Lecture 02

Introduction to C++. Built-in data types. Variables, pointers and references. Control structures. Functions.

Advanced Programming - SISSA, UniTS, 2025-2026

Giuseppe Alessio D'Inverno

07 Oct 2025

Why C++?

C++ is:

- Reasonably efficient in terms of CPU time and memory handling, being a **compiled language**.
- High demand in industry.
- A (sometimes exceedingly) complex language: if you know C++ you will learn other languages quickly.
- A **strongly typed**[1] language: safer code, less ambiguous semantic, more efficient memory handling.
- Supporting functional, object-oriented, and generic programming.
- Backward compatible (unlike Python... 😞). Old code compiles (almost) seamlessly.
- 🌲 It is green !

[1] Not everybody agrees on the definition of *strongly typed*.

Outline

1. Structure of a basic C++ program
2. Fundamental types
3. Memory management: variables, pointers, references, arrays
4. Conditional statements and control structures
5. Functions and operators
6. User-defined types: `enum`, `union`, `struct`, POD structs
7. Declarations and definitions
8. Code organization
9. The build toolchain in practice

Structure of a basic C++ program

Structure of a basic C++ program

- C++ program structure includes a collection of functions.
- Every C++ program must contain one `main()` function, which serves as the entry point.
- Other functions can be defined as needed.
- Statements within functions are enclosed in curly braces `{}`.
- Statements are executed sequentially unless control structures (e.g., loops, conditionals) are used.

Hello, world!

```cpp
#include <iostream>

int main() { // Or, more completely: int main(int argc, char** argv, char** envp)
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

- `#include <iostream>`: Includes the Input/Output stream library.
- `int main()`: Entry point of the program.
- `std::cout`: Standard output stream.
- `<<`: Stream insertion operator.
- `"Hello, world!"`: Text to be printed.
- `<< std::endl`: Newline character.
- `return 0;`: Indicates successful program execution.

How to compile and run

To compile the program:

```
g++ hello_world.cpp -o hello_world
```

To run the compiled program:

```
./hello_world [arg1] [arg2] ... [argN]
```

To check exit code:

```
echo $?
```

C++ as a strongly typed language

- C++ enforces strict type checking at compile time.
- Variables must be declared with a specific type.
- Type errors are detected and reported at compile time.
- This helps prevent runtime errors and enhances code reliability.

Example

```cpp
int x = 5;
char ch = 'A';
float f = 3.14;

x = 1.6;         // Legal, but truncated to the 'int' 1.
f = "a string"; // Illegal.

unsigned int y{3.0}; // Uniform initialization (since C++11): illegal.
```

Fundamental types

Fundamental types

Data Type	Size (Bytes)
`bool`	1
(`unsigned`) `char`	1
(`unsigned`) `short`	2
(`unsigned`) `int`	4
(`unsigned`) `long`	4 or 8
(`unsigned`) `long long`	8
`float`	4
`double`	8
`long double`	8, 12, or 16

Integer numbers

- C++ provides several integer types with varying sizes.
- Common integer types include `int`, `short`, `long`, and `long long`.
- The range of values that can be stored depends on the type.

Example

```
int age = 30;

short population = 32000;

long long large_number = 123456789012345;
```

Floating-point numbers

- C++ supports floating-point types for representing real numbers.
- Common floating-point types include `float`, `double`, and `long double`.
- These types can represent decimal fractions.

Example

```
float pi = 3.14;

double gravity = 9.81;
```

Floating-point arithmetic

Floating-point arithmetic is a method for representing and performing operations on real numbers $\pm f \cdot b^e$ in a binary format (i.e., $b = 2$).

- **Representation**: Floating-point numbers consist of three components: sign s (0: positive, 1: negative), significand f, and exponent e.
- **Normalized numbers**: In normalized form, the most significant bit of the significand is always 1, allowing for a wide range of values to be represented efficiently.
- **IEEE 754 standard**: The most commonly adopted standard for floating-point arithmetic is the IEEE 754 Standard for Floating-Point Arithmetic . This standard specifies the formats, precision, rounding rules, and handling of special values like NaN (Not-a-Number) and infinity.

Floating-point arithmetic limitations

