



Introduction to C / C++

Features • Program Structure • Tokens • Data Types

Fundamentals of Programming • Unit 1

Tap any item on the next slide to jump straight to that topic

Agenda

Click a topic to jump straight to it

01



Features of C & C++

What makes these languages powerful and widely used

Go

02



Structure of a Basic Program

The building blocks every C/C++ program is made of

Go

03



Tokens

The smallest meaningful units the compiler reads

Go

04



Data Types

How C/C++ classifies the kind of data you can store

Go

Features of C



[Back to Agenda](#)

The traits that made C the foundation of modern programming



Structured

Code is organized into functions and blocks — easy to read, test and reuse.



Middle-Level

Combines low-level access (pointers, memory) with high-level readability.



Portable

Source code runs on different machines with little or no change.



Fast & Efficient

Compiles close to machine code, so programs run with minimal overhead.



Rich Operators

A wide set of built-in operators for arithmetic, logic and bit-manipulation.



Extensible

New functionality can be added easily through user-defined functions & libraries.

Features of C++ — Built on Top of C

C++ is a superset of C: every C program is (almost) a valid C++ program, plus much more

C++

C (procedural core)

- Functions & structured code
- Structs & pointers
- printf / scanf I/O
- Manual memory management

C++ adds:

- Classes & Objects (OOP)
- Inheritance & Polymorphism
- Operator Overloading
- Templates (generic code)
- STL (containers & algorithms)
- cin / cout, exceptions

C vs C++ at a Glance

Paradigm	Procedural	Procedural + Object-Oriented
Approach	Top-down	Bottom-up (objects first)
Data security	Global data, less secure	Encapsulation via classes
Standard I/O	printf() / scanf()	cout / cin (streams)
Extras	—	Templates, STL, exceptions

Structure of a Basic Program

[Back to Agenda](#)

Every C/C++ program is assembled from the same set of building blocks, top to bottom

Documentation Section

Comments describing what the program does, author, date (`//` or `/* */`).

Preprocessor Directives

`#include` and `#define` lines — processed before compilation begins.

Global Declarations

Global variables and function prototypes, visible to the whole file.

`main()` Function

Local declarations + statements — where execution always begins.

User-Defined Functions

Additional functions called from `main()` or elsewhere in the program.

Only `main()` is mandatory — everything else is added as the program needs it.

Structure of C++ Program — A Real Example

Each highlighted line maps back to a section from the previous slide

```
// Program to add two numbers
#include <iostream>
using namespace std;

int sum(int, int);

int main() {
    int a = 5, b = 10;
    cout << sum(a, b);
    return 0;
}

int sum(int x, int y) {
    return x + y;
}
```

What Each Tag Means



Documentation

Notes for humans — ignored by the compiler.



Preprocessor

Handled before compilation; brings in libraries.



Global declaration

Visible everywhere in the file, declared outside main().



main() function

Program execution always starts and ends here.



User-defined function

Reusable block written by the programmer, called from main().

A Simple C++ Program

[Back to Agenda](#)

Example-1.

```
#include <iostream>    // C++ header

int main()
{
    std::cout << "Hello\n"; // Statement using iostream
    return 0;
}
```

- In C++, **std** is the **standard namespace** that contains all names (classes, functions, objects, templates) defined by the C++ Standard Library, such as cout, cin, string, vector, and algorithms.
- We use std to **organize library names under one scope** and to **avoid naming conflicts** between our code and the standard library (or other libraries)
- **What is a namespace?:**
 - A **namespace** is a named scope that groups related identifiers (variables, functions, classes) so the same name can be reused in different namespaces without collision.
- **What is std specifically?**
 - std stands for “**standard**”
 - The C++ standard specifies that **all standard library entities are defined inside namespace std** (including nested namespaces like std::chrono)
 - Headers like <iostream>, <string>, <vector>, <algorithm> declare their contents inside std.

A Simple C++ Program



[Back to Agenda](#)

Example-2.

