COMP 2011 Final Exam - Spring 2018 - HKUST

Date: May 29, 2018
Time Allowed: 2.5 hours
Instructions:

1. This is an open book, open notes examination. No electronic devices are allowed.
2. There are **5** questions on **28** pages (including this cover page and excluding the appendices).
3. Write your answers in the space provided in black/blue ink.
4. All programming codes in your answers must be written in the ANSI C++ version as taught in the lectures.
5. For programming questions, you are **NOT** allowed to define additional helper functions or structures, nor global variables unless otherwise stated. You also **cannot** use any library functions not mentioned in the questions.

Student Name	Solution
Student ID	
Email Address	
Seat Number	

For T.A. Use Only

Problem	Score
1	/ 20
2	/ 20
3	/ 20
4	/ 20
5	/ 20
Total	/ 100

Problem 1 [20 points] C++ Basics

(a) [2 points] If you have a function that needs to effectively pass back two numbers, how can you do it?

Answer:

Pass by reference

Or

Pass by array

(b) [3 points] Given the array:

```
const int SIZE = 5;
int arr[SIZE] = {3, 6, 4, 6, 7};
```

Assume the `void swap(int& a, int& b)` function works as given in notes which swaps two variables' values, what will the above array contain when the following code segment is executed?

```
for (int i = 0; i < SIZE-1; i++)
   for (int j = i+1; j < SIZE; j++)
      if (arr[i] > arr[j])
         swap(arr[i], arr[j]);
```

Answer:

Array element	a[0]	a[1]	a[2]	a[3]	a[4]
Value	3	4	6	6	7

(c) [3 points] Given the integer constant:

```
const int N = 4;
```

and the following function:

```
int fun(int a[][N], int row, int col)
{
   int n = 0;

   for (int i = 0; i < row; i++)
      if (i % 2)
         for (int j = 0; j < col; j++)
            if (a[i][j] > 0)
               n += a[i][j];

   return n;
}
```

What is the output for the following main function?

```
int main()
{
   int arr[4][N] = { {1, -2, 3, -4},
                     {5, -6, 7, 8},
                     {-9, 10, -11, 12},
                     {13, -14, 15, -16}};
   cout << fun(arr, 2, 3) << endl;
   return 0;
}
```

Answer:

12

(d) [6 points] Implement a function `swapParts()` to work with an array so that the elements before and after a certain cut-point are swapped. The first parameter is an integer array called `arr`. The second parameter is an integer representing the cut-point called `cutPt`, and the 3rd paramter is the size of array called `size`. You may assume `size` > 0 and $0 <$ `cutPt` $<$ `size`.

For example, if the cut-point is 3, and size is 10, the array `[0, 1, 2, 10, 4, 5, 6, 7, 8, 9]` becomes `[10, 4, 5, 6, 7, 8, 9, 0, 1, 2]`

Note: The function should be able to handle 1-dimensional arrays of any size. You are not allowed to define any additional array or data structure except integer variables and you are not allowed to call any function.

Here is an example of calling the `swapParts()` function:

```
int main() {
   int a[] = {0, 1, 2, 10, 4, 5, 6, 7, 8, 9};
   int s = sizeof(a)/sizeof(int);
   swapParts(a, 3, s);
   for (int i = 0; i < s; i++)
      cout << a[i] << " ";
   cout << endl;
   return 0;
}
```

will give the output:

```
10 4 5 6 7 8 9 0 1 2
```

Fill in the formal parameter list and the function body.

Answer:

```
void swapParts(                                                    )
{


}
```

```
void swapParts(int arr[], int cutPt, int size)
{
   for (int i=0; i<cutPt; i++)
   {
      int temp = arr[0];
      for (int j = 0; j < size - 1; j++)
      {
         arr[j] = arr[j + 1];
      }
      arr[size - 1] = temp;
   }
}
```

(e) [6 points] Define and implement a function `fun()` which takes an integer array and its size (`size` > 0) as the parameters. It replaces the ith element by the product of the ith element in the original array and the ith element of the reversed array.

Note: The function should be able to handle 1-dimensional arrays of any size. You are not allowed to define any additional array or data structure except integer variables.

For example, if the given array was `[1, 2, 3, 4, 5]`, then the reversed array would be `[5, 4, 3, 2, 1]`, and finally the array became `[5, 8, 9, 8, 5]`.

Write both the function header and the function body.

Answer:

```
void fun(int arr[], int size)
{
   for (int i=0; i<=((size-1)/2); i++)
   {
      arr[i] *= arr[size-i-1];
      arr[size-i-1] = arr[i];
   }
}
```

Problem 2 [20 points] Recursion

(a) [2 points] Since a recursive function calls itself, how can you prevent it from just doing exactly the same thing each time?