```cpp
double epsilon = 1.0; // Machine epsilon.

while (1.0 + epsilon != 1.0) {
    epsilon /= 2.0;
}
```

```cpp
double a = 0.1, b = 0.2, c = 0.3;

if (a + b == c) { // Unsafe comparison.
    // This may not always be true due to precision limitations.
}
```

```cpp
double x = 1.0, y = 1.0 / 3.0; double sum = y + y + y;

if (std::abs(x - sum) < tolerance) { // Safer comparison.
    // Use tolerance to handle potential rounding errors.
}
```

Characters and strings

- Characters are represented using the `char` type.
- Strings are sequences of characters and are represented using the `std::string` type.

Example

```
char comma = ',';

std::string name = "John";

std::string greeting = "Hello";

// Concatenate strings.
std::string message = greeting + comma + ' ' + name;
```

Boolean types

- C++ has a built-in Boolean type called `bool`.
- It can have two values: `true` or `false`.
- Useful for conditional statements and logical operations.
- Numbers can be converted to Boolean.

Example

```cpp
bool is_true = true;

bool is_false = false;

if (-1.5) // true.
    // ...

if (0) // false.
    // ...
```

Initialization and aliases

- Initialization sets the initial value of a variable at the time of declaration.
- C++ supports various forms of initialization, including direct, copy, and list initialization.

Example

```
int x = 5; // Direct initialization.
int y(10); // Constructor-style initialization.
int z{15}; // Uniform initialization (since C++11; preferred).
```

- Aliases enable to create new types based on already defined others.

Example

```
using number = double;
using int_array = int*;
```

`auto` and type conversions.

In many situations, the compiler can determine the correct type of an object using the initialization value.

```
auto a{42};         // int.
auto b{12L};        // long.
auto c{5.0F};       // float.
auto d{10.0};       // double.
auto e{false};      // bool.
auto f{"string"}; // char[7].

// C++11.
auto fun1(const int i) -> int { return 2 * i; }

// C++14.
auto fun2(const int i) { return 2 * i; }
```

In reality, things are much more complex: see, for instance, **Explicit type conversion** and related pages.

Memory management: variables, pointers, references, arrays

Heap vs. stack

- Programs use memory to store data and variables.

Stack memory

- **Stack**: A region of memory for function call frames.
- Variables stored on the stack have a fixed size and scope.
- Memory is allocated and deallocated automatically.
- Well-suited for small, short-lived variables.

Heap memory

- **Heap**: A region of dynamic memory for data with varying lifetimes.
- Variables on the heap have a dynamic size and longer lifetimes.
- Memory allocation and deallocation are explicit (manual).
- Used for objects with unknown or extended lifetimes.

Variables and pointers

- Variables stored on the stack are typically accessed directly.
- Pointers to stack variables can be used safely within their scope.
- Heap-allocated variables require pointers for access.
- Pointers to heap variables must be managed carefully.

```
int stack_var = 42; // Stack variable.
int* stack_ptr = &stack_var; // Pointer to stack variable.

int* heap_ptr = new int(42); // Pointer to heap variable.
// ...
delete heap_ptr;
heap_ptr = nullptr;
```

Lifetime and scope

- Stack variables have a limited lifetime within their scope.
- Heap variables can have a longer lifetime beyond their defining scope.
- Deallocating heap memory is the programmer's responsibility.

Best practices

- Use the stack for small, short-lived variables.
- Use the heap for dynamic data with extended lifetimes.
- Always deallocate heap memory to prevent memory leaks.

Variables

- Variables are named memory locations used to store data.
- They must be declared with a specific type before use.
- Variables can be modified and accessed in your program.

Example

```
int x = 5; // Declaration and initialization.
x = 10;    // Variable modification.

int y;  // Declaration with default initialization.
y = 20; // Initialization after declaration.

const double a = 3.7;
a = 5; // Error!
```

Pointers

- Pointers are variables that store memory addresses.
- They allow you to work with memory directly.
- Declared using `*` symbol.

Example

```
int number = 42;

int* pointer = &number; // Pointer to 'number'.

