Project 4
Steg-o-matic

Time due: 9 PM Thursday, March 12th

Before writing a single line of code, you must first read AND THEN RE-READ the *Requirements and Other Thoughts* section.

Introduction

The NachenSmall Software Corporation has been contacted by the Paranoid Students Users Group (PSUG), whose members believe that they're constantly being watched by "the man," to create a tool that helps them to communicate with each other in a secret manner. More specifically, the PSUG would like NachenSmall's team to build a steganography system capable of embedding hidden secret messages within ordinary web pages.

Steganography is the practice of embedding a digital message or file within another "carrier" digital medium. For example, you could encode a secret message like "Meet me in Ackerman at noon" by making certain very minor color changes in the pixels of a JPG image. The idea behind such a steganographic encoding is that someone viewing the carrier medium (e.g., the JPG) would not even know that it contains a secret message, so the carrier medium can be freely distributed or posted on the Internet for all to see. Yet the recipient, possessing the knowledge that the carrier medium actually contains an embedded message, and equipped with the proper tools, can extract that message and read it. Steganography has been used for years by spies, terrorist organizations, and almost certainly by paranoid individuals like those in PSUG[1].

So, in your last project for CS32 Winter 2015, your goal is to build a simple set of C++ classes that can be used to implement a steganography system. If you're able to prove to NachenSmall's reclusive and bizarre CEO, Carey Nachenberg, that you have the programming skills to build the simple steganography tool described in this specification, he'll hire you to build the complete project, and you'll be famous… at least in the paranoid student community.

Anatomy of a Steganography System

Steganography tools have two modes of operation:

1. Encoding a message/file within a carrier medium like a web page or JPG, and
2. Decoding or extracting a previously encoded message/file from a carrier medium

Encoding a Message

The steganographic tool that you will build encodes a secret message within a carrier web page, using the following steps:

[1] In fact, steganography is a key element in Carey Nachenberg's upcoming techno-thriller novel, *The Florentine Deception* – get your copy on April 14, and Carey will sign it!

1. It compresses the secret message, like “Meet me in Ackerman at noon”, into a more compact numeric form, like 7234 25115 3451 58345 45636. We will describe exactly how to do this in the specification below.
2. It converts the resulting series of numbers into a binary form, a string of 1 and 0 characters that represents the compressed message.
3. It converts this long string of 1s and 0s into a string made up of tabs and spaces (which in this specification we will make visible by representing them with – and _ respectively), so “101101” might be converted into “-_--_-”.
4. It retrieves a web page from the Internet that the user has chosen to use as a carrier file to hold the secret message.
5. It splits the web page into separate lines (where each line is terminated by a newline character).
6. It strips all existing whitespace (tabs and spaces) from the end of each line of the original web page, so, for example the line “<h1>Welcome to my web page</h1>_-_-” (the line ends with space tab space tab) would become “<h1>Welcome to my web page</h1>”.
7. It splits up the tabs-and-spaces string that encodes the secret message into substrings, one for each line of the web page, and appends each substring onto the end of a stripped line of that web page.
8. It joins the all of the updated lines of the web page into one string, which is now the carrier of the encoded message. This string might be saved to a file to be served up by a web server.

Someone looking at the newly-created web page in a browser will see nothing suspicious. The rare person who might look at the HTML source of the web page might not notice the extra tabs and spaces at the ends of lines.

Decoding a Message

To extract and decode a secret message that was previously hidden within a web page, the steganographic tool will use the following steps:

1. It retrieves the carrier web page from the Internet. This carrier HTML file was presumably posted on the Internet by the user that created it, and it holds a previously-encoded message.
2. It breaks up the file into different lines (where each line is terminated by a newline character).
3. From each line, it extracts all of the whitespace from the end of the line – this is where the secret message was added during the encoding step.
4. It concatenates all of the whitespace extracted from all of the lines together into a single long string of tabs and spaces.
5. It converts this tabs-and-spaces string into a string of 1 and 0 characters.
6. It converts the resulting string in binary form into a series of numbers, like 7234 25115 3451 58345 45636.

7. It uses a decompression algorithm to expand these numbers back into the original secret message, e.g., “Meet me in Ackerman at noon”.
8. It returns the original secret message to the user.

What Do You Need to Do?

So, at a high level, what do you need to build?

You’ll be building five complete classes and two new stand-alone functions, described at a high level below. Detailed specs follow in the later sections.

