C++ Hash Table Implementation: An Algorithms and Data Structures Problem

This document presents a foundational implementation of a hash table in C++. It's designed to be clear and easy to follow, providing a basic introduction to how hash tables work using custom-built functions.

Understanding the Logic

We've implemented the core functionalities of a hash table:

- `hash`: Computes an index for a given key.
- `put`: Inserts or updates a key-value pair.
- `search`: Finds a key's corresponding value.
- `get`: Retrieves a key's value.
- `deleteKey`: Removes a key-value pair.

For a deeper dive into the theoretical underpinnings and advanced concepts of hashing functions and hash tables, you may refer to classic texts like "Introduction to Algorithms" by Cormen, Leiserson, Rivest, and Stein.

Please take a moment to review the code. It demonstrates how data can be stored and retrieved using fundamental hashing principles. As an extension, you might experiment with simple transformations like XOR to encode values before storage, observing how this affects the data.

Compilation Notes

When compiling this C++ code from the command line, especially in environments like Dev-C++, remember to add the `-std=c++11` (or a newer version like `-std=c++14`, `-std=c++17`, `-std=c++20`) flag. Using `-std=c++17` is generally a good choice as it includes all C++11 features and more recent improvements.

Security Note

Important: The hashing and encoding techniques shown here, particularly simple ones like XOR, are for educational purposes only. They are **not suitable** for real-world security applications or protecting sensitive data, as they can be easily reversed with basic knowledge. The logic for secure hashing would be similar but involve far more complex, cryptographically secure algorithms.

The Code

(This code is from my github. I don't share it public hhe)

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>

using namespace std;

// Represents a node in the linked list for collision handling
struct Value
{
 string key;
 string value;
 Value* next;
 Value(string key, string value) : key(key), value(value), next(nullptr){}
};

// Represents the hash map structure
struct Map
{
 int maxSize = 100000; // Fixed size for the hash table array
 vector<Value*>table; // Vector of pointers to Value structs (linked lists)

 // Constructor to initialize the table with nullptrs
 Map()
 {
  table.resize(maxSize);
 }

 // Hashing function to convert a string key into an array index
 int hash(string& str)
 {
  int p = 33; // A common prime number used in polynomial hashing
  int hash = 0;
  for(int i = 0; i < str.size(); i++)
  {
   hash = (hash * p + str[i]) % maxSize; // Polynomial rolling hash
  }
  return hash;
 }

 // Inserts or updates a key-value pair in the hash map
 void put(string& key, string& value)
 {
```

```
 Value* x = search(key); // Check if the key already exists
 if(x!=nullptr)
 {
  x->value = value; // If key exists, update its value
  return;
 }
 int h = hash(key); // Get the hash index
 Value* node = new Value(key, value); // Create a new node
 node -> next = table[h]; // Link the new node to the head of the list at index h
 table[h] = node; // Make the new node the head
}

// Searches for a key and returns a pointer to its Value node
Value* search(string& key)
{
 Value* node = table[hash(key)]; // Start from the head of the list at the hash index
 while(node != nullptr)
 {
  if(node->key == key)
  {
   return node; // Key found
  }
  node = node -> next; // Move to the next node in the list
 }
 return nullptr; // Key not found
}

// Deletes a key-value pair from the hash map
void deleteKey(string& key)
{
 int h = hash(key); // Get the hash index
 Value* node = table[h]; // Current node
 Value* prevNode = nullptr; // Previous node

 while(node!=nullptr)
 {
  if(node -> key == key)
  {
   if(prevNode == nullptr)
   {
    table[h] = node -> next; // If it's the head node, update table[h]
   }
   else
   {
    prevNode -> next = node -> next; // Bypass the node to be deleted
    node -> next = nullptr; // Disconnect the deleted node
```

```
    }
    // Note: The deleted node is not explicitly 'delete'd from memory here
    // In a production environment, you would call delete node;
    return;
   }
   prevNode = node; // Move prevNode forward
   node = node -> next; // Move node forward
  }
 }

 // Retrieves the value associated with a key
 std::string get(string& key)
 {
  Value* node = search(key); // Search for the node
  return((node == nullptr)? "none" : node -> value); // Return value or "none" if not found
 }
};

// Main function to read commands from "map.in" and write output to "map.out"
int main()
{
 ifstream input("map.in"); // Input file stream
 ofstream output("map.out"); // Output file stream
 Map hashMap; // Create a hash map instance

 // Process commands until end of file
 while(!input.eof())
 {
  string str, key, value;
  input >> str; // Read command (put, get, delete)
  if(str == "put")
  {
   input >> key >> value; // Read key and value for "put"
   hashMap.put(key, value);
  }
  else if(str == "get")
  {
   input >> key; // Read key for "get"
   output << hashMap.get(key) << endl; // Get value and write to output file
  }
  else if(str == "delete")
  {
   input >> key; // Read key for "delete"
   hashMap.deleteKey(key);
  }
 }
```

```
 input.close(); // Close input file
 output.close(); // Close output file
 return 0;
}
```