

C++ Intermediate Cheatsheet

A comprehensive reference for intermediate C++ developers covering modern C++ features, best practices, and common patterns.

Table of Contents

- [Quick Keyword Index](#)
- [Language Basics](#)
- [Memory Management](#)
- [Object-Oriented Programming](#)
 - [Virtual Functions & Polymorphism](#)
 - [Override and final](#)
 - [Abstract Classes \(Interfaces\)](#)
 - [Object Slicing](#)
 - [RTTI: dynamic_cast and typeid](#)
- [Templates & Generics](#)
- [Standard Library](#)
- [Modern C++ Features](#)
 - [Attributes \(e.g., \[\[nodiscard\]\]\)](#)
- [Best Practices](#)
- [Common Patterns](#)
- [Error Handling](#)
- [Performance](#)

Language Basics

Data Types & Declarations

```
// Fundamental types
int x = 42;           // Signed integer
unsigned int y = 42u; // Unsigned integer
float f = 3.14f;      // Single precision
double d = 3.14159;   // Double precision
char c = 'A';         // Character
bool b = true;        // Boolean

// Fixed-width types (C++11)
#include <cstdint>
int8_t i8;   int16_t i16;   int32_t i32;   int64_t i64;
uint8_t u8;  uint16_t u16;  uint32_t u32;  uint64_t u64;

// Auto type deduction (C++11)
auto value = 42;           // int
auto ptr = &value;         // int*
auto lambda = [](){};      // lambda type
```

```
// References
int& ref = x;           // L-value reference
const int& cref = x;    // Const reference
int&& rref = std::move(x); // R-value reference (C++11)

// Pointers
int* ptr = &x;          // Raw pointer
const int* cptr = &x;   // Pointer to const
int* const pc = &x;     // Const pointer
void* vptr = &x;        // Void pointer
```

Control Flow

```
// Conditional statements
if (condition) {
    // code
} else if (other_condition) {
    // code
} else {
    // code
}

// Ternary operator
auto result = condition ? value1 : value2;

// Switch with modern features (C++11)
switch (value) {
    case 1:
        // code
        break;
    case 2: {
        // scoped block
        int x = 42;
        break;
    }
    default:
        // code
        break;
}

// Loops
for (int i = 0; i < 10; ++i) {
    // code
}

for (auto& item : container) { // Range-based for (C++11)
    // code
}

while (condition) {
    // code
}
```

```

}

do {
    // code
} while (condition);

```

Functions

```

// Function declaration and definition
return_type function_name(parameter_list) {
    // body
    return value;
}

// Default parameters
void func(int x, int y = 10, int z = 20);

// Function overloading
void func(int x);
void func(double x);
void func(int x, int y);

// Inline functions
inline int square(int x) { return x * x; }

// Constexpr functions (C++11)
constexpr int factorial(int n) {
    return n <= 1 ? 1 : n * factorial(n - 1);
}

// Lambda expressions (C++11)
auto lambda = [](int x) { return x * 2; };
auto lambda2 = [](int x, int y) -> int { return x + y; };

// Function pointers and std::function
using FuncPtr = int (*)(int);
std::function<int(int)> func_obj = [](int x) { return x; };

```

Memory Management

Smart Pointers (C++11)

```

#include <memory>

// Unique pointer - exclusive ownership
std::unique_ptr<int> uptr = std::make_unique<int>(42);
auto uptr2 = std::make_unique<std::vector<int>>(10);

```

```
// Shared pointer - shared ownership
std::shared_ptr<int> sptr = std::make_shared<int>(42);
auto sptr2 = sptr; // Reference count increases

// Weak pointer - breaks circular references
std::weak_ptr<int> wptr = sptr;
if (auto locked = wptr.lock()) {
    // Use the pointer safely
}

// Custom deleter
auto custom_deleter = [](int* ptr) { delete ptr; /* custom cleanup */ };
std::unique_ptr<int, decltype(custom_deleter)> uptr_custom(new int(42),
custom_deleter);
```

RAII (Resource Acquisition Is Initialization)

```
class Resource {
public:
    Resource() { /* acquire resource */ }
    ~Resource() { /* release resource */ }
    Resource(const Resource&) = delete;
    Resource& operator=(const Resource&) = delete;
    // Move operations if needed
};

// Usage
void function() {
    Resource r; // Resource acquired
    // Use resource
} // Resource automatically released
```

