SETS AND MAPS

Chapter 9
The set Functions

- Required methods:
  - testing set membership \( \text{find} \)
  - testing for an empty set \( \text{empty} \)
  - determining set size \( \text{size} \)
  - creating an iterator over the set \( \text{begin, end} \)
  - adding an element \( \text{insert} \)
  - removing an element \( \text{erase} \)

- There are no set union, set intersection, or set difference member functions
  - However, these operators are defined in the algorithm header for all containers, not just sets
The set Functions (cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
</table>
| template<typename II>
  set(II first, II last) | Constructs a set from the sequence of objects represented by the iterator range first...last. |
| iterator begin();
  const iterator begin() const | Returns an iterator to the first item in the set. |
| iterator end();
  const_iterator end() | Returns an iterator to one past the last item in the set. |
| bool empty() | Returns **true** if there are no items in the set. |
| int size() | Returns the number of items in the set. |
| pair<iterator, bool> insert(const Key_Type& item) | Inserts an item into the set. If the item is not in the set, the returned iterator will reference the inserted item, and the **bool** parameter will be **true**. If the item is already in the set, the returned iterator (the iterator parameter) will reference the item that is currently in the set, and the **bool** parameter will be **false**. |
| template<typename II>
  void insert(II first, II last) | Inserts the items from the sequence of objects represented by the iterator range first...last. Duplicate values are not inserted. The iterator range must not reference elements in the target set. |
| void erase(const Key_Type& item) | Removes the item from the set. |
| iterator find(const Key_Type& item) | Returns an iterator that references the item in the set. If the item is not present, **end()** is returned. |
The set Functions (cont.)

- The set is a template class that takes the following template parameters:
  - `Key_Type`: The type of the item contained in the set
  - `Compare`: A function class that determines the ordering of the keys; by default this is the less-than operator
  - `Allocator`: The memory allocator for key objects; we will use the library supplied default

- Although not a requirement, C++ stores items in a set as ordered by their `Compare` function

- If you iterate through a set, you get a sorted list of the contents
### The set Functions (cont.)

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>template&lt;typename II1, typename II2, typename OI, typename Compare&gt; OI set_difference(II1 first1, II1 last1, II2 first2, II2 last2, OI result, Compare less)</code></td>
<td>Forms the set difference of the element in the sorted sequence <code>first1 ... last1</code> and the elements in the sorted sequence <code>first2 ... last2</code>. The set difference is stored in <code>result</code>. An iterator to the end of <code>result</code> is returned.</td>
</tr>
<tr>
<td><code>template&lt;typename II1, typename II2, typename OI, typename Compare&gt; OI set_intersection(II1 first1, II1 last1, II2 first2, II2 last2, OI result, Compare less)</code></td>
<td>Forms the set intersection of the element in the sorted sequence <code>first1 ... last1</code> and the elements in the sorted sequence <code>first2 ... last2</code>. The set difference is stored in <code>result</code>. An iterator to the end of <code>result</code> is returned.</td>
</tr>
<tr>
<td><code>template&lt;typename II1, typename II2, typename OI, typename Compare&gt; OI set_union(II1 first1, II1 last1, II2 first2, II2 last2, OI result, Compare less)</code></td>
<td>Forms the set union of the element in the sorted sequence <code>first1 ... last1</code> and the elements in the sorted sequence <code>first2 ... last2</code>. The set union is stored in <code>result</code>. An iterator to the end of <code>result</code> is returned.</td>
</tr>
</tbody>
</table>
The set Functions (cont.)

```cpp
#include <set>
#include <string>
#include <iostream>
#include <algorithm>
#include <iterator>
#include "Set_Functions.h"      // for operator<<

using namespace std;
```
The set Functions (cont.)

```cpp
int main()
{
    set<string> set1;
    set<string> set2;
    set<string> set_u;
    set<string> set_d;
    set<string> set_i;

    string data1[] = {"Apples", "Oranges", "Pineapples"};
    string data2[] = {"Peaches", "Apples", "Grapes"};

    set1.insert(data1, data1+3);
    set2.insert(data2, data2+3);
    cout << "set1 is " << set1 << endl;
    cout << "set2 is " << set2 << endl;
```
The *set* Functions (cont.)

```cpp
class SetExample {
public:
  void processSets() {
    set<int> set1 = {1, 2, 3, 4};
    set<int> set2 = {3, 4, 5, 6};

    // Set union
    set_union(set1.begin(), set1.end(),
              set2.begin(), set2.end(),
              inserter(set_u, set_u.begin()));
    cout << "set1 + set2 is " << set_u << endl;

    // Set difference
    set_difference(set1.begin(), set1.end(),
                   set2.begin(), set2.end(),
                   inserter(set_d, set_d.begin()));
    cout << "set1 - set2 is " << set_d << endl;

    // Set intersection
    set_intersection(set1.begin(), set1.end(),
                     set2.begin(), set2.end(),
                     inserter(set_i, set_i.begin()));
    cout << "set1 * set2 is " << set_i << endl;

    bool is_member = (set1.find("Apples") != set1.end());
    cout << """Apples"" is an element of set1 is "
         << boolalpha << is_member << endl;

    return 0;
  }
};
```
The union operator (+) can be defined as

```cpp
/** Construct the union of two sets. */
template<typename Key_Type, typename Compare>
std::set<Key_Type, Compare> operator+(        
const std::set<Key_Type, Compare>& left,     
const std::set<Key_Type, Compare>& right) {  
typename std::set<Key_Type, Compare> result(left); 
result.insert(right.begin(), right.end());     
return result; }
```
The difference operator (-) can be defined as

```cpp
/** Construct the difference of two sets. */
template< typename Key_Type, typename Compare>
std::set<Key_Type, Compare> operator-(
    const std::set<Key_Type, Compare>& left,
    const std::set<Key_Type, Compare>& right) {
    typename std::set<Key_Type, Compare> result(left);
    for (typename std::set<Key_Type, Compare>::const_iterator
        itr = right.begin(); itr != right.end(); ++itr)
        result.erase(*itr);
    return result;
}
```
The set Operators +, −, *, and << (cont.)

