C++

Basic — fundamentals you must know

1. Hello world, structure, main

```cpp
#include <iostream>

// Entry point
int main() {
    std::cout << "Hello, world!\n";
    return 0; // optional in C++: falling off main returns 0
}
```

Explanation: #include brings in headers. main is program entry. std::cout prints to stdout. \n newline.

2. Variables, types, auto

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

int main() {
    int i = 42;
    double d = 3.14;
    bool flag = true;
    std::string name = "Arman";

    auto x = i + 10; // compiler deduces type (int)
    std::cout << name << " " << x << " " << d << "\n";
}
```

Explanation: Built-in types, std::string. auto lets compiler deduce type.

3. Control flow: if, switch, loops

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 5; ++i) {
        if (i % 2 == 0) {
            std::cout << i << " even\n";
        } else {
            std::cout << i << " odd\n";
        }
    }

    int choice = 2;
    switch (choice) {
        case 1: std::cout << "one\n"; break;
        case 2: std::cout << "two\n"; break;
        default: std::cout << "other\n";
    }
}
```

Explanation: Standard for-loop, if, switch, break to prevent fallthrough.

4. Functions and parameter passing

```cpp
#include <iostream>

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

void inc(int &x) { // pass by reference
    ++x;
}

int main() {
    int a = 5;
    std::cout << add(a, 3) << "\n"; // 8
    inc(a);
    std::cout << a << "\n"; // 6
}
```

Explanation: Functions, pass-by-value vs pass-by-reference (&).

5. Simple class & constructor

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

class Person {
public:
    Person(std::string name, int age) : name_(std::move(name)), age_(age) {}
    void greet() const { std::cout << "Hi, I'm " << name_ << " (" << age_ << ")\n"; }
private:
    std::string name_;
    int age_;
};

int main() {
    Person p("Arman", 25);
    p.greet();
}
```

Explanation: Class, constructor initializer list, const method, encapsulation.

Intermediate — common modern patterns & standard library

1. Vectors, iterators, range-based for (STL)

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {3, 1, 4, 1, 5};
    std::sort(v.begin(), v.end());
    for (int n : v) { // range-based for
        std::cout << n << " ";
    }
    std::cout << "\n";
}
```

Explanation: std::vector dynamic array, algorithms (std::sort), iterators.

2. Smart pointers & RAII

```cpp
#include <iostream>
#include <memory>

struct Node {
    int value;
    Node(int v): value(v) { std::cout << "Node created\n"; }
    ~Node() { std::cout << "Node destroyed\n"; }
};

int main() {
    {
        std::unique_ptr<Node> p = std::make_unique<Node>(10);
        std::cout << p->value << "\n";
    } // p goes out of scope -> Node destroyed

    std::shared_ptr<Node> s1 = std::make_shared<Node>(20);
    {
        std::shared_ptr<Node> s2 = s1;
        std::cout << s2->value << "\n"; // shared ownership
    } // s2 destroyed, s1 still owns
}
```

Explanation: unique_ptr exclusive ownership; shared_ptr shared ownership; RAII ensures deterministic destruction.

3. Lambdas and std::function

```cpp
#include <iostream>
#include <functional>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> v = {1,2,3,4,5};
    int factor = 2;

    std::for_each(v.begin(), v.end(), [factor](int x) {
        std::cout << x * factor << " ";
    });
    std::cout << "\n";

    // capturing by reference
    std::function<int(int)> adder;
    int base = 10;
    adder = [&base](int x){ return base + x; };
    std::cout << adder(5) << "\n"; // 15
}
```

Explanation: Anonymous functions (lambda); capture semantics ([=], [&], explicit capture).

