# Hash Tables in C++: A Refresher for C Programmers with AS/CR Style

Your Name

## Overview and Motivation

A **hash table** is a data structure that lets you store and retrieve key-value pairs in nearly constant time on average. If you've ever used a dictionary or an associative array, you've used something like a hash table.

### Why Are Hash Tables Important?

- Fast lookup: $O(1)$ average case for insertion, deletion, and search.
- Flexible: Can store almost anything if you define how to hash it.
- Foundation for many abstractions: caches, symbol tables, compiler internals, network routing tables, etc.
- Crucial for performance-oriented design—particularly important when building data-centric systems or modeling abstract relationships efficiently (e.g., graph traversal, sparse matrices).

## Intuitive Analogy (CR Friendly)

Imagine you're in a library where books are stored not by order, but by hashing the title into a number that tells you exactly which locker to open. If the locker is empty, place the book there. If there's a collision (another book already in the locker), you have to handle it by some clever strategy.

### Key Insight: Hash Function + Array = Superpower

Hash tables are made of two simple parts:

1. A **hash function** $h(key)$: maps the key to an array index.
2. An **array** (a contiguous block of memory): stores key-value pairs at these indices.

The power of a hash table lies in how cleverly you handle collisions and design your hash function.

## Struct Refresher (AS Foundation)

In C, a `struct` is a way to group variables (of possibly different types) under one name.

```
struct Book {
    int id;
    const char* title;
    float price;
};
```

Listing 1: Basic struct definition in C-style

In C++, this becomes richer with constructors and methods.

```cpp
struct Book {
    int id;
    std::string title;
    float price;

    // Constructor
    Book(int i, std::string t, float p) : id(i), title(t), price(p) {}
};
```

Listing 2: C++ struct with constructor

You'll see structs used in hash table nodes (e.g., key-value pairs).

## Pointers Refresher (CR-Style Insight)

Pointers let you reference memory locations directly, which is necessary when managing dynamic memory (especially with collision chains or rehashing).

```cpp
int a = 10;
int* ptr = &a;
std::cout << *ptr; // Outputs 10
```

Listing 3: Simple pointer usage

**Important**: In hash tables, we often use pointers to link items together (like in linked lists), or to allocate storage dynamically for flexible growth.

## Hash Table Components (AS Map)

- **Key**: What you're indexing by (e.g., name, number).
- **Value**: What you're storing.
- **Bucket**: The array slot where items get stored.
- **Hash function**: Converts the key to an index.
- **Collision resolution strategy**: What to do if two keys hash to the same index.

### Common Collision Resolution Techniques

- **Chaining**: Each bucket holds a linked list (or dynamic array) of entries.
- **Open addressing**: Look for the next available slot (e.g., linear probing, quadratic probing, double hashing).

## Coming Up Next

In the next module, we'll build a minimal working hash table in C++ using:

- Arrays of pointers to linked lists (for chaining).
- A simple hash function for strings or integers.
- Custom structs for key-value pairs.

**Preview**: You'll build something like this:

```
// HashTable
//   |--- Bucket 0: NULL
//   |--- Bucket 1: [ ("Alice", 100) -> ("Bob", 95) ]
//   |--- Bucket 2: NULL
//   |___ ...
```

## Design Overview

We will build a hash table that:

- Stores `std::string` keys and `int` values.
- Uses a basic hash function.
- Uses **chaining** with linked lists to handle collisions.

### High-Level Structure

- A fixed-size array of `Node*`, where each bucket is a pointer to the head of a linked list.
- Each `Node` holds a key, value, and pointer to the next node.

## Step 1: Define a Node Struct

```
struct Node {
    std::string key;
    int value;
    Node* next;

    Node(std::string k, int v) : key(k), value(v), next(nullptr) {}
};
```

Listing 4: Struct for Linked List Node

**AS insight**: We're grouping key-value pairs with a next pointer to form a singly linked list.

**CR insight**: This structure will allow us to "chain" multiple key-value pairs at the same array index if collisions occur.