Raw Memory Management (Use with Caution)

```
// Allocation
int* ptr = new int(42); // Single object
int* arr = new int[10]; // Array
int* placement = new (buffer) int(42); // Placement new

// Deallocation
delete ptr; // Single object
delete[] arr; // Array

// Smart alternatives (preferred)
auto uptr = std::make_unique<int>(42);
auto sptr = std::make_shared<int>(42);
```

Object-Oriented Programming

Classes and Structs

```
class MyClass {
public:
    // Constructors
    MyClass() = default; // Default
    MyClass(int x) : member_(x) {} // Member initializer list
    MyClass(const MyClass& other) = default; // Copy
    MyClass(MyClass&& other) noexcept = default; // Move (C++11)

    // Destructor
    ~MyClass() noexcept = default;

    // Assignment operators
    MyClass& operator=(const MyClass& other) = default;
    MyClass& operator=(MyClass&& other) noexcept = default;

    // Member functions
    int get_value() const { return member_; } // Const member function
    void set_value(int x) { member_ = x; }

    // Static members
    static int static_member_;
    static void static_function();

private:
    int member_ = 0; // Default member initializer (C++11)
};

// Static member definition (outside class)
int MyClass::static_member_ = 42;
```

Inheritance

```
class Base {
public:
    virtual ~Base() = default; // Virtual destructor
    virtual void virtual_func() {} // Virtual function
    void non_virtual_func() {}
};

class Derived : public Base {
public:
    void virtual_func() override {} // Override specifier (C++11)
    void non_virtual_func() {} // Hides base function
};

// Usage
```

```
std::unique_ptr<Base> ptr = std::make_unique<Derived>();
ptr->virtual_func(); // Calls Derived::virtual_func()
```

Virtual Functions & Polymorphism

- **What `virtual` does:** Enables *dynamic dispatch* (runtime polymorphism). When you call a **virtual** function through a base **pointer/reference**, the program chooses the most-derived override at runtime.
- **Where it matters:** `Base*` / `Base&` calls. If you call via a value (a sliced copy), you lose polymorphism (see [Object Slicing](#)).
- **Virtual destructor rule of thumb:** If a class has *any* virtual functions and is meant to be deleted via a base pointer, give it a **virtual destructor**.

```
struct Animal {
    virtual ~Animal() = default; // important for polymorphic base
types
    virtual void speak() const { // virtual enables dynamic dispatch
        std::cout << "??\n";
    }
};

struct Dog : Animal {
    void speak() const override { // checked by compiler
        std::cout << "woof\n";
    }
};

void demo(const Animal& a) { // reference keeps polymorphism
    a.speak(); // Dog prints "woof"
}
```

Override and final

- **override:** Put on a derived virtual function to make the compiler verify that you are *actually overriding* a base virtual function (catches signature mismatches).
- **final:**
 - On a **virtual function**: prevents further overriding.
 - On a **class**: prevents inheriting from it.

```
struct Base2 {
    virtual void f(int) {}
};

struct Derived2 : Base2 {
    void f(int) override {} // OK
    // void f(double) override {} // ERROR: doesn't override anything
    void f(int) final {} // no further overrides allowed
}
```

```
};

struct Leaf final : Base2 {           // cannot derive from Leaf
    void f(int) override {}
};
```

Abstract Classes (Interfaces)

- A **pure virtual** function (= 0) makes a class **abstract** (you cannot instantiate it).
- A common “interface” style in C++ is a class with:
 - a virtual destructor
 - only pure virtual methods

```
struct Activation {
    virtual ~Activation() = default;
    virtual double forward(double x) const = 0;    // pure virtual
    virtual double derivative(double x) const = 0; // pure virtual
};

struct ReLU : Activation {
    double forward(double x) const override { return x > 0 ? x : 0; }
    double derivative(double x) const override { return x > 0 ? 1 : 0; }
};
```

Object Slicing

- **Slicing** happens when you copy/assign a derived object into a base object **by value**. The derived part is “sliced off”.
- **Symptom**: virtual dispatch stops behaving like you expect (because you no longer have a derived object).

```
struct B { virtual ~B() = default; virtual void f() const { std::cout << "B\n"; } };
struct D : B { void f() const override { std::cout << "D\n"; } };

void by_value(B b) { b.f(); }           // prints "B" (sliced)
void by_ref(const B& b) { b.f(); }      // prints "D" for D

int main() {
    D d;
    by_value(d);
    by_ref(d);
}
```

RTTI: dynamic_cast and typeid

- **RTTI** (runtime type information) is what enables `dynamic_cast` and `typeid`.

