C++

Everyday C++17, with clear ownership.

01 Compile & run

```cpp
#include <iostream>
int main() {
  std::cout << "Hello, C++!\n";
  return 0;
}
```

Save as main.cpp. Compile: g++ -std=c++17 main.cpp -o app. Run ./app. Later examples are independent snippets: put headers above main and statements inside it.

02 Variables & types

```cpp
int count = 3;
double price = 19.95;
bool ready = true;
char grade = 'A';
const int limit = 10;
auto total = count + 2;
```

Initialize variables before reading them. auto infers a type; const prevents modification through that name. Use std::string for text, not a single char.

03 Operators & conversions

```cpp
int whole = 7 / 2;       // 3
double ratio = 7 / 2.0;  // 3.5
int remainder = 7 % 2;   // 1
bool ok = whole > 0 && remainder == 1;
int n = static_cast<int>(ratio); // 3
++whole;
```

Use + - * / %, == != < <= > >=, and && || !. Integer division truncates toward zero. Conversions can lose data; signed integer overflow is undefined behavior.

04 Conditions & switch

```cpp
int score = 82;
if (score >= 60) ++score; else score = 0;
switch (score) {
  case 83: --score; break;
  default: score = 0; break;
}
```

if requires a condition; zero and nullptr are false. switch handles integral or enum values. break prevents falling into the next case.

05 Loops

```cpp
int values[]{1, 2, 3};
int total = 0;
for (int n : values) { total += n; }
for (int i = 0; i < 3; ++i) { total += i; }
while (total > 0) { --total; }
```

Range-for visits each element of an array or container. break exits the nearest loop; continue skips to its next iteration.

06 Strings

```cpp
#include <string>
std::string name = "Maya";
name += "!";
auto length = name.size(); // 5
std::string part = name.substr(0, 2);
bool same = name == "Maya!";
```

std::string owns mutable text. == compares contents. substr takes a start and a count. Indices and size count char units, not Unicode characters.

07 Vectors

```cpp
#include <vector>
std::vector<int> nums{3, 1, 2};
nums.push_back(4);
nums.at(0) = 9;
auto count = nums.size();
nums.pop_back();
```

A vector grows dynamically. at checks bounds; [] does not. Reallocation invalidates its pointers, references, and iterators. pop_back requires a nonempty vector.

08 Functions

```cpp
int add(int a, int b = 1) {
  return a + b;
}
int total = add(2);    // 3
int other = add(2, 3); // 5
```

Define this function outside main, before its calls. Parameters are copied by value here. Default arguments fill omitted trailing parameters; void means no returned value.

09 References & const

```cpp
#include <string>
void rename(std::string& name) {
  name = "Maya";
}
// Inside main:
std::string name = "Leo";
rename(name);
```

Define rename outside main. T& aliases an existing object; const T& provides read-only access without copying. A reference must not outlive the object it refers to.

C++

Objects, resources, and useful APIs.

10 Pointers & lifetimes

```
int value = 7;
int* ptr = &value;
*ptr = 9; // value is now 9
int* empty = nullptr;
if (ptr != nullptr) { *ptr = 10; }
```

Raw pointers usually observe an object without owning it. Only dereference pointers to live objects. Do not return pointers or references to local variables.

11 Classes & construction

```
class Counter {
  int value_;
public:
  explicit Counter(int n) : value_(n) {}
  int value() const { return value_; }
};
// Inside main: Counter counter{3};
```

The constructor initializes value_. Class members are private by default; struct uses public. A const method cannot change ordinary data members. Define the class outside main.

12 RAII & smart pointers

```
#include <memory>
#include <utility>
auto owner = std::make_unique<int>(42);
auto next = std::move(owner);
int value = *next; // 42
// owner is now empty
```

RAII releases owned resources at destruction. Prefer values and unique_ptr for exclusive ownership. Moving a unique_ptr transfers ownership; shared_ptr is for shared ownership.

13 Maps

```
#include <map>
#include <string>
std::map<std::string, int> ages;
ages["Maya"] = 28;
auto it = ages.find("Leo");
bool found = it != ages.end();
```

map stores unique keys in sorted order. [] inserts a missing key with a value-initialized value (0 for int). find reads without insertion; end means not found.

14 Sets

```
#include <set>
std::set<int> ids{3, 1, 3};
ids.insert(2);
ids.erase(3);
bool found = ids.count(2) > 0;
auto total = ids.size(); // 2
```

set stores sorted unique values. count is 0 or 1. unordered_map and unordered_set use hashing with no sorted order; include their matching headers.

15 Standard algorithms

```
#include <algorithm>
#include <vector>
std::vector<int> nums{3, 1, 2};
std::sort(nums.begin(), nums.end());
auto it = std::find(
  nums.begin(), nums.end(), 2);
```

Algorithms use ranges [begin, end): the end is excluded. sort changes this vector; find returns end if absent. Never dereference an end iterator.

16 Lambdas & captures

```
int offset = 2;
auto add = [offset](int n) {
  return n + offset;
};
int value = add(3); // 5
```

[offset] copies a value into the lambda. Use [&offset] to refer to the original only while it stays alive. Captured references do not extend an object's lifetime.

17 Exceptions

```
#include <stdexcept>
#include <iostream>
try { throw std::runtime_error("Bad"); }
catch (const std::exception& error) {
  std::cerr << error.what() << '\n';
}
```

Catch exceptions by const reference. Local objects are destroyed as exceptions unwind the stack, so RAII protects resources. Handle errors at a useful boundary.

18 Files & stream errors

```
#include <fstream>
#include <stdexcept>
std::ofstream out{"notes.txt"};
if (!out) throw std::runtime_error("Open");
out << "Hello\n";
out.close();
if (!out) throw std::runtime_error("Write");
```

ofstream replaces existing contents. ifstream reads; getline reads text lines. Streams close at destruction; explicit close lets you check flush errors.