You need to build a hash table class template *HashTable*:

1. You need to be able to create a new *open hash table* with a specified number of buckets and a specified capacity.
2. You need to be able to add new associations (each association is a key→value mapping) to the hash table, e.g., 1234→“meet”, or “Ackerman”→45235.
3. You need to be able to efficiently search for an association in the hash table given a key.
4. You need to be able to determine if the hash table has reached its maximum capacity.
5. You need to be able to efficiently mark an association in the hash table as having been recently written (i.e., added or modified).
6. You need to be able to efficiently discard the least recently written association from the hash table. (This will be used if the hash table is full and its client needs to remove an association from the table to make room for a new one.

You need to build a class *Compressor*:

1. This class enables you to compress a secret message, like “Meet me in Ackerman at noon”, into a series of numbers, like 7234 25115 3451 58345 45636.
2. This class enables you to decompress a previously-created series of numbers, like 7234 25115 3451 58345 45636, back into the original secret message, “Meet me in Ackerman at noon”.

Your *Compressor* class **must** produce the same result as our prescribed compression and decompression algorithms exactly, or your program will not function properly with our test tools and you will receive zero credit on this part of the project. Your *Compressor* class must use your *HashTable* class template when compressing and decompressing – we’ll tell you exactly how that works in the sections below.

You need to build a class *BinaryConverter*:

1. Given a vector of unsigned short integers, this class must be able to convert the vector into a string containing just tabs and spaces, representing the binary form (1s and 0s) of the integers in the vector.
2. Given a string containing just tabs and spaces, this class must be able convert this back into its binary equivalent, and from there, back into a vector of unsigned short integers.

Since you haven't learned binary encoding yet, we will also provide you with a few functions to help you convert a single unsigned short into a binary string, and vice versa.

You need to build a class *Steg*:

1. This class allows its client to hide a secret message, like "Meet me in Ackerman at noon", within a string. This involves:
    a. Compressing the secret message using the *Compressor* class into a sequence of unsigned short integers .
    b. Converting those integers into their binary equivalent, and from that into to a corresponding string of tabs and spaces.
    c. Stripping existing tabs and spaces from the end of each line of the string.
    d. Appending the tabs and spaces that represent the secret message onto the ends of the lines of the stripped lines.
    e. Combining the lines and returning the string that contains the hidden message.
2. This class allows its client to extract and decode a secret message that was previously encoded inside a carrier string. This involves:
    a. Extracting the tabs and spaces from the end of each line of a string that previously had a message hidden in it, and creating one long string of tabs and spaces.
    b. Converting the tabs-and-spaces string to its binary equivalent, and then converting the binary string back to the original sequence of integers
    c. Using the *Compressor* class to decompress these integers into the original secret message.
    d. Returning the revealed secret message string.

You need to build a class *WebSteg*:

1. This class allows its client to specify a URL for a web page and a secret message, and produces a modified version of that page that contains the hidden message.
    a. The class must connect over the Internet to retrieve the web page specified by the URL (using the HTTP interface that we provide).
    b. The class must use the *Steg* class to hide the provided secret message within the page.
    c. The class must return a string with the content of the resulting page.

2. This allows its client to specify a URL for a web page that already contains a secret message embedded within it, returns the decoded secret message.
    a. The class must connect over the Internet to retrieve the web page specified by the URL. (This page contains the steganographically-encoded secret message.)
    b. The class must use the *Steg* class to extract and decode the secret message that is embedded within the page
    c. The class must return a string with the decoded secret message.

You need to create two overloaded hash functions, each named computeHash():

1. One *computeHash()* function must take a string argument and compute an unsigned integer hash value from that string.
2. The other *computeHash()* function must take an unsigned short integer and compute an unsigned integer hash value for this unsigned short integer. (This should be a pretty simple function!)

What Will We Provide?

We'll provide a simple *main.cpp* file that lets you test your overall steganography implementation.

We'll provide some helper code for you in *BinaryConverter.cpp* that converts unsigned short values to binary, and vice versa.

We'll provide an *HTTP* class that can be used to download a web page from a web server on the Internet (e.g., from *http://reddit.com*). If you specify the URL for a page, it will download the contents of the page and place them into a string.

The HTTP Class (aka But I don't know how to use C++ to access the Internet!)