- The membership test can be defined as

```cpp
template<typename Key_Type, typename Compare>
bool contains(const std::set<Key_Type, Compare>& s,
              const Key_Type& k) {
    return s.find(k) != s.end();
}
```
The set Operators $+, -, *,$ and $\ll$ (cont.)

- The `ostream` insertion operator ($\ll$) can be defined as

```cpp
// Overloading the ostream insertion operator
template<typename Item_Type>
std::ostream& operator<<(std::ostream& out,
    const std::set<Item_Type>& a_set) {
  out << "{";
  bool first = true;
  for (typename std::set<Item_Type>::const_iterator
    itr = a_set.begin(); itr != a_set.end(); ++itr) {
    if (first)
      out << *itr;
    else
      out << ", " << *itr;
    first = false;
  }
  return out << "}";
}
```
Comparison of vectors and sets

- Vectors and sets both have insert and erase functions, but these functions have different signatures and slightly different meanings.
  - With the vector you must specify the location where the item is to be inserted, but with the set you do not.
    - The set does have an insert function that takes a position argument as a “hint” to speed up the insertion, but the exact position where the item goes is not under the caller’s control.
  - In addition to the iterator referencing the inserted item, the set’s insert function also returns a bool value indicating whether the item was inserted.
Comparison of vectors and sets (cont.)

- Unlike a vector, a set does not have an subscript operator function (`operator[]`); therefore, elements cannot be accessed by index.

- If `seta` is a set object, the expression `seta[0]` would cause the following syntax error:

  ```
  no match for 'std::set<int, std::less<int>, std::allocator<int>>[int]' operator
  ```
Although you can’t reference a specific element of a set, you can iterate through all its elements using an iterator object.

The iterator \texttt{itr} must be type \texttt{const\_iterator} because you can’t change a set’s contents using an iterator.

```cpp
// Create an iterator to seta.
for (set<string>::const_iterator itr = seta.begin();
     itr != seta.end(); ++itr) {
    string next_item = *itr;
    // Do something with next\_item
    ...
}
```
The C++ standard library defines the class `pair` in the header `<utility>`. This class is a simple grouping of two values of different types. The members are named `first` and `second`. Pairs are used as the return type from functions that need to return two values, such as the `std::insert` function, and as the element type for maps. Since all of its members are public, it is declared as a `struct`.
template<typename Type1, typename Type2>
struct pair {
    Type1 first;
    Type2 second;

    // Construct a pair from two values
    pair(const Type1& x, const Type2& y) : first(x), second(y) {}

    // Construct a pair using default values for the types
    pair() : first(Type1()), second(Type2()) {}

    // Construct a pair from two values that are assignable to the
    // target types
    template<typename Other_T1, typename Other_T2>
    pair(const pair<Other_T1, Other_T2>& other) {
        first = other.first;
        second = other.second;
    }
};
A template function is defined to create a `pair` object from two arguments

```cpp
template< typename Type1, typename Type2 >
make_pair(const Type1& first_value, const Type2& second_value) {
    return pair< Type1&, Type2& >(first_value, second_value);
}
```
The less-than operator is defined for class `pair`:

```cpp
template<typename Type1, typename Type2>
bool operator<((pair<Type1, Type2>& left, pair<Type1, Type2>& right) {
    return (left.first < right.first)
        || (!right.first < left.first)
        && (left.second < right.second));
}
```
The multiset

- The multiset is the same as the set except that it does not impose the requirement that the items be unique.
- The insert function always inserts a new item, and duplicate items are retained.
- However, the erase function removes all occurrences of the specified item because there may be duplicates.
The functions `lower_bound` and `upper_bound` can be used to select the group of entries that match a desired value.

- If the item is present, both functions return iterators:
  - `lower_bound` returns an iterator to the first occurrence of the specified value.
  - `upper_bound` returns an iterator to the smallest item that is larger than the specified value.

- The desired entries are between the iterators returned by these two functions.

- If the item is not present, both `upper_bound` and `lower_bound` return an iterator to the smallest element that is larger than the specified entry.
The following function determines the number of occurrences of the string `target` in the multiset `string` `words_set`:

```cpp
int count_occurrences(const multiset<string>& words_set,
                       const string& target) {
    multiset<string>::const_iterator first_itr =
        words_set.lower_bound(target);
    multiset<string>::const_iterator last_itr =
        words_set.upper_bound(target);
    int count = 0;
    for (multiset<string>::const_iterator itr = first_itr;
         itr != last_itr; ++itr)
        ++count;
    return count;
}
```
These functions are also defined for the set
They can be used to define a subset by setting a pair of iterators to two values within the set
For example, if the set fruits is \{"Apples", "Grapes", "Oranges", "Peaches", "Pears", "Pineapples", "Tomatoes"\}, then
\[
\text{lower_bound("Peaches")}
\]
would return an iterator to "Peaches", and
\[
\text{upper_bound("Pineapples")}
\]
would return an iterator to "Tomatoes".
These two iterators would define the subset of fruits between "Peaches" and "Pineapples"
The C++ standard library defines the class `pair` in the header `<utility>`. This class is a simple grouping of two values of different types. The members are named `first` and `second`. Pairs are used as the return type from functions that need to return two values, such as the `set::insert` function, and as the element type for `maps`. Since all of its members are public, it is declared as a `struct`. 
Maps and Multimaps

Section 9.2
The map Functions

- A map is effectively defined as a set whose items are pairs.
- The member functions defined for both are the same except for the type of the parameters.
- The map is a template class that takes the following template parameters:
  - **Key_Type**: The type of the keys contained in the key set.
  - **Value_Type**: The type of the values in the value set.
  - **Compare**: A function class that determines the ordering of the keys; by default this is the less-than operator.
  - **Allocator**: The memory allocator for key objects; we will use the library-supplied default.
Items are stored in a `map`, ordered by their `Compare` function.

