CS 7B - Fall 2017 - Final Exam (take-home portion). Due Wednesday, 12/13 at 2 pm

Write your responses to following questions on separate paper. Use complete sentences where appropriate and write out code using proper style and syntax. You can hand write your responses or type set with a computer.

1. Consider the problem of writing a function, void `to_lower(char* s)`, that replaces all uppercase characters in the C-style string s with their lowercase equivalents. For example, Hello, World! becomes hello, world!. Do not use any standard library functions. A C-style string is a zero-terminated array of characters, so if you find a `char` with the value 0 you are at the end.

   Below you will find a number of solutions to that problem for comparison.

```
#include <iostream>
#include <string>
using namespace std;
// solution #1, using subscripting:
void to_lower(char* p) {
  // assume that lower and upper case characters have consecutive integer values
  for (int i = 0; p[i]; ++i) {
        if ('A'<=p[i] && p[i]<='Z') // is upper case (only characters are considered
                                    // upper case - e.g. not !@#$)
            p[i] -= ('A'-'a');     // make lower case
  }
}
```

   To avoid repeating `p[i]`, we could introduce a variable to hold the correct character:

```
// solution #1.1, using subscripting and a local variable:
void to_lower(char* p) {
  // assume that lower and upper case characters have consecutive integer values
  for (int i = 0; char c = p[i]; ++i) {
        if ('A'<=c && c<='Z') // is upper case (only characters are considered
    upper
                              // case - e.g. not !@#$)
            p[i] -= ('A'-'a');  // make lower case
  }
}
```

   (a) Is this really an improvement? How so? If not, why not?

   Here's another solution to the problem that uses pointer dereferencing:

```
void to_lower(char* p) {
  // assume that lower and upper case characters have consecutive integer values
  if (p==0) return; // just in case some user tries to_lower(0)
  for (; *p; ++p) { // p is already initialized
        if ('A'<=*p && *p<='Z') // is upper case
            *p -= ('A'-'a');  // make lower case
  }
}

void test(const string& ss) // rely on visual inspection
```

```
{
  string s = ss;  // take a copy to avoid modifying original
  cout << s << '\n';
  char* p = &s[0];  // the characters are stored with a terminating 0
  to_lower(p);
  cout << p << '\n';
}

int main()
{
  test("Hello, World!");

  string s; // read examples into std::string rather than C-style string
        // to avoid the possibility of overflow
  while (cin>>s && s!="quit") // take input, one word at a time, until "quit"
    test(s);
}
```

(b) Is this an improvement in efficiency, as measured by time/operation? Test this using the `chrono` library, as described in the next problem.
ANS: Here's the code we wrote to check this. The first `to_lower()` ("mine") was coded by student Dmitri.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>
#include <cstdio>

using namespace std;
using namespace std::chrono;

void to_lower_mine(char *p) {
    while(*p) {
        if(*p >= 'A' && *p <= 'Z') {
            *p -= ('A' - 'a');
        }
        p++;
    }
}
/// solution #1 , using subscripting :
void to_lower_1 (char* p) {
    /// assume that lower and upper case characters have consecutive integer
    values
    for(int i = 0; p[i]; ++i) {
        if( 'A' <= p[i] && p[i] <= 'Z') // is upper case ( only characters are
    considered
            /// upper case     e.g. not !@#$)
            p[i] -= ('A' - 'a'); /// make lower case
    }
}
/// solution #1.1 , using subscripting and a local variable :
void to_lower_2(char* p) {
    /// assume that lower and upper case characters have consecutive integer
    values
    for(int i = 0; char c = p[i]; ++i) {
        if('A' <= c && c <= 'Z') /// is upper case ( only characters are
    considered upper
                                 /// case     e.g. not !@#$)
```

