

CSC3002: Introduction to Computer Science Programming Paradigms

Assignment 2

Assignment description:

This assignment will be worth 10% of the final grade.

You should write your code for each question in a .cpp and .h file (please name it using the question name, e.g. q1.cpp). Please pack your **whole project** files into a single .zip file, name it using your student ID (e.g. if your student ID is 123456, then the file should be named as 123456.zip), and then submit the .zip file via Moodle.

You should create an empty project in QT and start your programming.

Please note that, the teaching assistant may ask you to explain the meaning of your program, to ensure that the codes are indeed written by yourself. Please also note that we may check whether your program is too similar to your fellow students' code using Moodle.

This assignment is due on 23:59PM, 20 November (Sunday). For each day of late submission, you will lose 10% of your mark in this assignment. If you submit more than three days later than the deadline, you will receive zero in this assignment.

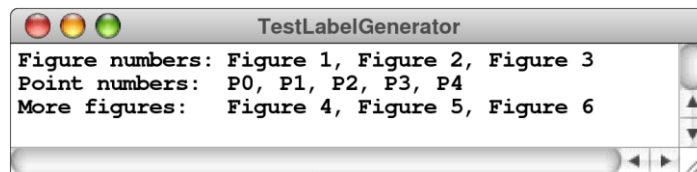
Reminders:

Marks will be given on the style of your coding. Please write comments and make your variable name meaningful.

6.7

1. For certain applications, it is useful to be able to generate a series of names that form a sequential pattern. For example, if you were writing a program to number figures in a paper, having some mechanism to return the sequence of strings "Figure 1", "Figure 2", "Figure 3", and so on, would be very handy. However, you might also need to label points in a geometric diagram, in which case you would want a similar but independent set of labels for points such as "P0", "P1", "P2", and so forth. If you think about this problem more generally, the tool you need is a label generator that allows the client to define arbitrary sequences of labels, each of which consists of a prefix string ("Figure " or "P" for the examples in the preceding paragraph) coupled with an integer used as a sequence number. Because the client may want different sequences to be active simultaneously, it makes sense to define the label generator as a `LabelGenerator` class. To initialize a new generator, the client provides the prefix string and the initial index as arguments to the `LabelGenerator` constructor. Once the generator has been created, the client can return new labels in the sequence by calling `nextLabel` on the `LabelGenerator`.

As an illustration of how the interface works, the main program shown in Figure 6-14 produces the following sample run:

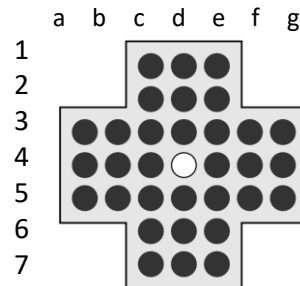


Write the files `labelgen.h` and `labelgen.cpp` to support this class.

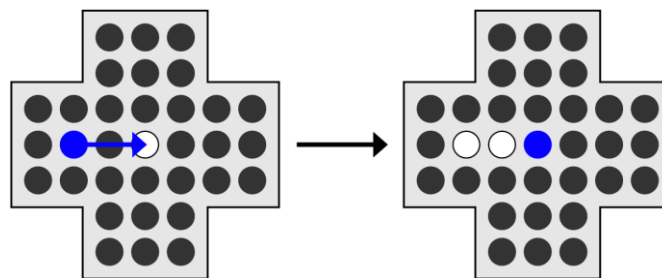
FIGURE 6-14 Main program to test the label generator

```
int main() {
    LabelGenerator figureNumbers("Figure ", 1);
    LabelGenerator pointNumbers("P", 0);
    cout << "Figure numbers: ";
    for (int i = 0; i < 3; i++) {
        if (i > 0) cout << ", ";
        cout << figureNumbers.nextLabel();
    }
    cout << endl << "Point numbers: ";
    for (int i = 0; i < 5; i++) {
        if (i > 0) cout << ", ";
        cout << pointNumbers.nextLabel();
    }
    cout << endl << "More figures: ";
    for (int i = 0; i < 3; i++) {
        if (i > 0) cout << ", ";
        cout << figureNumbers.nextLabel();
    }
    cout << endl;
    return 0;
}
```

2. In theory, the recursive backtracking strategy described in this chapter should be sufficient to solve most puzzles that involve performing a sequence of moves looking for some solution. In practice, however, some of those puzzles are too complex to solve in a reasonable amount of time. One puzzle that is just at the limit of what recursive backtracking can accomplish without using some additional cleverness is the *peg solitaire* puzzle, which dates from the 17th century. Peg solitaire is usually played on a board that looks like this:



The black dots in the diagram are pegs, which fill the board except for the center hole. On a turn, you are allowed to jump over and remove a peg, as illustrated in the following diagram, in which the colored peg jumps into the vacant center hole and the peg in the middle is removed:

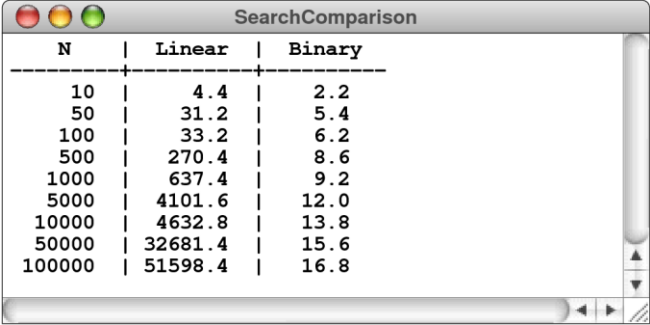


The object of the game is to perform a series of jumps that leaves only one peg in the center hole. Write a program to solve this puzzle.