If you iterate through a `map`, you get a sorted list of the contents.

The `Compare` function is used to create the function class `Key_Compare`, which compares only the `Key_Type` part of the `pair<const Key_Type, Value_Type>` items (called `Entry_Type`) that are stored in a `map`.

(The key for an entry can’t be changed, but the value can be)
struct Key_Compare {
    bool operator()(const Entry_Type& left,
                   const Entry_Type& right) const {
        return left.first < right.first;
    }
};
The map Functions (cont.)

- The map functions are all implemented by delegation to the corresponding set functions.
- In addition to the set functions, the map class overloads the subscript operator such that the key is used as an index (for example, \texttt{a_map[k]}).
- For this reason, a map is also known as an associative array.
The code that overloads the subscript operator is placed at the end of the public part of the class definition and begins as follows:

```cpp
Value_Type& operator[](const Key_Type& key) {
    std::pair<iterator, bool> ret
        = the_set.insert(Entry_Type(key, Value_Type()));
```
Because changing a value that is in a set could disrupt the ordering of the items, the `set::iterator`'s dereferencing operators always return a `const` reference to the object referenced.

We need to return a non-`const` reference to the `Value_Type` part of the `Entry_Type` object.

Thus we need to use a `const_cast` to remove the `const` qualification.

```cpp
Entry_Type& entry(const_cast<Entry_Type&>((*ret.first)));`
```cpp
#ifndef MAP_H_
#define MAP_H_

#include <set>
#include <utility>

namespace KW
{
    /** Definition of the map class using the std::set. */
    template<typename Key_Type, typename Value_Type>
    class map {

    public:

        // Define the Entry_Type.
        typedef std::pair<const Key_Type, Value_Type> Entry_Type;

        // Compare only the keys.
        struct Key_Compare {
            bool operator()(const Entry_Type& left,
                            const Entry_Type& right) const {
                return left.first < right.first;
            }
        };
    
```
The `map` Functions (cont.)

```cpp
// Define the iterator types
typedef typename std::set<Entry_Type, Key_Compare>::iterator iterator;
typedef typename std::set<Entry_Type, Key_Compare>::const_iterator const_iterator;

// Delegate the functions.
iterator begin() {
    return the_set.begin();
}

const_iterator begin() const {
    return the_set.begin();
}
```
The map Functions (cont.)

```cpp
iterator end() {
    return the_set.end();
}

const_iterator end() const {
    return the_set.end();
}

bool empty() {
    return the_set.empty();
}

int size() {
    return the_set.size();
}

std::pair<iterator, bool> insert(const Entry_Type& item) {
    return the_set.insert(item);
}
```
The map Functions (cont.)

```cpp
void erase(const Key_Type& key) {
    the_set.erase(Entry_Type(key, Value_Type()));
}

iterator find(const Key_Type& key) {
    return the_set.find(Entry_Type(key, Value_Type()));
}

// Overload the subscript operator
Value_Type& operator[](const Key_Type& key) {
    std::pair<iterator, bool> ret
        = the_set.insert(Entry_Type(key, Value_Type()));
    Entry_Type& entry(const_cast<Entry_Type&>(&(ret.first)));
    return entry.second;
}

private:
    std::set<Entry_Type, Key_Compare> the_set;

}; // End map

} // End namespace KW

#endif
```
Creating an Index of Words

- In Section 8.4 we used a binary search tree to store an index of words occurring in a term paper.
- Each element in the binary search tree consisted of a word followed by a three digit line number.
- If we store the index in a map, we can store all the line number occurrences for a word in a single index entry.
Creating an Index of Words (cont.)

- Each time a word is encountered, its list of line numbers is retrieved (using the word as key)
- The most recent line number is appended to this list
/** Reads each word (a key) in the data file in and stores it in a map along with a list of line numbers (a value).
pre:  index is an empty map.
post: lowercase form of each word with all its line numbers is stored in index.
@param in  An istream attached to the data file
@param index  The index
*/
void build_index(istream& in, map_type& index) {
    string next_line;  // Each data line
    int line_num = 0;  // Line number
    // Keep reading lines until done
    while (getline(in, next_line)) {
        line_num++;
        // Create a String_Tokenizer for the current data line
        // using punctuation and white space as delimiters
        String_Tokenizer tokenizer(next_line, " ,:-!?/%'""");
        // Insert each token in the index
        while (tokenizer.has_more_tokens()) {
            string word = tokenizer.next_token();
            to_lower(word);
            index[word].push_back(line_num);
        }
    }
}
Defining the Compare Function

- Assume that we want to use the class `Person`, with data fields `family_name` and `given_name` to be the key in a map; the `family_name` should determine the ordering of `Person` objects.

- However, if two `Person` objects have the same `family_name`, then we need to use the `given_name`.

