

Computer Science II — CSci 1200

Lecture 20

Hash Table Implementation

Review from Lecture 19

- Summary of data structures we have studied thus far
- Stacks and queues, a quick introduction
- Hashing:
 - Idea
 - Hash functions
 - Hash tables
 - Collision resolution

Today's Class

- We will complete our discussion of collision resolution
- Using a hash table to implement a **set**.
 - Function objects
 - Overall design
 - Iterators
 - Fundamental operations: find, insert and erase.

A Set As a Hash Table

- The class is templated over both the key type and the hash function type.

```
template < typename KeyType, typename HashFunc >
class hash_set {
    ...
}
```

We will discuss the mechanism for templating over a hash function below.

- We use separate chaining for collision resolution. Hence the main data structure inside the class is

```
std::vector< std::list<KeyType> > m_table;
```

- We will use automatic resizing to handle the possibility that our table is too full.
 - This resize is analogous to the automatic reallocation that occurs inside the vector `push_back` function rather than the explicit `resize` function call.

Function Objects

- Our hash function is implemented as an object with a function call operator.
- The basic form of the function call operator is

```
class name {
public:
    // ...

    return_type operator() ( ... args );
};
```

where any specific number of arguments can be used.

- Here is an example of a templated function object implementing the less-than comparison operation:

```
template <typename T>
class less_than {
public:
    bool operator() ( const T& x, const T& y ) { return x<y; }
};
```

This is the default 3rd argument to `std::sort`.

- Constructors of functions objects can be used to specify parameters for use in comparisons. For example

```
class between_values {
private:
    int x, y;
public:
    between_values( int in_x, int in_y ) : x(in_x), y(in_y) {}
    bool operator() ( int z ) { return x <= z && z <= y; }
};
```

This can be used in combination with `find_if`. For example if `v` is a vector of integers, then

```
between_values low_range( -99, 99 );
if ( std::find_if( v.begin(), v.end(), low_range ) != v.end() )
    std::cout << "A value between -99 and 99 is in the vector.\n";
```

Our Hash Function Object

```
class hash_string_obj {
public:
    unsigned int operator() ( std::string const& key ) const
    {
```

```

// This implementation comes from
// http://www.partow.net/programming/hashfunctions/
unsigned int b    = 378551;
unsigned int a    = 63689;
unsigned int hash = 0;

for(unsigned int i = 0; i < str.length(); i++)
{
    hash = hash * a + str[i];
    a    = a * b;
}

return hash;
};

```

The type `hash_string_obj` is one of the template parameters to the declaration of a `hash_set`:

```
typedef hash_set<std::string, hash_string_obj> hash_set_type;
```

Iterators

- Iterators move through the hash table in the order the values are stored rather than the ordering imposed by (say) an `operator<`. The order depends on the hash function and the table size.
 - Hence the increment operators must move to the next entry in the current list or, if the end of the current list is reached, to the first entry in the next non-empty list.
- The declaration is nested inside the `hash_set` declaration in order to avoid explicitly templating the iterator over the hash function type.
- The iterator must store a pointer to the hash table it is associated with.
 - This reflects a subtle point about types: even though the `iterator` class is declared inside the `hash_set`, this does not mean an iterator automatically knows about any particular `hash_set`.
- The iterator must also store
 - The index of the current list in the hash table.
 - An iterator referencing the current location in the current list.
- Because of the way the classes are nested, the `iterator` class object must declare the `hash_set` class as a friend, but the reverse is unnecessary.

begin() and end()

- They are member functions of the `hash_set` class, but they create iterator objects.
- `begin()`:
 - It must find the first key in the table, perhaps skipping over empty lists.
 - It must tie the iterator being created to the particular `hash_set` object it is applied to. This is done by passing the `this` pointer to the iterator constructor.
- `end()`:
 - Associates the table with the iterator as well.
 - Assigns an index of -1.
 - Does not assign the particular list iterator.
- **Exercise:** Implement the `begin()` function.

Iterator Operators

- The increment operators must find the next key, either in the current list, or in the next non-empty list.
- The decrement operator must check if the iterator in the list is at the beginning and if so it must proceed to find the previous non-empty list and then find the last entry in that list.
 - While this sounds expensive, remember that the lists should be very short.
- The comparison operators must accommodate the fact that when (at least) one of the iterators is the `end`, the internal list iterator will not have a useful value.

Now we turn to the main functions of the `hash_set`.

Insert

- Computes the hash function value and then the index location.
- If the key is already in the list at the index location then no changes are made to the set, but

- an iterator is created referencing the location of the key
 - a pair is returned with this iterator and `false`
- If the key is not in the list at the index location, then the key should be inserted in the list (at the front is fine), and
 - an iterator is created referencing the location of the newly-inserted key
 - a pair is returned with this iterator and `true`
- **Exercise:** Implement the `insert()` function, ignoring for now the `resize`

Find

- This is similar to `insert`, and requires computation of the hash function and the index, following by a `std::find` operation.

Erase

- Two versions are implemented, one based on a key value and one based on an iterator.
- These are based on finding the appropriate iterator location in the appropriate list, and applying the list erase function.

Resizing the Table

- Must copy the contents of the current vector into a scratch vector.
- The current vector must be resized.
- Each key must then be inserted into the resized vector.
- **Exercise:** Write `resize_table`

Iterators and Changes to the Hash Set

- Any insert operation invalidates *all* `hash_set` iterators because the insert operation could cause a resize of the table.
- The erase function only invalidates an iterator that references the current object.

Summary

- The algorithmic steps are not particularly difficult.
- The use of C++ is more sophisticated than recent examples.
- The standard library is exploited to simplify the implementation.
- This imposes some hidden costs in the implementation, the most substantial being in `resize` where
 - lists are copied and then reallocated, and
 - hash function values are recomputed.