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.

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

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.

4. Do the drill from Chapter 21.