```cpp
struct Compare_Person {
    bool operator()(const Person& p1, const Person& p2) {
        if (p1.family_name < p2.family_name)
            return true;
        else
            return (p1.family_name == p2.family_name)
                && (p1.given_name < p2.given_name);
    }
};
```
The multimap

- Like the multiset, the multimap removes the restriction that the keys are unique
- The subscript operator is not defined for the multimap
- Instead, lower_bound and upper_bound must be used to obtain a range of iterators that reference the values mapped to a given key
The `multimap` (cont.)

```cpp
void print_index(const multimap<string, int>& index) {
    for (multimap<string, int>::const_iterator itr = index.begin();
         itr != index.end(); ) {
        // Print the word
        string word = itr->first;
        cout << word << " : ";
        // Print the first line number
        cout << itr->second;
        // Print the rest of the line numbers
        multimap<string, int>::const_iterator next_word
            = index.upper_bound(word);
        ++itr;
        while (itr != next_word) {
            cout << ", " << itr->second;
            ++itr;
        }
        cout << endl;
    }
}
```
Section 9.3

Hash Tables
Open Addressing

- We now consider two ways to organize hash tables:
  - open addressing
  - chaining
- In open addressing, linear probing can be used to access an item (type `Entry_Type*`) in a hash table
  - If the index calculated for an item's key is occupied by an item with that key, we have found the item
  - If that element contains an item with a different key, increment the index by one
  - Keep incrementing until you find the key or a NULL entry (assuming the table is not full)
Algorithm for Accessing an Item in a Hash Table with Linear Probing

1. Compute the index by taking the key’s hash function modulo the size of the table.
2. if table[index] is NULL
3. The item is not in the table.
4. else if table[index]->first is equal to the key
5. The item is in the table.
   else
6. Continue to search the table by incrementing the index until either the item is found or a NULL entry is found.
Implementing the Hash Table

Section 9.4
# The `KW::hash_map` ADT

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
</table>
| iterator begin();
const iterator begin() const | Returns an iterator to the first entry in the map. |
| iterator end();
const_iterator end() | Returns an iterator to one past the last entry in the map. |
| bool empty() | Returns `true` if there are no entries in the map. |
| size_t size() | Returns the number of entries in the map. |
| pair<iterator, bool> insert(const Entry_Type& entry) | Inserts an entry into the map. If the key is not in the map, the returned iterator will reference the inserted entry, and the `bool` result will be `true`. If the key is already in the map, the returned iterator will reference the entry that is currently in the map with this key, and the `bool` result will be `false`. |
| void erase(const Key_Type& key) | Removes the item from the map. |
| iterator find(const Key_Type& key) | Returns an iterator that references the item in the map. If the item is not present, `end()` is returned. |
| Value_Type& operator[](const Key_Type& key) | Returns a reference to the value associated with key. If key is not associated with a value, inserts a default value into the map and returns a reference to the inserted object. |
The **Entry_Type**

- **Type Entry_Type is defined as follows:**

