C++ Cheatsheet

Basic Structure and Compilation Environment

A C++ program generally consists of `#include` preprocessor directives, a `main` function, and statements containing execution logic.

```
#include <iostream> // Include input/output stream library

// Main function, the entry point of the program
int main() {
    std::cout << "Hello, World!" << std::endl; // Send message to standard output stream
    return 0; // Program terminated successfully
}
```

- Compilation and Execution (G++):

```
g++ -o hello hello.cpp # Compile and generate executable
./hello                # Run the generated executable
```

Data Types, Variables, and Operators

- Basic Data Types: `int` (integers), `double` (floating-point), `char` (character), `bool` (boolean), `float` (floating-point).
- `std::string`: Standard class type for string processing.
- `const`: Used to declare constants. `const double PI = 3.14;`
- `auto`: The compiler automatically deduces the type of the variable. `auto x = 5;`
- Main Operators: Arithmetic (`+`, `-`, `*`, `/`, `%`), Relational (`==`, `!=`, `<`, `>`), Logical (`&&`, `||`, `!`).

Control Statements (Conditionals and Loops)

- `if-else` Conditional:

```
if (condition) {
    // Executed when condition is true
} else if (another_condition) {
    // Executed when another condition is true
} else {
    // Executed when all conditions are false
}
```

- `switch` Statement:

```
switch (variable) {
    case 1:
        // Executed when value is 1
        break;
    case 2:
        // Executed when value is 2
        break;
    default:
        // Executed when no values match
}
```

- Loops:
  - `for` Loop: `for (int i = 0; i < 5; ++i) { ... }`
  - Range-based `for` Loop (C++11+:

```
std::vector<int> v = {1, 2, 3};
for (int item : v) {
    std::cout << item << std::endl;
}
```

  - `while` Loop: `while (condition) { ... }`

Functions and Overloading

- Function Definition:

```
return_type function_name(parameter_type parameter_name) {
    // Function logic
    return return_value;
}
```

- Function Overloading: Multiple functions can be defined with the same name but different parameter types or counts.
- Default Arguments: `void print(int value, int base=10);`

Pointers and References

- Pointer (`*`): A variable that stores the memory address of another variable.

```
int var = 10;
int* ptr = &var; // Store address of var into ptr
std::cout << *ptr; // Output the actual value (10) pointed to by ptr (dereferencing)
```

- Reference (`&`): Another name (alias) for an existing variable. Must be initialized upon declaration.

```
int var = 10;
int& ref = var; // ref becomes an alias for var
ref = 20; // value of var also changes to 20
```

Object-Oriented Programming (Classes and Objects)

- Class Definition:

```
class MyClass {
public: // Accessible from outside
    // Constructor
    MyClass(int val) : my_field(val) {}

    // Member function (Method)
    void myMethod() {
        std::cout << "Field is " << my_field << std::endl;
    }

private: // Accessible only within the class
    // Member variable (Field)
    int my_field;
};
```

- Object Creation: `MyClass obj(10);`
- Member Access: `obj.myMethod();`
- Inheritance: `class Derived : public Base { ... };`

Core STL (Standard Template Library) Containers

`std::vector`

The most commonly used dynamic array container.

```
#include <vector>
std::vector<int> vec;
vec.push_back(10); // Add element
vec.push_back(20);
int first = vec[0]; // Access via index
vec.size(); // Check number of stored elements
```

`std::map`

An associative container that maintains key-value pairs in a sorted state.

```
#include <map>
std::map<std::string, int> ages;
ages["Alice"] = 30;
ages["Bob"] = 25;
```

`std::string`

A class providing powerful features for string manipulation.

```
#include <string>
std::string s = "Hello";
s += ", World!";
s.substr(0, 5); // Extract 5 characters starting from index 0 ("Hello")
s.find("World"); // Search for the position of a specific string
```

Algorithm Usage

The `<algorithm>` header provides various functions for container processing.

- `std::sort(vec.begin(), vec.end());`
- `std::find(vec.begin(), vec.end(), value);`
- `std::for_each(vec.begin(), vec.end(), my_func);`

Data Input/Output (I/O)

Console I/O (`iostream`)

```
#include <iostream>
int x;
std::cout << "Enter a number: ";
std::cin >> x; // Receive value from standard input
std::cerr << "Error message output" << std::endl; // Standard error stream
```

File I/O (`fstream`)

```
#include <fstream>
#include <string>
```

```cpp
// Writing data to a file
std::ofstream outFile("data.txt");
outFile << "Hello, file!" << std::endl;
outFile.close();

// Reading data from a file
std::ifstream inFile("data.txt");
std::string line;
while (std::getline(inFile, line)) {
    std::cout << line << std::endl;
}
inFile.close();
```