- **dynamic_cast:**
 - safe downcast for polymorphic types (types with at least one virtual function)
 - returns `nullptr` for pointers on failure; throws `std::bad_cast` for references

```
struct Shape { virtual ~Shape() = default; };
struct Circle : Shape { void draw_circle() {} };

void try_downcast(Shape* s) {
    if (auto* c = dynamic_cast<Circle*>(s)) {
        c->draw_circle();
    }
}

void print_type(const Shape& s) {
    std::cout << typeid(s).name() << "\n";    // implementation-defined
    string
}
```

Access Specifiers and Friends

```
class MyClass {
private:
    int private_member_;
protected:
    int protected_member_;
public:
    int public_member_;

    // Friend declarations
    friend class FriendClass;
    friend void friend_function(MyClass& obj);
    friend bool operator==(const MyClass& lhs, const MyClass& rhs);
};
```

Operator Overloading

```
class Complex {
public:
    Complex(double r, double i) : real_(r), imag_(i) {}

    // Arithmetic operators
    Complex operator+(const Complex& other) const {
        return Complex(real_ + other.real_, imag_ + other.imag_);
    }

    Complex& operator+=(const Complex& other) {
        real_ += other.real_;
```

```

        imag_ += other.imag_;
        return *this;
    }

    // Comparison operators
    bool operator==(const Complex& other) const {
        return real_ == other.real_ && imag_ == other.imag_;
    }

    // Conversion operators
    explicit operator double() const { return real_; }

    // Increment/decrement
    Complex& operator++() { ++real_; return *this; }
    Complex operator++(int) { Complex temp = *this; ++real_; return temp; }

private:
    double real_, imag_;
};

// Non-member operators
Complex operator+(double lhs, const Complex& rhs) {
    return Complex(lhs + rhs.real_, rhs.imag_);
}

```

Templates & Generics

Function Templates

```

// Basic function template
template <typename T>
T max(T a, T b) {
    return a > b ? a : b;
}

// Multiple template parameters
template <typename T, typename U>
auto add(T a, U b) -> decltype(a + b) { // Trailing return type (C++11)
    return a + b;
}

// Template specialization
template <>
const char* max(const char* a, const char* b) {
    return std::strcmp(a, b) > 0 ? a : b;
}

// Variadic templates (C++11)
template <typename... Args>
void print(Args... args) {

```

```
(std::cout << ... << args) << std::endl; // Fold expression (C++17)
}
```

Class Templates

```
template <typename T, std::size_t N>
class Array {
public:
    T& operator[](std::size_t index) { return data_[index]; }
    const T& operator[](std::size_t index) const { return data_[index]; }

    std::size_t size() const { return N; }

private:
    T data_[N];
};

// Template with default parameters
template <typename T = int, typename Allocator = std::allocator<T>>
class Vector {
    // Implementation
};

// Template specialization
template <>
class Vector<bool> { // Space-optimized bool vector
    // Compact bit representation
};
```

Template Metaprogramming

```
// Compile-time factorial
template <std::size_t N>
struct Factorial {
    static constexpr std::size_t value = N * Factorial<N-1>::value;
};

template <>
struct Factorial<0> {
    static constexpr std::size_t value = 1;
};

// Usage
constexpr auto fact5 = Factorial<5>::value; // 120

// Type traits (C++11)
#include <type_traits>

template <typename T>
```

```
void process(T value) {
    if constexpr (std::is_integral_v<T>) {           // C++17
        std::cout << "Integer: " << value << std::endl;
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Float: " << value << std::endl;
    } else {
        std::cout << "Other type" << std::endl;
    }
}
```

Concepts (C++20)

```
#include <concepts>

// Define a concept
template <typename T>
concept Numeric = std::is_arithmetic_v<T>;

// Use concept in template
template <Numeric T>
T add(T a, T b) {
    return a + b;
}

// Concept-based constraints
template <typename T>
concept Container = requires(T t) {
    typename T::value_type;
    t.begin();
    t.end();
    t.size();
};

template <Container C>
void print_size(const C& container) {
    std::cout << container.size() << std::endl;
}
```

