

Page 4: Strings in C++

Strings store sequences of characters (text).

C++ supports two primary approaches:

C-style strings These are null-terminated arrays of char. The string ends with a special character '\0' that marks its boundary. They are inherited from C and require manual memory management. Functions from <cstring> (e.g., strlen, strcpy) operate on them. They are error-prone due to possible buffer overflows.

std::string (recommended) A class provided by the Standard Library (#include <string>). It manages memory automatically, grows dynamically, and offers convenient methods such as:

- Concatenation with +
- Length via .length() or .size()
- Substring extraction
- Searching and replacing

std::string is safer, more intuitive, and integrates seamlessly with other Standard Library components. Beginners should almost always use std::string unless interfacing with legacy C code.

Both types can be used for input/output with std::cin and std::cout, but std::string works more naturally.

Page 5–6: Pointers in C++

Pointers are variables that store memory addresses rather than data values directly. They are one of C++'s most distinctive and powerful features, enabling dynamic memory allocation, efficient parameter passing, and implementation of complex data structures.

Core concepts:

- **Address-of operator &**: Retrieves the memory address of a variable.
- **Dereference operator ***: Accesses the value stored at the address held by a pointer.
- Pointer declaration: The type specifies what kind of data it points to, e.g., int* p; declares a pointer to an integer.

Pointers allow indirect access: changing the value at the pointed-to location affects the original variable.

Special cases:

- nullptr: A null pointer constant indicating the pointer points to nothing (safer than old NULL or 0).

- Array-pointer relationship: The name of an array decays to a pointer to its first element, explaining why arrays and pointers are often used interchangeably in function parameters.
- Pointer arithmetic: Adding an integer to a pointer moves it by that many elements (not bytes), respecting the pointed-to type's size.

References (int& ref = var;) are closely related: they act as aliases for existing variables, must be initialized immediately, and cannot be null or reassigned.

Common uses of pointers:

- Dynamic memory allocation with new and delete.
- Building linked lists, trees, and other data structures.
- Passing large objects to functions efficiently (by reference or pointer).

Dangers include:

- Dangling pointers (pointing to freed memory).
- Memory leaks (allocated memory not freed).
- Null pointer dereference (program crash).

Modern C++ encourages smart pointers (std::unique_ptr, std::shared_ptr) to manage ownership automatically and reduce errors.

Mastering pointers gives deep insight into how programs interact with memory and is crucial for understanding advanced C++ features.

Conclusion

These fundamentals—data types, loops, arrays, strings, and pointers—form the bedrock of C++ programming. Once comfortable with them, you can explore functions, classes, templates, the Standard Template Library (STL), and modern C++ features (C++11/14/17/20/23).

The best way to learn is by writing small programs, experimenting, and gradually building more complex projects. Happy coding!