C++ Practical Cheatsheet — Quick Reference for Everyday Programming

Goal: A compact, portable reference that covers the essential C++ knowledge you need to *write, read, debug,* and *reason about* code. It's derived from the concepts in your projects (class design, memory management, operators, I/O, error handling) but generalized so you can rely on it any time you forget how to do something in C++.

Layout & Export

- Designed for PDF: clear headings, monospace code blocks, and concise examples.
- Export tip: use a monospaced font for code and medium page margins.

Quick Syntax & Types (1-page glance)

- Fundamental types: `int`, `long`, `short`, `char`, `bool`, `float`, `double`.
- Fixed-width: `<cstdint>` — `int32_t`, `uint64_t`.
- Compound: pointers `T*`, references `T&`, arrays `T[]`, `std::array<T,N>`, `std::vector<T>`.
- `auto` for type deduction.
- `const` for immutability, `constexpr` for compile-time constants.

Example — declare & print:

```
int x = 42;
const double pi = 3.14159;
auto s = "hello"; // const char*
std::cout << x << " " << pi << " " << s << '
';
```

Explanation: `auto` deduces the initializer type. `const` prevents modification.

Control Flow

- `if`, `else if`, `else` — usual conditional branches.
- Loops: `for (init; cond; step)`, range-based `for (auto &v : container)`, `while`, `do/while`.
- Early return and `continue`/`break` for loop control.

Example — range loop:

```
std::vector<int> v = {1,2,3};
for (auto &val : v) val *= 2; // modifies values in-place
```

Explanation: Range-based loop is concise and safe; use `auto &` to modify elements.

Functions

- Declaration: `ReturnType name(Params)`. Put `const` after method when it doesn't modify `this`.
- Default arguments allowed. Overload functions by signature.
- Prefer `pass-by-reference` for large objects: `void f(const MyType& t)`.

Example — small function:

```
int add(int a, int b = 0) { return a + b; }
```

Explanation: `b` has a default value; `add(2)` returns 2.

Classes & Objects — Essentials

- Encapsulation: `private` members, `public` interface.
- Always initialize members (use constructor initializer lists).
- RAII: resources acquired in ctor, released in dtor.

Minimal class:

```
class Point {
  double x, y;
public:
  Point(double x_, double y_) : x(x_), y(y_) {}
  double length() const { return std::hypot(x,y); }
};
```

Explanation: initializer list (`: x(x_), y(y_)`) constructs members directly and is efficient.

Constructors, Destructor, and The Rule of Three/Five

- If your class manages raw resources (heap memory, file handle), implement:
- **Rule of Three:** copy constructor, copy assignment, destructor.
- **Rule of Five:** add move constructor and move assignment for efficient moves.

- Prefer to avoid raw `new` / `delete` when possible and use smart pointers (`std::unique_ptr` , `std::shared_ptr`).

Example — copy-and-swap idiom (exception-safe assignment):

```
void swap(MyClass &a, MyClass &b) { using std::swap; swap(a.ptr,b.ptr);
swap(a.n,b.n); }
MyClass& MyClass::operator=(MyClass other) { swap(*this, other); return *this; }
```

Explanation: Pass-by-value makes a copy; swapping makes assignment exception-safe and handles self-assignment.

Move Semantics (short)

- Move ctor: `MyClass(MyClass&& other) noexcept;` steal resources and leave `other` in a valid empty state.
- Move assignment: free existing resources, steal other's resources.

Example — pseudo:

```
MyClass(MyClass&& o) noexcept : ptr(o.ptr) { o.ptr = nullptr; }
MyClass& operator=(MyClass&& o) noexcept { if (this!=&o){ delete ptr; ptr =
o.ptr; o.ptr=nullptr;} return *this; }
```

Explanation: Moves avoid expensive deep copies.

Memory Management & Smart Pointers

- Prefer `std::unique_ptr<T>` for exclusive ownership; `std::shared_ptr<T>` for shared ownership.
- Avoid raw `new` / `delete` in modern C++ unless teaching/edge cases.

Example:

```
auto up = std::make_unique<MyClass>(args...);
std::vector<std::unique_ptr<MyClass>> pool;
pool.push_back(std::move(up));
```

Explanation: `std::make_unique` constructs safely and prevents leaks even if an exception occurs.

C-Strings vs `std::string`

- Use `std::string` for text unless you interoperate with C APIs.
- To get C-string: `mystdstring.c_str()`.

Example:

```
std::string a = "Hello";
const char* c = a.c_str();
```

Explanation: `std::string` manages memory; avoid manual `char*` handling.

Containers (STL) — essentials

- `std::vector<T>` — dynamic array (most used).
- `std::array<T,N>` — fixed-size array.
- `std::list`, `std::deque`, `std::map`, `std::unordered_map`, `std::set` — higher-level containers.
- Use `std::vector` + algorithms in most cases.