Standard Library

Containers

```
#include <vector>
#include <array>
#include <deque>
#include <list>
#include <forward_list>
#include <set>
```

```

#include <map>
#include <unordered_set>
#include <unordered_map>

// Sequence containers
std::vector<int> vec = {1, 2, 3, 4, 5};
std::array<int, 5> arr = {1, 2, 3, 4, 5}; // Fixed size
std::deque<int> deq = {1, 2, 3}; // Double-ended queue
std::list<int> lst = {1, 2, 3}; // Doubly-linked list

// Associative containers
std::set<int> s = {3, 1, 4, 1, 5}; // Unique, sorted
std::map<std::string, int> m = {"one", 1}, {"two", 2};
std::unordered_set<int> us = {3, 1, 4, 1, 5}; // Hash-based
std::unordered_map<std::string, int> um = {"one", 1}, {"two", 2};

// Container operations
vec.push_back(6); // Add to end
vec.pop_back(); // Remove from end
vec.insert(vec.begin(), 0); // Insert at position
vec.erase(vec.begin()); // Erase at position
vec.clear(); // Remove all elements

// Access elements
auto first = vec.front(); // First element
auto last = vec.back(); // Last element
auto elem = vec[0]; // Random access (vector only)
auto elem2 = vec.at(0); // Bounds-checked access

```

Algorithms

```

#include <algorithm>
#include <numeric>

std::vector<int> vec = {3, 1, 4, 1, 5, 9, 2};

// Non-modifying algorithms
auto it = std::find(vec.begin(), vec.end(), 4); // Find element
auto count = std::count(vec.begin(), vec.end(), 1); // Count occurrences
bool all_even = std::all_of(vec.begin(), vec.end(), [](int x){ return x % 2 == 0; });
bool any_even = std::any_of(vec.begin(), vec.end(), [](int x){ return x % 2 == 0; });
bool none_even = std::none_of(vec.begin(), vec.end(), [](int x){ return x % 2 == 0; });

// Modifying algorithms
std::sort(vec.begin(), vec.end()); // Sort in-place
std::reverse(vec.begin(), vec.end()); // Reverse in-place
std::transform(vec.begin(), vec.end(), vec.begin(), [](int x){ return x * 2; });

```

```
// Numeric algorithms
#include <numeric>
auto sum = std::accumulate(vec.begin(), vec.end(), 0); // Sum elements
auto product = std::accumulate(vec.begin(), vec.end(), 1,
std::multiplies<int>()); // Product

// Partitioning
auto it = std::partition(vec.begin(), vec.end(), [](int x){ return x % 2 ==
0; });
// Elements before 'it' are even, after are odd
```

Strings and Streams

```
#include <string>
#include <sstream>
#include <iomanip>

// String operations
std::string str = "Hello, World!";
str.length(); // Get length
str.empty(); // Check if empty
str.substr(7, 5); // Extract substring
str.find("World"); // Find substring
str.replace(7, 5, "Universe"); // Replace substring

// String streams
std::stringstream ss;
ss << "Value: " << 42 << " Hex: " << std::hex << 42;
std::string result = ss.str();

// Formatted output
std::cout << std::setw(10) << std::left << "Name"
<< std::setw(5) << "Age" << std::endl;
std::cout << std::setw(10) << std::left << "John"
<< std::setw(5) << 25 << std::endl;

// File I/O
#include <fstream>
std::ofstream outfile("output.txt");
outfile << "Hello, File!" << std::endl;
outfile.close();

std::ifstream infile("input.txt");
std::string line;
while (std::getline(infile, line)) {
    std::cout << line << std::endl;
}
```

Utilities

```
#include <utility>
#include <tuple>
#include <optional>
#include <variant>

// Pairs and tuples
std::pair<int, std::string> p = {42, "answer"};
auto [num, text] = p; // Structured binding (C++17)

std::tuple<int, double, std::string> t = {1, 3.14, "pi"};
auto [i, d, s] = t; // Structured binding

// Optional (C++17)
std::optional<int> maybe_value = 42;
if (maybe_value.has_value()) {
    int value = maybe_value.value();
    int safe_value = maybe_value.value_or(0); // Default if no value
}

// Variant (C++17) - type-safe union
std::variant<int, double, std::string> v = 42;
v = 3.14;
v = "hello";

// Visit variant
std::visit([](auto&& arg) {
    std::cout << arg << std::endl;
}, v);
```

Modern C++ Features

C++11 Features

```
// Range-based for loops
for (auto& element : container) {
    // Use element
}

// nullptr instead of NULL
int* ptr = nullptr;

// Enum classes
enum class Color { Red, Green, Blue };
Color c = Color::Red;

// Static assertions
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");
```

```
// Type aliases
using StringVector = std::vector<std::string>;
```