## Step 2: Define the Hash Table Class

```
class HashTable {
private:
    static const int TABLE_SIZE = 10;
    Node* table[TABLE_SIZE];

    int hashFunction(std::string key);

public:
    HashTable();
    ~HashTable();

    void insert(std::string key, int value);
    int search(std::string key);
    void remove(std::string key);
};
```

Listing 5: Basic Hash Table Class Skeleton

## Step 3: Constructor and Destructor

```
HashTable::HashTable() {
    for (int i = 0; i < TABLE_SIZE; ++i) {
        table[i] = nullptr;
    }
}

HashTable::~HashTable() {
    for (int i = 0; i < TABLE_SIZE; ++i) {
        Node* entry = table[i];
        while (entry != nullptr) {
            Node* prev = entry;
            entry = entry->next;
            delete prev;
        }
    }
}
```

**Why do this?**

- We initialize all buckets to `nullptr`.
- In the destructor, we walk through each bucket and free any nodes. This avoids memory leaks.

## Step 4: A Simple Hash Function

```
int HashTable::hashFunction(std::string key) {
    int hash = 0;
    for (char c : key) {
        hash += c;
    }
    return hash % TABLE_SIZE;
}
```

Listing 6: Hash Function for Strings

**Note**: This is a weak hash, but it works for small demos.

## Step 5: Insert Operation

```
void HashTable::insert(std::string key, int value) {
    int index = hashFunction(key);
    Node* newNode = new Node(key, value);
    newNode->next = table[index];
    table[index] = newNode;
}
```

This adds the new node to the front of the list at `table[index]`.

## Step 6: Search Operation

```
int HashTable::search(std::string key) {
    int index = hashFunction(key);
    Node* entry = table[index];

```

```
    while (entry != nullptr) {
        if (entry->key == key) return entry->value;
        entry = entry->next;
    }
    return -1; // Key not found
}
```

## Step 7: Remove Operation

```
void HashTable::remove(std::string key) {
    int index = hashFunction(key);
    Node* entry = table[index];
    Node* prev = nullptr;

    while (entry != nullptr && entry->key != key) {
        prev = entry;
        entry = entry->next;
    }

    if (entry == nullptr) return; // Not found

    if (prev == nullptr) {
        table[index] = entry->next; // Remove head
    } else {
        prev->next = entry->next;
    }

    delete entry;
}
```

## Test It All Together

```
int main() {
    HashTable ht;
    ht.insert("Alice", 100);
    ht.insert("Bob", 95);
    ht.insert("Charlie", 85);

    std::cout << "Alice: " << ht.search("Alice") << std::endl;
    std::cout << "Bob: " << ht.search("Bob") << std::endl;

    ht.remove("Bob");
    std::cout << "Bob after removal: " << ht.search("Bob") << std::endl;

    return 0;
}
```

Listing 7: Main function to test the hash table

## CR Wrap-up: Hidden Powers

By understanding this structure, you now know how to:

- Use linked data structures with dynamic memory.

- Control memory layout and access.
- Prepare for advanced modeling like sparse data graphs, lookup acceleration, or key-indexed maps.

## Better Hash Functions

In Module 2, our hash function was:

```
int hash = 0;
for (char c : key) {
    hash += c;
}
return hash % TABLE_SIZE;
```

### Why it's bad (AS/CR dual view):

- It treats permutations of characters the same (`"abc"` == `"cab"`).
- It increases collision probability for short ASCII strings.
- It doesn't scale well when 'TABLE_SIZE' increases.

### Improved Version: DJB2 Hash

The DJB2 hash function is simple, fast, and surprisingly effective.

```
unsigned long HashTable::hashFunction(std::string key) {
    unsigned long hash = 5381;
    for (char c : key) {
        hash = ((hash << 5) + hash) + c; // hash * 33 + c
    }
    return hash % TABLE_SIZE;
}
```

This gives better key distribution and sets you up for future hashing hardware-accelerated logic.

## Dynamic Resizing (a.k.a. Rehashing)

**Problem:** A fixed table size means eventual performance decay (more collisions, longer chains).
**Solution:** Track the load factor $\alpha = \frac{\text{num\_entries}}{\text{table\_size}}$. When $\alpha > 0.75$, double the size and rehash.

### Steps to Resize

1. Store the old table.
2. Allocate a new table (2x size).
3. Reinsert every key-value pair into the new table.
4. Free the old table.

We'll cover the implementation in a later advanced module to stay focused on concepts.

## Why `std::vector` is Your Friend

Instead of a fixed array:

```
Node* table[TABLE_SIZE];
```

We use:

```
std::vector<Node*> table;
```

Advantages:

- Auto-resizing (with `resize()`).
- Cleaner memory handling.
- Essential for GPU-style data chunking (next section).