Answer:

- Can pass variables as parameters that are with different values in each recursive call, for example, a variable with the level (or other variables) that so that what the recursive call does depends on the level.
- Perhaps different random numbers are picked (or different values for a certain local variable are generate) in the recursive function so that what the recurive call does depends on the different values.

(b) [3 points] Assume the function `oneDice()` returns the value of rolling one dice once, meaning a random integer value between 1 to 6 inclusively, what does the following code calculate (i.e. what is being stored in the variable `total` after executing the code)?

```
#include <iostream>
using namespace std;

int oneDice();

int main()
{
   int total = 0, value;
   for (int i = 1; i <= 10; i++)
   {
      value = oneDice();
      if ( (i % 2) == 1 )
         total += value;
      else
         total -= value;
   }

   cout << "total = " << total << endl;

   return 0;
}
```

Answer:

Rolls the dice 10 times and accumulates the values where odd values are subtracted and even values are added.

(c) [3 points] What will happen when the function `oneMinionDice()` is excuted? You may assume that `srand()` has been called once at the begining of the `main()` program.

```
int oneMinionDice()
{
   int value = 1 + rand()%6;
   if ((value % 2) == 1)
      return oneMinionDice() - value;
   else
      return oneMinionDice() + value;
}
```

Answer:

Infinite recursion occurs as there is no base case/stopping criteria.

(d) [6 points] Given the following recursive `minionDice()` function for the suggested answer to Minion Mission #5:

```
int minionDice()
{
   int value1 = 1 + rand()%6;
   int value2 = 1 + rand()%6;
   int value3 = 1 + rand()%6;
   cout << "roll: " << value1 << " " << value2 << " " << value3 << endl;
   if (value1 == 6)
      value1 += minionDice();
   if (value2 == 6)
      value2 += minionDice();
   if (value3 == 6)
      value3 += minionDice();
   return (value1 + value2 + value3);
}
```

Assume the maximum number of dices is given by the gobal integer constant, NDICE, modify the above function to take an integer parameter, `level`, which represents the current level of rolling (i.e. the number of recursive calls it takes to reach the current function call). If the function is of level N, it will uses $N+1$ dices, where $N \leq NDICE$. Hence, no further rolling of dices when the level reaches beyond N.

For example, the initial call from the `main()` function is of level zero, i.e. `minionDice(0)`. The function will give 1 value (dice) only. If that value is 6, it will invoke a recursive call of level 1, which will give 2 values (dices). If the two values are [6 6], they will invoke two recursive calls of level 2, which both calls will give 3 values (dices).

For example, an instance of calling the modified function will give the following output:

```
level 0: 6
level 1: 6 6
level 2: 1 5 6
level 3: 1 6 6 1
level 4: 4 3 1 4 3
level 4: 1 3 1 3 2
level 2: 5 1 2
```

And correspondingly the function call, `minionDice(0)`, returns 77 as the sum of all values.

Line 1 is the output of the initial call (level 0) using 1 dices.
Line 2 is the output of the level 1 recursive call using 2 dices.
Line 3 and 7 are the outputs of the level 2 recursive calls using 3 dices.
Line 4 is the output of the level 3 recursive call using 4 dices.

Line 5 and 6 is the output of the level 4 recursive call using 5 dices.

Write the function definition (including the function header and body) below which will output the rolls (values) in each level and return the sum for all rolls (values).

Note: You may define additional data structre, e.g. array, if necessary.

Answer:

```
int minionDice(int level)
{
   if (level >= NDICE)
      return 0;

   int values[NDICE];

   for (int i=0; i<=level; i++)
      values[i] = 1 + rand()%6;

   cout << "level " << level << ": ";
   for (int i=0; i<=level; i++)
      cout << values[i] << " ";
   cout << endl;

   int sum = 0;
   for (int i=0; i<=level; i++)
   {
      sum += values[i];
      if (values[i] == 6)
         sum += minionDice(level + 1);
   }
   return sum;
}
```

(e) [6 points] Write a recursive function, `recursiveSort()`, to sort an array of integers into ascending order using the following idea: place the smallest element in the first position, then sort the rest of the array by a recursive call.

The function takes 3 parameters, namely, the integer array, the size of the array and the current index, respecitively.

Here is an example of calling `recursiveSort()` in the main function:

```
#include <iostream>
using namespace std;

void recursiveSort(int arr[], int size, int index);

int main()
{
   int a[] = {10, 6, 4, 5, 3};
   recursiveSort(a, sizeof(a)/sizeof(int), 0);

   for (int i=0; i<sizeof(a)/sizeof(int); i++)
      cout << a[i] << " ";
   cout << endl;

   return 0;
}
```

which will give an output:

```
3 4 5 6 10
```

Note: your implementation should be able to handle array of any size. You are not allowed to define any other array or data structure besides integer variables. You are also not allowed to call any functions.