Attributes (e.g., `[[nodiscard]]`)

- **Attributes** are standardized annotations written as `[[...]]` that give the compiler extra information (warnings, optimization hints, intent).
- They generally **do not change runtime behavior**, but they can cause warnings/errors and improve correctness.

```
// [[nodiscard]]: warn if return value is ignored (C++17, some compilers
support earlier)
[[nodiscard]] int parse_int(std::string_view s);

parse_int("123");           // warning: ignoring nodiscard return value
int x = parse_int("123");    // OK

// [[nodiscard]] on a type: warn when a temporary of this type is discarded
struct [[nodiscard]] Status {
    bool ok;
    std::string msg;
};
Status do_work();
do_work();                  // warning: result discarded

// [[maybe_unused]]: silence "unused variable/parameter" warnings (C++17)
void f([[maybe_unused]] int debug_flag) {}

// [[deprecated]]: warn when used (C++14)
[[deprecated("use new_api() instead")]] void old_api();

// [[fallthrough]]: document intentional switch fallthrough (C++17)
switch (n) {
    case 1:
        do_one();
        [[fallthrough]];
    case 2:
        do_two();
        break;
}

// [[likely]] / [[unlikely]]: branch likelihood hints (C++20)
if (condition) [[likely]] {
    /* fast path */
} else [[unlikely]] {
    /* slow path */
}
```

C++14 Features

```
// Generic lambdas
auto lambda = [](auto x, auto y) { return x + y; };

// Lambda capture with initializer
auto lambda = [value = compute_value()]() { return value; };

// Binary literals and digit separators
int binary = 0b1010'1010;
int large_number = 1'000'000;
```

C++17 Features

```
// Structured bindings
std::map<std::string, int> m = {"a", 1}, {"b", 2};
for (auto [key, value] : m) {
    std::cout << key << ": " << value << std::endl;
}

// if with initializer
if (auto it = container.find(key); it != container.end()) {
    // Use it
}

// constexpr if
template <typename T>
void process(T value) {
    if constexpr (std::is_integral_v<T>) {
        // Compile-time branch
    }
}

// std::optional and std::variant (mentioned above)
```

C++20 Features

```
// Concepts (mentioned above)

// Three-way comparison operator
struct Point {
    int x, y;
    auto operator<=>(const Point&) const = default; // Generates all
    comparisons
};

// Coroutines (simplified)
#include <coroutine>
struct Generator {
    struct promise_type {
```

```
        int current_value;
        // ... coroutine promise implementation
    };
};

// Ranges library
#include <ranges>
auto even_numbers = std::views::filter(numbers, [](int x) { return x % 2 == 0; });
for (int n : even_numbers) {
    // Process even numbers
}

// Modules (future feature)
import std; // Instead of #include <iostream>, etc.
```

Best Practices

Code Organization

```
// Header file structure
#pragma once

// System includes (alphabetical)
#include <memory>
#include <string>
#include <vector>

// Project includes
#include "myproject/config.hpp"

// Namespace
namespace myproject {

// Forward declarations
class Helper;

// Main class declaration
class MyClass {
public:
    // Public interface
private:
    // Private implementation
};

// Implementation
} // namespace myproject
```

Dependency Policy (learning NN from scratch)

If your goal is “I implement the neural network logic myself”, use dependencies only for **tooling and validation**, not for core math/ML.

- **Use (recommended)**
 - **Standard library:** containers (`std::vector`), utilities (`std::span` in C++20), random (`std::mt19937`), timing (`<chrono>`), IO.
 - **Testing** (pick one): **Catch2** or **GoogleTest**. (Does not implement NN logic; it verifies your code.)
 - **Dev tools** (not part of the library): `clang-format`, `clang-tidy`, sanitizers (ASan/UBSan).
 - Optional ergonomics: `fmt` (printing), `spdlog` (logging), Google Benchmark (benchmarks).
- **Avoid (they “solve” your project)**
 - Linear algebra / tensor libraries: Eigen, Armadillo, Blaze, xtensor
 - ML frameworks: libtorch, TensorFlow C++, ONNX Runtime
 - Autodiff libraries: anything that computes gradients for you
- **Exception (later, dev-only)**
 - It's okay to use a trusted math library **only in tests/benchmarks** as a reference to compare correctness/performance—keep your core library independent.

