Thoughts on Project 8
Tables in the C++ STL & Elsewhere

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, November 20, 2020

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

`ggchappell@alaska.edu`

Thoughts on Project 8

Thoughts on Project 8
Overview

Project 8 has two exercises:

- In Exercise A, you will write a complete C++ program (including `main`!) that uses an STL Table implementation.
- In Exercise B, you will write a test program, using the *doctest* framework, for a simple class.

Thoughts on Project 8
Exercise A [1/3]

In Exercise A you will write a program that is given a filename. It reads the file with that name and breaks it into words. Then it prints certain information about those words.

You are to choose an appropriate STL Table implementation and use it in your program.

The program needs to *work*, of course. But your choice of Table implementation, and proper use of it, will also be factor in the grading.

Thoughts on Project 8 Exercise A [2/3]

A **word** is a sequence of non-space characters. For example, suppose your program is given the name of a file containing the following text.

```
dog dog? dog
dog dog?   cat
```

This file contains 6 words having 3 distinct values. In lexicographic order, these values are the following:

- `cat`
- `dog`
- `dog?`

Thoughts on Project 8 Exercise A [3/3]

If it can read the file, then your program should do the following.

- Print a message indicating the number of *distinct* words in the file.
- Go through these words in lexicographic order. For each, print, on one line, the word, a colon, a blank, and then the number of times that word appears in the file.

For the given file, the following should be printed.

```
Number of distinct words: 3

cat: 1
dog: 3
dog?: 2
```

Text in file:

```
dog dog? dog
dog dog?   cat
```

Thoughts on Project 8
Exercise B [1/9]

In Exercise B you will write a test program, using the *doctest* framework, for a very simple class called `Squarer`.

Objects of class `Squarer` are function objects. The function is a template that returns the square of its parameter.

So `Squarer` should be usable as follows.

```
Squarer sq;
int n = sq(5);         // Sets n to 5 squared: 25
double d = sq(1.1);    // Sets d to 1.1 squared: 1.21
```

Thoughts on Project 8
Exercise B [2/9]

We could write class `Squarer` as follows.

```
// class Squarer. Class invariants: None.
class Squarer {
public:
    // operator(). Returns square of its parameter.
    // Req's on types: Num must have op*, copy ctor.
    // Throws what & when Num ops throw.
    // Strong guarantee
    // Exception neutral.
    template<typename Num>
    Num operator()(const Num & k) const
    { return k * k; }

    // Default ctor, copy ctor, move ctor, copy =, move =, dctor:
    // automatically generated versions used.
};
```

Thoughts on Project 8
Exercise B [3/9]

I provide a skeleton source file for a test program.

Your finished source file should contain into a number of **test cases**. Each should look something like this:

```
TEST_CASE("Squarer: negative ints")
{
    …
}
```

A printable message identifying the test case.

Thoughts on Project 8
Exercise B [4/9]

Within a test case are one or more **tests**, which are done with a `REQUIRE` directive. Inside parentheses after `REQUIRE` is an expression of type `bool`, which is `true` if the test passes, and `false` otherwise.

Other code may be included in a test case.

```
TEST_CASE("Squarer: negative ints")
{
    Squarer sq;
    REQUIRE(sq(-5) == 25);
    …
}
```

Thoughts on Project 8
Exercise B [5/9]

You should also include `INFO` directives. Each of these takes a string, which is printed if any following test fails.

```
TEST_CASE("Squarer: negative ints")
{
    Squarer sq;
    INFO("-5 squared is 25");
    REQUIRE(sq(-5) == 25);
    …
}
```

Thoughts on Project 8
Exercise B [6/9]

Stream insertion (<<) may be used in an `INFO` directive.

```
TEST_CASE("Squarer: negative ints")
{
    Squarer sq;
    int arg = -5;
    int result = 25;
    INFO(arg << " squared is " << result);
    REQUIRE(sq(arg) == result);
    …
}
```

Thoughts on Project 8
Exercise B [7/9]

