COMP 2011 Final Exam - Fall 2015 - HKUST

Date: December 16, 2015 (Wednesday)
Time Allowed: 3 hours, 4:30–7:30pm
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **10** questions on **27** pages (including this cover page).
3. Write your answers in the space provided in black/blue ink. *NO pencil please, otherwise you are not allowed to appeal for any grading disagreements.*
4. All programming codes in your answers must be written in ANSI C++.
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	Model Answer
Student ID	0000
Email Address	
Lecture & Lab Section	

For T.A. Use Only

Problem	Score
1	/ 5
2	/ 10
3	/ 6
4	/ 7
5	/ 6
6	/ 6
7	/ 14
8	/ 15
9	/ 16
10	/ 15
Total	/ 100

Problem 1 [5 points] True or false

Indicate whether the following statements are *true* or *false* by circling **T** or **F**. You get 1.0 point for each correct answer, −0.5 for each wrong answer, and 0.0 if you do not answer.

T **F** (a) Defining a reference variable requires no static or dynamic allocation of memory.

T **F** (b) A program with an infinite recursion will run forever until you manually stop the program.

T **F** (c) Elements in a (one-dimensional) C++ array are guaranteed to be contiguous in memory regardless of whether the array is static or dynamic.

T **F** (d) A `10x10` 2D array allocated statically occupies the same amount of memory as a `10x10` 2D array allocated dynamically.

T **F** (e) Every class must have an explicitly defined destructor.

Problem 2 [10 points] Multiple choice questions

Circle the letter to the left of the correct answers. There is only one correct answer per question. Each question is worth **2** points.

I. Which of the following statements about function overloading and default arguments is correct?

(a) An overloaded function cannot have default arguments.

(b) An overloaded function can have no more than one default argument.

(c) No default arguments of an overloaded function can be of user-defined data types.

(d) All arguments of an overloaded function can be default.

II. The following program

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

int main( )
{
    int x;
    int* xp = &x;

    delete xp;
    xp = NULL;

    return 0;
}
```

(a) does not compile and there will be compilation error(s).

(b) compiles but there will be compilation warning(s). Then it runs with no error(s).

(c) compiles but when it runs, it terminates with runtime error(s).

(d) compiles and runs with no error(s).

III. A programmer wants to store and manipulate 10 integer values. Which of the following data structures occupies the most amount of memory?

(a) A binary tree with 10 nodes.

(b) A linked list with 10 nodes.

(c) An array with 10 entries.

(d) They all occupy the same amount of memory.

IV. Given the following program code:

```
const int a = 10;
const int* b = &a;
const int* const c = &a;
int d = 20;
```

which of the following is legal (i.e., syntactically correct)?

(a) `*b += 1;`

(b) `b = &d;`

(c) `c = &d;`

(d) `b = &c;`

V. Which of the following about `const` member functions is correct?

(a) They must be public (member functions).

(b) They must be put before the non-`const` member functions in the class definition.

(c) They cannot modify any private class members but only public class members.

(d) They cannot modify any class members.

Problem 3 [6 points] Scope and Functions

Determine the output of the following program.

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

int n = 10;

int fn(int a, int b)
{
    cout << "fn(int, int) = ";
    n = (a*b)/2;
    return (a + b + n);
}

int fn(double a, int b)
{
    cout << "fn(double, int) = ";
    int n = (a*b)/2;
    return (a + b + n);
}

int main( )
{
    int result;
    result = fn(n/2.0, n);
    cout << result << ", n = " << n << endl;

    result = fn(n/2, n);
    cout << result << ", n = " << n << endl;
    return 0;
}
```

```
fn(double, int) = 40, n = 10
fn(int, int) = 40, n = 25
```

Answer: _____

Grading scheme: 1 point each

Problem 4 [7 points] Array and Pointer

(a) Determine the output when the following program is compiled and executed on a 32-bit machine.