Const Correctness

```
class Example {
public:
    // Const member functions don't modify object
    int get_value() const { return value_; }

    // Overloads for const and non-const access
    const std::vector<int>& get_data() const { return data_; }
    std::vector<int>& get_data() { return data_; }

    // Parameters that aren't modified should be const references
    void process_data(const std::vector<int>& data) {
        // Don't modify data
    }

private:
    mutable int cache_ = 0; // Can be modified in const methods
    int value_ = 0;
    std::vector<int> data_;
};
```

Resource Management

```
// RAII class
class FileHandler {
public:
```

```

    explicit FileHandler(const std::string& filename)
        : file_(filename) {
        if (!file_.is_open()) {
            throw std::runtime_error("Cannot open file");
        }
    }

    ~FileHandler() {
        if (file_.is_open()) {
            file_.close();
        }
    }

    // Rule of five
    FileHandler(const FileHandler&) = delete;
    FileHandler& operator=(const FileHandler&) = delete;
    FileHandler(FileHandler&&) noexcept = default;
    FileHandler& operator=(FileHandler&&) noexcept = default;

    void write(const std::string& data) {
        file_ << data;
    }

private:
    std::ofstream file_;
};

// Usage - automatic cleanup
void process_file(const std::string& filename) {
    FileHandler handler(filename);
    handler.write("Hello, World!");
} // File automatically closed

```

Error Handling

```

#include <system_error>
#include <stdexcept>

// Custom exception hierarchy
class MyError : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};

// Error codes
enum class ErrorCode {
    Success = 0,
    FileNotFound,
    PermissionDenied
};

```

```

std::error_code make_error_code(ErrorCode e) noexcept;

// Function that may fail
std::expected<int, std::string> divide(int a, int b) { // C++23
    if (b == 0) {
        return std::unexpected("Division by zero");
    }
    return a / b;
}

// Usage with error handling
auto result = divide(10, 0);
if (!result) {
    std::cerr << "Error: " << result.error() << std::endl;
} else {
    std::cout << "Result: " << *result << std::endl;
}

```

Common Patterns

Singleton Pattern

```

class Singleton {
public:
    static Singleton& get_instance() {
        static Singleton instance; // Thread-safe in C++11
        return instance;
    }

    // Delete copy/move operations
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    Singleton(Singleton&&) = delete;
    Singleton& operator=(Singleton&&) = delete;

private:
    Singleton() = default;
    ~Singleton() = default;
};

```

Factory Pattern

```

// Abstract product
class Shape {
public:
    virtual ~Shape() = default;
    virtual void draw() const = 0;
};

```

```

// Concrete products
class Circle : public Shape {
public:
    void draw() const override { std::cout << "Drawing circle" <<
std::endl; }
};

class Square : public Shape {
public:
    void draw() const override { std::cout << "Drawing square" <<
std::endl; }
};

// Factory
class ShapeFactory {
public:
    enum class ShapeType { Circle, Square };

    static std::unique_ptr<Shape> create_shape(ShapeType type) {
        switch (type) {
            case ShapeType::Circle:
                return std::make_unique<Circle>();
            case ShapeType::Square:
                return std::make_unique<Square>();
            default:
                return nullptr;
        }
    }
};

```

Observer Pattern

```

// Observer interface
class Observer {
public:
    virtual ~Observer() = default;
    virtual void update(int value) = 0;
};

// Subject
class Subject {
public:
    void add_observer(std::weak_ptr<Observer> observer) {
        observers_.push_back(observer);
    }

    void set_value(int value) {
        value_ = value;
        notify_observers();
    }
}

```

```

private:
    void notify_observers() {
        for (auto& weak_obs : observers_) {
            if (auto obs = weak_obs.lock()) {
                obs->update(value_);
            }
        }
    }

    std::vector<std::weak_ptr<Observer>> observers_;
    int value_ = 0;
};

```

CRTP (Curiously Recurring Template Pattern)

```

template <typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
};

class Derived : public Base<Derived> {
public:
    void implementation() {
        std::cout << "Implementation in derived class" << std::endl;
    }
};

```

PIMPL (Pointer to Implementation) Idiom

```

// In header
class Widget {
public:
    Widget();
    ~Widget();
    void do_something();

private:
    class Impl; // Forward declaration
    std::unique_ptr<Impl> pimpl_;
};

// In implementation file
class Widget::Impl {
public:
    void do_something_internal() {