If you do not want the `INFO` string to be printed for *all* tests that follow in the test case, then you can put it, and the associated `REQUIRE`, in a **subcase**. The `INFO` string will go away at the end of the subcase.

```
TEST_CASE("Squarer: negative ints")
{
    Squarer sq;
    SUBCASE("Square -5") {
        INFO("-5 squared is 25");
        REQUIRE(sq(-5) == 25);
    }
    SUBCASE("Square -1") {
        INFO("-1 square is 1");
        REQUIRE(sq(-1) == 1);
    }
    …
```

Thoughts on Project 8
Exercise B [8/9]

Remember that floating-point arithmetic is inexact. When doing floating-point tests, we usually want to test for *approximate* equality. *doctest* enables this with `doctest::Approx`, used as follows.

```
TEST_CASE("Squarer: doubles")
{
    Squarer sq;
    INFO("1.1 squared is 1.21");
    REQUIRE(sq(1.1) == doctest::Approx(1.21));
    …
```

Thoughts on Project 8 Exercise B [9/9]

Issues to consider when writing your test program:

- Do both const & non-const `Squarer` objects work?
- Does `Squarer` work properly for a wide range of values?
- Does `Squarer` work properly for both positive and negative values?
- Does `Squarer` work properly for both integer and floating-point arguments?
- Special cases: the squares of `0`, `1`, and `-1` should be correct.
- When a test fails, is the message printed both correct and helpful?

doctest has other features that you may wish to use: `CHECK` directives, `CAPTURE` directives, etc. However, you are not required to use any of these.

Review

Review
Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

A solution to this problem is a **container**.

In a **generic container**, client code can specify the value type.

Unit Overview
Tables & Priority Queues

Major Topics

- ✓ Introduction to Tables ← Lots of lousy implementations
- ✓ Priority Queues
- ✓ Binary Heap Algorithms
- ✓ Heaps & Priority Queues in the C++ STL

  } Idea #1: Restricted Table

- ✓ 2-3 Trees
- ✓ Other self-balancing search trees

  } Idea #2: Keep a tree balanced

- ✓ Hash Tables

  } Idea #3: Magic functions

- ✓ Prefix Trees ← A special-purpose implementation: "the Radix Sort of Table implementations"
- Tables in the C++ STL & Elsewhere

Overview of Advanced Table Implementations

We cover the following advanced Table implementations.

- Self-balancing search trees
  - To make things easier, allow more children (?):
    - ✓ **2-3 Tree**
      - Up to 3 children
    - ✓ **2-3-4 Tree**
      - Up to 4 children
    - ✓ **Red-Black Tree**
      - Binary Tree representation of a 2-3-4 Tree
  - Or back up and try for a strongly balanced Binary Search Tree again:
    - ✓ **AVL Tree**
- Alternatively, forget about trees entirely:
  - ✓ **Hash Table**
- Finally, "the Radix Sort of Table implementations":
  - ✓ **Prefix Tree**

Idea #2:
Keep a tree balanced

Later, we will cover other self-balancing search trees: B-Trees, B+ Trees.