Implement the recursive function below:

```
void recursiveSort(int arr[], int size, int index)
{
   // Answer here:

}


/*
 Answer:
*/

#include <iostream>
//#include <cstdlib>

using namespace std;

void recursiveSort(int arr[], int size, int index)
{
   if (index == (size - 1))
      return;
   int min_index = index;
   for (int i=index + 1; i<size; i++)
      if (arr[min_index] > arr[i])
         min_index = i;
   int temp = arr[index];
   arr[index] = arr[min_index];
   arr[min_index] = temp;

   recursiveSort(arr, size, index+1);
}

int main()
{
   int a[] = {10, 6, 4, 5, 3};
   recursiveSort(a, sizeof(a)/sizeof(int), 0);

   for (int i=0; i<sizeof(a)/sizeof(int); i++)
      cout << a[i] << " ";
   cout << endl;

   return 0;
}
```

Problem 3 [20 points] Linked List and Dynamic Array

(a) [2 points] Where is the most difficult (in terms of link updates) to insert a node into a linked list – the beginning, the middle, or the end? And why?

Answer:

The middle. Because you have more (two) links to update to insert the node. (Head & tail insertion only requires one link update.)

Or

The end, because you need more iterations(operations) to reach the last node.

(b) [3 points] Assume the linked list is defined as in the lecture notes, pages 55 to 64, what is the output of the following main function? (Note: the definition of the linked list is given in Appendix A for your reference.)

```
#include "ll_cnode.h"

int main()
{
   ll_cnode* head = ll_create("zmxrbsawt");
   ll_delete(head, 'a');
   ll_insert(head, 'n', 3);
   ll_cnode* temp = ll_search(head, 's');
   ll_delete_all(temp->next);
   int n = ll_length(head);
   ll_insert(head, 'f', n/2);
   ll_print(ll_search(head, 'f'));

   return 0;
}
```

Answer:

fnrbs

(c) [3 points] Assume the linked list is defined as in the lecture notes, pages 55 to 64, explain what the following code does? Also, state and explain whether memory leak may occur. (Note: the definition of the linked list is given in Appendix A for your reference.)

```
#include "ll_cnode.h"

void ll_mystery(ll_cnode*& head, char c)
{
   ll_cnode *prev, *p;
   ll_cnode *node = new ll_cnode;
   node->data = c;
   node->next = nullptr;
   if ( (head == nullptr) || (head->next == nullptr) )
      head = node;
   else {
      p = head;
      while (p->next != nullptr)
      {
         prev = p;
         p = p->next;
      }
      prev->next = node;
   }
}
```

Answer:

Create a new node with the data as c and next pointer as nullptr. If the linked list is empty, add it as the new head node. Otherwise, if the linked list is not empty, the original tail node is replaced by the new node. Yes, there is memory leak, the old tail node is not deallocated.

(d) [6 points] Assume the linked list is defined as in the lecture notes, pages 55 to 64, implement a function `deleteN()` to delete only the N-th node (if any) in a given linked list pointed by head. If there are less than N elements in the linked list, do nothing. You may also assume N is a positive integer (i.e. $N > 0$).

For example,

```
#include "ll_cnode.h"

void deleteN(ll_cnode*& head, int N);

int main()
{
   ll_cnode* head = ll_create("abcdef");
   ll_print(head);
   deleteN(head, 5);
   ll_print(head);
   deleteN(head, 4);
   ll_print(head);
   deleteN(head, 1);
   ll_print(head);
   deleteN(head, 6);
   ll_print(head);

   return 0;
}
```

will produce the following output:

```
abcdef
abcdf
abcf
bcf
bcf
```

You should ensure that no memory leak will occur in your implementation and you cannot call any functions. (Note: the definition of the linked list is given in Appendix A for your reference.)

Implement the function `deleteN()` on the next page.

```
void deleteN(ll_cnode*& head, int N)
{
   // Answer here:

}
```

```
#include "ll_cnode.h"

void deleteN(ll_cnode*& head, int N)
{
    if (head == nullptr)
       return;
    if (N == 1)
    {
       ll_cnode* p = head;
       head = p->next; // or head = head->next;
       delete p;
       return;
    }
    ll_cnode* prev;
    ll_cnode* p = head;

    for (int i=0; i<N-1; i++)
    {
       if (p == nullptr)
          return;
       prev = p;
       p = p->next;
    }
    prev->next = p->next;
    delete p;
}
```

(e) [6 points] In this question, you will write a program to create a dictionary of words. Just like a normal dictionary, words are grouped according to the first alphabet of each word under the same section. For example, "apple", "ant", "anyone" are grouped together under the section 'a'.