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

int main( )
{
    char name[50] = "introduction to object-oriented programming";
    char c = *(name + sizeof(name) / 10);
    cout << c << ' ';

    char* a = name;
    c = *(a + sizeof(a)) + 1;
    cout << c << endl;
    return 0;
}
```

d p

Answer: _____

(b) Given the following definition and initialization of the variable `types`,

```
char types[3][10] = {"int", "double", "float"};
```

complete the table below assuming the statements are compiled and executed on a 32-bit machine. The equivalent indexing expression must only use array indexing operator with *no* pointer arithmetic *nor* dereferencing operator.

Expression	Equivalent Indexing Expression	Value
**types	types[0][0]	'i'
*(types[1])	types[1][0]	'd'
*(types[2] + 1)	types[2][1]	'l'
((types + 1) + 3)	types[1][3]	'b'

Grading scheme: (a) 2 points each (b) 0.5 points each

Problem 5 [6 points] Constructor and Destructor

What is the output when the following program is run?

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

class Foo
{
  public:
    Foo( ) { cout << "C "; }
    ~Foo( ) { cout << "D "; }
};

void func( ) { Foo* p = new Foo; p = NULL; }

int main( )
{
    Foo a;

    for (int i = 0; i < 1; i++)
    {
        Foo b;

        if (true)
        {
            Foo c;
        }

        func( );
    }

    return 0;
}
```

C C C D C D D

Answer: _____

Grading scheme: (a) 0.8 points each; full mark when all are correct

Problem 6 [6 points] Stack

We discussed in class an array implementation of stack. To refresh your memory, the header file is shown below:

```
#include <iostream>                                    /* File: int-stack.h */
#include <cstdlib>
using namespace std;
const int BUFFER_SIZE = 100;

class int_stack
{
  private:
    int data[BUFFER_SIZE];                          // Use an array to store data
    int top_index;                                  // Start from 0; -1 when empty

  public:
    int_stack(void);                                // Default constructor
    bool empty(void) const;                         // Check if the stack is empty
    bool full(void) const;                          // Check if the stack is full
    int size(void) const;                           // Give the number of items currently stored
    int top(void) const;                            // Retrieve the value of the top item
    void push(int);                                 // Add a new item to the top of the stack
    void pop(void);                                 // Remove the top item from the stack
};
```

We now define two `void` non-member functions with their function prototypes given below:

```
void insert_at_bottom(int_stack& stack, int item);
void reverse_stack(int_stack& stack);
```

Unlike the `push()` member function which adds a new item to the top of the stack, `insert_at_bottom()` adds a new item to the bottom of the stack if the stack is not full. As for `reverse_stack()`, it reverses the order of all the items in the stack. For example, if the items in the stack from top to bottom are $(10, 20, 30)$, applying `reverse_stack()` to the stack will change their order to $(30, 20, 10)$.

Assuming that the function `insert_at_bottom()` has been implemented, implement `reverse_stack()` using `insert_at_bottom()` and the member functions of `int_stack` (and possibly recursive call to itself). You are not allowed to define any additional object such as a stack or an array except a single `int` variable.

Answer:

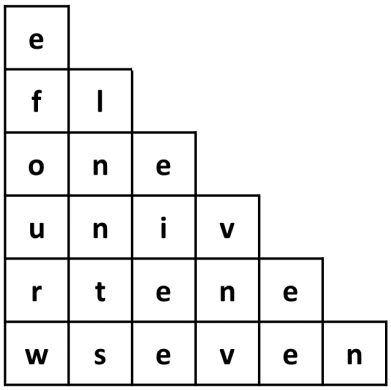
```
/*
 * General guideline:
 * max -2 points for syntax errors. Repeated syntax errors count only once.
 */

void reverse_stack(int_stack& stack)
{
    if (stack.empty())
        return;                              // 0.5 point for correct behavior when empty
    else
    {
        int top_item = stack.top();
        stack.pop();                                 // 1.5 points for top and pop together
        reverse_stack(stack);                // 2 points for recursion on the reduced stack
        insert_at_bottom(stack, top_item);                                    // 2 points
    }
}
```