```

```

        // Implementation details
    }
private:
    // Private data members
    int data_;
    std::string name_;
};

Widget::Widget() : pimpl_(std::make_unique<Impl>()) {}
Widget::~Widget() = default; // Unique_ptr handles deletion

void Widget::do_something() {
    pimpl_->do_something_internal();
}

```

Error Handling

Exception Hierarchy

```

#include <stdexcept>
#include <system_error>

// Standard exceptions
class MyException : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};

class ValidationError : public MyException {
public:
    using MyException::MyException;
};

// Error codes with system_error
enum class NetworkError {
    ConnectionFailed = 1,
    Timeout,
    InvalidResponse
};

class NetworkException : public std::system_error {
public:
    NetworkException(NetworkError ec, const std::string& what)
        : std::system_error(std::make_error_code(ec), what) {}
};

```

Exception Safety Guarantees

```

class Container {
public:
    // No-throw guarantee
    ~Container() noexcept {
        // Cleanup must not throw
    }

    // Strong exception safety
    void add_element(T element) {
        // Make a copy first
        auto temp = data_;
        temp.push_back(std::move(element));

        // If this throws, we're still in original state
        validate_state(temp);

        // Commit the change
        data_ = std::move(temp);
    }

    // Basic exception safety
    void unsafe_operation() noexcept(false) {
        // May throw, but object remains in valid state
    }

private:
    std::vector<T> data_;
    void validate_state(const std::vector<T>& data) const; // May throw
};

```

Error Handling Strategies

```

// Return value with error code
std::pair<bool, std::string> process_data(const Data& data) {
    if (!validate(data)) {
        return {false, "Invalid data"};
    }

    if (!process(data)) {
        return {false, "Processing failed"};
    }

    return {true, "Success"};
}

// Out parameter with return code
bool process_data(const Data& input, Result& output, std::string& error) {
    if (!validate(input)) {
        error = "Invalid input";
        return false;
    }
}

```

```

    }

    output = do_processing(input);
    return true;
}

// Exception-based (modern C++)
void process_data(const Data& input) {
    validate_input(input); // Throws on invalid input
    auto result = do_processing(input); // May throw on failure
    save_result(result); // May throw on I/O error
}

```

Performance

Memory Alignment and Padding

```

// Structure padding awareness
struct BadLayout {
    char c; // 1 byte
    int i; // 4 bytes (3 bytes padding)
    char c2; // 1 byte (3 bytes padding)
}; // Total: 12 bytes

struct GoodLayout {
    int i; // 4 bytes
    char c; // 1 byte
    char c2; // 1 byte (2 bytes padding)
}; // Total: 8 bytes

// Alignment specification (C++11)
struct alignas(16) AlignedStruct {
    int data;
};

// Cache line alignment
struct CacheLine {
    alignas(64) std::atomic<int> flag;
    char padding[64 - sizeof(std::atomic<int>)];
};

```

Move Semantics and RVO

```

class Matrix {
public:
    // Copy constructor (expensive)
    Matrix(const Matrix& other) : data_(other.data_) {}
}

```

```

// Move constructor (cheap)
Matrix(Matrix&& other) noexcept
    : data_(std::move(other.data_)) {}

// Copy assignment
Matrix& operator=(const Matrix& other) {
    if (this != &other) {
        data_ = other.data_; // Copy
    }
    return *this;
}

// Move assignment
Matrix& operator=(Matrix&& other) noexcept {
    if (this != &other) {
        data_ = std::move(other.data_); // Move
    }
    return *this;
}

private:
    std::vector<double> data_;
};

// Function return value optimization (RVO)
Matrix create_matrix() {
    Matrix m;
    // Initialize m
    return m; // Compiler may elide copy/move
}

// Named return value optimization (NRVO)
Matrix create_matrix_named() {
    Matrix result;
    // Initialize result
    return result; // Often optimized away
}

```

Cache-Friendly Code

```

// Good: Sequential access
void process_good(const std::vector<int>& data) {
    for (size_t i = 0; i < data.size(); ++i) {
        data[i] *= 2;
    }
}

// Bad: Random access (cache thrashing)
void process_bad(const std::vector<int>& data, const std::vector<size_t>&
indices) {
    for (size_t idx : indices) {