Detailed Analysis of STL Containers

Sequence Containers

```cpp
#include <vector>
#include <deque>
#include <list>
#include <array>

// std::vector: Dynamic array
std::vector<int> vec = {1, 2, 3, 4, 5};
vec.reserve(100);  // Pre-allocate memory capacity
vec.shrink_to_fit();  // Return unused capacity
vec.emplace_back(6);  // Construct object directly from arguments (better performance)

// std::deque: Double-ended queue (fast insertion/deletion at both front/back)
std::deque<int> dq = {1, 2, 3};
dq.push_front(0);
dq.push_back(4);

// std::list: Doubly linked list
std::list<int> lst = {1, 2, 3, 4, 5};
lst.sort();  // Use list-specific sort member function

// std::array: Object version of C-style array (fixed size)
std::array<int, 5> arr = {1, 2, 3, 4, 5};
arr.fill(0);  // Fill all elements with a specific value
```

Associative Containers

```cpp
#include <map>
#include <set>
#include <unordered_map>
#include <unordered_set>

// std::map: Sorted key-value storage
std::map<std::string, int> scores;
scores.emplace("Bob", 87);
auto it = scores.find("Alice");
if (it != scores.end()) std::cout << it->second << std::endl;

// std::set: Sorted set of unique elements
std::set<int> numbers = {3, 1, 4, 1, 5};
auto [iter, inserted] = numbers.insert(3); // C++17 structured bindings

// std::unordered_map: Hash-based fast search
std::unordered_map<std::string, int> cache;
cache.reserve(1000); // Pre-allocate hash table size
```

Container Adapters

```cpp
#include <stack>
#include <queue>
#include <priority_queue>

// std::stack: LIFO structure
std::stack<int> stk;
stk.push(1);
int top = stk.top();
stk.pop();

// std::queue: FIFO structure
std::queue<int> q;
q.push(1);
int front = q.front();
q.pop();

// std::priority_queue: Extraction based on priority (max-heap by default)
std::priority_queue<int> pq;
pq.push(3);
int max_val = pq.top(); // Extract the largest value
```

STL Algorithm Usage

Searching and Sorting

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

std::vector<int> vec = {5, 2, 8, 1, 9, 3};

// Use sorting library
std::sort(vec.begin(), vec.end()); // Ascending
std::sort(vec.begin(), vec.end(), std::greater<int>()); // Descending

// Binary search (only on sorted range)
bool found = std::binary_search(vec.begin(), vec.end(), 5);
auto it = std::lower_bound(vec.begin(), vec.end(), 5); // Lower bound search

// Find element with specific condition
auto it_cond = std::find_if(vec.begin(), vec.end(), [](int x) { return x > 5; });
```

Data Modification and Manipulation

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

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

// Batch transform
std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) { return x * 2; });

// Conditional copy
std::vector<int> evens;
std::copy_if(vec.begin(), vec.end(), std::back_inserter(evens),
             [](int x) { return x % 2 == 0; });

// Remove specific elements (Erase-Remove idiom)
vec.erase(std::remove(vec.begin(), vec.end(), 3), vec.end());
```

Numeric Data Processing

```cpp
#include <numeric>
#include <vector>

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

// Accumulate (Sum)
int sum = std::accumulate(vec.begin(), vec.end(), 0);

// Adjacent difference calculation
std::vector<int> diffs(vec.size());
std::adjacent_difference(vec.begin(), vec.end(), diffs.begin());
```

Advanced Memory Management and Smart Pointers

Using Smart Pointers

```cpp
#include <memory>

// std::unique_ptr: Pointer with unique ownership
std::unique_ptr<int> uptr = std::make_unique<int>(42);
std::unique_ptr<int> uptr2 = std::move(uptr); // Only ownership transfer is possible

// std::shared_ptr: Pointer with shared ownership based on reference counting
std::shared_ptr<int> sptr = std::make_shared<int>(42);
std::shared_ptr<int> sptr2 = sptr; // Increase reference count

// std::weak_ptr: Weak reference to solve shared_ptr circular reference issues
std::weak_ptr<int> wptr = sptr;
if (auto locked = wptr.lock()) {
    // Check validity before use
```

```
    std::cout << *locked << std::endl;
}
```

Move Semantics

```
#include <string>
#include <vector>

class Resource {
    int* buffer;
public:
    // Move constructor: Efficiently construct by taking resources from existing object
    Resource(Resource&& other) noexcept : buffer(other.buffer) {
        other.buffer = nullptr;
    }

    // Move assignment operator
    Resource& operator=(Resource&& other) noexcept {
        if (this != &other) {
            delete[] buffer;
            buffer = other.buffer;
            other.buffer = nullptr;
        }
        return *this;
    }

    ~Resource() { delete[] buffer; }
};

Resource r1;
Resource r2 = std::move(r1); // Perform explicit resource move
```