```cpp
typedef std::pair<const Key_Type, Value_Type> Entry_Type;
```
## Data Fields Class `hash_map` as Implemented by `Hash_Table_Open.h`

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hash&lt;Key_Type&gt; hash_fcn</code></td>
<td>The function object that will compute the hash function for the Key_Type.</td>
</tr>
<tr>
<td><code>size_t num_keys</code></td>
<td>The number of keys in the table, excluding keys that were deleted.</td>
</tr>
<tr>
<td><code>std::vector&lt;Entry_Type*&gt; the_table</code></td>
<td>The hash table vector.</td>
</tr>
<tr>
<td><code>static const size_t INITIAL_CAPACITY</code></td>
<td>The initial capacity.</td>
</tr>
<tr>
<td><code>const double LOAD_THRESHOLD</code></td>
<td>The maximum load factor.</td>
</tr>
<tr>
<td><code>size_t num_deletes</code></td>
<td>The number of deleted keys.</td>
</tr>
<tr>
<td><code>const Entry_Type dummy</code></td>
<td>A dummy entry to represent deleted keys.</td>
</tr>
<tr>
<td><code>const Entry_Type* DELETED</code></td>
<td>A pointer to the dummy entry.</td>
</tr>
</tbody>
</table>
/** Definition of the hash_map class. This definition is similar to the unordered_map that has been proposed for the next version of the C++ standard. **/ 
@param Key_Type The type of the keys 
@param Value_Type The type of the values 
*/ 

template<typename Key_Type, typename Value_Type>
class hash_map {

public:

// Typedefs
typedef std::pair<const Key_Type, Value_Type> Entry_Type;

// Forward declaration of iterator
class iterator;
class const_iterator;

/** Construct an empty hash_map. */
hash_map() :
    hash_fcn(hash<Key_Type>(), num_keys(0),
             the_table(INITIAL_CAPACITY, NULL),
             LOAD_THRESHOLD(0.75),
             num_deletes(0)) {}
...

private:

/** The hash function object */
hash<Key_Type> hash_fcn;

/** The number of items currently in the map */
size_t num_keys;

/** The vector containing the hash_table */
std::vector<Entry_Type> the_table;

/** The initial capacity */
static const size_t INITIAL_CAPACITY = 100;
Class hash_map as Implemented by Hash_Table_Open.h

/** The maximum load factor */
const double LOAD_THRESHOLD;

/** The number of deleted keys */
size_t num_deletes;

/** A dummy entry and a pointer to it */
static Entry_Type dummy;
static Entry_Type* const DELETED;

}; // End hash_map

template<typename Key_Type, typename Value_Type>
typename hash_map<Key_Type, Value_Type>::Entry_Type
hash_map<Key_Type, Value_Type>::dummy =
std::pair<const Key_Type, Value_Type>(
  Key_Type(), Value_Type());

template<typename Key_Type, typename Value_Type>
typename hash_map<Key_Type, Value_Type>::Entry_Type* const
hash_map<Key_Type, Value_Type>::DELETED =
  &hash_map<Key_Type, Value_Type>::dummy;
The locate Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t locate(const Key_Type&amp; key)</td>
<td>Returns the index of the specified key if present in the table; otherwise, returns the index of the first available slot.</td>
</tr>
<tr>
<td>void rehash()</td>
<td>Doubles the capacity of the table and permanently removes deleted items.</td>
</tr>
</tbody>
</table>

Algorithm for `hash_map::locate(const Key_Type& key)`

1. Set `index` to `hash_fcn(key) % the_table.size()`, where `hash_fcn` is the hash function
2. while the_table[index] is not empty and the key is not at the_table[index]
3. Increment `index` modulo `the_table.size()`
4. Return the index.
/** Locates the entry in the hash table that contains the target key or finds the next free entry. */
@par key: The key being sought
@return: The position of the key or the first empty slot if 
the key is not in the table

size_t locate(const Key_Type& key) {
    size_t index = hash_fcn(key) % the_table.size();
    while (the_table[index] != NULL 
        && (the_table[index] == DELETED 
            || the_table[index]->first != key))
        index = (index + 1) % the_table.size();
    return index;
}
The insert Function

Algorithm for `hash_map::insert(const Entry_Type& entry)`

1. Check for the need to rehash
2. Find the first table element that is empty or the table element that contains the key
3. if an empty element was found
4. Insert the new item and increment `num_keys`
5. Return `make_pair(iterator to the inserted item, true)`
6. else // key was found
7. Return `make_pair(iterator to the found item, false)`
/** Inserts an item into the map. 
   post: The key is associated with the value in the map. 
   @param entry The key, value pair to be inserted 
   @return An iterator to the inserted item and true 
                   if the entry was inserted; an iterator to the existing 
                   item and false if the item is already present 
*/

std::pair<iterator, bool> insert(const Entry_Type& entry) {
    double load_factor = double(num_keys + num_deletes) / the_table.size();
    if (load_factor > LOAD_THRESHOLD) {
        rehash(); // Double the size of the table.
    }
    // Find the position in the table.
    size_t index = locate(entry.first);
    // See whether it is empty.
    if (the_table[index] == NULL) {
        // Create a new entry.
        the_table[index] = new Entry_Type(entry);
        num_keys++;
        return std::make_pair(iterator(this, index), true);
    } else {
        // Item is already in the table.
        return std::make_pair(iterator(this, index), false);
    }
}
The Subscript Operator (operator[])

Algorithm for the Subscript Operator (operator[])

1. Call `insert` to insert a new entry consisting of the key and a default `Value_Type` object

2. Use the iterator returned from the call to `insert` to return a reference to the value that corresponds to the key
/**
 * Accesses a value in the map, using the key as an index.
 * @param key The key of the item being sought
 * @return A reference to the associated value. If the
 *         key was not in the map, a default value is inserted and
 *         a reference to this value is returned.
 */
Value_Type& operator[](const Key_Type& key) {
   // Try to insert a dummy item.
   std::pair<iterator, bool> ret = insert(Entry_Type(key, Value_Type()));
   // Return a reference to the value found or inserted.
   return ret.first->second;
}
The erase Function

Algorithm for erase (const Key_Type& key)
1. Find the first table element that is empty or the table element that contains the key
2. if an empty element is found
3. Return
4. else
5. The key is found
   Remove this table element by setting it to point to DELETED, increment num_deletes, and decrement num_keys
The rehash Function

Algorithm for rehash

1. Allocate a new hash table that is double the size
2. Reset the number of keys and number of deletions to 0
3. Reinsert each table entry that has not been deleted in the new hash table
The rehash Function (cont.)

/** Expand the table size when load_factor exceeds LOAD_THRESHOLD. 
   post: The size of the table is doubled. 
   Each nondeleted entry from the original table is 
   reinserted into the expanded table. 
   The value of num_keys is reset to the number of items 
   actually inserted; num_deletes is reset to 0. 
*/
void rehash() {
  // Create a new table whose size is double the current table.
  std::vector<Entry_Type*> other_table(the_table.size() * 2, NULL);

  // Swap this table with the current table.
  the_table.swap(other_table);

  // Reinsert all items from old table to new.
  num_deletes = 0;
  for (size_t i = 0; i < other_table.size(); i++) {
    if ((other_table[i] != NULL) && (other_table[i] != DELETED)) {
      size_t index = locate(other_table[i]->first);
      the_table[index] = other_table[i];
    }
  }
}
The Copy Constructor, Assignment Operator, and Destructor

- Because the `vector<Entry_Type*>` table contains pointers to dynamically allocated `Entry_Type` objects, we need to implement the copy constructor and assignment operators so that they make copies of the objects pointed to when a `hash_map` is copied.

- We also need to delete these dynamically allocated objects when a `hash_map` is destroyed.

