C++ Cheatsheet for Experienced Developers

Memory Management & RAII

Smart Pointers

```cpp
// Unique ownership
auto p1 = std::make_unique<T>(args);
auto p2 = std::unique_ptr<T>(new T(args));

// Shared ownership
auto sp = std::make_shared<T>(args);
auto wp = std::weak_ptr<T>(sp); // Non-owning

// Using weak_ptr
if (auto locked = wp.lock()) {
    // Use locked as shared_ptr
}

// Custom deleters
auto del = [](T* p) {
    // Custom cleanup logic
    delete p;
};
std::unique_ptr<T, decltype(del)> p(new T(), del);
```

Resource Management

```cpp
// RAII pattern
class Resource {
private:
    FILE* handle;
public:
    Resource(const char* filename)
        : handle(fopen(filename, "r")) {
        if (!handle)
            throw std::runtime_error("File error");
    }
    ~Resource() {
        if (handle) fclose(handle);
    }
    // Prevent copying
    Resource(const Resource&) = delete;
    Resource& operator=(const Resource&) = delete;

    // Allow moving
    Resource(Resource&& other) noexcept
        : handle(other.handle) {
        other.handle = nullptr;
    }
    Resource& operator=(Resource&& other) noexcept {
        if (this != &other) {
            if (handle) fclose(handle);
            handle = other.handle;
            other.handle = nullptr;
        }
        return *this;
    }
};
```

Templates & Metaprogramming

SFINAE & Type Traits

```cpp
// SFINAE pattern
template <typename T,
          typename = std::enable_if_t<
              std::is_integral_v<T>>>
void func(T value) { /*...*/ }

// Concepts (C++20)
template <std::integral T>
void func(T value) { /*...*/ }

// Type traits
static_assert(std::is_same_v<T, U>);
if constexpr (std::is_arithmetic_v<T>) {
    // For arithmetic types
} else {
    // For other types
}
```

Variadic Templates

```cpp
// Recursive variadic templates
template<typename T>
void print(const T& t) {
    std::cout << t << '\n';
}

template<typename T, typename... Args>
void print(const T& t, const Args&... args) {
    std::cout << t << ' ';
    print(args...);
}

// Fold expressions (C++17)
template<typename... Args>
auto sum(Args... args) {
    return (args + ...); // Unary right fold
}
```

Modern C++ Features

Move Semantics

```cpp
// Perfect forwarding
template<typename T, typename... Args>
std::unique_ptr<T> make(Args&&... args) {
    return std::unique_ptr<T>(
        new T(std::forward<Args>(args)...));
}

// Move operations
class Widget {
    std::vector<int> data;
public:
    // Move constructor
    Widget(Widget&& other) noexcept
        : data(std::move(other.data)) {}

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

Lambdas & Closures

```cpp
// Basic lambda
auto add = [](int a, int b) { return a + b; };

// Lambda with capture
int multiplier = 5;
auto times = [multiplier](int x) {
    return x * multiplier;
};

// Mutable lambda
auto counter = [count = 0]() mutable {
    return ++count;
};

// Generic lambda (C++14)
auto generic = [](auto x, auto y) { return x + y; };

// Lambda with explicit return type
auto divide = [](double a, double b) -> double {
    return a / b;
};

// Capture with initialization (C++14)
auto func = [ptr = std::make_unique<int>(42)]() {
    return *ptr;
};
```

Structured Bindings (C++17)

```cpp
// With array
int arr[3] = {1, 2, 3};
auto [a, b, c] = arr;

// With tuple
auto [name, age] = std::make_tuple("Alice", 30);

// With pair (common in map operations)
auto [iter, success] = myMap.insert({key, value});

// With struct
struct Point { int x, y; };
Point p{1, 2};
auto [px, py] = p;
```

Range-Based For Loop

```cpp
// Basic usage
for (const auto& item : container) { /*...*/ }

// With structured bindings (C++17)
for (const auto& [key, value] : map) { /*...*/ }

// With initialization (C++20)
for (std::vector v{1,2,3}; auto& elem : v) {
    /*...*/
}
```

STL & Algorithms

Algorithm Patterns