// Create a dynamic integer with new.
int* dynamic_variable = new int;
*dynamic_variable = 5;

// Deallocate it.
delete dynamic_variable;
dynamic_variable = nullptr;
```

Pointers: common problems

```cpp
int* arr = new int[5]; // Dynamically allocate an integer array.

// Access and use the array beyond its allocated size.
for (int i = 0; i <= 5; ++i) {
    arr[i] = i;
}

// Forgot to delete the dynamically allocated array, causing a memory leak.
// delete[] arr;

// Attempt to access memory beyond the allocated array's bounds, causing undefined behavior.
std::cout << arr[10] << std::endl;
```

References

- References provide an alias for an existing variable.
- Declared using `&` symbol.
- Provide an alternative way to access a variable.

Example

```cpp
int a = 10;
int& ref = a; // Reference to 'a'.

ref = 20; // Modifies 'a'.

int b = 10;
ref = b;
ref = 5; // What's now the value of 'a' and 'b'?
```

Arrays

- Arrays are collections of elements of the same type.
- Elements are accessed by their index (position).
- C++ provides the much safer `std::array<type>`, `std::vector<type>` (to be covered in a sequent lecture).

Example

```
int numbers[5]; // Array declaration.
numbers[0] = 1; // Assigning values to elements.

int* dynamic_array = new int[5];

for (int i = 0; i < 5; ++i) {
    dynamic_array[i] = 2 * i;
}

delete[] dynamic_array;
```

Conditional statements and control structures

`if` ... `else if` ... `else`

- Conditional statements allow you to execute different code based on conditions.
- In C++, we use `if`, `else if`, and `else` statements for conditional execution.

Example

```cpp
int x = 10;

if (x > 5) {
    std::cout << "x is greater than 5." << std::endl;
} else if (x > 3) {
    std::cout << "x is greater than 3 but not greater than 5." << std::endl;
} else {
    std::cout << "x is not greater than 5." << std::endl;
}
```

`switch` ... `case`

- The `switch` statement is a control flow structure alternative to using multiple `if` ... `else` statements based on the value of an expression.

Example

```
switch (expression) {
    case constant1:
        // Code to execute if expression == constant1.
        break;
    case constant2:
        // Code to execute if expression == constant2.
        break;
    // ... more cases ...
    default:
        // Code to execute if expression doesn't match any case.
}
```

`for` loop

- A `for` loop is used to execute a block of code a specific number of times. It is often used when the number of iterations is known beforehand.

```
for (initialization; condition; post-iteration operation) {
    // Code to execute in each iteration.
}
```

Example

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

NB: `i` is usually named `counter`. A more general concept, called `iterator` will provide a generalization for custom objects.

`while` loop

- A `while` loop is used to repeat a block of code as long as the specified condition remains `true`. It is useful when the number of iterations is not predetermined.

```
while (condition) {
    // Code to execute while the condition is true.
}
```

Example

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

NB: A `do { ... } while(...)` loop is similar to a `while` loop but guarantees that the block of code will be executed at least once, as the condition is checked after the loop body.

Functions and operators

Functions

- Functions are blocks of code that perform a specific task.
- Functions are defined with a return type, name, and parameters.
- They can be called to execute their code.

Example

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

int result = add(3, 4); // Calling the 'add' function.
```

`void`

- `void` is a data type that represents the absence of a specific type.
- It indicates that a function does not return any value or that a pointer does not have a defined type.
- Dangerous to use.