Oh, we knew you were going to say that! Such a whiner! But wouldn't you like to learn how to write a program that interacts with other computers over the Internet? We thought so. So we're going to provide you with a reasonably functional Internet HTTP interface that is capable of downloading pages off of the Internet for you. HTTP is the protocol used by web browsers to download web pages from servers on the Internet into your browser.

When you use our interface, you don't have to worry about the details of how to communicate over the Internet yourself. Of course, if you want to see how our interface works, you're welcome to do so… and before you know it, you'll be forming your own start-up Internet company to compete against Google[2]. Our HTTP interface's primary public function (get) is as easy to use as this:

```
#include "http.h"

int main()
{
    string url = "http://en.wikipedia.org/wiki/Bald";
    string page;  // to hold the contents of the web page

      // The next line downloads a web page for you.  So easy!
    if (HTTP().get(url, page))
        cout << page;  // prints the page's data out
    else
        cout << "Error fetching content from URL " << url << endl;
    ...
}
```

Note that you don't need to declare an HTTP variable. The call above looks as if it calls a function named HTTP, then calls a *get()* member function on what it returns.

A challenge when testing a program that analyzes the contents of web pages is that you have no control over those contents. Our HTTP interface lets you set up a pseudo-web of pages with URLs and contents of your choosing:

```
int main()
{
    HTTP().set("http://a.com", "This is a test page.");
    HTTP().set("http://b.com", "Here is another.");
    HTTP().set("http://c.com", "<html>Everyone loves CS 32</html>");
    string page;
    if (HTTP().get("http://b.com", page))
        cout << page << endl;  // writes  Here is another.
}
```

You call *set()* to associate a URL with a string. From that point on, calling *get()* with that URL will retrieve that string. (Once you call *set()*, *get()* will no longer retrieve pages from the real web; it will instead consult only the pages that you installed with *set()*.)

Details: The Classes You Must Write

You must write correct versions of the following classes to obtain full credit on this project. Your classes must work correctly with our provided code, and you must **not** modify our provided code to make it work with your code. Doing so will result in a **zero score** on that part of the project.

[2] By your agreeing to use our HTTP code for Project 4, this license entitles NachenSmall to a 20% cut of all profits.

HashTable Class Template

You must write a class template named *HashTable* that lets a client associate items of a key type with items of a (usually different) value type, with the ability to look up items by key efficiently. Your implementation **must** use an *open hash table*.

A HashTable whose key type is string and value type is int, for example, lets a client add A number of key→value associations (e.g., "carey"→1, "david"→2, "lily"→3, etc.), and allows the client to search using a particular key and find the associated value.

When you construct a new *HashTable* object, you must specify both the number of *buckets* in the hash table, as well as its *capacity*, the maximum number of associations that can be stored in the table. If a *HashTable* object is filled to its capacity and the client tries to add an association with a key that's not already present in the table, then the addition will fail. However, even if the hash table is at capacity, the client may still update an association that has an existing key in the table by replacing its associated value with another value.

Of course, when you insert an association into a hash table, you need to be able to compute the hash of the key for that association. Therefore, in addition to defining your *HashTable* class template, you must also define a non-member function named *computeHash* for each type of key you wish to put into your hash table. Look for examples of these functions in the code snippets below.

For example, someone could use your *HashTable* class template to associate the following key strings with integer values:

```
#include "HashTable.h"

  // here's a computeHash() function that hashes a C++ string to an
  // unsigned int hash value

unsigned int computeHash(string s)
{
      unsigned int hashValue;

        // Here you'll write code that computes an unsigned int hash
        // value from a C++ string, and returns it.  A HashTable whose
        // key type is string will call this function as part of
        // determining a bucket number.

      return hashValue;
}

void argh()
{
      // create a hash table object that has 100 buckets
      // and has a capacity of 200 associations

      HashTable<string, int> nameToAge(100, 200);
```

```
	if ( ! nameToAge.set("Carey", 43) )
		cout << "Error associating Carey with 43!\n";
	else
		cout << "Carey is mapped to 43\n";

	if ( ! nameToAge.set("David", 97)
		cout << "Error associating David with 97!\n";
	else
		cout << "David is mapped to 97\n";

	int age;
	if ( nameToAge.get("Carey", age) )
		cout << "Carey is associated with age: " << age;
	else
		cout << "Unable to find an association for Carey\n";

	…
}
```