- Write a program that generates a table comparing the performance of two algorithms—linear and binary search—when used to find a randomly chosen integer key in a sorted `Vector<int>`. The linear search algorithm simply goes through each element of the vector in turn until it finds the desired one or determines that the key does not appear. The binary search algorithm, which is implemented for string vectors in Figure 7-5, uses a divide-and-conquer strategy by checking the middle element of the vector and then deciding which half of the remaining elements to search.

The table you generate in this problem, should calculate the number of comparisons made against elements of the vector. To ensure that the results are not completely random, your program should average the results over several independent trials. A sample run of the program might look like this:

10.5



N	Linear	Binary
10	4.4	2.2
50	31.2	5.4
100	33.2	6.2
500	270.4	8.6
1000	637.4	9.2
5000	4101.6	12.0
10000	4632.8	13.8
50000	32681.4	15.6
100000	51598.4	16.8

FIGURE 7-5 Divide-and-conquer implementation of binary search

```

/*
 * Function: findInSortedVector
 * Usage: int index = findInSortedVector(key, vec);
 * -----
 * Searches for the specified key in the Vector<string> vec, which
 * must be sorted in lexicographic (character code) order. If the
 * key is found, the function returns the index in the vector at
 * which that key appears. (If the key appears more than once in
 * the vector, any of the matching indices may be returned). If the
 * key does not exist in the vector, the function returns -1. This
 * implementation is simply a wrapper function; all of the real work
 * is done by the more general binarySearch function.
 */

int findInSortedVector(string key, Vector<string> & vec) {
    return binarySearch(key, vec, 0, vec.size() - 1);
}

/*
 * Function: binarySearch
 * Usage: int index = binarySearch(key, vec, p1, p2);
 * -----
 * Searches for the specified key in the Vector<string> vec, looking
 * only at indices between p1 and p2, inclusive. The function returns
 * the index of a matching element or -1 if no match is found.
 */

int binarySearch(string key, Vector<string> & vec, int p1, int p2) {
    if (p1 > p2) return -1;
    int mid = (p1 + p2) / 2;
    if (key == vec[mid]) return mid;
    if (key < vec[mid]) {
        return binarySearch(key, vec, p1, mid - 1);
    } else {
        return binarySearch(key, vec, mid + 1, p2);
    }
}

```

4. Rewrite the implementation of the merge sort algorithm from Figure 10-3 so that it sorts an array rather than a vector. Your function should use the prototype

11.7

```
void sort(int array[], int n)
```

FIGURE 10-3 Implementation of the merge sort algorithm

```
/*
 * Function: sort
 * -----
 * This function sorts the elements of the vector into increasing order
 * using the merge sort algorithm, which consists of the following steps:
 *
 * 1. Divide the vector into two halves.
 * 2. Sort each of these smaller vectors recursively.
 * 3. Merge the two vectors back into the original one.
 */

void sort(Vector<int> & vec) {
    int n = vec.size();
    if (n <= 1) return;
    Vector<int> v1;
    Vector<int> v2;
    for (int i = 0; i < n; i++) {
        if (i < n / 2) {
            v1.add(vec[i]);
        } else {
            v2.add(vec[i]);
        }
    }
    sort(v1);
    sort(v2);
    vec.clear();
    merge(vec, v1, v2);
}

/*
 * Function: merge
 * -----
 * This function merges two sorted vectors, v1 and v2, into the vector
 * vec, which should be empty before this operation. Because the input
 * vectors are sorted, the implementation can always select the first
 * unused element in one of the input vectors to fill the next position.
 */

void merge(Vector<int> & vec, Vector<int> & v1, Vector<int> & v2) {
    int n1 = v1.size();
    int n2 = v2.size();
    int p1 = 0;
    int p2 = 0;
    while (p1 < n1 && p2 < n2) {
        if (v1[p1] < v2[p2]) {
            vec.add(v1[p1++]);
        } else {
            vec.add(v2[p2++]);
        }
    }
    while (p1 < n1) vec.add(v1[p1++]);
    while (p2 < n2) vec.add(v2[p2++]);
}
```

5. Even though programmers tend to think of strings as relatively simple entities, their implementation turns out to involve the full range of techniques you have seen in this chapter. In this exercise, your mission is to define a class called **MyString** that approximates the behavior of the **string** class from the Standard C++ libraries. Your class should export the following methods:
- A constructor **MyString(str)** that creates a **MyString** object from a C++ **string**.
 - A destructor that frees any heap storage allocated by the **MyString**.
 - A **toString()** method that converts a **MyString** to a C++ **string**.
 - A method **length()** that returns the number of characters in the string.
 - A method **substr(start, n)** that returns a substring of the current string object. As in the library version of the **string** class, the substring should begin at the index position **start** and continue for **n** characters or through the end of the string, whichever comes first. The parameter **n** should be optional; if it is missing, the substring should always extend through the end of the original string.
 - A redefinition of the operator **+** that concatenates two **MyString** objects. It also makes sense to overload the operator **+=** so that it appends a character or a string to the end of an existing one.
 - A redefinition of the operator **<<** so that **MyString** objects can be written to output streams.
 - A redefinition of the bracket-selection operator so that **str[i]** returns by reference the character at index position **i** in **str**. As an improvement over the **string** class in the C++ libraries, your implementation of the bracket operator should call **error** if the index is outside the bounds of the string.
 - A redefinition of the relational operators **==**, **!=**, **<**, **<=**, **>**, and **>=** that compare strings lexicographically.
 - A redefinition of the assignment operator and the copy constructor for the **MyString** class so that any copying operations make a deep copy that creates a new character array.

Your code should work directly with your underlying representation and should make no calls to any of the methods in the C++ **string** class. Your interface and implementation should also be **const** correct so that both clients and the C++ compiler know exactly what methods can change the value of the string.