In the program, words (i.e. C Strings) are stored and organized in an array of structure objects, `wordSection`. There are 26 elements (i.e. 26 `wordSection` objects) in the `dictionary` array which stores the sections of 'a' to 'z'. For example, the first element is for the alphabet 'a' section, the fifth element is for the alphabet 'e' section and the 26th element is for the alphabet 'z'.

Words with the same first alphabet are stored in a `wordSection` object as a dynamic array of C Strings where the address of the dynamic array is stored in the pointer member, `words` and the number of words is stored in the integer member, `num`.

Four functions are designed for the dictionary, namely, `initDictionary()`, `printDictionary()`, `addWordToDictionary()`, and `deleteDictionary()`.

Here are the structure definition, the function declarations, the function definitions of the main function and two of the functions, `initDictionary()` and `printDictionary()`:

```
#include <iostream>
#include <cstring>
using namespace std;

struct wordSection
{
  char** words;
  int num;
};

void initDictionary(wordSection d[], int size);
void printDictionary(wordSection d[], int size);
void addWordToDictionary(wordSection d[], int size, const char* newWord);
void deleteDictionary(wordSection dictionary[], int size);

int main()
{
  const int SIZE = 26;
  wordSection dictionary[SIZE];

  initDictionary(dictionary, SIZE);
  addWordToDictionary(dictionary, SIZE, "hello");
  addWordToDictionary(dictionary, SIZE, "happy");
  addWordToDictionary(dictionary, SIZE, "computer");
  addWordToDictionary(dictionary, SIZE, "science");
  addWordToDictionary(dictionary, SIZE, "minion");
  addWordToDictionary(dictionary, SIZE, "stuart");
  addWordToDictionary(dictionary, SIZE, "bob");
```

```
  addWordToDictionary(dictionary, SIZE, "handsome");
  addWordToDictionary(dictionary, SIZE, "kevin");
  printDictionary(dictionary, SIZE);
  deleteDictionary(dictionary, SIZE);

  return 0;
}

void initDictionary(wordSection d[], int size)
{
   for (int i = 0; i < size; i++) {
      d[i].words = nullptr;
      d[i].num = 0;
   }
}

void printDictionary(wordSection d[], int size)
{
   for (int i = 0; i < size; i++) {
      cout << "Section " << static_cast<char>('a' + i) << ": ";
      for (int j = 0; j < d[i].num; j++)
         cout << d[i].words[j] << " ";
      cout << endl;
   }
}
```

which gives the following output:

```
Section a:
Section b: bob
Section c: computer
Section d:
Section e:
Section f:
Section g:
Section h: hello happy handsome
Section i:
Section j:
Section k: kevin
Section l:
Section m: minion
Section n:
Section o:
Section p:
Section q:
Section r:
Section s: science stuart
Section t:
Section u:
```

```
Section v:
Section w:
Section x:
Section y:
Section z:
```

Based on the above, complete the implementation of the function `addWordToDictionary()` to add a given C String, `newWord`, to the appropriate `wordSection` in the given array, `d`. You may assume all the characters are already in lowercase and the first character must be an alphabet ('a' to 'z'), and the C String in `newWord` does not exist in the dictionary, `d`, before the function being called. Note: your implementation should ensure no memory leak.

You may use the following two C String functions:

```
// strlen() calculates the length of the string pointed to by s, not including
// the terminating null character.
int strlen(const char* s);

// strcpy() copies the string pointed to by s2 to the string pointed to by s1
// returning pointer s1
char* strcpy(char* s1, const char* s2);
```

Complete the implementation of the function `addWordToDictionary()` on the next page.

```
void addWordToDictionary(wordSection d[], int size, const char* newWord)
{
   // Answer here:

}
```

```
void addWordToDictionary(wordSection d[], int size, const char* new_word)
{
   int sectionIndex = new_word[0] - 'a';
   char** new_list = new char*[d[sectionIndex].num + 1];

/* Approach 1: */
   for (int i=0; i<d[sectionIndex].num; i++)
   {
      new_list[i] = d[sectionIndex].words[i];
   }
   new_list[ d[sectionIndex].num ] = new char[strlen(new_word) + 1];
   strcpy(new_list[ d[sectionIndex].num ], new_word);

   d[sectionIndex].num++;
   delete [] d[sectionIndex].words;
   d[sectionIndex].words = new_list;

/* Approach 2:

   for (int i=0; i<d[sectionIndex].num; i++)
   {
      new_list[i] = new char[ strlen(d[sectionIndex].words[i]) + 1 ];
      strcpy(new_list[i], d[sectionIndex].words[i]);
   }
   new_list[ d[sectionIndex].num ] = new char[strlen(new_word) + 1];
   strcpy(new_list[ d[sectionIndex].num ], new_word);

   d[sectionIndex].num++;
   for (int i=0; i<d[sectionIndex].num; i++)
      delete [] d[sectionIndex].words[i];
   delete [] d[sectionIndex].words;
   d[sectionIndex].words = new_list;
*/
}
```

Problem 4 [20 points] C++ Class

(a) [2 points] In the "Bulbs and Lamps" example in the lecture notes of "C++ Class" (pages 35 to 41), what is the primary relationship between the two Classes, Lamp and Bulb? (Note: the class definitions of Lamp and Bulb are given in Appendix B for your reference.)