### Update Constructor with Vector

```
HashTable::HashTable() {
    table.resize(TABLE_SIZE, nullptr);
}
```

## Spatial Memory Intuition (CR Spatial Insight)

Your hash table's bucket array is like a 1D grid in RAM. With `chaining`, each cell points to a mini list. In GPU terms:

- The table is **device global memory**.
- Each thread can operate on a disjoint bucket or chain.
- All operations must respect memory access rules.

**CUDA Insight:** Buckets can be parallelized. Each block/thread gets:

- A bucket index.
- A pointer to walk through the chain.

This maps cleanly to warp-style access patterns.

## Designing with Parallelism in Mind

Even though we're still in CPU-C++, it pays to design the interface so that:

- Insertions/searches can occur in parallel buckets.
- Hash functions are pure (no state/memory dependencies).
- Data locality is improved (you'll cache better).

## Your New Optimized Table Class (Sketch)

```
class HashTable {
private:
    std::vector<Node*> table;
    int size;

    unsigned long hashFunction(std::string key);

public:
    HashTable(int tableSize = 10);
    void insert(std::string key, int value);
    int search(std::string key);
    void remove(std::string key);
};
```

**AS Parallel-Ready Abstraction:**

- Input: key
- Compute: index
- Operate: on table[index] independently

## Coming Up in Module 4

You'll begin designing:

- A rehashable, vectorized hash table.
- Hooks for benchmarking insert/search speed.
- A "simulated GPU kernel" for parallel insertions (emulated with threads or loops).

**Bonus:** We'll preview how to port the core structure to a CUDA kernel-friendly format (e.g., SoA/CoA, global vs. shared memory constraints).

## Dynamic Resizing (a.k.a. Rehashing)

Your table should grow when too many items crowd into a small number of buckets (i.e., high load factor).

### When to Rehash?

Let:

$$\alpha = \frac{\text{num\_elements}}{\text{table\_size}}$$

Trigger rehash when $\alpha > 0.75$.

### Rehashing Steps

1. Save the old table.
2. Create a new table with size = $2\times$ old size.
3. Reinsert each element.
4. Delete old nodes to avoid memory leaks.

## Implementation Details

New member variables:

```cpp
int capacity;       // total slots
int count;          // total elements
```

Add this to ‘insert()‘:

```cpp
if ((float)count / capacity > 0.75) {
    rehash();
}
```

Rehash implementation:

```cpp
void HashTable::rehash() {
    int old_capacity = capacity;
    std::vector<Node*> old_table = table;

    capacity *= 2;
    table.clear();
    table.resize(capacity, nullptr);
    count = 0;

    for (int i = 0; i < old_capacity; ++i) {
        Node* entry = old_table[i];
        while (entry) {
            insert(entry->key, entry->value); // reinsert into new table
            Node* prev = entry;
            entry = entry->next;
            delete prev;
        }
    }
}
```

**Note:** We avoid directly copying pointers—this preserves hashing consistency with the new size.

## Performance Measurement Hooks

Use the standard C++ ‘¡chrono¿‘ library:

```cpp
#include <chrono>

auto start = std::chrono::high_resolution_clock::now();

// Perform insertions/searches

auto end = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> duration = end - start;
std::cout << "Time: " << duration.count() << "s\n";
```

This gives you fine-grained timing for benchmarking insert/search.

## Simulating Parallelism (Pre-CUDA Mental Model)

Imagine multiple threads inserting into the hash table. What happens?

**Challenge:** Two threads inserting into the same bucket may cause race conditions.

### CR Insight: Lockless Chaining Design

A CUDA-friendly design requires:

- Preallocated node storage (pool).
- Atomic inserts (e.g., lock-free append).
- Per-bucket synchronization (e.g., fine-grained locks or atomics).

**CPU Emulation Idea:**
Use ‘std::thread‘ to simulate parallel insertions:

```
void threadedInsert(HashTable* ht, std::string key, int val) {
    ht->insert(key, val);
}

std::thread t1(threadedInsert, &ht, "Alice", 100);
std::thread t2(threadedInsert, &ht, "Bob", 200);

t1.join();
t2.join();
```

**BUT:** You must add locks around ‘insert()‘ if you do this!

## Preparing for GPU-Style Design (Preview)

On a GPU:

- Buckets must be stored in device memory.
- Each thread gets a bucket index (e.g., via hash).
- You’ll need:
  - Flat memory representation (no pointers).
  - Atomic memory ops (e.g., `atomicCAS`).

