CSE 109: Systems Programming

Fall 2018

Program 6: **Due on Sunday, November 18th at 9pm on CourseSite.**

Checkpoint Due: **Due on Saturday, November 10th at 9pm via checkpointer.**

Collaboration Reminder:

1. You must submit your own work.
2. In particular, you may not:
   (a) Show your code to any of your classmates
   (b) Look at or copy anyone else's code
   (c) Copy material found on the internet
   (d) Work together on an assignment

Assignment: Preparation

1. Make a *Prog6* directory in your class folder.
2. Copy *sha1.h* and *sha1.o* from the *prog6student* folder.
3. You may copy the answer executable from the *prog6student* folder. Note that it only can give you a comparision for the first two parts of the assignment (not the templated portion).
4. For the first two parts, you can get a *.o* that you can link to that will allow you to use the same testing structure as the answer executable.

   Do so after you've written Hash.h with the following command:

   ```
   ~jloew/CSE109/requestProg6.pl
   ```
5. You will be writing a *Hash.h*, *Hash.cpp*, *HashGeneric.h* and *HashGeneric.cpp* file for this assignment.

   We will not be providing a tester for HashGeneric.

   This assignment is in C++, you must use a **class** and it's appropriate forms.

6. All source code files must have the comment block as shown at the end of this document. All files must be contained in your *Prog6* directory.

7. You will need to add *-lssl -lcrypto* to your compilation line in order to link in the SSL and Crypto libraries.

Background:

- The basic idea of a Chaining Hash Table is that we maintain a set of buckets (an array of buckets) and each bucket acts as an array of elements.

  When we store data, it goes into a bucket based on how the data is hashed by some hash function. A particular piece of data will always go to the same bucket (assuming we don't change the number of buckets). If we want to look for something, we hash it and look in the corresponding bucket.

- A hash function takes data and generates some non-reconstructable representation of it. For a given data, we will always generate the same result. This allows us to organize our data by what the resulting hash is and taking our number of buckets into account.

Assignment:

In this assignment, you will be creating chaining hash table data structures. The nature of the hash table is as follows:

- For the first hash table, the elements stored in the hash will be unsigned ints.

- The number of buckets in the hash is determined at construction time and can only be changed by an explicit resizing of the hash as requested by the user.

- The hash function used will be determined at construction time by passing a function pointer during construction.

  sha1.o represents a hash function that you can use for testing purposes.

- For the second hash table, the elements stored in the hash will be determined at compile time, via construction of the hash table by templates.

How we apply the hash function will also be determined at compile time, via construction of the hash table by templates.

1. **First Hash**

   (a) Define a Hash_t object:

      i. For part 1, add *#define NOITERATE* to your *Hash.h* file.
      ii. It will hold objects of type: unsigned int.
      iii. The number of buckets in the Hash_t will be defined at the creation time of the Hash_t object - use 10 if the user does not explicitly provide this value.
      iv. Each bucket is a list of elements. You should use *vector* for this.
      v. A function pointer that will be used to hash an element. The function pointer is for a function that takes *(const void*, size_t)* as arguments and returns an int: *int (*)(const void*, size_t)*. If the constructor does not provide this, assign it to the *sha1_hash* provided.
      vi. A size_t called *_size* to track the number of elements within the hash.
      vii. None of the instance variables are permitted to be publically accessible.

   (b) Constructors:

      i. Default constructor: As per description.
      ii. Hash_t(size_t buckets): Use the bucket size given.
      iii. Hash_t(size_t buckets, int(*hashfunc)(const void*, size_t)): Use the bucket size given and use *hashfunc* as the function for hashing.

         This will be used to provide the a hash function to your Hash_t object.

      Initialize your buckets. Initialize *_size* to 0 - use this to keep track of how many elements are in the hash.

   (c) Copy constructor: Deep copies a Hash_t object.

   (d) Hash_t& operator=(const Hash_t&): Deep assigns a Hash_t object.