Here’s another example:

```
  // hashes a Rectangle value to an unsigned int hash value
unsigned int computeHash(Rectangle r)
{
        // Here’s an example of how we might compute a hash for a
        // rectangle.

	return r.getX() + r.getY() + r.getWidth() + r.getHeight();
}

void hashABunchOfRects()
{
	Rectangle a(2, 5, 10, 20);
	Rectangle b(0, 0, 30, 40);
	Rectangle c(10, 7, 100, 15);

	HashTable<Rectangle,unsigned int> rectToArea;

	rectToArea.set(a, a.getArea());  // maps a to its area
	rectToArea.set(b, b.getArea());  // maps b to its area
	rectToArea.set(c, c.getArea());  // maps c to its area

	unsigned int area;
	if ( rectToArea.get(b, area) )
		cout << "The area of rectangle b is: " << area;
}
```

Note that the *HashTable* class must maintain no more than one association with a particular key, so if a client associates “Joe”→5 and then asks to associate “Joe”→17, there will be only on association in the table with the key “Joe”, and that association will have the value 17. The earlier mapping to 5 was replaced by the later mapping to 17.

Unfortunately for you, you're not just going to implement a conventional open hash table for this problem. There's a special requirement that will make implementing this class a bit (a lot?) more challenging.

Your version of *HashTable* must be able to track how each association in the hash table was written (i.e., added or modified). So for instance, imagine that we add a bunch of associations into a *HashTable* instance over time:

```
void argh()
{
      HashTable<string, int> nameToAge(100, 200);

        // add a bunch of associations in this order
      nameToAge.set("Carey", 43);
      nameToAge.set("David", 97);
      nameToAge.set("Lily", 18);
      nameToAge.set("Sally", 22);
      nameToAge.set("David", 55);
}
```

The resulting *HashTable* object will hold:

Carey → 43
Lily → 18
Sally → 22
David → 55

David's association will be the most recently written association in the HashTable, with Sally's being the next most recently written one, then Lily's, and then Carey's. Your *HashTable* class template must track the order of how recently each association has been written within the table.

Specifically, any time you *add* or *update* an association, that association becomes the most recently written one, and all earlier-written items fall down one position in the ranking.

There is one exception to this rule. When adding an association with a key not already in the hash table using the *set()* method, you may designate the item as being "permanent." Permanent items ***must not*** be tracked in your hash table's recently written association list. The client of your class may designate an item as permanent by passing a third argument value of *true* to your *set()* method when adding a new association. (A third argument of *false*, just like no third argument at all, means the association is not permanent, and its recency of being written must be tracked.)

```
void argh()
{
      HashTable<string, int> nameToAge(100,200);

        // add a bunch of associations in this order
      nameToAge.set("Carey", 43, true);   // permanent
```

```
        nameToAge.set("Lily", 18);
        nameToAge.set("Sally", 22);
        nameToAge.set("David", 97, true);    // permanent
        nameToAge.set("Diego",17);
        nameToAge.set("Sally",23);
}
```

In the above example, both Carey and David's associations would be designated as permanent in your hash table and must not be tracked in your recently-written list. Lily's, Sally's and Diego's associations would be tracked in your list, with Sally's being the most recently written, then Diego's, and finally Lily's.

You must also enable the client of your *HashTable* class to be able to manually "touch" an existing non-permanent item that is already in the *HashTable* without actually changing its value, to indicate that it should be brought to the top of the most recently written list:

```
void argh()
{
        HashTable<string, int> nameToAge(100,200);

        nameToAge.set("Cameron", 25);
        nameToAge.set("Lily", 18);
        nameToAge.set("Timothy", 43, true);
        nameToAge.set("Sally", 22);
        nameToAge.set("Mikey", 28);

          // This will move Cameron to the top of the most-recently
          // written list
        nameToAge.touch("Cameron");
}
```

After the last "touch" line above, Cameron's association will be the most recently written one in the HashTable, followed by Mikey's, Sally's, and Lily's. Timothy's will not be in the recently-written list, since it is marked as permanent.

Touching a permanent item in the hash table has no effect on the association, since it's not being tracked in your recently-written list.

Note that only *adding*, *updating* or *touching* a non-permanent association may adjust that associations's position in the recently-written list. Simply looking up an association in the hash table, for example, **must not** change the assocation's position in the recently-written list.