Problem 7 [14 points] Word Search Puzzle

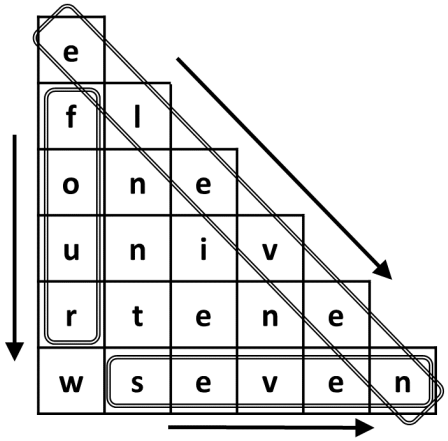
A word search puzzle presents a grid of seemingly random letters and a list of words. The goal of the puzzle is to find and mark all the words hidden within the grid, searching along straight lines in three directions:

- *horizontal*: from left to right
- *vertical*: from top to bottom
- *diagonal*: from upper left to bottom right

Below is an example of a word search puzzle in a triangular grid.



You can find the word "eleven" in the diagonal direction, the word "four" in the vertical direction and "seven" in the horizontal direction.



Below is an incomplete program to generate a word search puzzle in a triangular grid with a given list of words. Right now, it only has the definitions of various data structures, function prototypes, and the main function. You are asked to complete the implementation of 3 of the functions. (You don't need to implement the remaining 2 functions: `longest_word_length` and `print_grid`.)

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

enum Direction { HORIZONTAL, VERTICAL, DIAGONAL };

struct Position { int row; int col; };                    // both indexes start from zero

struct PuzzleWord
{
    const char* word;                                     // the content of the word
    Position pos;              // the starting position (row, col) in the word search puzzle
    Direction dir;                  // the direction of the word in the word search puzzle
};

// add a word to the 2D puzzle array
bool add_word(char** puzzle, int n, const PuzzleWord* candidate_word);
void init_grid(char**& puzzle, int n);          // dynamically create the 2D puzzle array
void destroy_grid(char**& puzzle, int n);       // deallocate the dynamic 2D puzzle array

/* YOU DON'T NEED TO IMPLEMENT THE FOLLOWING 2 FUNCTIONS */
// return the length of the longest word in the words array with n items
int longest_word_length(PuzzleWord words[ ], int n);
void print_grid(const char* const* puzzle, int n);    // print contents of the 2D puzzle

int main( )
{
    const int NUM_WORDS = 8;
    PuzzleWord candidate_words[NUM_WORDS] = {
        {"one", {2,0}, HORIZONTAL},
        {"eleven", {0,0}, DIAGONAL},
        {"nine", {2,1}, DIAGONAL},
        {"four", {1,0}, VERTICAL},
        {"ten", {4,1}, HORIZONTAL},
        {"seven", {5,1}, HORIZONTAL},
        {"twelve", {1,1}, DIAGONAL},
        {"two", {2,1}, VERTICAL} };

    cout << "The list of words:  ";
    for (int i = 0; i < NUM_WORDS; i++)
        cout << candidate_words[i].word << ' ';
```

```
    cout << endl;

    int size = longest_word_length(candidate_words, NUM_WORDS);
    char** word_search_puzzle = NULL;

    cout << endl << "Initializing the grid:" << endl;
    init_grid(word_search_puzzle, size);
    print_grid(word_search_puzzle, size);

    cout << endl << "Adding words to the grid:" << endl;
    for (int i = 0; i < NUM_WORDS; i++)
        if (!add_word(word_search_puzzle, size, &candidate_words[i]))
            cout << "Fail to add the word:  " << candidate_words[i].word << endl;

    print_grid(word_search_puzzle, size);
    destroy_grid(word_search_puzzle, size);
    return 0;
}
```

The program will give the following output:

```
The list of words: one eleven nine four ten seven twelve two

Initializing the grid:
-
-,-
-,-,-
-,-,-,-
-,-,-,-,-
-,-,-,-,-,-

Adding words to the grid:
Fail to add the word: twelve
Fail to add the word: two
e
f,l
o,n,e
u,_,i,v
r,t,e,n,e
_,s,e,v,e,n
```

(a) [4 points] Complete the implementation of the function `init_grid(char**& puzzle, int n)` to dynamically create a triangular array of `n` rows and initialize every element in the array by the character '_'. A triangular array is a 2-dimensional array in which the first row is just big enough to store only 1 element (character in our case), the second row is just big enough to store only 2 elements, and so on. The address of the created triangular array is stored in the variable, `puzzle`. You may assume that n is a positive integer and is always given correctly.

Answer:

```
void init_grid(char**& puzzle, int n)
{
    puzzle = new char*[n];                    // 1 point for allocating this column

    for (int i = 0; i < n; i++)
    {
        puzzle[i] = new char[i+1];            // 2 points for allocating the rows

        for (int j = 0; j <= i; j++)          // 1 point for initialization
            puzzle[i][j] = '_';
    }
}
```

(b) [4 points] Complete the implementation of the function `destroy_grid(char**& puzzle, int n)` to deallocate the dynamic triangular array pointed by the variable, `puzzle`, which has `n` rows. Reset the puzzle to NULL after it is destroyed in your function. Again, you may assume that n is a positive integer and is always given correctly.

Answer:

```
void destroy_grid(char**& puzzle, int n)
{
    for (int i = 0; i < n; i++)           // 2 points for deallocating the rows
        delete [ ] puzzle[i];

    delete [ ] puzzle;                    // 1 point for deallocating the column
    puzzle = NULL;                        // 1 point for resetting it to NULL
}
```

(c) [6 points] Complete the implementation of the function
`add_word(char** puzzle, int n, const PuzzleWord* candidate_word)`
to add one word in the `PuzzleWord` object pointed to by `candidate_word` to the specified position and in the specified direction. For example, in the program, the word "eleven" is to be added starting at the row with index 0 and the column with index 0 in the diagonal direction, while the word "four" is to be added starting at the row with index 1 and column with index 0 in the vertical direction. Return true if the word can be fitted in the triangular array, `puzzle`; otherwise, return false. Again, you may assume that n is a positive integer and is always given correctly.

Note: You may use the function `strlen(const char* str)` to find the length of the C string `str`, assuming that the `cstring` library has been properly included.

Answer:

```
bool add_word(char** puzzle, int n, const PuzzleWord* candidate_word)
{
    int row = candidate_word->pos.row;
    int col = candidate_word->pos.col;
    int len = strlen(candidate_word->word);
    Direction d = candidate_word->dir;

    // Checking out-of-bound cases
    if ((row < 0) || (col < 0) || (col > row))
        return false;                                          // 0.5 point

    if ((d == HORIZONTAL) && ((col + len) > (row + 1)))
        return false;                                          // 0.5 point

    else if ((d == VERTICAL) && ((row + len) > n))
        return false;                                          // 0.5 point

    else if ((d == DIAGONAL) && (((row + len) > n) || ((col + len) > n)))
        return false;                                          // 0.5 point
```

```
    /* Basic Solution */
    // 2 points for setting up the loop
    int row_change = 0, col_change = 0;

    if (d == HORIZONTAL)
        col_change = 1;

    else if (d == VERTICAL)
        row_change = 1;

    else if (d == DIAGONAL)
        row_change = col_change = 1;

    for (int i = 0, r = row, c = col ; i < len; i++, r += row_change, c += col_change)
    {
        // 1 point for checking whether the positions are valid
        // i.e. the positions are empty or not occupied by other characters
        if ((puzzle[r][c] != '_') && (candidate_word→word[i] != puzzle[r][c]))
            return false;
    }