Answer:

Bulb objects are contained in each Lamp object.

Or

A Lamp object includes Bulb object as data member via a Bulb pointer to a dynamic array.

For questions 4(b) to 4(e), we are going to extend the class definitions to allow each Bulb object in the Lamp object to have diferent wattages and prices. To do this, the following modifications are made:

(I) Adding a new data member called `max_num_bulbs`, which contains the maximum number of Bulb objects that a Lamp object can hold. Therefore, the number of bulbs correctly installed in the lamp is stored in `num_bulbs`, which initially starts at zero and can range up to `max_num_bulbs`.

(II) The member functions are also modified accordingly:

i. The constructor takes 2 parameters which set the `max_num_bulbs` and `price` respectively. (Note: `num_bulbs` is initialized to zero.)

ii. The member function `install_bulbs()` is removed.

iii. A new member function called `add_bulbs()` is added. It takes 3 parameters which set the wattage of the bulbs to be added, the price for each bulb and how many of this type of bulbs to add. For example, `add_bulbs(1, 2, 3)` will add three 1 Watt bulbs that costs $2 each.

iv. The member function `total_price()` is modified to sum up all bulbs' prices and the price of the lamp.

v. The member function `total_power()` is modified to sum up all bulbs' powers.

Here is the extended definition of the `Lamp` class:

```
#include "bulb.h"       /* File: lamp.h */
class Lamp
{
  private:
    int  num_bulbs; // A lamp MUST have 1 or more light bulbs
    int max_num_bulbs; // the maximum number of bulbs
    Bulb* bulbs; // Dynamic array of light bulbs installed onto a lamp
    float price; // Price of the lamp, not including its bulbs

  public:
    Lamp(int n, float p);      // n = maximum number of bulbs; p = lamp's price
    ~Lamp();

    int total_power() const;   // Total power/wattage of its bulbs
    float total_price() const; // Price of a lamp PLUS its bulbs

    // Print out a lamp's information; see outputs from our example
    void print(const char* prefix_message) const;

    // Add n light bulbs to a lamp with:
    // w = a light bulb's wattage; p = a light bulb's price
    void add_bulbs(int w, float p, int n);
};
```

(b) [3 points] With the extended definition of the `Lamp` class, what does the following program output?

```
#include <iostream>
#include "lamp.h"
using namespace std;

int main()
{
    Lamp chandelier(100, 1000); // chandelier costs 1000 and maximum 100 bulbs

    chandelier.add_bulbs(20, 40, 10); // add 10 bulbs of 20 Watts, each costs 40
    chandelier.add_bulbs(11, 12, 30); // add 30 bulbs of 11 Watts, each costs 12
    chandelier.add_bulbs(14, 22, 20); // add 20 bulbs of 14 Watts, each costs 22

    int price = chandelier.total_price();
    int power = chandelier.total_power();

    cout << price << " " << power << endl;

    return 0;
}
```

Answer:

2200 810

(c) [3 points] Implement the constructor for the extended `Lamp` class as described above.

```
Lamp::Lamp(int n, float p)
{
   // Answer here:

}
```

```
Lamp::Lamp(int n, float p)
{
   max_num_bulbs = n;
   num_bulbs = 0;
   price = p;
   bulbs = new Bulb [n];
}
```

(d) [6 points] Implement the member function `add_bulbs()` for the extended `Lamp` class as described above. (Note: do nothing if there is not enough room for adding the requested number of bulbs.)

```
void Lamp::add_bulbs(int w, float p, int n)
{
   // Answer here:

}
```

```
void Lamp::add_bulbs(int w, float p, int n)
{
    if ((num_bulbs + n) > max_num_bulbs)
       return;

    for (int j = num_bulbs; j < (num_bulbs + n); ++j)
        bulbs[j].set(w, p);
    num_bulbs += n;
}
```

(e) [6 points] Implement the member function `total_price()` for the extended `Lamp` class as described above.

```
float Lamp::total_price() const
{
   // Answer here:


}
```

```
float Lamp::total_price() const
{
  float total = price;
  for (int i=0; i<num_bulbs; i++)
     total += bulbs[i].get_price();
  return total;
}
```

Problem 5 [20 points] Stack and Queue

(a) [2 points] What is the **major difference** between Stack and Queue?