Why would you want to track when each item in the *HashTable* was last written? Well, it is possible that the client of your hash table will eventually insert a large number of associations and the table will reach its capacity. In this situation, the client may wish to discard one or more old associations from the hash table to make room for new ones. By maintaining a log of which items were least recently written, you can easily allow the client to discard old associations to make room for new ones. Your *HashTable* class template must therefore provide a *discard()* method for this purpose.

Each time it is called, the *discard()* method must discard the least recently written non-permanent association in the hash table (removing it completely), assuming there is at least one non-permanent association in the hash table. Here's how it might be used:

```
unsigned int computeHash(std::string s)
{
        // Don't forget to write this!
}

void argh()
{
      HashTable<string, int> nameToAge(100, 200);

      nameToAge.set("Carey", 43);
      nameToAge.set("David", 97);
      nameToAge.set("Timothy", 43, true);
      nameToAge.set("Ivan", 28);
      nameToAge.set("Sally", 22);
      nameToAge.set("David", 55);

      nameToAge.touch("Carey");

      // let's discard the two least recently written items

      for (int k = 0; k < 2; k++)
      {
            string discardedName;
            int discardedAge;
            if (nameToAge.discard(discardedName,discardedAge))
                  cout << "Discarded " << discardedName
                       << " who was " << discardedAge
                       << " years old.\n";
            else
                  cout << "There are no items to discard!\n";
      }
}
```

The first loop iteration will discard Ivan's association, since his was the least recently written one in the hash table. The second loop iteration would discard Sally's association, since after Ivan's was discarded, hers was the least recently written one.

After the loop is completed, the hash table holds just Timothy's, David's, and Carey's associations, with Carey's being the most recently written one, and David's being the least recently written one. Timothy's, being permanent, would not be in the recently-written list. David's would be the next association discarded if *discard()* were called again.

The HashTable Class Template: Details

Your *HashTable* implementation **must** have the following public interface; you must **not** change or add to the public:

```
template <typename KeyType, typename ValueType>
class HashTable
{
public:
  HashTable(unsigned int numBuckets, unsigned int capacity);
  ~HashTable();
  bool isFull() const;
  bool set(const KeyType& key, const ValueType& value, bool permanent = false);
  bool get(const KeyType& key, ValueType& value) const;
  bool touch(const KeyType& key);
  bool discard(KeyType& key, ValueType& value);

private:
    // We prevent a HashTable from being copied or assigned by declaring the
    // copy constructor and assignment operator private and not implementing them.
  HashTable(const HashTable&);
  HashTable& operator=(const HashTable&);
};
```

Here are the general requirements for your *HashTable* class:

1. You **must** implement your own open hash table in your *HashTable* class template (i.e., define your own *Node* struct/class, maintain an array of pointers to nodes, etc.).
2. You **must not** use any STL containers to implement *HashTable* (e.g., no map, set, unordered_map, unordered_set, vector, list, queue, stack, etc.).
3. Your *HashTable* class template **must** use the public interface documented above. You may add only private members; you must **not** add other public members to *HashTable*. Doing so will result in a score of **zero** for this part of the project.
4. You may assume that the key type of any instantiation of the *HashTable* class template has the comparison operators == and != defined for it (certainly ints and strings do). If strings are keys, they are case-sensitive, so "bill" and "Bill" would be different keys.
5. You **must** define one non-member *computeHash* function for every type of key you wish to store in a hash table. These non-member functions may be implemented either in your *HashTable.h* file with the *inline* keyword or in your .cpp files.

Requirements for `HashTable(unsigned int numBuckets, unsigned int capacity)`

The constructor must allocate and initialize an empty hash table with the specified number of buckets. It must also remember the capacity of the hash table. It must run in O(B) time, where B is numBuckets.

Requirements for `~HashTable()`

The destructor must free all memory associated with the hash table's nodes. This method must run in O(B+N) time, where N is the number of items in the hash table and B is the number of buckets in the hash table.

Requirements for `bool isFull() const`

The *isFull()* method must return *true* if the hash table is full to capacity (i.e., the number of associations in the hash table is equal to the capacity), otherwise *false*.

Requirements for `bool set(const KeyType& key, const ValueType& value, bool permanent = false)`

Let B be the number of buckets in the hash table, and C be its capacity. If the indicated key is not already in the table: If the table already contains C associations, the *set()* method returns *false* without changing anything; otherwise it adds a new association key→value, which will be permanent if the third argument is *true*, otherwise non-permanent.