    // 1 point for adding the word to the puzzle
    for (int i = 0, r = row, c = col ; i < len; i++, r += row_change, c += col_change)
        puzzle[r][c] = candidate_word→word[i];

    return true;
}
```

```
/****** Alternate Solution for the actual adding of the word *****/

    for (int i = 0; i < len; i++)
    {
        if ((puzzle[row][col] != '_') && (candidate_word→word[i] != puzzle[row][col]))
            return false;

        if (d == HORIZONTAL)
            col += 1;
        else if (d == VERTICAL)
            row += 1;
        else if (d == DIAGONAL)
        {
            row += 1; col += 1;
        }
    }

    row = candidate_word→pos.row;
    col = candidate_word→pos.col;

    for (int i = 0; i < len; i++)
    {
        puzzle[row][col] = candidate_word→word[i];

        if (d == HORIZONTAL)
            col += 1;
        else if (d == VERTICAL)
            row += 1;
        else if (d == DIAGONAL)
        {
            row += 1; col += 1;
        }
    }
```

Problem 8 [15 points] Recursion

We discussed in class a recursive function for solving the Tower of Hanoi problem. The function, renamed as `toh3()` here for the Tower of Hanoi problem with 3 pegs, is rewritten as follows:

```
void move(int num_discs, char src, char dest)
{
    cout << "Move disc " << num_discs << " from peg " << src
         << " to peg " << dest << endl;
}

void toh3(int num_discs, char src, char aux, char dest)
{
    if (num_discs == 0)
        return;
    else
    {
        toh3(num_discs - 1, src, dest, aux);
        move(num_discs, src, dest);
        toh3(num_discs - 1, aux, src, dest);
    }
}
```

(a) [5 points] Define a function called `toh3_numcalls()` with the following function prototype that, in addition to printing the same output as in `toh3()`, returns the number of function calls invoked:

```
int toh3_numcalls(int num_discs, char src, char aux, char dest);
```

For example, `toh3_numcalls(2, 'A', 'B', 'C')` returns 3 and `toh3_numcalls(3, 'A', 'B', 'C')` returns 7. You may assume without checking that `num_discs` is a nonnegative integer.

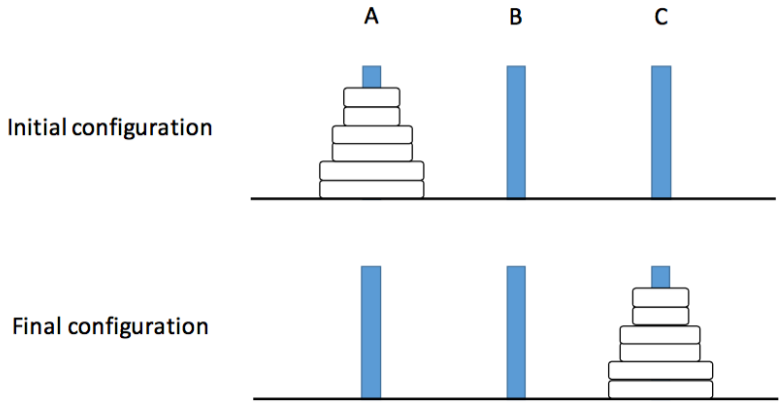
Answer:

```
int toh3_numcalls(int num_discs, char src, char aux, char dest)
{
    if (num_discs == 0)
        return 0;                                          // 1 point for the base case
    else
    {
        int count = 1;                                     // 0.5 for counting the current case
        count += toh3_numcalls(num_discs - 1, src, dest, aux);   // 1 point
        move(num_discs, src, dest);                        // 1 point
        count += toh3_numcalls(num_discs - 1, aux, src, dest);   // 1 point
        return count;                                      // 0.5 point
    }
}
```

(b) [5 points] Define a function called `dtoh3()` with the following function prototype that extends `toh3()` to solve the Double Tower of Hanoi problem:

```
void dtoh3(int num_discs, char src, char aux, char dest);
```

The Double Tower of Hanoi problem starts with an even number ($2n$) of discs on the source peg where there are two discs for each of n different sizes (see the figure below for an example with $n = 3$). Like the original Tower of Hanoi problem, only one disc may be moved at a time and at no time may a larger disc be placed on top of a smaller one. However, a disc may be placed on top of another one of the same size. Like `toh3()`, `dtoh3()` should be implemented in such a way that the number of moves needed is the minimum. You may assume without checking that `num_discs` is a nonnegative even integer.



Answer:

```
void dtoh3(int num_discs, char src, char aux, char dest)
{
    if (num_discs == 0)
        return;                                          // 1 point for the base case
    else
    {
        dtoh3(num_discs - 2, src, dest, aux);                          // 1 point

        cout << "Move discs " << num_discs - 1 << " and " << num_discs
             << " in two steps from peg " << src << " to peg " << dest
             << endl;                                                  // 2 points

        dtoh3(num_discs - 2, aux, src, dest);                          // 1 point
    }
}
```

(c) [5 points] Define a function called `toh4()` with the following function prototype that extends `toh3()` to solve the Tower of Hanoi problem with 4 pegs:

```
void toh4(int num_discs, char src, char aux1, char aux2, char dest);
```

With one more auxiliary peg, `aux2`, in general fewer moves are needed to move all the discs from the source peg to the destination peg. For example, while `toh3(4, 'A', 'B', 'C')` takes 15 moves to complete, `toh4(4, 'A', 'B', 'C', 'D')` only takes 9 steps with the following moves by taking advantage of both auxiliary pegs:

```
Move disc 1 from peg A to peg D
Move disc 2 from peg A to peg B
Move disc 1 from peg D to peg B
Move disc 3 from peg A to peg C
Move disc 4 from peg A to peg D
Move disc 3 from peg C to peg D
Move disc 1 from peg B to peg C
Move disc 2 from peg B to peg D
Move disc 1 from peg C to peg D
```

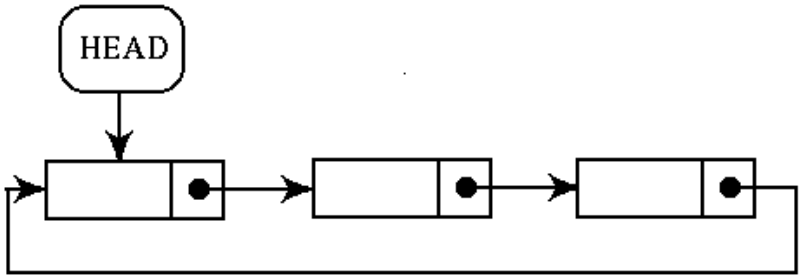
You may assume without checking that `num_discs` is a nonnegative integer. Note that you *must* use both auxiliary pegs in a meaningful way in your scheme with an aim at minimizing the number of moves in the general TOH problem with $n \geq 3$.

Answer:

```
void toh4(int num_discs, char src, char aux1, char aux2, char dest)
{
    if (num_discs == 0)
        return;                                    // 0.5 point for the base case I
    else if (num_discs == 1)
        move(num_discs, src, dest);                // 0.5 point for the base case II
    else
    {
        toh4(num_discs - 2, src, aux2, dest, aux1);                    // 1 point
        move(num_discs - 1, src, aux2);
        move(num_discs, src, dest);
        move(num_discs - 1, aux2, dest);           // 2 points for the 3 moves
        toh4(num_discs - 2, aux1, src, aux2, dest);                    // 1 point
    }
}
```

Problem 9 [16 points] Circular Linked List

A circular linked list (CLL) is a variation of the linked list in which the last node points back to the first node (i.e., the head). This data structure can be useful in situations where one wants to repeatedly iterate over the nodes, since no special handling is necessary to return to the beginning of the list.



In this question, the structure of a node in a CLL, called `cll_node`, is defined as follows to store characters:

```
struct cll_node
{
    char data;                          // the stored information
    cll_node* next;                     // the link to the next node
};
```

Note: In the special case when a CLL consists of only one node, the node's next pointer points to itself.

Implement the following functions of a CLL.

(a) [3 points] Implement the function `cll_create`. It creates a new `cll_node` which stores the given character `c` and its `next` points to what the given `next_node` points to.