Answer:

Stacks adds and removes from the same end of a list (i.e. the top) while Queue adds and removes from opposit ends (i.e. the back and front respectively.

Or

Stack follows LIFO (FILO) policy while Queue follows FIFO (LILO) policy.

(b) [3 points] Assuming the queue class, `int_queue`, as defined in the lecture notes. What will the output of the following program be? (Note: the definition of `int_queue` is given in Appendix C for your reference.)

```
#include "int-queue.h"

void print_queue_info(const int_queue& a) {
    cout << "No. of data currently on the queue = " << a.size() << "\t";
    if (!a.empty()) cout << "Front item = " << a.front();
    cout << endl << "Empty: " << boolalpha << a.empty();
    cout << "\t\t" << "Full: " << boolalpha << a.full() << endl << endl;
}

int main() {
   int n;
   int_queue q;
   q.enqueue(5);
   q.enqueue(3);
   q.dequeue();
   q.enqueue(7);
   q.enqueue(1);
   n = q.front();
   q.enqueue(n);
   q.enqueue(9);
   q.dequeue();
   n = q.size();
   q.enqueue(n);
   if (q.full())
      q.dequeue();
   print_queue_info(q);
   return 0;
}
```

Answer:

```
No. of data currently on the queue = 4  Front item = 1
Empty: false            Full: false
```

For questions 5(c) to 5(e), we are going to implement a class called, `priority_queue`. A priority queue is an abstract data type just like a queue. But different from a regular queue, in a priority queue, elements are assigned with priorities. The element with the highest priority is served or removed first. For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.

In the `priority_queue` class, linked list is used to store the elements (characters). The structure definition, `ll_cnode`, is modified with an additional member variable, `priority`. (Note: a smaller value means a higher priority.)

You are given the struct definition of `ll_cnode` and the class definition of `priority_queue` in the header file "priority-queue.h":

```
#include <iostream>
using namespace std;

struct ll_cnode
{
   char data;
   int priority;
   ll_cnode* next;
};

class priority_queue
{
   private:
      ll_cnode* head;

   public:
      priority_queue();
      ~priority_queue();

      char front() const;
      int size() const;
      bool empty() const;
      void print() const;
      int mystery(int) const;

      void enqueue(char, int);
      void dequeue();
};
```

Here is a sample main function for using the `priority_queue` class:

```
#include "priority-queue.h"

int main()
{
   priority_queue queue;
```

```
    queue.enqueue('a', 5);
    queue.print();
    queue.enqueue('b', 1);
    queue.print();
    queue.enqueue('c', 2);
    queue.print();
    queue.dequeue();
    queue.print();
    queue.enqueue('d', 3);
    queue.print();
    queue.enqueue('e', 2);
    queue.print();
    queue.enqueue('f', 2);
    queue.print();

    return 0;
}
```

And the corresponding output:

```
From front(i.e. highest priority) to end(i.e. lower priority):
(a, 5)
From front(i.e. highest priority) to end(i.e. lower priority):
(b, 1) (a, 5)
From front(i.e. highest priority) to end(i.e. lower priority):
(b, 1) (c, 2) (a, 5)
From front(i.e. highest priority) to end(i.e. lower priority):
(c, 2) (a, 5)
From front(i.e. highest priority) to end(i.e. lower priority):
(c, 2) (d, 3) (a, 5)
From front(i.e. highest priority) to end(i.e. lower priority):
(c, 2) (e, 2) (d, 3) (a, 5)
From front(i.e. highest priority) to end(i.e. lower priority):
(c, 2) (e, 2) (f, 2) (d, 3) (a, 5)
```

(c) [3 points] Suppose `priority_queue` has the following member function, what does it return?

```
int priority_queue::mystery(int n) const
{
   int i = 0;
   for (ll_cnode* current = head; current != nullptr; current = current->next)
   {
      if (n > current->priority)
         i++;
   }
   return i;
}
```

Answer:

It counts the number of nodes (elements) in the priority queue with higher priority (smaller priority values) than n.