If an association with the indicated key is already in the table, it is updated with the indicated value replacing the value currently stored in that association. In this case, the third argument is ignored; the permanent/non-permanent status of the association may or may not be what is indicated by the third parameter, and is not, for example, changed because of it.

Each non-permanent, newly added or updated association must be made the most recently written item in the recently-written list. Permanent associations must not be tracked in the recently-written list. The method return *true* if an association was added or updated.

Attempting to add or update an association must run in O(C/B) time, as must updating the recently-written list.

Requirements for `bool get(const Ketyype& key, ValueType& value) const`

If the indicated key is not in the table, the *get()* method returns *false* and the second parameter is unchanged. Otherwise, the second parameter is set to the value associated with the key in the table, and the method returns *true*. This method does **not** adjust the order of any items within the recently-written list.

If B is the number of buckets in the hash table, and C is its capacity, this method must run in O(C/B) time.

Requirements for `bool touch(const KeyType& key)`

If a non-permanent association in the table has the indicated key, the *touch()* method moves it to the top of the recently-written list, just as if the association had actually been updated, and returns *true*. Otherwise, it does nothing and returns *false*.

If B is the number of buckets in the hash table, and C is its capacity, this method must run in O(C/B) time.

Requirements for `bool discard(KeyType& key, ValueType& value)`

If there are no non-permanent associations in the table, the *discard()* method does nothing (leaving its parameters unchanged) and returns *false*. Otherwise it must identify the least recently written non-permanent association in the table. It must set the key and value parameters to the key and value of that association, delete that association from the table (ensuring that the recently-written list is still consistent so that future calls to *discard()* will properly discard the next least-recently used association), and return *true*.

If B is the number of buckets in the hash table, and C is its capacity, this method must run in O(C/B) time.

Requirements for your *computeHash()* functions

So what are these *computeHash()* functions that you need to define?

Well, as we learned in class, when using a hash table, you need some way of determining in which bucket to find the information associated with a given key. This is done with a hash function.

The hash function is expected to take a key as its argument and produce an unsigned integer as its result. That result can then be used to compute the bucket number in the hash table.

If you're using several hash tables, you must have a different hash function, each named *computeHash()*, for every *type* of key you will be using in those hash tables. If a application needs two hash tables that use ints as keys, five that use strings as keys, and one that uses *Rectangle*s as keys, you'll need three *computeHash()* functions, one for each type. The five hash tables that use strings as keys will all use the same *computeHash()* function, the one for strings.

As an example, for this *HashTable* that maps *Rectangle* objects to *Color*s:

```
HashTable<Rectangle, Color> rectToColor;
```

you'd need to define a hashing function that is capable of computing an unsigned integer value for given a *Rectangle*; here's one possibility:

```
unsigned int computeHash(Rectangle r)
{
```

```
        unsigned int hashValue = r.getX() + r.getY() +
                                 r.width() + r.height();
        return hashValue;
}
```

If you wanted to have a *HashTable* mapping *Student* objects to GPAs (a double):

```
HashTable<Student, double> gpas;
```

you'd need to define a hashing function that is capable of computing an unsigned integer value for each *Student*; here's one possibility:

```
unsigned int computeHash(Student stud)
{
        return stud.getID();  // use student ID as hash value
}
```

These *computeHash()* functions must be defined as non-member functions implemented either in your *HashTable.h* file with the *inline* keyword or in your .cpp files.

When you write your *HashTable* class template, you can then assume that a client of your *HashTable* will supply a suitable *computeHash()* function for the type of key the table will use. You can use it determine which bucket to put a new association into. Here's a little hint how this might be used:

```
template<typename KeyType, typename ValueType>
class HashTable
{
        …
        unsigned int getBucketForKey(const KeyType& key)
        {
                // The computeHash function must be defined
                // for each type of key that we use in some
                // hash table.
               unsigned int computeHash(KeyType); // prototype
               unsigned int result = computeHash(key);
               unsigned int bucketNum =
                            determineBucketFromHash(result);
               return bucketNum;
        }
        …
};
```

By using this approach, you enable your *HashTable* class to work the same way no matter what C++ type is being used as the key. **Important note**: Before calling *computeHash*() in a *HashTable* member function, make sure that the member function has declared a prototype for the *computeHash()* function. While some compilers (e.g., Visual C++ 2013, older versions of g++) are forgiving in some circumstances, Standard C++ has arcane "two-phase lookup" rules for names used in templates that often surprise even experienced C++ programmers.