Example

```
void greet() {
    std::cout << "Hello, world!" << std::endl;
}

void* generic_ptr;
int x = 10;

generic_ptr = &x; // Can point to any data type.
```

Pass by value vs. pointer vs. reference (1/2)

```cpp
void modify_by_copy(int x) {
    // Creates a copy of 'x' inside the function.
    x = 20; // Changes the copy 'x', not the original value.
}

void modify_by_ptr(int* ptr) {
    *ptr = 30; // Modifies the original value via the pointer.
}

void modify_by_ref(int& ref) {
    ref = 40; // Modifies the original value through the reference.
}
```

Pass by value vs. pointer vs. reference (2/2)

```cpp
int value = 10;

modify_by_copy(value); // Pass by value.
std::cout << value << std::endl; // Output: 10.

modify_by_ptr(&value); // Pass by pointer
std::cout << value << std::endl; // Output: 30.

modify_by_ref(value); // Pass by reference
std::cout << value << std::endl; // Output: 40.
```

Best practices

- Pass by value for small, non-mutable data.
- Pass by pointer for modifying values or working with arrays.
- Pass by reference for efficiency and direct modification of values.

Return by value vs. pointer vs. reference (1/2)

```cpp
int get_copy() {
    return 42; // Return a copy of the value.
}

int* get_ptr() {
    int* arr = new int[5];
    // ...
    return arr; // Return a pointer to the array.
}

int& get_ref() {
    static int value = 10; // static means that the variable persists
                           // through multiple calls to this function.
                           // Beware: if not static, undefined behavior
                           // (value gets destroyed).
    return value; // Return a reference to 'value'.
}
```

Return by value vs. pointer vs. reference (2/2)

```cpp
int result1 = get_copy(); // Return by value.

int* result2 = get_ptr(); // Return by pointer.
result2[2] = 5;
delete[] result2; // Beware: memory leaks.

int& result3 = get_ref(); // Return by reference.
result3 = 20;
```

Best practices

- Return by value for small, non-mutable data.
- Return by pointer for dynamically allocated data.
- Return by reference for efficiency and direct modification of data.

`const` correctness (1/2)

```cpp
void print_value(const int x) {
    // x = 42; // Error: Cannot modify 'x'.
}

const int get_copy() {
    const int x = 42;
    return x;
}

int result = get_copy();
result = 10; // Safe, it's a copy!

const int age = 30; // Immutable variable.
const int* ptr_to_const = &age; // Pointer to an integer which is constant.

ptr_to_const = &result; // Now pointing to another variable.
*ptr_to_const = 42; // Error: cannot modify pointed object.
```

Question: how to declare a constant pointer to a non-constant `int`?

`const` correctness (2/2)

Benefits

- Prevents unintended modifications: Helps avoid accidental data modifications, enhancing code safety.
- Self-documenting code: Makes code more self-documenting by indicating the intent of data usage.
- Compiler optimizations: Allows the compiler to perform certain optimizations, as it knows that const data won't change.

Best practices

- Const correctness is a valuable practice for writing safe and maintainable C++ code.
- Use `const` to indicate read-only data and functions.
- Incorrectly using `const` can lead to compiler errors or unexpected behavior.

Operators

- Operators are symbols used to perform operations on variables and values.
- Arithmetic operators: `+`, `-`, `*`, `/`, `%`
- Arithmetic and assignment operators: `+=`, `-=`, `*=`, `/=`, `%=`
- Comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`, `<=>` (C++20)
- Logical operators: `&&`, `||`, `!`

Example

```cpp
int x = 5, y = 3;
bool is_true = (x > y) && (x != 0); // Logical expression.
int z = (x > y) ? 2 : 1; // Ternary operator.

x += 2; // 7.
y *= 4; // 12.
z /= 2; // 1.
```

Increment operators

1. **Pre-increment (`++var`)**:
   - Increases the variable's value before using it.
   - The updated value is immediately reflected. No temporary needed: **more efficient**.
2. **Post-increment (`var++`)**:
   - Uses the current value of the variable before incrementing.
   - The variable's value is increased after its current value is used.

```cpp
int a = 5;
int b = ++a; // Pre-increment.
// a is now 6, b is also 6.

int c = a++; // Post-increment.
// a is now 7, but c is 6.
```

Function overloading

- Function overloading is a feature in C++ that allows you to define multiple functions with the same name but different parameters.
- The compiler selects the appropriate function based on the number or types of arguments during the function call.

```cpp
void print(int x) {
    std::cout << "Integer value: " << x << std::endl;
}

void print(double x) {
    std::cout << "Double value: " << x << std::endl;
}

print(3); // Calls the int version.
print(2.5); // Calls the double version.
```

User-defined types

`enum`

- Enumerations (enums) allow you to define a set of named values.
- Enums provide a way to create user-defined data types.