(d) [6 points] Implement a member function, `dequeue()`, for performing the operation **dequeue** of a priority queue. Make sure there is no memory leak.

```
char priority_queue::dequeue()
{
   // Answer here:


}
```

```
void priority_queue::dequeue()
{
   if (head != nullptr)
   {
      ll_cnode* temp = head;
      head = head->next;
      delete temp;
   }
}
```

(e) [6 points] Implement the member function `enqueue()`. Insert a node with data as d and priority as p into the appropriate position of the linked list.

```
void priority_queue::enqueue(char d, int p)
{
   // Answer here:



}
```

```
void priority_queue::enqueue(char d, int p)
{
   ll_cnode* temp = head;
   ll_cnode* new_node = new ll_cnode;
   new_node->data = d;
   new_node->priority = p;
   new_node->next = nullptr;

   if (head == nullptr) {
      head = new_node;
      return;
   }

   if (head->priority > p ) {
      // insert head
      new_node->next = head;
      head = new_node;
   } else {

      while ((temp->next != nullptr) && (temp->next->priority <= p))
         temp = temp->next;
      new_node->next = temp->next;
      temp->next = new_node;
   }
}
```

Appendix A

```
#include <iostream> /* File: ll_cnode.h */
using namespace std;

struct ll_cnode
{
    char data;          // Contains useful information
    ll_cnode* next;     // The link to the next node
};

const char NULL_CHAR = '\0';
ll_cnode* ll_create(char);
ll_cnode* ll_create(const char []);
int ll_length(const ll_cnode*);
void ll_print(const ll_cnode*);
ll_cnode* ll_search(ll_cnode*, char c);
void ll_insert(ll_cnode*&, char, unsigned);
void ll_delete(ll_cnode*&, char);
void ll_delete_all(ll_cnode*&);

#include "ll_cnode.h"   /* File: ll_create.cpp */
// Create a ll_cnode and initialize its data
ll_cnode* ll_create(char c)
{
    ll_cnode* p = new ll_cnode; p->data = c; p->next = nullptr; return p;
}

// Create a linked list of ll_cnodes with the contents of a char array
ll_cnode* ll_create(const char s[])
{
    if (s[0] == NULL_CHAR) // Empty linked list due to empty C string
        return nullptr;

    ll_cnode* head = ll_create(s[0]); // Special case with the head

    ll_cnode* p = head; // p is the working pointer
    for (int j = 1; s[j] != NULL_CHAR; ++j)
    {
        p->next = ll_create(s[j]); // Link current cnode to the new cnode
        p = p->next; // p now points to the new ll_cnode
    }

    return head; // The WHOLE linked list can be accessed from the head
}

#include "ll_cnode.h"   /* File: ll_print.cpp */

void ll_print(const ll_cnode* head)
{
    for (const ll_cnode* p = head; p != nullptr; p = p->next)
        cout << p->data;
    cout << endl;
}
```

```cpp
#include "ll_cnode.h"   /* File: ll_search.cpp */

// The returned pointer may be used to change the content
// of the found ll_cnode. Therefore, the return type
// should not be const ll_cnode*.

ll_cnode* ll_search(ll_cnode* head, char c)
{
    for (ll_cnode* p = head; p != nullptr; p = p->next)
    {
        if (p->data == c)
            return p;
    }

    return nullptr;
}

#include "ll_cnode.h" /* File: ll_length.cpp */

int ll_length(const ll_cnode* head)
{
    int length = 0;
    for (const ll_cnode* p = head; p != nullptr; p = p->next)
        ++length;
    return length;
}

#include "ll_cnode.h"   /* File: ll_insert.cpp */

// To insert character c to the linked list so that after insertion,
// c is the n-th character (counted from zero) in the list.
// If n > current length, append to the end of the list.

void ll_insert(ll_cnode*& head, char c, unsigned n)
{
    // STEP 1: Create the new ll_cnode
    ll_cnode* new_cnode = ll_create(c);

    // Special case: insert at the beginning
    if (n == 0 || head == nullptr)
    {
        new_cnode->next = head;
        head = new_cnode;
        return;
    }


    // STEP 2: Find the node after which the new node is to be added
    ll_cnode* p = head;
    for (int position = 0;
         position < n-1  &&  p->next != nullptr;
         p = p->next, ++position)
         ;

    // STEP 3,4: Insert the new node between
    //           the found node and the next node
    new_cnode->next = p->next; // STEP 3
    p->next = new_cnode;       // STEP 4
}
```

```
#include "ll_cnode.h"   /* File: ll_delete.cpp */
// To delete the character c from the linked list.
// Do nothing if the character cannot be found.
void ll_delete(ll_cnode*& head, char c)
{
    ll_cnode* prev = nullptr; // Point to previous ll_cnode
    ll_cnode* current = head; // Point to current ll_cnode

    // STEP 1: Find the item to be deleted
    while (current != nullptr && current->data != c)
    {
        prev = current;      // Advance both pointers
        current = current->next;
    }

    if (current != nullptr)  // Data is found
    {   // STEP 2: Bypass the found item
        if (current == head) // Special case: delete the first item
            head = head->next;
        else
            prev->next = current->next;

        delete current;      // STEP 3: Free up the memory of the deleted item
    }
}

#include "ll_cnode.h"   /* File: ll_delete_all.cpp */

// To delete the WHOLE linked list, given its head by recursion.
void ll_delete_all(ll_cnode*& head)
{
    if (head == nullptr) // An empty list; nothing to delete
        return;

    // STEP 1: First delete the remaining nodes
    ll_delete_all(head->next);

    // For debugging: this shows you what are deleting
    cout << "deleting " << head->data << endl;

    delete head;         // STEP 2: Then delete the current nodes
    head = nullptr;      // STEP 3: To play safe, reset head to nullptr
}
```