```
            p[i] -= ('A' - 'a'); /// make lower case
        }
}
void to_lower_3(char* p) {
    /// assume lower/upper case chars are consecutive integer
    if(p == 0) return ; /// just in case some user tries to lower (0)
    for (; *p; ++p) { /// p is already i n i t i a li z ed
        if('A' <= *p && *p <= 'Z') /// is upper case
            *p -= ('A' - 'a'); /// make lower case
    }
}
void test(const string& ss) { /// rely on visual inspection
    string s = ss ; /// take a copy to avoid modifying original
    cout << s << '\n';
    char* p = &s[0]; /// the characters are stored with a terminating 0
    to_lower_3(p);
    cout << p << '\n';
}
int main() {
    string s = "Hello, World!"; /// read examples into std::string rather than
    C style string
                /// to avoid the possibility of overflow
    char *p = &s[0];
    chrono::time_point<chrono::system_clock> t_1, t_2;
    chrono::duration<double> elapsed_seconds;
    t_1 = chrono::system_clock::now();
    for(int i = 0; i < 100000000; i++ ){
        to_lower_mine(p);
        s = "Hello, World";
    }
    t_2 = chrono::system_clock::now();
    elapsed_seconds = t_2 - t_1;
    cout << "\nto_lower_mine() elapsed time: "
         << elapsed_seconds.count();
    t_1 = chrono::system_clock::now();
    for(int i = 0; i < 100000000; i++ ){
        to_lower_1(p);
        s = "Hello, World";
    }
    t_2 = chrono::system_clock::now();
    elapsed_seconds = t_2 - t_1;
    cout << "\nto_lower_1() elapsed time: " << elapsed_seconds.count();
    t_1 = chrono::system_clock::now();
    for(int i = 0; i < 100000000; i++ ){
        to_lower_2(p);
        s = "Hello, World";
    }
    t_2 = chrono::system_clock::now();
    elapsed_seconds = t_2 - t_1;
    cout << "\nto_lower_2() elapsed time: " << elapsed_seconds.count();
    t_1 = chrono::system_clock::now();
    for(int i = 0; i < 100000000; i++ ){
        to_lower_3(p);
        s = "Hello, World";
    }
    t_2 = chrono::system_clock::now();
    elapsed_seconds = t_2 - t_1;
    cout << "\nto_lower_3() elapsed time: " << elapsed_seconds.count();
    return 0;
}
```

...and here's the (very much machine/compiler dependent) output I'm getting now:

```
to_lower_mine() elapsed time: 12.964
to_lower_1() elapsed time: 13.059
to_lower_2() elapsed time: 13.192
to_lower_3() elapsed time: 13.073
```

The differences are in how the loops are formed, what the conditional for the loop is, how the "`to_upper()` action is executed and how the update is performed. Dmitri's `to_lower()` uses a `while` loop conditioned on the `NUL` character

2. Look at PPP20.7.1 and time the insert/delete operations using the `chrono` library as partially illustrated below:

```
#include "std_lib_facilities.h"
#include <chrono>
#include <ctime>

int main()
{
    std::chrono::time_point<std::chrono::system_clock> start, end;

    int initializer[7] = {1,2,3,4,5,6,7};
    int* first = initializer;
    int* last  = initializer+7;

    //To compare, we'll do exactly the same with a vector:
    start = std::chrono::system_clock::now();

    auto beginning = std::chrono::high_resolution_clock::now();
    for(int i = 0; i < 100000; ++i)
    {
        vector<int> v(first, last);
        vector<int>::iterator p = v.begin(); // take a vector
        ++p; ++p; ++p;                       // point to its 3rd element
        vector<int>::iterator q = p;
        ++q;                                 // point to it's 4th element

        p = v.insert(p,99);                  // p points at the inserted element
        p = v.erase(p);                      // p points at the element after the erased
    one
    }
    auto ending = std::chrono::high_resolution_clock::now();
    end = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_seconds = end-start;
    std::time_t end_time = std::chrono::system_clock::to_time_t(end);

    std::cout << "finished computation at " << std::ctime(&end_time)
              << "elapsed time: " << elapsed_seconds.count() << "s\n";
    cout << "My poorly-written function took: "
         << std::chrono::duration_cast<std::chrono::nanoseconds>(ending-beginning).
    count()
         << " nanoseconds\n";
    //To compare, we'll do exactly the same with a list (provide timing code):
    {
        list<int> v(first, last);
        list<int>::iterator p = v.begin();   // take a list
        ++p; ++p; ++p;                       // point to its 3rd element
```