Idea #3:
Magic functions

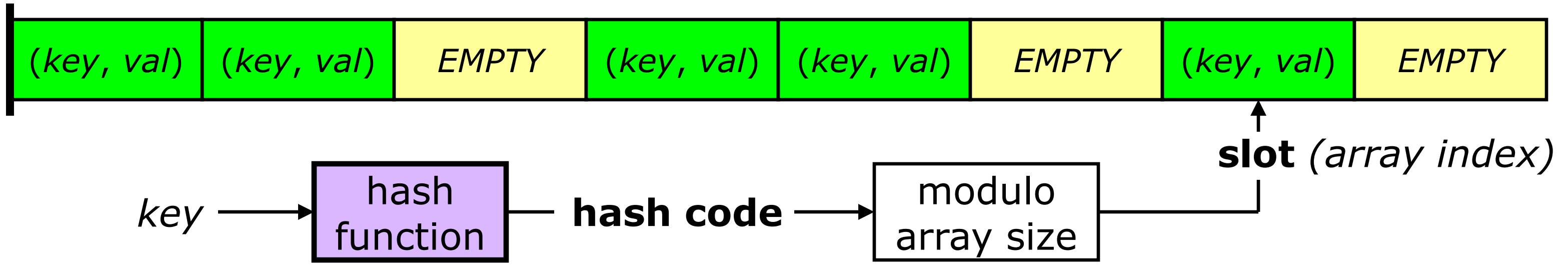
DONE

Review
Hash Tables [1/5]

A **Hash Table** is a Table implementation that stores key-value pairs in an unsorted array. Array indices are **slots**.

Or just keys, if there are no associated values.

- A key’s slot is computed using a **hash function**.
- An array location can be *EMPTY*.

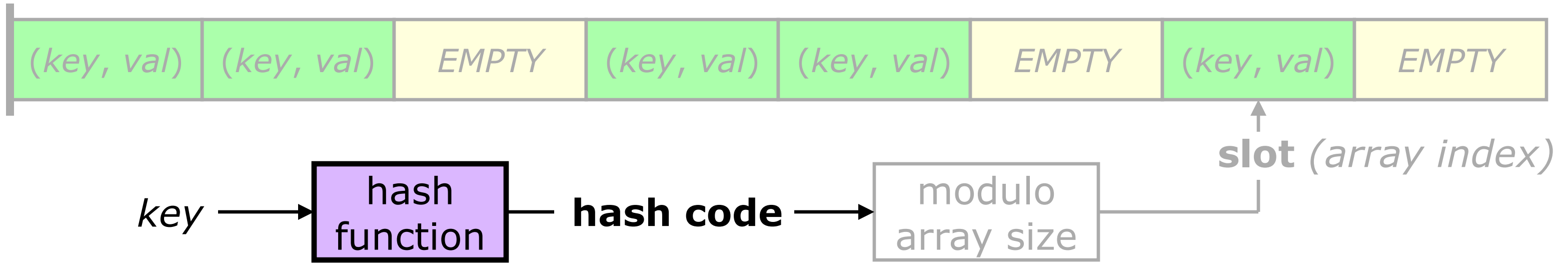


Collision: when two items with unequal keys get the same slot. Collisions are generally an unavoidable problem, because there are often far more possible keys than slots.

Needed

- Hash function (typically separate from the Hash Table implementation).
- **Collision-resolution** method.

Review
Hash Tables [2/5]



A hash function *must*:

- Take a key and return a nonnegative integer (**hash code**).
- Be **deterministic**: the output depends only on the input. Passing the same key multiple times results in the same hash code every time.
- Return the same hash code for keys that compare equal (==).

Consistency requirement mentioned previously

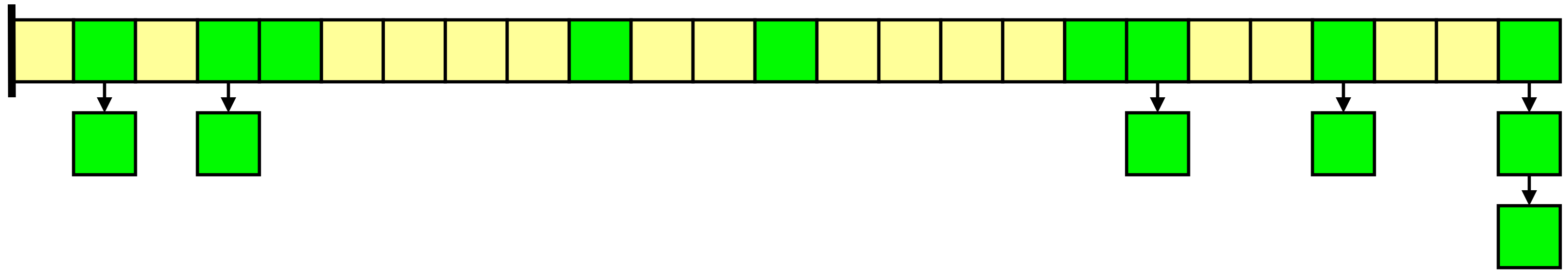
A *good* hash function:

- Is fast.
- Spreads out its results evenly over the possible output values.
- Turns patterns in its input into unpatterned output.