```
cll_node* cll_create(char c, cll_node* next_node = NULL)
{
    // ADD YOUR CODE HERE
    cll_node* p = new cll_node;                          // 1 point
    p->data = c;                                         // 0.5 point
    p->next = next_node;                                 // 0.5 point
    return p;                                            // 0.5 point
}

// Give full mark of 3 points when all are correct.
```

(b) [3 points] Implement the function `cll_length` that finds the number of nodes in a CLL pointed to by the given `head` pointer.

```
int cll_length(const cll_node* head)
{
    // ADD YOUR CODE HERE
    if (head == NULL)
        return 0;                                // 0.5 points for the base case

    int length = 1;
    for (const cll_node* p = head->next; p != head; p = p->next)      // 1.5 points
        ++length;                                                     // 0.5 point

    return length;                                                    // 0.5 point
}
```

(c) [6 points] Complete the function `cll_insert` by following the comments in the code. It inserts the given character `c` to the CLL given its `head` pointer so that after insertion, `c` is the n-th character (counted from zero) in the list. If n is greater then the current length of the CLL, append the character to the end of the list.

```
void cll_insert(cll_node*& head, char c, int n)
{
    if (head == NULL)                                   // Special case: empty list
    {                                                  // ADD YOUR CODE HERE
        head = cll_create(c);                                          // 0.5 point
        head->next = head;                                             // 0.5 point
        return;                                       // missing return -0.5 point
    }

    if (n == 0)                                         // Special case: insert at head
    {                                  // ADD YOUR CODE HERE /* shorter solution */
        cll_node* p = cll_create(head->data, head->next);              // 1 point
        head->data = c; head->next = p;                                // 1 point
        return;                                       // missing return -0.5 point

        /* Alternative longer solution */
        cll_node* p = head;
        for (; p->next != head; p = p->next)                // 1 point for the loop
            ;

        head = p->next = cll_create(c, p->next);                       // 1 point
        return;                                       // missing return -0.5 point
    }

    // Find the node after which the new node is to be added
    // Then create and insert new node between found node and next node
    // ADD YOUR CODE HERE
    // 1 point for the position loop; 1.5 point for the p loop
    cll_node* p = head;
    for (int position = 0;
         position < n-1 && p->next != head;
         p = p->next, ++position)
        ;
    p->next = cll_create(c, p->next);                                  // 0.5 point
}
```

(d) [4 points] Complete the following function `cll_quick_insert_at_end` which creates a new `cll_node` with content `c` and inserts it at the end of the CLL with the given `head` pointer. That is, after the insertion, the new node is pointed to by the former last node of the CLL and it is now pointing to the head. To simplify the problem, you may assume that the list contains at least 3 nodes.

Requirement: You are **NOT** allowed to traverse the entire list in the process of determining where to insert the new node. Your function should be able to quickly insert the new node in an amount of time that is *independent* of the length of the list.

Hint: Note that the head pointer is passed by reference.