```cpp
// Finding elements
auto it = std::find_if(begin(c), end(c),
    [](const auto& x) { return x > 10; });

// Transforming elements
std::transform(begin(src), end(src), begin(dest),
    [](const auto& x) { return x * 2; });

// Functional operations
auto sum = std::accumulate(begin(c), end(c), 0);
auto product = std::accumulate(begin(c), end(c), 1,
    std::multiplies<>());

// Removing elements (erase-remove idiom)
// Note: remove_if doesn't change container size
auto new_end = std::remove_if(begin(v), end(v),
    [](int x) { return x % 2 == 0; });
v.erase(new_end, end(v));

// Sorting with custom comparator
std::sort(begin(v), end(v),
    [](const auto& a, const auto& b) {
        return a.priority > b.priority;
    });
```

Containers & Iterators

```cpp
// Container adaptors
std::priority_queue<T, std::vector<T>, Compare> pq;
std::stack<T> stack;
std::queue<T> queue;

// Using iterators
auto it = container.begin();
std::advance(it, 5);    // Move forward by 5
auto dist = std::distance(container.begin(), it);

// Iterator adapters
auto rbegin = container.rbegin();  // Reverse
std::back_inserter(container);     // Insert at end

// C++20 ranges
auto view = std::ranges::views::filter(
    container, [](const auto& x) {
        return x > 0;
    });
```

Concurrency

```cpp
// Thread creation
std::thread t([](int x) {
    /* thread work */
}, 42);
t.join();  // Wait for thread completion

// Mutex and lock
std::mutex mtx;
{
    std::lock_guard<std::mutex> lock(mtx);
    // Critical section
}

// More flexible locking
std::unique_lock<std::mutex> lock(mtx,
    std::defer_lock);  // Don't lock yet
// ... some code ...
lock.lock();  // Now lock

// Condition variable
std::condition_variable cv;
cv.wait(lock, []{ return ready; });
cv.notify_one();  // Wake one waiting thread
```

Classes & OOP

Special Member Functions

```cpp
class MyClass {
public:
    // Constructor
    MyClass() = default;

    // Custom constructor (explicit prevents implicit)
    explicit MyClass(int val) : value(val) {}

    // Destructor
    ~MyClass() = default;

    // Copy operations
    MyClass(const MyClass&) = delete;  // No copying
    MyClass& operator=(const MyClass&) = delete;

    // Move operations
    MyClass(MyClass&&) noexcept = default;
    MyClass& operator=(MyClass&&) noexcept = default;

private:
    int value = 0;
};
```

Inheritance Patterns

```cpp
// Abstract base class
class Shape {
public:
    virtual ~Shape() = default;  // Virtual destructor
    virtual double area() const = 0;  // Pure virtual
    virtual void draw() const {
        // Default implementation
    }
};

// CRTP pattern (static polymorphism)
template<typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }
protected:
    // Default implementation if needed
    void implementation() { /* default */ }
};
```

Type Deduction

```cpp
// auto vs decltype
auto x = expr;  // Type from initialization
decltype(expr) y;  // Type of expr (no evaluation)

// Function return type deduction (C++14)
auto func() { return value; }

// Trailing return type
auto func() -> decltype(expr) { return expr; }

// decltype(auto) (C++14)
decltype(auto) f() { return (x); }  // Returns ref

// Forwarding references
template<typename T>
void f(T&& param);  // Universal/forwarding ref
```

Common Patterns & Idioms

```cpp
// Const-correctness
void func(const T& arg) const noexcept;

// Rule of zero/five
// Either:
// 1. Define no special members (zero)
// or
// 2. Define all five: destructor, copy ctor,
//    copy assign, move ctor, move assign

// Prevent narrowing conversions
T x{narrower};  // Error if narrowing occurs

// Make non-copyable but movable
X(const X&) = delete;
X& operator=(const X&) = delete;
X(X&&) = default;
X& operator=(X&&) = default;

// Virtual inheritance (diamond problem)
class B : public virtual A { /*...*/ };

// const member function overloading
T& operator[](size_t idx);
const T& operator[](size_t idx) const;
```