Review
Hash Tables [3/5]

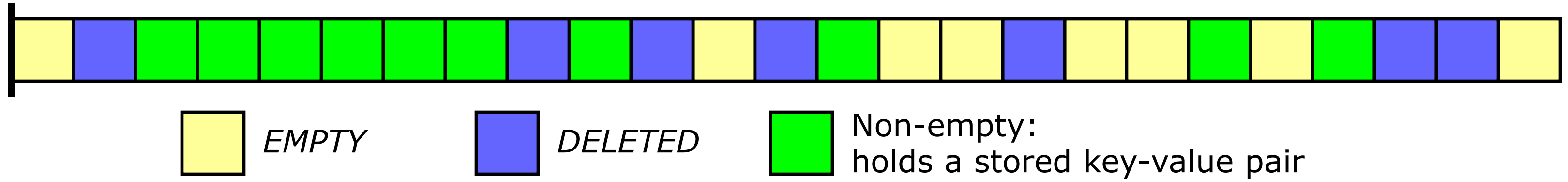
Collision resolution methods, category #1: **Open Hashing**

- An array item (**bucket**) can store multiple key-value pairs.
- Buckets are virtually always Singly Linked Lists.
- To find a key, determine which bucket to look in based on the hash code. Do a Sequential Search on that bucket.



Collision resolution methods, category #2: **Closed Hashing**

- An array item holds one key-value pair, or is *EMPTY* or *DELETED*.
- To find a key, begin at the slot given by the hash code, and **probe** in a sequence of slots: the **probe sequence**.

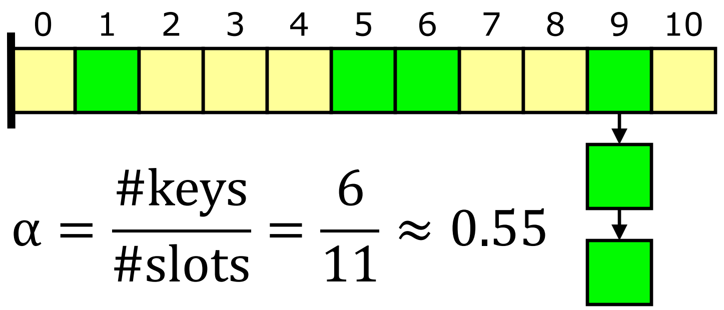


Review
Hash Tables [4/5]

Worst-case time for all operations is linear time.

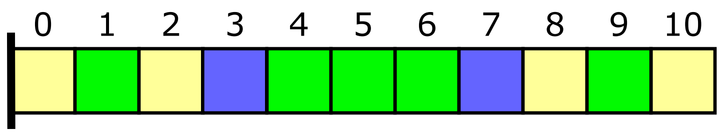
Average-case performance of a Hash Table can be analyzed based on its **load factor**: α = (# of keys present) / (# of slots).

Open Hashing



$$\alpha = \frac{\#\text{keys}}{\#\text{slots}} = \frac{6}{11} \approx 0.55$$

Closed Hashing



$$\alpha = \frac{\#\text{keys}}{\#\text{slots}} = \frac{5}{11} \approx 0.45$$

The load factor is kept small—well below 1.

Average-case time for *retrieve* and *delete* is constant time.

When the load factor gets too high, **rehash**: rebuild the Hash Table in a larger array. So average-case time for *insert* is amortized constant.