(e) bool insert(unsigned int): Inserts an unsigned int element into the hash. True if inserted. No duplicates are allowed. Insert the data in numeric order within the bucket.

(f) Hash_t& operator+=(unsigned int): similar to insert(unsigned int)

(g) bool find(unsigned int) const: Finds an unsigned int element in the hash. True if found.

(h) bool remove(unsigned int): Removes an unsigned int element from the hash. True if removed.

(i) Hash_t& operator-=(unsigned int): similar to remove(unsigned int)

(j) string to_string() const: Creates a string with all of the bucket data, see sample output. This assumes the hash holds unsigned ints

(k) size_t size() const: Returns the number of elements in the hash.

(l) bool resize(size_t): Changes the number of buckets to the specified amount. Returns false is 0 is provided, otherwise must be successful. All elements in the hash must end up in their correct locations and we must not leak memory.

(m) friend ostream& operator<<(ostream&, const Hash_t&): Overloaded output operator for the hash, uses to_string().

(n) private: getIndex(const unsigned int&) const: Calls the hash function for the hash object and mods it by the number of buckets.

2. **First Hash: Part 2: Iteration**

   (a) This part will have you provide basic iteration semantics to the Hash.

   (b) Remove *#define NOITERATE* from your *Hash.h* file.

   (c) Methods new to the Hash:

      i. iterator begin(); Returns an iterator object that reflects the first element in the hash (first element found in earliest bucket).

      ii. iterator end(); Returns an iterator object that reflects the end of the hash. This is technically out-of-bounds. We need it to know when the iterator we get from begin has reached the end of our data.

iii. iterator remove(iterator& it): Removes the element that the iterator refers to from the hash. Returns an iterator to the next logical element.

(d) The Hash will also define a subclass, class iterator.

i. The first statement within the subclass should be "friend class Hash_t". This will allow the Hash access to the private fields of the iterator. (you may find this useful). It is not required though.

ii. Instance data that allows you to know where you current are within the Hash. This data must be private. You will also want a pointer or a reference to the Hash's buckets so you can view the elements (non-const).

Note that external operations, such as insert/remove invalidate all existing iterators, this is also why Hash_t::remove returns an iterator, to replace the invalidated one.

iii. Explicit construction: Construct the iterator with whatever instance data you use to determine where you are within the Hash.

iv. iterator& operator=(const iterator&): Assignment operator.

v. bool operator!=(const iterator&) const: Non-equivalence operator.

vi. bool operator==(const iterator&) const: Equivalence operator.

vii. iterator operator+(size_t amt) const: Returns a new iterator, moved ahead by amt elements. Undefined if it ends up going past the end iterator.

viii. iterator& operator+=(size_t amt): Returns the current iterator, moved ahead by amt elements. Undefined if it ends up going past the end iterator.

ix. iterator operator-(size_t amt) const: Returns a new iterator, moved back by amt elements. Undefined if it ends up going out of bounds.

x. iterator& operator-=(size_t amt): Returns the current iterator, moved back by amt elements. Undefined if it ends up going out of bounds.

xi. iterator& operator++(): Returns the current iterator, moved ahead by one element. Undefined if it was equivalent to the end iterator. Prefix.

xii. iterator& operator−−(): Returns the current iterator, moved back by one element. Undefined if it was equivalent to the begin iterator. Prefix

xiii. iterator operator++(int): Returns an iterator, moved ahead by one element. Undefined if it was equivalent to the end iterator. Postfix.

xiv. iterator operator−−(int): Returns an iterator, moved back by one element. Undefined if it was equivalent to the begin iterator. Postfix.

xv. const unsigned int& operator*() const: Returns a const reference to the element that the iterator refers to. Don't let the user change our data!

3. **Second Hash**

Note that we will only concern ourselves with the differences between the first and second hash. You may copy your first hash files into the second hash and then convert it to a templated class.

