C++ Programming Guide: 10 Case Studies with Illustrated Solutions

July 2025

Contents

1 Introduction

This guide presents 10 case studies demonstrating key C++ programming concepts, complete with detailed solutions and code examples. Each case study addresses a practical problem, illustrating fundamental and advanced C++ techniques. The guide is designed for students, developers, and professionals seeking to deepen their C++ expertise.

2 Case Studies

2.1 Case Study 1: Stack Implementation

Problem: Design a stack data structure supporting push, pop, and peek operations with dynamic resizing.
Solution: Use a dynamic array with a doubling strategy for resizing. Include error handling for stack overflow and underflow.

```cpp
#include <stdexcept>

class Stack {
private:
    int* arr;
    int capacity;
    int top;
public:
    Stack(int size = 10) : capacity(size), top(-1) {
        arr = new int[capacity];
    }
    ~Stack() { delete[] arr; }
    void push(int value) {
        if (top + 1 == capacity) {
            int* temp = new int[capacity * 2];
            for (int i = 0; i < capacity; ++i) temp[i] = arr[i];
            delete[] arr;
            arr = temp;
            capacity *= 2;
        }
        arr[++top] = value;
    }
    int pop() {
        if (top < 0) throw std::runtime_error("Stack underflow");
        return arr[top--];
    }
    int peek() const {
        if (top < 0) throw std::runtime_error("Stack empty");
        return arr[top];
    }
};
```

Explanation: The stack uses a dynamic array, resizing by doubling when full. Exceptions handle underflow and empty stack conditions.

2.2 Case Study 2: Binary Search Tree (BST)

Problem: Implement a BST with insert, search, and delete operations.
Solution: Use a recursive approach for tree operations, ensuring balanced operations

for efficiency.

```cpp
#include <memory>

struct Node {
    int data;
    std::unique_ptr<Node> left, right;
    Node(int val) : data(val) {}
};

class BST {
private:
    std::unique_ptr<Node> root;
    std::unique_ptr<Node> insert(std::unique_ptr<Node> node, int val) {
        if (!node) return std::make_unique<Node>(val);
        if (val < node->data) node->left = insert(std::move(node->left), val);
        else node->right = insert(std::move(node->right), val);
        return node;
    }
public:
    void insert(int val) { root = insert(std::move(root), val); }
};
```

Explanation: Smart pointers (`unique_ptr`) ensure memory safety. Recursive insertion maintains BST properties.

2.3 Case Study 3: Matrix Multiplication

Problem: Multiply two matrices efficiently.
Solution: Use nested loops with error checking for dimension compatibility.

```cpp
#include <vector>

std::vector<std::vector<int>> multiply(const std::vector<std::vector<int>>& A,
                                       const std::vector<std::vector<int>>& B) {
    if (A[0].size() != B.size()) throw std::invalid_argument("Invalid dimensions");
    std::vector<std::vector<int>> result(A.size(), std::vector<int>(B[0].size(), 0));
    for (size_t i = 0; i < A.size(); ++i)
        for (size_t j = 0; j < B[0].size(); ++j)
            for (size_t k = 0; k < A[0].size(); ++k)
                result[i][j] += A[i][k] * B[k][j];
    return result;
}
```

Explanation: The solution checks for valid matrix dimensions and uses vectors for dynamic sizing.

2.4 Case Study 4: String Pattern Matching (KMP Algorithm)

Problem: Implement Knuth-Morris-Pratt (KMP) for efficient string matching.
Solution: Compute a prefix table to avoid redundant comparisons.

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

std::vector<int> computePrefix(const std::string& pattern) {
    std::vector<int> pi(pattern.size(), 0);
    for (size_t i = 1, j = 0; i < pattern.size(); ++i) {
        while (j > 0 && pattern[i] != pattern[j]) j = pi[j-1];
        if (pattern[i] == pattern[j]) ++j;
        pi[i] = j;
    }
    return pi;
}

int kmpSearch(const std::string& text, const std::string& pattern) {
    std::vector<int> pi = computePrefix(pattern);
    for (size_t i = 0, j = 0; i < text.size(); ++i) {
        while (j > 0 && text[i] != pattern[j]) j = pi[j-1];
        if (text[i] == pattern[j]) ++j;
        if (j == pattern.size()) return i - j + 1;
    }
    return -1;
}
```

Explanation: KMP uses a prefix table to skip unnecessary comparisons, improving efficiency over naive matching.

2.5 Case Study 5: Thread-Safe Singleton

Problem: Implement a thread-safe singleton class.
Solution: Use double-checked locking with a mutex.