Review
Hash Tables [5/5]

Efficiency Comparison (duplicate keys not allowed)

	Idea #1	Idea #2	Idea #3	
	Priority Queue using Heap	Self-Balancing Search Tree	Hash Table: worst case	Hash Table: average case
Retrieve	Constant*	Logarithmic	Linear	Constant
Insert	Amortized** logarithmic	Logarithmic	Linear	Amortized constant***
Delete	Logarithmic*	Logarithmic	Linear	Constant

*Priority Queue *retrieve* & *delete* are not Table operations in full generality. Only the item with the highest priority (key) can be retrieved/deleted.

**Logarithmic if enough memory is preallocated. Otherwise, occasional reallocate-and-copy—linear time—may be required. Time per *insert*, averaged over many consecutive *inserts*, will be logarithmic. Thus, *amortized logarithmic time* (which is not a term I expect you to know).

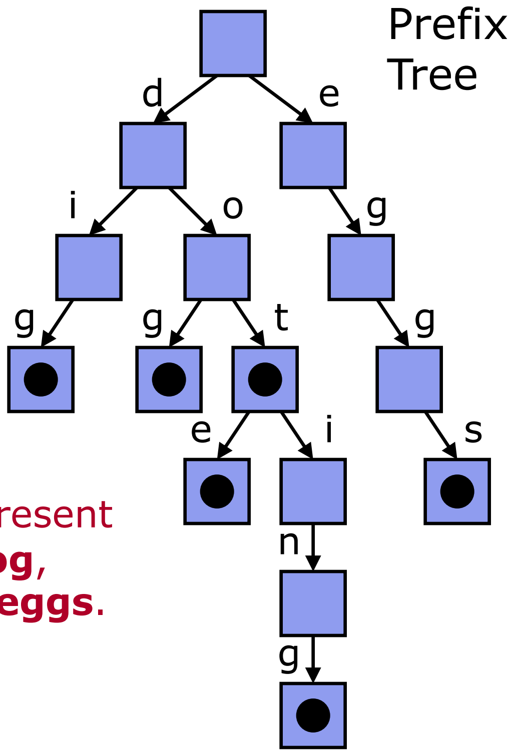
***Hash Table *insert* is constant-time only in a *double average* sense: averaged both over all possible inputs and over a large number of consecutive *inserts*.

Review
Prefix Trees

A **Prefix Tree** (a.k.a. **Trie**) is a Table implementation in which the keys are **strings**—in a general sense, as for Radix Sort.

A Prefix Tree is a kind of tree.

- A node has:
  - A Boolean—whether it represents a stored key.
  - Child pointers—one for each possible character.
  - The value associated with a key, if needed.



Nodes with dots represent stored keys: **dig**, **dog**, **dot**, **dote**, **doting**, **eggs**.

Prefix Trees are easy to implement well!

Tables in the C++ STL & Elsewhere

Tables in the C++ STL & Elsewhere
Overview

Now we look briefly at Table implementations as they exist in the C++ STL and also in the broader world of programming.

- C++ STL
  - Set: `std::set`
  - Key-value structure: `std::map`
  - Hash Table versions: `std::unordered_set`, `std::unordered_map`
  - Tables allowing duplicate keys
- Other Programming Languages
  - Python
  - Perl
  - JavaScript
- Other Data Structures
  - Hash Trees
  - Skip Lists

Tables in the C++ STL & Elsewhere
C++ STL: std::set — Introduction

The simplest STL Table implementation is `std::set`, in `<set>`.

- An item is simply a key; there is no associated value.
- Duplicate (equivalent) keys are not allowed.
- The spec. was written with a self-balancing search tree in mind. Implementations will typically use a Red-Black Tree or variant.

Declare a `std::set` as follows:

```
std::set<valuetype> s;
```

The comparison is specified as for `std::priority_queue`.

- Default: use `operator<`.
- Optional second template parameter: the *type* of the comparison.
- If necessary, pass the comparison itself as a constructor argument.

Tables in the C++ STL & Elsewhere
C++ STL: std::set — Iterators [1/2]

`std::set` has bidirectional iterators. `begin`/`end` work as usual.