(a) Define a Hash_t object:

i. It will be templated:

```
template<typename T, pair<const void*, size_t>(*D)(const T&),
 int(*H)(const void*, size_t)>
```

- It will hold objects of type T.
- It will be defined with a function that determines what data needs to be hashed.
- It will be defined with a hash function that will be used by the hash object.
- Since the hash function used will be provided via the template mechanism, it is no longer an argument at construction time.

ii. At compile time, we must provide a function that returns a pair object containing two values. The first being a const void* and the second being a size_t that reflects how much data the void* is pointing to. This will be used by the hash function to determine where the data goes. This is provided as part of the template, and not part of construction.

iii. An example for string:

```
pair<const void*, size_t> stringHash(const string& str)
```

```
{
    const void* data = str.c_str();
    size_t size = str.size();
    return make_pair(data, size);
}
```

iv. Because our code is a template, at the end of the header file, but before the *#endif*, you will include the .cpp file. You will not compile the .cpp file for the template, just including the .h file is sufficient in these cases.

v. Our Hash will only work with objects that have certain behaviors defined, such as equality, copy construction, comparision, overloaded output.

(b) Hash_t(size_t): Default constructor no longer takes a hash function argument.

(c) Hash_t<T,D,H>& operator=(const Hash_t<T,D,H>&): Deep assigns a Hash_t<T> object.

(d) bool insert(T): Inserts a T element into the hash. True if inserted. No duplicates are allowed. Insert the data in order within the bucket.

(e) Hash_t<T,D,H>& operator+=(T): similar to insert(T)

(f) bool find(T) const: Finds a T element in the hash. True if found.

(g) bool remove(T): Removes a T element from the hash. True if removed.

(h) Hash_t<T,D,H>& operator-=(T): similar to remove(T)

(i) string to_string() const: Note that this should probably use a stringstream to create what you want.

(j) friend ostream& operator<<(ostream&, const Hash_t<T1,D1,H1>&): Overloaded output operator for the hash, uses to_string().

This must be templated at declaration and definition time.

(k) private: getIndex(const T&) const: Calls the hash function for the hash object and mods it by the number of buckets.

Sample output for operator<<

Note the space before the single-digit indices. The indices should be lined up neatly based on the width of the largest index value. Do not line up the data elements within the buckets.

```
 0:  (Empty)
 1:  (Empty)
 2:  (Empty)
 3:  (Empty)
 4:  (Empty)
 5:  (Empty)
 6:  (Empty)
 7:  18
 8:  71  19
 9:  33  20
10:  34
11:  (Empty)
12:  23
```

Checkpointing:

1. The following methods of the unsigned int *Hash_t* object (Hash.cpp and Hash.h) are due at checkpoint time:
   (a) Constructors (all)
   (b) Destructor (if necessary, no memory leaks should happen)
   (c) insert(unsigned int)
   (d) Hash_t& operator+=(unsigned int)
   (e) bool find(unsigned int)
   (f) string to_string() const
   (g) size_t size() const
   (h) friend ostream& operator<<(ostream&, const Hash_t&)
2. To submit your checkpoint:

```
cp Hash.h checkpoint6.c
cat Hash.cpp >> checkpoint6.c
~jloew/CSE109/submitCheckpoint.pl 6
```

   That is lowercase PL, followed by the number 6.

3. You may submit your Checkpoint up to ten times before the deadline.

4. Ideally, it will tell you which functions are incorrect. It is possible that the functions are incorrect but pass the checkpoint. Although, they should be mostly correct or completely correct if they pass the checkpoint.

5. The checkpoint will not check for memory corruption or leaks. You will need to handle that yourself.

Style:

For assignments, we follow the Allman style of braces and indentation.

1. **Review the Style document on Coursesite**

Submission:

1. Once ready to submit, you can package up the assignment as a .tgz file

   tar -czvf Prog6.tgz Prog6

   You must use this command in the directory that contains the *Prog6* folder, not within the directory.

2. Transfer *Prog6.tgz* to the Program 6 submission area of CourseSite.

Comment Block:

```
/*
    CSE 109: Fall 2018
    <Your Name>
    <Your user id (Email ID)>
    <Program Description>
    Program #6
*/
```