4. Move semantics (rvalue refs) & std::move

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

std::string make_string() {
    std::string s = "temporary";
    return s; // move or RVO
}

int main() {
    std::string s1 = make_string();            // likely moved / RVO
    std::string s2 = std::move(s1);           // s1 becomes unspecified
    std::cout << "s2: " << s2 << "\n";
    std::cout << "s1 now: '" << s1 << "'\n";  // s1 is valid but unspecified content
}
```

Explanation: std::move turns lvalue into rvalue to enable moving resources (avoids copies).

5. Exceptions

```cpp
#include <iostream>
#include <stdexcept>

int safe_divide(int a, int b) {
    if (b == 0) throw std::runtime_error("divide by zero");
    return a / b;
}

int main() {
    try {
        std::cout << safe_divide(10, 0) << "\n";
    } catch (const std::exception &e) {
        std::cerr << "Error: " << e.what() << "\n";
    }
}
```

Explanation: Throw/catch exceptions for error handling (std::exception base).

Advanced — modern C++ power features

1. **constexpr and compile-time computation (C++17/C++20)**

```cpp
#include <iostream>

constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

int main() {
    constexpr int f5 = factorial(5); // computed at compile time
    std::cout << f5 << "\n"; // 120
}
```

Explanation: constexpr functions evaluated at compile-time when possible.

2. **Templates (generic programming) & simple type traits**

```cpp
#include <iostream>
#include <type_traits>

template<typename T>
T add(T a, T b) {
    return a + b;
}

// enable_if example: only enable for integral types
template<typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
twice(T x) {
    return x * 2;
}

int main() {
    std::cout << add(1, 2) << "\n";
    std::cout << add(1.5, 2.5) << "\n";
    std::cout << twice(5) << "\n";
    // twice(3.14); // compile error: not integral
}
```

Explanation: Templates for generic code, enable_if + type_traits to constrain templates (SFINAE technique).

3. **C++20 Concepts (cleaner constraints)**

```cpp
#include <concepts>
#include <iostream>

template<typename T>
concept Addable = requires(T a, T b) {
    { a + b } -> std::convertible_to<T>;
};

template<Addable T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << add(1,2) << "\n";
    // add(std::string("a"), std::string("b")) also works
}
```

Explanation: Concepts (C++20) make template constraints readable and produce better error messages.

4. **Move-only types, custom move ctor/assign**

```cpp
#include <iostream>
#include <utility>

struct NonCopyable {
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;            // not copyable
    NonCopyable& operator=(const NonCopyable&) = delete;
    NonCopyable(NonCopyable&&) noexcept { std::cout << "moved\n"; } // movable
    NonCopyable& operator=(NonCopyable&&) noexcept { std::cout << "move assign\n"; return *this; }
};

int main() {
    NonCopyable a;
    NonCopyable b = std::move(a); // uses move ctor
}
```

Explanation: Control copy/move operations explicitly using = delete or custom implementations.

5. **Multithreading (C++11 threads)**

```cpp
#include <iostream>
#include <thread>
#include <vector>

void worker(int id) {
    std::cout << "worker " << id << " running\n";
}

int main() {
    std::vector<std::thread> threads;
    for (int i = 0; i < 4; ++i) {
        threads.emplace_back(worker, i);
    }
    for (auto &t : threads) t.join(); // wait for all
}
```

Explanation: std::thread, join threads to avoid premature program exit. For real code, use synchronization (mutexes, locks) where needed.

6. Coroutines (C++20) — simplified example using generator-like coroutine

Note: Compiler support (and libraries) are evolving; requires -std=c++20 and often extra flags.

7. Template metaprogramming (type-level computation) — short taste

```cpp
#include <iostream>

// compute Fibonacci at compile time (simple)
template<int N>
struct Fib {
    static constexpr int value = Fib<N-1>::value + Fib<N-2>::value;
};
template<> struct Fib<0> { static constexpr int value = 0; };
template<> struct Fib<1> { static constexpr int value = 1; };

int main() {
    constexpr int f7 = Fib<7>::value; // computed at compile time
    std::cout << f7 << "\n"; // 13
}
```

Explanation: Using templates for compile-time computation (classic TMP).

Practical tips, style & compilation

- Prefer modern C++ (C++11 and later): auto, range-based for, smart pointers, move semantics.
- Use RAII: resource management in constructors/destructors rather than raw new/delete.
- Prefer std::vector and STL containers over raw arrays.
- Mark functions const where appropriate.
- Use -Wall -Wextra -Werror during development to catch warnings early.
- Compile commands:
    - g++ -std=c++17 -Wall -Wextra file.cpp -o prog
    - For C++20: g++ -std=c++20 file.cpp -o prog

Quick reference (common keywords)

class, struct, public, private, protected, virtual, override, template, constexpr, auto, decltype, static, mutable, friend, namespace, using, typedef/using, try/catch/throw.