Items appear in sorted order; `set` is basically a SortedSequence.

`set` does not have **mutable** iterators. We cannot do "`*iter = v;`", and `begin`/`end` cannot be used to modify items.

Q. Why not?

A. Items are sorted. Changing an item might break this invariant.

Range-based for-loops can be used.

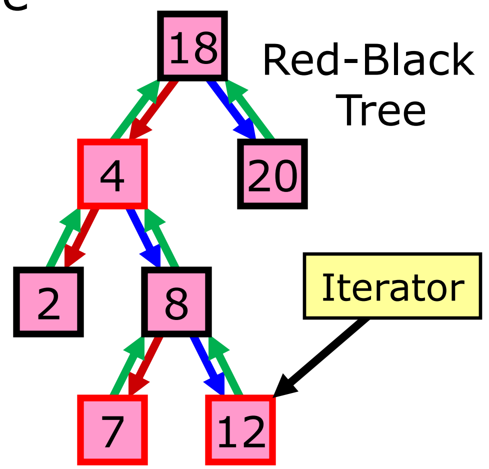
```
for (const auto & k : s)
{
    cout << "Key: " << k << endl;
}
```

Tables in the C++ STL & Elsewhere
C++ STL: std::set — Iterators [2/2]

`std::set` iterators and references are valid until the referenced item is destroyed.

What does this tell us about the implementation?

- A Red-Black tree may be reorganized by an insertion or deletion. So iterators must not store information about the structure of the tree.
- But we must be able to navigate around the tree efficiently, starting at a leaf node.
- So the tree must have **parent pointers**.
- An iterator can simply be a wrapper around a pointer to a node—together with algorithms for navigating the tree.



Tables in the C++ STL & Elsewhere
C++ STL: std::set — Operations [1/2]

Table Insert: member function `insert`

- Given an item (same as a key, for `set`). Inserts this into the set.
- **Does nothing** if an equivalent item (key) is already in the set.
- Returns `pair<`*iterator*`, bool>`. Iterator points to inserted item or already present item. The `bool` is `true` if the insertion happened.

```
set<int> s;
auto p = s.insert(3);
if (!p.second) cout << "3 was already present";
```

Table Delete: member function `erase`

- Given key *or* iterator. Deletes the proper item, if any, from the set.

```
s.erase(5);
s.erase(p.first());
```

Tables in the C++ STL & Elsewhere
C++ STL: std::set — Operations [2/2]

Table Retrieve #1: member function `find`

- Given a key. Returns iterator to the item, or `end()` if not found.

```
auto iter = s.find(3);
if (iter != s.end()) cout << "3 found";
```

Table Retrieve #2: member function `count`

- Given a key. Returns number of times key occurs in set (`0` or `1`).

```
if (s.count(3) != 0) cout << "3 found";
```

Why not use `std::find` or `std::binary_search` (or a variant)?

- Both work! But both are Θ(*n*): `find` because it does Sequential Search, `binary_search` because `std::set` is not random-access.
- However, member function `find` is Θ(log *n*) [Red-Black Tree].

Tables in the C++ STL & Elsewhere
C++ STL: std::map — Introduction

The other main STL Table is `std::map`, in `<map>`.

- An item is a key along with associated data.
  - The key type & **data type** are both specified.
  - The value type is `pair<const` *keytype*`,` *datatype*`>`.
- Duplicate (equivalent) keys are not allowed.
- The spec. was written with a self-balancing search tree in mind.

STL-speak for *type of the associated value.*

Declare a `std::map` as follows:

```
std::map<keytype, datatype> m;
```

An optional comparison can be specified. The default is `operator<`.

Most `map` operations are much the same as for `set`.

- Insert: member function `insert`, given $\boxed{\text{item.}}$
- Delete: member function `erase`, given $\boxed{\text{key}}$ or iterator.
- Retrieve: member function `find` or `count`, given key.

For `map`, these two are different!