```

```

        data[idx] *= 2; // Random memory access
    }
}

// Structure of arrays (SoA) vs Array of structures (AoS)
struct PointAoS {
    double x, y, z;
};

struct PointsSoA {
    std::vector<double> x, y, z;
};

// AoS: Good for single point operations
void process_point(PointAoS& p) {
    p.x *= 2;
    p.y *= 2;
    p.z *= 2;
}

// SoA: Good for bulk operations on coordinates
void process_points_soa(PointsSoA& points) {
    for (auto& x : points.x) x *= 2; // Cache-friendly
    for (auto& y : points.y) y *= 2;
    for (auto& z : points.z) z *= 2;
}

```

Compiler Optimizations

```

// Likely/unlikely branches (GCC)
#define likely(x)    __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)

if (likely(count > 0)) {
    // This branch is usually taken
}

// Restrict keyword (GCC)
void add_arrays(const double* restrict a, const double* restrict b, double*
restrict c, size_t n) {
    for (size_t i = 0; i < n; ++i) {
        c[i] = a[i] + b[i];
    }
}

// Inline hints
inline __attribute__((always_inline)) int fast_function(int x) {
    return x * x;
}

__attribute__((noinline)) void cold_function() {

```

```
// Rarely called function
}
```

Profiling and Optimization Tips

```
// Benchmarking
#include <chrono>

template <typename Func, typename... Args>
auto benchmark(Func&& func, Args&&... args) {
    auto start = std::chrono::high_resolution_clock::now();
    auto result = func(std::forward<Args>(args)...);
    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>
(end - start);
    std::cout << "Time: " << duration.count() << " microseconds" <<
std::endl;

    return result;
}

// Usage
auto result = benchmark(my_function, arg1, arg2);
```

Quick Reference

Common Type Traits

```
#include <type_traits>

// Primary type categories
std::is_void<T>
std::is_integral<T>
std::is_floating_point<T>
std::is_array<T>
std::is_pointer<T>
std::is_reference<T>
std::is_function<T>

// Composite type categories
std::is_arithmetic<T>    // integral or floating point
std::is_scalar<T>        // arithmetic, pointer, enum, nullptr_t
std::is_object<T>        // not function, not reference, not void

// Type properties
std::is_const<T>
std::is_volatile<T>
```

```
std::is_signed<T>
std::is_unsigned<T>

// Supported operations
std::is_constructible<T, Args...>
std::is_copy_constructible<T>
std::is_move_constructible<T>
std::is_assignable<T, U>
std::is_copy_assignable<T>
std::is_move_assignable<T>
std::is_destructible<T>

// Relationships
std::is_same<T, U>
std::is_base_of<Base, Derived>
std::is_convertible<From, To>
```

Common Algorithms Complexity

Algorithm	Best	Average	Worst	Notes
std::sort	O(n)	O(n log n)	O(n log n)	Intro-sort hybrid
std::stable_sort	O(n)	O(n log n)	O(n log² n)	Maintains relative order
std::find	O(1)	O(n)	O(n)	Linear search
std::lower_bound	O(1)	O(log n)	O(log n)	Binary search
std::reverse	-	O(n)	O(n)	In-place
std::unique	-	O(n)	O(n)	Removes consecutive duplicates

Container Selection Guide

Container	Access	Insert/Erase	Memory	Use Case
vector	O(1)	O(1) end, O(n) middle	Contiguous	Default choice
deque	O(1)	O(1) ends	Segmented	Queue operations
list	O(n)	O(1)	Node-based	Frequent insertions/deletions
set/map	O(log n)	O(log n)	Node-based	Ordered unique elements
unordered_set/map	O(1) avg	O(1) avg	Hash table	Fast lookup, unordered

Memory Model (C++11)

```
#include <atomic>
#include <thread>

// Atomic operations
std::atomic<int> counter = 0;
counter.fetch_add(1, std::memory_order_relaxed);

// Memory barriers
std::atomic_thread_fence(std::memory_order_acquire);
std::atomic_thread_fence(std::memory_order_release);

// Memory orders
std::memory_order_relaxed    // No ordering constraints
std::memory_order_consume    // Data dependency ordering
std::memory_order_acquire    // Acquire operation
std::memory_order_release    // Release operation
std::memory_order_acq_rel    // Both acquire and release
std::memory_order_seq_cst    // Sequential consistency (default)
```

This cheatsheet covers intermediate C++ concepts. For advanced topics like template metaprogramming, low-level optimizations, or specific library usage, consult the C++ standard or specialized resources.