Example

```
enum Color : unsigned int {
    Red = 0,
    Green,
    Blue
};

Color my_color = Green;
```

`union`

- Unions allow you to define a type that can hold different data types.
- Only one member of a union can be accessed at a time.
- Useful for optimizing memory usage.

Example

```
union Duration {
    int seconds;
    short hours;
};

Duration d;
d.seconds = 259200;

short h = d.hours; // Contains garbage: undefined behavior.
```

`struct`

- Structs (structures) allow you to group related data members into a single unit.
- Members can have different data types.
- Structs provide a way to create custom data structures.

Example

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

Point p;
p.x = 3;
p.y = 5;
```

Actually, in C++ `struct` is just a special type of `class`. When Referring to C-style structs, a more proper name would be **Plain Old Data (POD) structs**.

Plain Old Data (POD) structs

- POD structs are classes with simple data members and no user-defined constructors or destructors.
- They have C-like semantics and can be used in low-level operations.

Example

```cpp
struct Rectangle {
    double width;
    double height;
};

Rectangle r;
r.width = 10;
r.height = 20;

Rectangle s{5, 10};
Rectangle t = s; // POD structs are trivially copyable.
```

Looking towards classes

- Object-oriented programming (OOP) is a programming paradigm that uses classes and objects.
- C++ is an object-oriented language that supports OOP principles.
- Classes are user-defined data types that encapsulate data and behavior.
- Classes extend structs by including **member functions** other than data.
- OOP promotes code reusability, modularity, and organization.

Declarations and definitions

Declaration

- Declarations inform the compiler about the existence of variables or functions.
- They provide type information but do not allocate memory or provide implementation.

```
int x; // Declaration of 'x'.
extern int y; // Declaration of 'y'.

struct X; // Forward-declaration. What if I want to use both X in Y and Y in X?
struct Y { X var; };
```

Definition

- Definitions provide the actual implementation of variables or functions.
- They allocate memory for variables or specify the behavior of functions.

```
int x = 5; // Definition of 'x'.
```

Declaring functions

- Function declarations provide enough information for the compiler to use the function.
- They specify the return type, name, and parameter types.
- Function declarations are typically placed in header files.

Example

```
int add(int a, int b); // Declaration of 'add' function.
```

Defining functions

- Function definitions specify the implementation of a function.
- They include the function's return type, name, parameters, and code block.
- They are typically placed in source files.

Example

```
int add(int a, int b) { // Definition of 'add' function.
    return a + b;
}
```

Code organization

Modular programming

- Modular programming divides code into separate modules or units.
- Each module focuses on a specific task or functionality.
- Benefits:
  - Improved code organization and readability
  - Easier maintenance and debugging
  - Code reusability
  - Encapsulation of functionality

Building blocks of C++ code modules

- C++ code modules consist of:
- Header files (`.h` or `.hpp`) for declarations
- Source files (`.cpp`) for definitions
- Implementation files (`.cpp`) for non-template classes
- Header files contain function prototypes and class declarations.
- Source files contain function and class definitions.

Header files

- Header files (`.h` or `.hpp`) contain declarations and prototypes.
- They define the interface to a module or class.
- Header files are included in source files to access declarations.

```
// my_module.hpp
int add(int a, int b); // Function prototype.
```

Best practices

- Use include guards or `#pragma once` to prevent multiple inclusions.
- Include only necessary headers to reduce compilation time.
- Keep header files concise and focused on declarations.
- Use descriptive and unique names for header files.
- Document complex or non-obvious declarations.

Source files

- Source files (`.cpp`) contain the definitions of functions and classes.
- They implement the functionality declared in header files.
- Source files include header files for access to declarations.

```cpp
// my_module.cpp
#include "my_module.hpp" // Include the corresponding header.

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

The need for header guards

- Header guards (or include guards) prevent multiple inclusions of the same header file.
- They ensure that a header file is included only once during compilation.
- Header guards are essential to avoid redefinition errors.