```
43          list<int>::iterator q = p;
            ++q;                                    // point to it's 4th element
45      }
    }
```

3. (PPP20.20) Run a small timing experiment to compare the cost of using vector and list. You can find an explanation of how to time a program in §26.6.1. Generate N random int values in the range [0:N). As each int is generated, insert it into a vector⟨int⟩ (which grows by one element each time). Keep the vector sorted; that is, a value is inserted after every previous value that is less than or equal to the new value and before every previous value that is larger than the new value. Now do the same experiment using a list⟨int⟩ to hold the ints. For which N is the list faster than the vector? Try to explain your result.
ANS: In class we developed code for this purpose:

```
#include <iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
using namespace std::chrono;

int main() {
    srand(time(0));
    chrono::time_point<chrono::system_clock> t1, t2;
    chrono::duration<double> elapsed_seconds;
    ///putIn();
    int randNo{0};
    vector<int> vint;
    vector<int>::iterator vit = vint.begin();
    list<int> lint;
    list<int>::iterator lit = lint.begin();
    int N = 100;
    int n = 50000; /// repeat do_something() n times
    t1 = system_clock::now(); /// begin time
    vint.insert(vit,rand()%N);
    //cout << "\nvint[0] = " << vint[0];
    for (int i = 0; i<n; i++) {  /// timing loop
        randNo = rand()%N;
        vit = vint.begin();
        ///cout << "\n*vit = " << *vit;
        while(*vit<randNo && !(vit==vint.end())) ++vit;
        vint.insert(vit, randNo);
        ///for(j:vint) cout << j << " ";
    }
    t2 = system_clock::now(); // end time
    elapsed_seconds = t2-t1;
    cout << "\nInsterting into sorted vector " << n << " times took "
         << elapsed_seconds.count() << " seconds\n";
    ///<< duration_cast<seconds>(t2-t1).count() << "seconds\n";
    t1 = system_clock::now(); /// begin time
    //vint.insert(vit,rand()%N);
    //cout << "\nvint[0] = " << vint[0];
    //cin.get();
    for (int i = 0; i<n; i++) {  /// timing loop
        randNo = rand()%N;
        lit = lint.begin();
        ///cout << "\n*vit = " << *vit;
        while(*lit<randNo && !(lit==lint.end())) ++lit;
        lint.insert(lit, randNo);
        ///for(j:vint) cout << j << " ";
    }
    t2 = system_clock::now(); // end time
    elapsed_seconds = t2-t1;
    cout << "\nInsterting into sorted list " << n << " times took "
         << elapsed_seconds.count() << " seconds\n";
}
```

Perhaps it's strange, but, while inserting into `vector` is slower for smaller `N`, for larger `N`, inserting into `list` is slower! I put the thing in a loop with N = powers of 10:

```
N = 10:
Inserting into sorted vector 50000 times took 7.402 seconds
Inserting into sorted list 50000 times took 6.492 seconds

N = 100:
Inserting into sorted vector 50000 times took 30.364 seconds
Inserting into sorted list 50000 times took 35.798 seconds

N = 1000:
Inserting into sorted vector 50000 times took 69.311 seconds
Inserting into sorted list 50000 times took 72.806 seconds

N = 10000:
Inserting into sorted vector 50000 times took 98.779 seconds
Inserting into sorted list 50000 times took 112.134 seconds
```

For an explanation and bigger picture on this, see cpp-benchmark-vector-list-deque "https://baptiste-wicht.com/posts/2012/12/cpp-benchmark-vector-list-deque.html"
"List is clearly slower than the other two data structures that exhibit the same performance. This comes from the very slow linear search. Even if the two other data structures have to move a lot of data, the copy is cheap for small data types."

4. Do the drill from Chapter 20.