Advanced Template Metaprogramming (TMP)

Class Template Specialization

```
// General template definition
template<typename T>
class Container {
public:
    void info() { std::cout << "General Type" << std::endl; }
};

// Full specialization for specific type
template<>
class Container<std::string> {
public:
    void info() { std::cout << "String Type" << std::endl; }
};

// Partial specialization for pointer types
template<typename T>
class Container<T*> {
public:
    void info() { std::cout << "Pointer Type" << std::endl; }
};
```

Variadic Templates

```
#include <iostream>

// Variadic argument processing using C++17 Fold expressions
template<typename... Args>
auto sumAll(Args... args) {
    return (args + ...); // Parameter pack expansion
}

// Recursive template processing
template<typename T>
void printAll(T arg) { std::cout << arg << std::endl; }

template<typename T, typename... Args>
void printAll(T first, Args... args) {
    std::cout << first << ", ";
    printAll(args...);
}
```

Multithreading and Concurrency Programming

Thread Creation and Management

```
#include <thread>
#include <mutex>
#include <future>

void task(int id) { std::cout << "Thread " << id << " is running" << std::endl; }

int main() {
    std::thread t1(task, 1);
    t1.join(); // Wait for thread completion
}
```

Synchronization Mechanisms

```
#include <mutex>
#include <shared_mutex>

class SafeCounter {
    mutable std::shared_mutex mtx;
    int val = 0;
public:
    int get() const {
        std::shared_lock lock(mtx); // Shared lock for read-only
        return val;
    }
    void add() {
        std::unique_lock lock(mtx); // Exclusive lock for write
        val++;
    }
};
```

Asynchronous Result Processing (Future/Promise)

```
#include <future>

// Asynchronous function execution via async
auto fut = std::async(std::launch::async, []() {
    return 42;
});

int result = fut.get(); // Blocking until asynchronous result is ready
```

Code Optimization and Performance Enhancement Techniques

Compiler Optimization Hints

```
// inline: Recommend removing function call overhead
inline int add(int a, int b) { return a + b; }

// C++20 [[likely]] / [[unlikely]]: Optimize branch prediction
if (cond) [[likely]] {
    // When likely true
} else [[unlikely]] {
    // When likely false
}
```

Memory Layout and Cache Efficiency

```
// Row-major traversal recommended for cache hit rate
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < M; ++j) {
        process(matrix[i][j]); // Contiguous memory access
    }
}
```

Performance Benchmarking (using Chrono)

```
#include <chrono>

auto start = std::chrono::high_resolution_clock::now();
// Code to measure
auto end = std::chrono::high_resolution_clock::now();

std::chrono::duration<double, std::milli> ms = end - start;
std::cout << "Execution time: " << ms.count() << "ms" << std::endl;
```

Google Style Guide

Naming Conventions

- File Names: `my_useful_class.cc` (lowercase, underscores)
- Types (Classes/Structs): `MyExcitingClass` (UpperCamelCase)
- Variables (Local/Data Members): `my_local_variable`, `my_member_variable_` (lowercase, members have trailing underscore)
- Constants/Enums: `kDaysInAWeek` (CamelCase starting with k)
- Functions: `MyExcitingFunction()` (UpperCamelCase)
- Namespaces: `my_namespace` (lowercase)
- Macros: `MY_MACRO_THAT_SCARES_SMALL_CHILDREN`

Header Files

- Self-contained: Headers should be compilable by themselves.
- #define Guard: Compliance with `#ifndef PROJECT_PATH_FILE_H_` format.
- Include Order: Related header -> C systems -> C++ standard -> other libraries -> project headers.
- Inline Functions: Define in header only if short (approx. 10 lines or fewer).

Classes

- Constructors: No virtual function calls in constructors. Use `explicit` for single-argument constructors.
- Struct vs Class: Use `struct` for simple data transfer, `class` for logic.
- Inheritance: Use `public` inheritance only. Avoid multiple inheritance. Specify `override` or `final`.
- Access Control: Keep data members `private`.

Programming Practices

- Smart Pointers: Clarify ownership models (prefer `std::unique_ptr`).
- Namespaces: Place code inside namespaces. `using namespace std;` is forbidden.
- Output Parameters: Prefer return values for results. Consider pointers over references for output parameters.
- C++20: Aim to use the latest standards (C++20) but avoid non-standard extensions.
- Exceptions: Avoid internally at Google, use as appropriate in open source projects.