```cpp
/** Copy Constructor. 
 * @param other The other map to be copied 
 */
hash_map(const hash_map<Key_Type, Value_Type>& other) :
    hash_fcn(hash<Key_Type>(), num_keys(0),
             the_table(other.the_table.size(), NULL),
             LOAD_THRESHOLD(0.75),
             num_deletes(0)) {
    for (Size_t i = 0; i < other.the_table.size(); i++) {
        if (other.the_table[i] != NULL && other.the_table[i] != DELETED)
            insert(Entry_Type(other.the_table[i]->first,
                               other.the_table[i] -> second));
    }
}
```
### Class `hash_map` as Implemented by `Hash_Table_Chain.h`

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hash&lt;Key_Type&gt; hash_fcn</code></td>
<td>The hash function object.</td>
</tr>
<tr>
<td><code>size_t num_keys</code></td>
<td>The number of keys in the map.</td>
</tr>
<tr>
<td><code>std::vector&lt;std::list&lt;Entry_Type&gt; &gt; the_buckets</code></td>
<td>A vector of lists containing the items.</td>
</tr>
<tr>
<td><code>static const size_t INITIAL_CAPACITY</code></td>
<td>The initial size of the vector.</td>
</tr>
<tr>
<td><code>static double LOAD_THRESHOLD</code></td>
<td>The maximum load factor before rehashing.</td>
</tr>
</tbody>
</table>
/* Definition of the hash_map class. This definition is similar to the unordered_map that has been proposed for the next version of the C++ standard. */

/** @param Key_Type The type of the keys */
/** @param Value_Type The type of the values */

template<typename Key_Type, typename Value_Type>
class hash_map {
    public:

    // Typedefs
    typedef std::pair<const Key_Type, Value_Type> Entry_Type;

    // Forward declaration of iterator
    class iterator;
    class const_iterator;

    /** Construct an empty hash_map. */
    hash_map();
    hash_fcn(hash<Key_Type>(), num_keys(),
             the_buckets(INITIAL_CAPACITY), LOAD_THRESHOLD(3.0)) {
    }
    ...

    private:

    /** The hash function object */
    hash<Key_Type> hash_fcn;

    /** The number of items currently in the map */
    size_t num_keys;

    /** Vector of lists containing the buckets */
    std::vector<std::list<Entry_Type>> the_buckets;

    /** The initial capacity */
    static const size_t INITIAL_CAPACITY = 100;

    /** The maximum load factor */
    const double LOAD_THRESHOLD;

}; // End hash_map
The insert Function

Algorithm for insert(const Entry_Type& entry)

1. Check for need to rehash
2. Set index to hash_fcn(key) % the_buckets.size()
3. Search the list at the_buckets[index] to find the key
4. if not found
5. Append a new entry to the end of this list
6. Return make_pair(iterator to the inserted item, true)
7. else
8. Return make_pair(iterator to the found item, false)
The **insert Function (cont.)**

```cpp
/** Inserts an item into the map.
 post: The key is associated with the value in the map.
 @param entry The key, value pair to be inserted
 @return an iterator to the inserted item and true
     if the entry was inserted, an iterator to the existing
     item and false if the item is already present
 */

std::pair<iterator, bool>
insert(const Entry_Type& entry) {

    // Check for the need to rehash.
    double load_factor = double(num_keys) / the_buckets.size();
    if (load_factor > LOAD_THRESHOLD) {
        rehash();
    }

    // Find the position in the table.
    size_t index = hash_fcn(entry.first) % the_buckets.size();

    // Search for the key.
    typename std::list<Entry_Type>::iterator pos
        = the_buckets[index].begin();
    while (pos != the_buckets[index].end())
        if (pos->first != entry.first) ++pos;
    if (pos == the_buckets[index].end()) { // Not in table
        the_buckets[index].push_back(Entry_Type(entry));
        num_keys++;
        return std::make_pair(iterator(this, index,
                                        --(the_buckets[index].end())),
                               true);
    } else { // Already there
        return std::make_pair(iterator(this, index, pos), false);
    }
}
```
The erase Function

Algorithm for `hash_map::erase(const Key_Type& key)`

1. Set index to
   \[ \text{hash	extunderscore fcn}(key) \mod \text{the	extunderscore buckets	extunderscore size}() \]
2. Search the list at `table[index]` to find the key
3. if the search is successful
4. Erase the entry with this key and decrement `num	extunderscore keys`
Because `Hash_Table_Chain.h` uses a `std::vector<std::list<Entry_Type> >` to hold the hash table:

- The default copy constructor and assignment operator will make a deep copy of the `hash_map`.
- The default destructor will delete any dynamically allocated objects.
Testing the Hash Table Implementation

- Write a method to
  - create a file of key-value pairs
  - read each key-value pair and insert it in the hash table
  - observe how the hash table is filled

- Implementation
  - Write a `to_string` method that captures the index of each non-`NULL` table element and the contents of the table element
  - For open addressing, the contents consists of the string representation of the key-value pair
  - For chaining, an iterator can traverse the linked list at the table element and append each key-value pair to the result string
Testing the Hash Table Implementation (cont.)

- Cases to examine:
  - Does the array index wrap around as it should?
  - Are collisions resolved correctly?
  - Are duplicate keys handled appropriately? Is the new value retrieved instead of the original value?
  - Are deleted keys retained in the table but no longer accessible via a `operator[]`?
  - Does rehashing occur when the load factor reaches 0.75 (3.0 for chaining)?

- Step through the `insert` method to
  - observe how the table is probed
  - examine the search chain followed to access or retrieve a key
Alternatively, insert randomly generated integers in the hash table to create a large table with little effort.

```c
for (int i = 0; i < SIZE; i++) {
    int next_int = rand();
    hash_table[next_int] = next_int;
}
```
Insertion of randomly generated integers into a table allows testing of tables of very large sizes, but is less helpful for testing for collisions.

After the table is complete, you can interactively enter items to retrieve, delete, and insert and verify that they are handled properly.
If you are using open addressing, you can add statements to count the number of items probed each time an insertion is made—these can be totaled and divided by the number of insertions to determine the average search chain length.

If you are using chaining, you can also count the number of probes made and display the average.

After all items are inserted, you can calculate the average length of each linked list and compare that with the number predicted by the formula discussed in Section 9.3.
Implementation Considerations for the hash_map

Section 9.5
Defining the Hash Function Class

/** Hash Function Objects Template */
template<typename Key_Type>
struct hash {
    size_t operator()(const Key_Type&);
};
$s_0 \times 31^{n-1} + s_1 \times 31^{n-2} + s_2 \times 31^{n-3} + \cdots + s_{n-1}$

// Specialization for string
#include <string>
template<>
struct hash<std::string> {
    size_t operator()(const std::string& s) {
        size_t result = 0;
        for (size_t i = 0; i < s.length(); i++) {
            result = result * 31 + s[i];
        }
        return result;
    }
};
Specialization for int

- Using an int value as a hash function does not tend to distribute the keys evenly
- A better approach is to multiply the int value by a large prime number and take the modulo