All hash values for keys **must** be computed using an appropriate *computeHash()* function, which **must** have a prototype in the following form, where *sometype* is the appropriate key type:

```
unsigned int computeHash(sometype key);
```

This will enable us to test your class with our test code, and let us use your hash table with types that it's never seen before. Failure to properly implement this requirement will result in a **zero** on this part of the project.

BinaryConverter Class

[This section will be filled in soon.]

Compressor Class

[This section will be filled in soon.]

Steg Class

[This section will be filled in soon.]

WebSteg Class

[This section will be filled in soon.]

Requirements and Other Thoughts

Make sure to read this entire section before beginning your project!

1. In Visual C++, make sure to change your project from UNICODE to Multi Byte Character set, by going to Project → Properties → Configuration Properties → General → Character Set
2. In Visual C++, make sure to add *wininet.lib* to the set of input libraries, by going to Project → Properties → Linker → Input → Additional Dependencies ; otherwise, you'll get a linker error!

3. The entire project can be completed in roughly 500 lines of C++ code beyond what we've already written for you, so if your program is getting much larger than this, talk to a TA – you're probably doing something wrong.
4. Before you write a line of code for a class, think through what data structures and algorithms you'll need to solve the problem. How will you use these data structures? Plan before you program!
5. Don't make your program overly complex – use the simplest data structures possible that meet the requirements.
6. You must not modify any of the code in the files we provide you that you will not turn in; since you're not turning them in, we will not see those changes. We will incorporate the required files that you turn in into a project with special test versions of the other files.
7. Make sure to implement and test each class independently of the others that depend on it. Once you get the simplest class coded, get it to compile and test it with a number of different unit tests. Only once you have your first class working should you advance to the next class.
8. We'll provide you with a working version of the *HashTable* class template that uses types fro the STL that you are forbidden from using in your implementation. You can use this class template to build and test your other classes even if you can't figure out how to implement your *HashTable* class template.
9. You may use only those STL containers (e.g., vector, list) that are not forbidden by this spec. For *HashTable*, this means you must use none at all. For other classes, this means you must not use *map*, *multimap*, *unordered_map*, *unordered_multimap*, or the nonstandard *hash_map*; use your *HashTable* if you need a map, for example.
10. Try your best to meet our big-O requirements for each method in this spec. If you can't figure out how, then solve the problem in a simpler, less efficient way, and move on. Then come back and improve the efficiency of your implementation later if you have time.

If you don't think you'll be able to finish this project, then take some shortcuts. For example, use the substitute *HashTable* we will provide instead of creating your own *HashTable* class if necessary to save time.

You can still get a good amount of partial credit if you implement most of the project. Why? Because if you fail to complete a class (e.g., *HashTable* or *Compressor*), we will provide a correct version of that class and test it with the rest of your program. If you implemented the rest of the program properly, it should work perfectly with our version of the *HashTable* class template and we can give you credit for those parts of the project you completed.

But whatever you do, make sure that ALL CODE THAT YOU TURN IN BUILDS without errors with both Visual Studio and either clang++ or g++!

What to Turn In

You will turn in **six** files:

BinaryConverter.cpp	Contains your binary converter implementation
Compressor.cpp	Contains your compressor implementation
Steg.cpp	Contains your steganographic transformer for strings implementation
WebSteg.cpp	Contains your steganographic transformer for web pages implementation
HashTable.h	Contains your HashTable class template
report.docx, report.doc, or report.txt	Contains your report

You are to define your classes' declarations and all member function implementations directly within the specified .h and .cpp files, as well as implementations of *computeHash()* for int and for string arguments. You may add any #includes or constants you like to these files. You may also add support functions for these classes if you like. Make sure to properly comment your code.

You must submit a brief (You're welcome!) report that describes:

1. Whether any of your classes have bugs or other problems that we should know about.
2. Whether or not each satisfies our big-O requirements, and if not, what you did instead and what the big-O is for your version.
3. How three of your methods work – use high-level pseudocode to describe these:
    - *HashTable*'s *set()* method
    - *HashTable*'s *touch()* method
    - *HashTable*'s *discard()* method

Grading

- 95% of your grade will be assigned based on the correctness of your solution.
- 5% of your grade will be based on your report.

Good luck!