Tables in the C++ STL & Elsewhere
C++ STL: std::map — Bracket Operator [1/3]

A very convenient operation: *datatype* `& operator[](`*key*`)`
This allows a `map` to be used like an array. Examples:

```
map<string, int> m;
m["abc"] = 7;
cout << m["abc"] << endl;
m["abc"] += 2;
```

`operator[]` can be defined as follows (`k` is the given key):

```
(*((m.insert(make_pair(k, datatype()))).first)).second
```

Make sure key `k` *is in the* `map`*, and give me the associated value.*

Tables in the C++ STL & Elsewhere
C++ STL: std::map — Bracket Operator [2/3]

More `operator[]` examples:

```
map<int, int> m2;
m2[0] = 34;
m2[123456789] = 28;  // Very little memory used!

map<string, string> id;
id["Cuthbert Gump"] = "abc";
cout << id["Frederica Murg"] << endl;
// The above line inserts
//     pair<string, string> ("Frederica Murg", string())
// into the map.
```

Tables in the C++ STL & Elsewhere
C++ STL: std::map — Bracket Operator [3/3]

`operator[]` for `map` is useful and convenient. However, it *always* calls `insert`. So it has no const version.

```
void printAbcValue(const map<string, int> & mm)
{
    cout << ~~mm["abc"]~~ << endl;   // DOES NOT COMPILE!
}
```

Due to the insertion, `operator[]` is a poor way to check whether a key is already in the `map`. Use member function `count`.

```
map<Foo, Bar> m3;
Foo theKey;   // We want to check whether theKey is in m3
if (m3.count(theKey) != 0)   // GOOD way to check
if (m3[theKey] == …)         // BAD way to check
```

Tables in the C++ STL & Elsewhere
C++ STL: std::map — Iterators

Iterators for `map` are much as for `set`.

- They are bidirectional iterators.
- Items appear in sorted order, by key.
- They are not **mutable**. We cannot do "`*iter = v;`".

However, we can do "`(*iter).second = d;`". ← This is legal, but we would normally write `iter->second = d;`

Q. How is this possible?

A. The value type is `pair<const` *keytype*, *datatype*`>`.

Remember that a `std::map` item is a key-value pair.

```
for (const auto & kvpair : m)
{
    cout << "Key: "   << kvpair.first  << "  "
         << "value: " << kvpair.second << endl;
}
```

Tables in the C++ STL & Elsewhere
C++ STL: Hash Tables

The 2011 C++ Standard added Hash Table versions of `set` & `map`:

- `std::unordered_set`, in header `<unordered_set>`.
- `std::unordered_map`, in header `<unordered_map>`.

These are very similar to `set` & `map`, respectively.

- Value types are identical.
- Member functions `insert`, `erase`, `find`, & `count` work the same.
- `unordered_map` has `operator[]`—which inserts.

Primary differences:

- Efficiency is as for a Hash Table, not a self-balancing search tree.
- Table traverse is not sorted—thus “unordered”.
- No ordering is used. A custom **hash function** and **equality comparison** can be specified.

The interfaces were written with open hashing in mind. In particular, iteration through a single bucket is supported. (Why? I could not say.)

Tables in the C++ STL & Elsewhere
C++ STL: Tables Allowing Duplicate Keys

The STL also has Tables that allow duplicate keys:

- `std::multiset`
- `std::multimap`
- `std::unordered_multiset`
- `std::unordered_multimap`

Each is declared in the same header as its non-multi version.

- E.g., `std::multiset` is declared in header `<set>`.

Each is similar to its non-multi version. Important differences:

- The `count` member function may return values greater than `1`.
- `multimap` & `unordered_multimap` have no `operator[]`.
- In practice, when using these containers, we might deal with a *range* of items having equivalent/equal keys. Relevant member functions include `equal_range`, `lower_bound`, `upper_bound`.

Tables in the C++ STL & Elsewhere
TO BE CONTINUED …

Tables in the C++ STL & Elsewhere will be continued next time.