Without header guards, if a header file is included multiple times in a source file or across multiple source files, it can lead to redefinition errors.

How to implement header guards

- Place `#ifndef`, `#define`, and `#endif` or `#pragma once` directives in the header file.
- Use a unique identifier (usually based on the filename) as the guard symbol.

Example (file `my_module.hpp`):

```
#ifndef MY_MODULE_HPP__
#define MY_MODULE_HPP__

// ...

#endif // MY_MODULE_HPP__
```

Modern compilers also support:

```
#pragma once

// ...
```

Preventing header file inclusion issues

To avoid issues with header file inclusions:

- Include necessary headers in your source files.
- Avoid circular dependencies (A includes B, and B includes A).
- Use forward declarations when possible to minimize dependencies.
- Follow a consistent naming convention for header guards.

Managing scope in C++

- Scope determines the visibility and lifetime of variables and functions.
- C++ uses blocks, functions, and namespaces to manage scope.
- Variables declared inside a block have block scope.
- Variables declared outside of any function or class have global scope.
- Namespaces help organize code and avoid naming conflicts.

```cpp
int x = 10;

{ // Manually define a scope.
    int y = 20;
    // ...
} // Destroy all variables local to the scope.
// Beware: dynamically allocated variables must be deleted manually.

std::cout << y << std::endl; // Error: 'y' is undefined here.
```

Using namespaces for organization

- Namespaces group related declarations to avoid naming collisions.
- They provide a way to organize code into logical units.
- Namespace members are accessed using the `::` operator.

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

int result1 = Math::add(3, 4); // Accessing a namespace member.

using namespace Math; // Useful, but dangerous due to possible name clashes.
int result2 = add(3, 4);
```

Anonymous (i.e. unnamed) namespaces are only accessible from the current compilation unit.

The build toolchain in practice

Preprocessor and compiler

- The preprocessor (`cpp`) handles preprocessing directives.
- It includes headers, performs macro substitution, and removes comments.
- The compiler (`g++`, `clang++`) translates source code into object files.
- Preprocessor and compiler commands are combined when you run `g++` or `clang++`.

Example (project with three files: `module.hpp`, `module.cpp`, `main.cpp`):

```
# Preprocessor.
g++ -E module.cpp -I/path/to/include/dir -o module_preprocessed.cpp
g++ -E main.cpp -I/path/to/include/dir -o main_preprocessed.cpp

# Compiler.
g++ -c module_preprocessed.cpp -o module.o
g++ -c main_preprocessed.cpp -o main.o
```

Linker

- The linker (`ld`) combines object files and resolves external references.
- It creates an executable program from multiple object files.
- Linker errors occur if functions or variables are not defined.

Example

```
g++ module.o main.o -o my_program
```

Link against an external library:

```
g++ module.o main.o -o my_program -lmy_lib -L/path/to/my/lib
```

In this example, the `-lmy_lib` flag is used to link against the library `libmy_lib.so`. The `-l` flag is followed by the library name without the `lib` prefix and without the file extension `.so` (dynamic) or `.a` (static).

Preprocessor, compiler, linker: a simplified procedure

For small projects with few dependencies, the following command performs the preprocessing, compilation and linking phase:

```
g++ module1.cpp module2.cpp main.cpp -I/path/to/include/dir -o my_program
```

⚠️ Warning: different compilers lead to different behavior

Please keep in mind that different compilers can yield different behaviors and trigger distinct warnings or errors or print them in a less/more human-readable format.

For a demonstration, see this example on GodBolt comparing the output of GCC and Clang on the same code.

Loader

- The loader loads the executable program into memory for execution.
- It allocates memory for the program's data and code sections.
- The operating system's loader handles this task.

Example

```
./my_program
```

If linked against an external dynamic library, the loader has to know where it is located. The list of directories where to find dynamic libraries is contained in the colon-separated environment variable `LD_LIBRARY_PATH`.

```
export LD_LIBRARY_PATH+=:/path/to/my/lib
./my_program
```

➡️ Classes and object-oriented programming