Example:

```
std::vector<int> v = {3,1,4,1};
std::sort(v.begin(), v.end());
```

Explanation: Algorithms work with iterators and many avoid manual loops.

Iterators & Algorithms

- Algorithms live in `<algorithm>`: `std::sort`, `std::find`, `std::accumulate`, `std::transform`.
- Prefer algorithms + lambdas to explicit loops when expressive and clear.

Example — transform:

```
std::vector<int> in = {1,2,3};
std::vector<int> out; out.resize(in.size());
std::transform(in.begin(), in.end(), out.begin(), [](int x){ return x*x; });
```

Explanation: `std::transform` applies the lambda to each element.

Lambdas (quick)

- Syntax: `[captures](params)->ret { body }`.
- Capture by value `[=]`, by reference `[&]`, or explicit `[a, &b]`.

Example:

```
int offset = 10;
auto f = [offset](int x){ return x + offset; };
std::cout << f(5); // 15
```

Explanation: Lambdas are lightweight function objects ideal for algorithms.

Templates — the basics

- Function templates and class templates provide type-generic code.

Example:

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

Explanation: The compiler generates concrete functions for used types.

Exceptions & Error Handling

- Use `try` / `catch` to handle exceptional conditions.
- Use `throw std::runtime_error("msg")` or specific exception types.
- Prefer `noexcept` where functions must not throw (e.g., destructors, move ctors often `noexcept`).

Example:

```
if (index < 0) throw std::out_of_range("index");

try { risky(); } catch (const std::exception &e) { std::cerr << e.what(); }
```

Explanation: Exceptions are for exceptional conditions; use return codes only when unavoidable.

Input / Output

- Streams: `std::cin`, `std::cout`, `std::cerr`.
- Use `std::getline` to read full lines; `operator>>` to read tokens.

Example:

```
std::string line;
std::getline(std::cin, line);
std::cout << "you typed: " << line << '
';
```

Explanation: `getline` reads including spaces until newline.

Header & Source Organization

- Put declarations in `.h` / `.hpp`, definitions in `.cpp`.
- Include guards or `#pragma once` in headers.

Header example:

```
// mylib.h
#pragma once
class MyClass { public: void f(); };
```

Source example:

```
// mylib.cpp
#include "mylib.h"
void MyClass::f() { /*...*/ }
```

Explanation: Keep headers minimal to reduce compile times.

Build & Debug Tips

- Compile with warnings enabled: `-Wall -Wextra -Werror` (GCC/Clang).
- Use sanitizer for runtime bugs: `-fsanitize=address,undefined`.
- Use `gdb` or IDE debugger; add `-g` for debug symbols.

Explanation: Warnings and sanitizers catch many common mistakes early.

Common Pitfalls & Checklist

- Forgetting `delete[]` when using raw `new[]` → prefer `std::vector` or `std::unique_ptr`.
- Off-by-one with NUL terminator for C-strings — allocate `len + 1`.
- Shallow copies of owning pointers — implement Rule of Three/Five.
- Not checking stream state after I/O.
- Ignoring `const` correctness (use `const` everywhere it applies).

Quick checklist before committing:

- Use RAII for resources.
- Prefer `std::string` over `char*`.
- Prefer `std::vector` over raw arrays.
- Add boundary checks when indexing.
- Enable compile warnings and address sanitizer.

Short mini-examples with explanations

1. **RAII file reader (safe resource):**

```
struct FileReader {
  std::ifstream f;
  FileReader(const std::string &path) : f(path) { if(!f) throw
std::runtime_error("open"); }
  std::string readLine(){ std::string s; std::getline(f, s); return s; }
};
```

Explanation: `std::ifstream` is managed by RAII inside the struct.

1. **Simple class with copy + move (skeleton):**

```
class Box { int n; int* data;
public:
  Box(int n): n(n), data(new int[n]){}
  ~Box(){ delete[] data; }
  Box(const Box& o): n(o.n), data(new int[o.n]){
std::copy(o.data,o.data+o.n,data); }
  Box(Box&& o) noexcept: n(o.n), data(o.data){ o.data=nullptr; }
  Box& operator=(Box o){ swap(*this,o); return *this; }
};
```

Explanation: Copy allocates, move steals pointer, assignment uses copy-and-swap.

1. **Algorithm + lambda:**

```
std::vector<int> a = {5,3,1,4};
std::sort(a.begin(), a.end(), [](int x,int y){ return x>y; }); // descending
```

Explanation: Lambda passed to `std::sort` defines custom ordering.

Final Practical Tips

- When in doubt: prefer standard library solutions (`<algorithm>`, containers, smart pointers).
- Keep functions small and single-purpose.
- Write tests or small run examples for tricky ownership logic.