```cpp
// Specialization for int
template<>
struct hash<int> {
    size_t operator()(int i) {
        return size_t(4262999287U * i);
    }
};
```
Specialization for Your Own Classes

- To use objects of your own classes as keys in a `hash_map`, define the equality operator (==) and specialize the hash function class.
- The hash function is used to start the search, and the equality operator is used to finish it.
- The hash function must obey the following contract:

\[
\text{If } \text{obj1} == \text{obj2} \text{ is true, then } \text{hash<type>}(\text{obj1}) == \text{hash<type>}(\text{obj2})
\]

where `obj1` and `obj2` are objects of type `type`.

- You should make sure that your function uses the same data field(s) as your equality operator.
Class `Person` has data field `IDNumber`, which is used to determine whether two `Person` objects are equal. The equality operator returns true only if the objects’ `IDNumber` fields have the same contents.

```cpp
bool operator==(const Person& other) const {
    return IDNumber == other.IDNumber;
}
```

To satisfy its contract, function `hash<Person>` must also be specialized as follows:

```cpp
template<>
struct hash<Person> {
    size_t operator()(const Person& p) {
        return hash<string>()(p.IDNumber);
    }
};
```
# The `hash_map::iterator` and `hash_map::const_iterator`

<table>
<thead>
<tr>
<th>Data Field</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>hash_map&lt;Key_Type, Value_Type&gt;* the_parent</code></td>
<td>A pointer to the <code>hash_map</code> that this iterator is iterating through.</td>
</tr>
<tr>
<td><code>size_t the_index</code></td>
<td>The index in the table for the current entry.</td>
</tr>
</tbody>
</table>

## Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Entry_Type&amp; operator*()</code></td>
<td>Returns a reference to the <code>Entry_Type</code> that is referenced by this iterator.</td>
</tr>
<tr>
<td><code>Entry_Type* operator-&gt;()</code></td>
<td>Returns a pointer to the <code>Entry_Type</code> that is referenced by this iterator.</td>
</tr>
<tr>
<td><code>iterator&amp; operator++()</code></td>
<td>Prefix increment operator. Advances this iterator to the next entry in the table that is occupied.</td>
</tr>
<tr>
<td><code>iterator operator++(int)</code></td>
<td>Postfix increment operator. Saves the current value of this iterator. Increments this iterator and then returns the saved value.</td>
</tr>
<tr>
<td><code>bool operator==(const iterator&amp; other) const</code></td>
<td>Returns <code>true</code> if this iterator and the other iterator are the same.</td>
</tr>
</tbody>
</table>

## Private Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iterator(const hash_map&lt;Key_Type&gt;* parent, size_t index)</code></td>
<td>Constructs an <code>iterator</code> that starts at the specified <code>index</code>, or at the next occupied table entry following <code>index</code>.</td>
</tr>
<tr>
<td><code>void advance()</code></td>
<td>If the <code>_index</code> is not at an occupied position in the table, advances it to the next occupied position.</td>
</tr>
</tbody>
</table>
The `hash_map::iterator` and `hash_map::const_iterator` (cont.)

- The `const_iterator` must provide a public constructor that converts from an iterator to a `const_iterator`

- The definition of this constructor is:

```cpp
const_iterator(const typename hash_map<Key_Type, Value_Type>::iterator& other) :
  the_parent(other.the_parent), the_index(other.the_index) {}
```
The `hash_map::iterator` and `hash_map::const_iterator` (cont.)

- The other constructors for `iterator` and `const_iterator` are private because we do not want the client programs to create arbitrary iterators.

- The only valid iterator objects are ones created by the member functions of the `hash_map` that owns the iterator:
  - `iterator` and `const_iterator` classes must declare the `hash_map` as a friend.
  - `iterator` must also declare the `const_iterator` as a friend for the conversion constructor just described.
Additional Applications of Maps

Section 9.6
Implementing the Phone Directory Using a Map

- Problem
  - Use a map to obtain a more efficient implementation (better than $O(n)$) of our `Phone_Directory` ADT (previously implemented as an array and later as a vector)
Implementing the Phone Directory Using a Map (cont.)

- Analysis

- A map will associate the name (the key) with a list of phone numbers (value)

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane Smith</td>
<td>215-555-1234</td>
</tr>
<tr>
<td>John Smith</td>
<td>215-555-1234</td>
</tr>
<tr>
<td>Bill Jones</td>
<td>508-555-6123</td>
</tr>
</tbody>
</table>
Implementing the Phone Directory Using a Map (cont.)

□ Analysis

We can implement the Phone_Directory ADT by using a `map<string, string>` object for the phone directory. The `map<string, string>` object would contain the key-value pairs

\{
  ("Jane Smith", "215-555-1234"),
  ("John Smith", "215-555-1234"),
  ("Bill Jones", "508-555-6123"
}\}

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane Smith</td>
<td>215-555-1234</td>
</tr>
<tr>
<td>John Smith</td>
<td>215-555-1234</td>
</tr>
<tr>
<td>Bill Jones</td>
<td>508-555-6123</td>
</tr>
</tbody>
</table>
Implementing the Phone Directory Using a Map (cont.)

- Design

<table>
<thead>
<tr>
<th>Phone Directory API</th>
<th>map API</th>
</tr>
</thead>
<tbody>
<tr>
<td>add_or_change_entry</td>
<td>operator[]</td>
</tr>
<tr>
<td>lookup_entry</td>
<td>find</td>
</tr>
<tr>
<td>remove_entry</td>
<td>erase</td>
</tr>
<tr>
<td>load_data</td>
<td>None</td>
</tr>
<tr>
<td>save</td>
<td>None</td>
</tr>
</tbody>
</table>
The following function is slightly inefficient since we do two searches of the map:

```cpp
string Phone_Directory::add_or_change_entry(const string& name, const string& number) {
    string old_number = the_directory[name];
    the_directory[name] = number;
    modified = true;
    return old_number;
}
```
Implementation – add or change entry (revised)

The revised function:

```cpp
string Phone_Directory::add_or_change_entry(
    const string& name, const string& number) {
    string old_number = "";
    pair<iterator, bool> ret =
        the_directory.insert(pair<string, string>(name, number));
    if (!ret.second) { // Name already in the directory
        old_number = ret.first->second;
        ret.first->second = number;
    }
    modified = true;
    return old_number;
}
```
Implementing the Phone Directory Using a Map (cont.)

- Implementation – look up an entry

```cpp
/** Look up an entry.
   *
   * @param name The name of the person
   * @return The number. If not in the directory, an empty string
   */

string Phone_Directory::lookup_entry(const string& name) const {
  const_iterator itr = the_directory.find(name);
  if (itr != the_directory.end())
    return itr->second;
  else
    return "";
}
```
Implementation - remove

```cpp
string Phone_Directory::remove_entry(const string& name) {
    string old_number = the_directory[name];
    the_directory.erase(name);
    modified = old_number != string();
    return old_number;
}
```
Implementing the Phone Directory
Using a Map (cont.)

- Implementation – load data
  - The `load_data` function reads the entries from a data file and stores them in a map
  - We write the loop that does the read and store operations. It uses the subscript operator to add an entry with the given name and number

```cpp
while (getline(in, name)) {
    if (getline(in, number)) {
        the_directory[name] = number;
    }
}
```
Implementation – saving

To save the directory, we need to extract each name-number pair sequentially from the map and write them out. We can use a for loop and an iterator:

```cpp
for (iterator itr = the_directory.begin();
     itr != the_directory.end(); ++itr) {
    out << itr->first << "\n";
    out << itr->second << "\n";
}
```
Testing

To test this code, modify the `PD_Application.cpp` file to include `Map_Based_PD.h` and compile and link this modified source file with the `Map_Based_PD.cpp`

The rest of the `main` function used to test the application will be the same
Huffman Coding

- Problem
  - Build an array of (weight, symbol) pairs, where weight is the frequency of occurrence of each symbol for any data file
  - Encode each symbol in the input file by writing the corresponding bit string for that symbol to the output file
Huffman Coding (cont.)

- **Analysis**
  - For each task in the problem, we need to look up a symbol in a table
  - Using a map ensures that the lookup is expected $O(\log n)$
Huffman Coding (cont.)

- **Analysis**
  - For the frequency table, we need to read a file and count the number of occurrences of each symbol in the file.
  - The symbol will be the key, and the value will be the count of its occurrences.
  - As each symbol is read, we retrieve its map entry and increment the corresponding count.
  - If the symbol is not yet in the frequency table, the map subscript operator will insert a zero the first time we reference it.
Huffman Coding (cont.)

- Analysis
  - Once we have the frequency table, we can construct the Huffman tree using a priority queue as explained in Section 8.6
  - Then we build a code table that stores the bit string code associated with each symbol to facilitate encoding the data file
  - Storing the code table in a `map<char, Bit_String>` object makes the encoding process more efficient, because we can look up the symbol and retrieve its bit string code (O(log n) process)
  - To build the code table, we do a preorder traversal of the Huffman tree
Huffman Coding (cont.)

□ Design

**Algorithm for build_frequency_table**

1. while there are more characters in the input file
2. Read a character
3. Increment the entry in the map associated with this character
4. for each entry in the map
5. Store its data as a weight-symbol pair in the vector<Huff_Data>
6. Return the vector<Huff_Data>
**Algorithm for Function** `build_code`

1. Get the data at the current root
2. if reached a leaf node
3. Insert the symbol and bit string code so far as a new code table entry
4. else
5. Append a 0 to the bit string code so far
6. Apply the function recursively to the left subtree
7. Append a 1 to the bit string code
8. Apply the function recursively to the right subtree
Huffman Coding (cont.)

- Design

**Algorithm for Function encode**

1. while there are more characters in the input file
2. Read a character and get its corresponding bit string code
3. Write its bit string to the output file
Huffman Coding (cont.)

- Implementation

```cpp
vector<Huff_Data> Huffman_Tree::build_frequency_table(istream& in) {
  map<char, int> frequencies;
  char c;
  while (in.get(c)) {
    frequencies[c]++;
  }
  vector<Huff_Data> result;
  for (map<char, int>::iterator itr = frequencies.begin();
       itr != frequencies.end(); ++itr) {
    result.push_back(Huff_Data(itr->second, itr->first));
  }
  return result;
}
```
Huffman Coding (cont.)

- Testing
  - Download class `Bit_String` and write a main method that calls the methods in the proper sequence.
  - For interim testing, read a data file and display the frequency table to verify its correctness.
  - Use the `string` class instead of `Bit_String` in functions `build_code` and `encode` to build a code of characters ('0' or '1') instead of bits; verify its correctness.