```
void cll_quick_insert_at_end(cll_node*& head, char c)
{
    // ADD YOUR CODE HERE

    // 1.5 points for duplicating the old head
    cll_node* newhead = cll_create(head→data, head→next);

    // 2 points for changing the old head so that it becomes the last node
    head→data = c;
    head→next = newhead;

    // 0.5 point for resetting the new head
    head = newhead;
}
```

Problem 10 [15 points] C++ Class: Lamp and Bulbs

This question involves two classes of objects: class `Bulb` and class `Lamp` defined in the files, "bulb.h" and "lamp.h" respectively in the following pages. There is at least one light bulb on any lamp. In `Lamp`'s constructor, the number of light bulbs must be specified, and the constructor will dynamically allocate the required number of `Bulb` objects. `Lamp`'s destructor must also deallocate all the `Bulb` objects owned by a `Lamp` object.

Note that

- all bulbs of a lamp are the same in terms of price and wattage (power);
- the price of a lamp that is passed to the `Lamp`'s constructor does not include the price of its bulbs which have to be bought separately;
- one installs bulb(s) onto a lamp by calling the member function `install_bulbs`.

Below is a testing program "lamp-test.cpp" of the two classes:

```
#include "lamp.h"

int main( )
{
    Lamp lamp1(4, 100.5);              // lamp1 costs $100.5 itself and needs 4 bulbs
    Lamp lamp2(2, 200.5);              // lamp2 costs $200.5 itself and needs only 2 bulbs

    // Install 4 bulbs of 20 Watts, each costing $30.1 on lamp1
    lamp1.install_bulbs(20, 30.1);
    lamp1.print("lamp1");

    // Install 2 bulbs of 60 Watts, each costing $50.1 on lamp2
    lamp2.install_bulbs(60, 50.1);
    lamp2.print("lamp2");

    return 0;
}
```

Here is its output.

```
lamp1: total power = 80W , total price = $220.9
lamp2: total power = 120W , total price = $300.7
```

```
/* File: bulb.h */
class Bulb                                                    // Light bulbs
{
  private:
    int wattage;                                  // A light bulb's power in watt (W)
    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 = a light bulb's wattage; p = a light bulb's price
};

/* File: lamp.h */
#include "bulb.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 the price of its bulbs

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

    // Total power/wattage of the light bulbs
    int total_power( ) const;

    // Price of a lamp PLUS its light bulbs
    float total_price( ) const;

    // 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);
};
```

Implement all member functions of the 2 classes in 2 separate files called "bulb.cpp" and "lamp.cpp", and do not forget to include other necessary files. That means, if all files are in the same folder and the executable is called "lamp-test", the testing program may be compiled as follows:

```
g++ -o lamp-test lamp-test.cpp bulb.cpp lamp.cpp
```

(a) [4 points] Implement all member functions of class `Bulb`.

Answer: `/* File "bulb.cpp" */`

```
#include "bulb.h"                                                    // 1 point

int Bulb::get_power( ) const { return wattage; }   // 1 point

float Bulb::get_price( ) const { return price; }   // 1 point

void Bulb::set(int w, float p) { wattage = w; price = p; }   // 1 point
```

(b) [11 points] Implement all member functions of class `Lamp`.

Answer: `/* File "lamp.cpp" */`

```
#include "lamp.h"                                                    // 0.5 point
#include <iostream>
using namespace std;                                  // both statements together 0.5 point

Lamp::Lamp(int n, float p)
{
    num_bulbs = n;                                                   // 0.5 point
    price = p;                                                       // 0.5 point
    bulbs = new Bulb [num_bulbs];                                    // 1 point
}

Lamp::~Lamp( ) { delete [ ] bulbs; }   // 2 points

int Lamp::total_power( ) const                                       // 1 points
{
    return num_bulbs * bulbs[0].get_power( );
}

float Lamp::total_price( ) const                                     // 1 points
{
    return price + num_bulbs * bulbs[0].get_price( );
}

void Lamp::print(const char* prefix_message) const                   // 2 points
{
    cout << prefix_message
         << ":  total power = " << total_power( ) << "W"
         << " , total price = $" << total_price( ) << endl;
}

void Lamp::install_bulbs(int w, float p)                             // 2 points
{
    for (int j = 0; j < num_bulbs; ++j)
        bulbs[j].set(w, p);
}
```