Appendix B

```
/* File: bulb.h */

class Bulb
{
  private:
    int wattage;        // A light bulb's power in watt
    float price;        // A light bulb's price in dollars

  public:
    int get_power() const;
    float get_price() const;
    void set(int w, float p); // w = bulb's wattage; p = its price
};

#include "bulb.h"       /* File: lamp.h */
class Lamp
{
  private:
    int  num_bulbs; // A lamp MUST have 1 or more light bulbs
    Bulb* bulbs; // Dynamic array of light bulbs installed onto a lamp
    float price; // Price of the lamp, not including its bulbs

  public:
    Lamp(int n, float p);      // n = number of bulbs; p = lamp's price
    ~Lamp();

    int total_power() const;   // Total power/wattage of its bulbs
    float total_price() const; // Price of a lamp PLUS its bulbs

    // Print out a lamp's information; see outputs from our example
    void print(const char* prefix_message) const;

    // All light bulbs of a lamp have the same power/wattage and price:
    // w = a light bulb's wattage; p = a light bulb's price
    void install_bulbs(int w, float p);
};

/* File: bulb.cpp */

#include "bulb.h"

int Bulb::get_power() const { return wattage; }

float Bulb::get_price() const { return price; }

void Bulb::set(int w, float p) { wattage = w; price = p; }

#include "lamp.h"       /* File: lamp.cpp */
#include <iostream>
using namespace std;

Lamp::Lamp(int n, float p)
          { num_bulbs = n; price = p; bulbs = new Bulb [n]; }
Lamp::~Lamp() { delete [] bulbs; }
int Lamp::total_power() const
          { return num_bulbs*bulbs[0].get_power(); }
float Lamp::total_price() const
```

```
          { return price + num_bulbs*bulbs[0].get_price(); }
void Lamp::print(const char* prefix_message) const
{
    cout << prefix_message << ": total power = " << total_power()
         << "W" << " , total price = $" << total_price() << endl;
}
void Lamp::install_bulbs(int w, float p)
{
    for (int j = 0; j < num_bulbs; ++j)
        bulbs[j].set(w, p);
}
```

Appendix C

```
#include <iostream>     /* File: int-queue.h */
#include <cstdlib>
using namespace std;
const int BUFFER_SIZE = 5;

class int_queue // Circular queue
{
  private:
    int data[BUFFER_SIZE]; // Use an array to store data
    int num_items;         // Number of items on the queue
    int first;             // Index of the first item; start from 0

  public:
    // CONSTRUCTOR member functions
    int_queue();           // Default constructor

    // ACCESSOR member functions: const => won't modify data members
    bool empty() const;    // Check if the queue is empty
    bool full() const;     // Check if the queue is full
    int size() const;      // Give the number of data currently stored
    int front() const;     // Retrieve the value of the front item
    // MUTATOR member functions
    void enqueue(int);     // Add a new item to the back of the queue
    void dequeue();        // Remove the front item from the queue
};

#include "int-queue.h"  /* File: int-queue1.cpp */

          /***** Default CONSTRUCTOR member function *****/
// Create an empty queue
int_queue::int_queue() { first = 0; num_items = 0; }

          /***** ACCESSOR member functions *****/
// Check if the int_queue is empty
bool int_queue::empty() const { return (num_items == 0); }

// Check if the int_queue is full
bool int_queue::full() const { return (num_items == BUFFER_SIZE); }

// Give the number of data currently stored
int int_queue::size() const { return num_items; }

// Retrieve the value of the front item
int int_queue::front() const
{
    if (!empty())
        return data[first];

    cerr << "Warning: Queue is empty; can't retrieve any data!" << endl;
    exit(-1);
}

#include "int-queue.h" /* File: int-queue2.cpp */

void int_queue::enqueue(int x)  // Add a new item to the back of the queue
{
    if (!full())
```

```
    {
        data[(first+num_items) % BUFFER_SIZE] = x;
        ++num_items;
    } else {
        cerr << "Error: Queue is full; can't add (" << x << ")!" << endl;
        exit(-1);
    }
}

void int_queue::dequeue()       // Remove the front item from the queue
{
    if (!empty())
    {
        first = (first+1) % BUFFER_SIZE;
        --num_items;
    } else {
        cerr << "Error: Queue is empty; can't remove any data!" << endl;
        exit(-1);
    }
}
```

---------- END OF PAPER ----------