```cpp
#include <mutex>

class Singleton {
private:
    static Singleton* instance;
    static std::mutex mtx;
    Singleton() {}
public:
    static Singleton* getInstance() {
        if (instance == nullptr) {
            std::lock_guard<std::mutex> lock(mtx);
            if (instance == nullptr) instance = new Singleton();
        }
        return instance;
    }
};
Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

Explanation: Double-checked locking ensures thread safety while minimizing mutex overhead.

2.6 Case Study 6: Graph Traversal (DFS)

Problem: Implement depth-first search (DFS) for a graph.
Solution: Use recursion with an adjacency list representation.

```cpp
#include <vector>
#include <unordered_set>

class Graph {
private:
    std::vector<std::vector<int>> adj;
    void dfs(int v, std::unordered_set<int>& visited) {
        visited.insert(v);
        for (int u : adj[v])
            if (visited.find(u) == visited.end())
                dfs(u, visited);
    }
public:
    Graph(int vertices) : adj(vertices) {}
    void addEdge(int u, int v) { adj[u].push_back(v); }
    void DFS(int start) {
        std::unordered_set<int> visited;
        dfs(start, visited);
    }
};
```

Explanation: DFS explores nodes recursively, using a set to track visited nodes.

2.7 Case Study 7: Priority Queue

Problem: Implement a priority queue using a binary heap.
Solution: Use a max-heap for priority-based operations.

```cpp
#include <vector>

class PriorityQueue {
private:
    std::vector<int> heap;
    void heapifyUp(int index) {
        while (index > 0 && heap[(index-1)/2] < heap[index]) {
            std::swap(heap[index], heap[(index-1)/2]);
            index = (index-1)/2;
        }
    }
public:
    void push(int val) {
        heap.push_back(val);
        heapifyUp(heap.size()-1);
    }
    int top() const {
        if (heap.empty()) throw std::runtime_error("Queue empty");
        return heap[0];
    }
};
```

Explanation: The max-heap ensures the highest priority element is always at the root.

2.8 Case Study 8: File Parser

Problem: Parse a CSV file into a structured format (simulated with string input).
Solution: Use string streams for parsing.

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

std::vector<std::vector<std::string>> parseCSV(const std::string& input) {
    std::vector<std::vector<std::string>> result;
    std::stringstream ss(input);
    std::string line;
    while (std::getline(ss, line)) {
        std::vector<std::string> row;
        std::stringstream ls(line);
        std::string cell;
        while (std::getline(ls, cell, ',')) row.push_back(cell);
        result.push_back(row);
    }
    return result;
}
```

Explanation: String streams simplify parsing by handling delimiters and line breaks.

2.9 Case Study 9: LRU Cache

Problem: Implement a Least Recently Used (LRU) cache.
Solution: Use a combination of a hash map and doubly linked list.

```
#include <unordered_map>
#include <list>

class LRUCache {
private:
    int capacity;
    std::list<std::pair<int, int>> dll;
    std::unordered_map<int, std::list<std::pair<int, int>>::iterator> map;
public:
    LRUCache(int cap) : capacity(cap) {}
    int get(int key) {
        if (map.find(key) == map.end()) return -1;
        dll.splice(dll.begin(), dll, map[key]);
        return map[key]->second;
    }
    void put(int key, int value) {
        if (map.find(key) != map.end()) {
            dll.splice(dll.begin(), dll, map[key]);
            map[key]->second = value;
            return;
        }
        if (dll.size() >= capacity) {
            map.erase(dll.back().first);
            dll.pop_back();
        }
        dll.emplace_front(key, value);
        map[key] = dll.begin();
    }
```

```
};
```

Explanation: The doubly linked list maintains order, and the hash map provides O(1) access.

2.10 Case Study 10: Merge Sort

Problem: Implement merge sort for sorting an array.
Solution: Use divide-and-conquer with a merge function.

```
#include <vector>

void merge(std::vector<int>& arr, int left, int mid, int right) {
    std::vector<int> temp(right - left + 1);
    int i = left, j = mid + 1, k = 0;
    while (i <= mid && j <= right)
        temp[k++] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];
    for (i = 0; i < k; ++i) arr[left + i] = temp[i];
}

void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Explanation: Merge sort divides the array recursively and merges sorted halves.