// C++ program for sum of 2 numbers

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    int a, b, sum;
```

```
    a = 10;
```

```
    b = 15;
```

```
    sum = a + b;
```

```
    cout << "Sum of " << a << " and " << b
```

```
        << " is: " << sum; // perform addition operation
```

```
    return 0;
```

```
}
```

Tokens

[Back to Agenda](#)

The smallest individual units of a program that the compiler can recognize



Keywords

Reserved words with fixed meaning.

`int, if, return, class`



Identifiers

Names given to variables, functions.

`total, calculateSum`



Constants

Fixed values that never change.

`10, 3.14, 'A'`



Strings

Sequence of characters in quotes.

`"Hello, World"`



Operators

Symbols that act on operands.

`+ - * / == &&`



Punctuators

Special symbols that structure code.

`;, { } ( ) [ ]`

Tokens

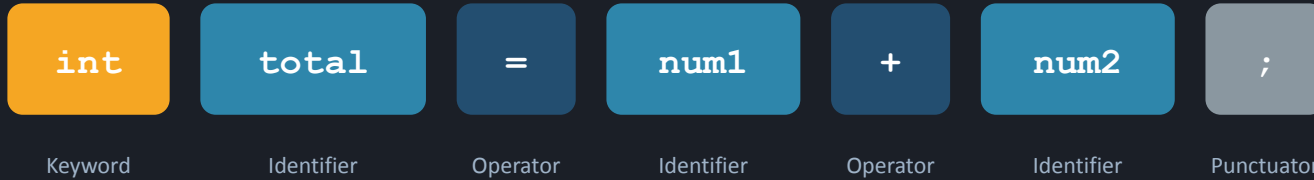
[Back to Agenda](#)

1. **Keywords:** Reserved words with fixed meaning (e.g., int, float, if, while, return; in C++ also class, template, namespace)
2. **Identifiers:** Names you choose for variables, functions, classes, etc. (must start with a letter or _, then letters/digits/_)
3. **Constants (Literals):** Fixed values such as 42, 3.14, 'A', "text"
4. **Operators:** Symbols that perform operations (+ - * / %, == != < >, && || !, = +=, ++ --, bitwise & | ^ ~ << >>, etc.)
5. **Separators / Special symbols:** Punctuation like ; , () { } [] and in C++ also ::, ->*, etc.
6. **String:** String is a group of characters.

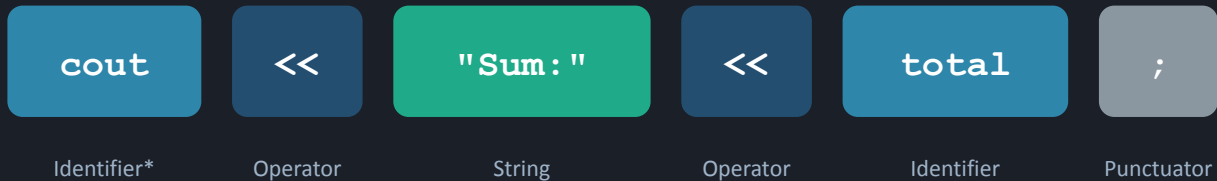
Tokens

*Token is a smallest unit of a program, such as variable (identifier) or keywords.
The compiler scans source code and splits it into tokens before doing anything else*

int total = num1 + num2;



cout << "Sum:" << total;



**cout is technically a predefined identifier from the iostream library, not a keyword.*

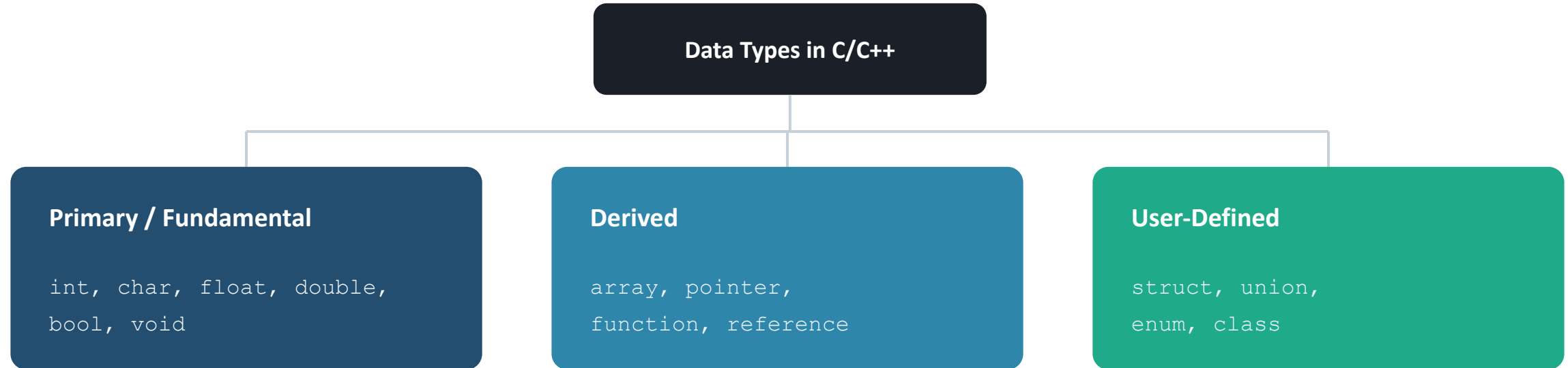
Legend:

 Keyword	 Identifier	 Operator	 String	 Punctuator
---	--	--	--	--

Data Types

[Back to Agenda](#)

1. A **data type** in C++ tells the compiler **what kind of value a variable can store, how that value should be interpreted, and what operations can be performed on it.**
2. *Every variable has a type that tells the compiler what kind of data it holds*



Primary types are built into the language; derived and user-defined types are built from them.

Primary/Fundamental Data Types

Typical sizes on a modern 32/64-bit system — always confirm with sizeof() on your compiler

Type	Size	Typical Range / Notes
char	1 byte	-128 to 127 (or 0 to 255 if unsigned)
int	4 bytes	-2,147,483,648 to 2,147,483,647
float	4 bytes	~6-7 significant decimal digits
double	8 bytes	~15-16 significant decimal digits
bool (C++)	1 byte	Only two values: true or false
void	0 bytes	Represents "no value" — used for functions with no return

Derived types (array, pointer) and user-defined types (struct, class) build on these primary types.

Primary /Fundamental Data Types

- Fundamental data types are the basic types directly provided by C++.
- The major fundamental types are:

Data Type	Purpose	Example
int	Whole numbers	25
char	Single character	'A'
float	Decimal numbers	12.5f
double	Higher-precision decimal numbers	12.56789
bool	True/false values	true
void	No value	void display()
wchar_t	Wide character	L'A'
char16_t	UTF-16 character/code unit	u'A'
char32_t	UTF-32 character/code unit	U'A'

Derived types (array, pointer) and user-defined types (struct, class) build on these primary types.

C++ Data Types and Their Sizes

- The exact size of a C++ data type can **depend on the compiler, operating system, processor architecture, and implementation**. The table below gives the sizes commonly encountered on modern systems.

Data Type	Typical Size	Typical Range / Purpose
bool	1 byte	true / false
char	1 byte	Character
signed char	1 byte	Small signed integer
unsigned char	1 byte	Small unsigned integer
short	2 bytes	Small integer
unsigned short	2 bytes	Small positive integer
int	4 bytes	General-purpose integer
unsigned int	4 bytes	Non-negative integer
long	4 or 8 bytes	Larger integer
unsigned long	4 or 8 bytes	Larger non-negative integer
long long	8 bytes	Very large integer
unsigned long long	8 bytes	Very large non-negative integer
float	4 bytes	Single-precision decimal
double	8 bytes	Double-precision decimal
long double	8, 12, or 16 bytes	Extended precision decimal
wchar_t	2 or 4 bytes	Wide-character type
char16_t	2 bytes	UTF-16 code unit
char32_t	4 bytes	UTF-32 code unit
void	No storage size	Represents absence of a value

How to check sizes on your computer

- To check the size of Data type Use the **sizeof** operator:

```
#include <iostream>
int main()
{
    std::cout << "bool: " << sizeof(bool) << " byte(s)\n";
    std::cout << "char: " << sizeof(char) << " byte(s)\n";
    std::cout << "short: " << sizeof(short) << " byte(s)\n";
    std::cout << "int: " << sizeof(int) << " byte(s)\n";
    std::cout << "long: " << sizeof(long) << " byte(s)\n";
    std::cout << "long long: " << sizeof(long long) << " byte(s)\n";
    std::cout << "float: " << sizeof(float) << " byte(s)\n";
    std::cout << "double: " << sizeof(double) << " byte(s)\n";
    std::cout << "long double: " << sizeof(long double) << " byte(s)\n";
    std::cout << "wchar_t: " << sizeof(wchar_t) << " byte(s)\n";
    std::cout << "char16_t: " << sizeof(char16_t) << " byte(s)\n";
    std::cout << "char32_t: " << sizeof(char32_t) << " byte(s)\n";
    return 0;
}
```

Note: Do not assume that every C++ data type has one fixed size on every computer. The C++ standard specifies minimum requirements, while the actual implementation determines many sizes.



Important Points

- **C** — is a fast, structured, portable middle-level language; C++ extends it with OOP, templates and the STL.
- **Program structure** — flows top to bottom: documentation, preprocessor directives, global declarations, `main()`, then other functions.
- **`main()`** — is the only mandatory part — it's always where execution begins.
- **Tokens** — are the smallest units the compiler recognizes: keywords, identifiers, constants, strings, operators, punctuators.
- **Data types** — fall into three families — primary (built-in), derived (array/pointer), and user-defined (struct/class).