### CR Anticipation: SoA vs. AoS

Instead of:

```
struct Node { string key; int val; Node* next; };
```

Think:

- `keys[]`, `values[]`, `next_index[]` → Structure of Arrays (SoA)

This lets you parallelize better in CUDA (coalesced memory access).

## Coming in Module 5:

We’ll finalize the full optimized C++ version with:

- Rehashable vectorized table
- Benchmark CLI interface
- Final project file layout
- Then start CUDA-friendly architectural planning!

## Final Hash Table Class: Overview

- Fully dynamic and rehashable
- Uses `std::vector` for bucket array
- Benchmarked via `std::chrono`
- Structurally sound for CUDA-style refactoring

## Header File: hash_table.hpp

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

struct Node {
    std::string key;
    int value;
    Node* next;
    Node(std::string k, int v) : key(k), value(v), next(nullptr) {}
};

class HashTable {
private:
    std::vector<Node*> table;
    int capacity;
    int count;

    unsigned long hashFunction(std::string key);
    void rehash();

public:
    HashTable(int initialSize = 10);
    ~HashTable();

    void insert(std::string key, int value);
    int search(std::string key);
    void remove(std::string key);
};
```

## Implementation File: hash_table.cpp

```cpp
#include "hash_table.hpp"
#include <iostream>

HashTable::HashTable(int initialSize) : capacity(initialSize), count(0) {
    table.resize(capacity, nullptr);
}

HashTable::~HashTable() {
    for (int i = 0; i < capacity; ++i) {
        Node* entry = table[i];
        while (entry) {
            Node* prev = entry;
            entry = entry->next;
```

```
            delete prev;
        }
    }
}

unsigned long HashTable::hashFunction(std::string key) {
    unsigned long hash = 5381;
    for (char c : key)
        hash = ((hash << 5) + hash) + c;
    return hash % capacity;
}

void HashTable::rehash() {
    int old_capacity = capacity;
    std::vector<Node*> old_table = table;

    capacity *= 2;
    table.clear();
    table.resize(capacity, nullptr);
    count = 0;

    for (int i = 0; i < old_capacity; ++i) {
        Node* entry = old_table[i];
        while (entry) {
            insert(entry->key, entry->value);
            Node* prev = entry;
            entry = entry->next;
            delete prev;
        }
    }
}

void HashTable::insert(std::string key, int value) {
    if ((float)count / capacity > 0.75)
        rehash();

    int index = hashFunction(key);
    Node* newNode = new Node(key, value);
    newNode->next = table[index];
    table[index] = newNode;
    ++count;
}

int HashTable::search(std::string key) {
    int index = hashFunction(key);
    Node* entry = table[index];
    while (entry) {
        if (entry->key == key)
            return entry->value;
        entry = entry->next;
    }
    return -1;
}

void HashTable::remove(std::string key) {
    int index = hashFunction(key);
    Node* entry = table[index];
    Node* prev = nullptr;

```

```cpp
    while (entry && entry->key != key) {
        prev = entry;
        entry = entry->next;
    }

    if (!entry) return;

    if (!prev) table[index] = entry->next;
    else prev->next = entry->next;

    delete entry;
    --count;
}
```

## Benchmark File: main.cpp

```cpp
#include "hash_table.hpp"
#include <iostream>
#include <chrono>

int main() {
    HashTable ht;
    const int N = 10000;

    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 0; i < N; ++i) {
        ht.insert("Key" + std::to_string(i), i);
    }
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> diff = end - start;
    std::cout << "Inserted " << N << " items in " << diff.count() << " seconds.\n"
        ;

    std::cout << "Search result for Key1234: " << ht.search("Key1234") << "\n";
}
```

## Project Structure

```
hash_table_project/
|--- hash_table.hpp
|--- hash_table.cpp
|--- main.cpp
|__ Makefile (optional)
```

## Makefile (Optional)

```makefile
all:
    g++ -std=c++17 -O2 main.cpp hash_table.cpp -o hash_table
```

## Next Steps (Toward CUDA)

Now that your C++ base is solid, you're ready to:

- Flatten your data layout (SoA instead of pointers)
- Preallocate all memory
- Implement per-bucket insertions in threads
- Use atomic primitives or warp-serial execution

## Bonus Thought: Sparse Graphs and Algebraic Models

Your hash table could now easily:

- Represent sparse graphs (edges as keys, weights as values)
- Store fast-access memoization results for recurrence relations
- Index elements by prime-factor "key" patterns

This opens the door to using hash maps in symbolic computation, parallel dataflow models, or even algebraic DSLs you design yourself.