COMP 2011 Final Exam - Fall 2016 - HKUST

Date: December 17, 2016 (Saturday)
Time Allowed: 2.5 hours, 9–11.30 am
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **10** questions on **30** pages (including this cover page, 2 appendices, and 3 blank pages at the end).
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 the ANSI C++ version as taught in the class.
5. For programming questions, unless otherwise stated, you are **NOT** allowed to define additional structures, use global variables nor any library functions not mentioned in the questions.
6. For question 8, 9 and 10, you are allowed to define additional helper functions, and use the 2 C string functions in Appendix II.

Student Name	
Student ID	
Email Address	
Seat Number	

For T.A. Use Only

Problem	Score
1	/ 10
2	/ 6
3	/ 5
4	/ 6
5	/ 4
6	/ 5
7	/ 8
8	/ 23
9	/ 16
10	/ 17
Total	/ 100

Problem 1 [10 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) None of the following statements contain a reference declaration.

i. `int *i;`

ii. `swap(&x, &y)`

iii. `int x, *pt; pt = &x;`

iv. `int x, &pt = x;`

T **F** (b) All of the following statements are correct initialization of the C string "TREE'.

i. `char plant[] = "TREE";`

ii. `char plant[] = {'T', 'R', 'E', 'E', '\0'};`

iii. `char plant[80] = "TREE";`

iv. `char plant[80] = {'T', 'R', 'E', 'E', '\0'};`

T **F** (c) The following code is legal. That is, it is syntactically correct and can be compiled without errors.

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

void fun(double a) { cout << a << endl; }
void fun(double a, int b) { cout << a << ", " << b << endl; }
int fun(double a = 2.3, int b = 3) { cout << a << ", " << b << endl; }

int main()
{
    fun();
    return 0;
}
```

T **F** (d) Given 2 variables `x` and `y`, and `y` is a reference to `x`, then the 2 variables have the same address. For example, if we have

```
int x = 5;
int& y = x;
```

then `cout << &x;` and `cout << &y;` will print out the same value.

T **F** (e) The following statement is syntactically incorrect, and will produce errors when it is compiled.

```
int * const temp;
```

T **F** (f) The following code can be executed without errors and the output will be:

```
cabinet    10
```

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

struct item
{
    char* name;
    int id;
};

void func(item& a)
{
    char text[20] = "cabinet";
    item temp = { &text[0], 10 };
    a = temp;
}

int main()
{
    item a;
    func(a);
    cout << a.name << "\t" << a.id << endl;
    return 0;
}
```

T **F** (g) Let `x` be the identifier for a 2D array. Regardless of whether `x` represents a static or dynamic array, its array element `x[i][j]` is equivalent to the following expression: `*(*(x+i)+j)`.

T **F** (h) One cannot create objects of class X given its following definition:

```
class X
{
    int a;
    float b;
};
```

because all its data members are private and there are no public constructor(s).

T **F** (i) A class definition may have multiple (i.e., more than one) constructors and multiple destructors insofar as they have different function signatures as required by function overloading.

T **F** (j) The following code is legal, that is, it is syntactically correct and can be compiled without errors.

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

class item
{
  private:
    int data;
    void func1(int n = 10) { data = n; }
  public:
    void func2() const { func1(); }
};

int main()
{
    item temp;
    temp.func2();
    return 0;
}
```

Problem 2 [6 points] Loop and Control

What are the outputs of the following program?

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

bool odd(unsigned int x)
{
    return (x%2);
}

int Russian_peasant(unsigned int m, unsigned int n)
{
    int sum = 0;

    if (odd(m))
        sum = n;

    while (m > 1)
    {
        m = m/2;
        n = 2*n;

        if (odd(m))
            sum += n;
    }

    return sum;
}

int main()
{
    cout << Russian_peasant(4, 5) << " ";
    cout << Russian_peasant(3, 8) << " ";
    cout << Russian_peasant(213, 100) << endl;
    return 0;
}
```

Answer: _____

Problem 3 [5 points] Mysterious Recursion

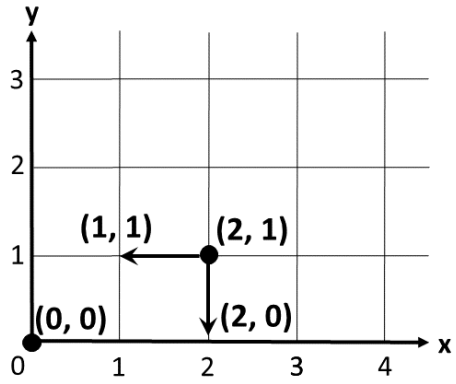
What is the output of the following program?

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

void mysterious(int n)
{
    if (n > 0)
    {
        cout << 0;
        mysterious(n-1);

        cout << 1;
        mysterious(n-2);
    }
}

int main()
{
    mysterious(3);
    cout << endl;
    return 0;
}
```

Answer: _____

Problem 4 [6 points] Recursive Path Counting

Given an integer grid where the x-coordinates and y-coordinates of all the points on the grid are non-negative integers, suppose that you are at the point (x, y) and want to go to the origin $(0, 0)$. From each point (x, y), you are allowed only to move either to the left point $(x - 1, y)$ or the point $(x, y - 1)$ below.



For example, at point $(2, 1)$, you can move to either the point $(2, 0)$ or $(1, 1)$. Then, at point $(2, 0)$, you can move only to the point $(1, 0)$. At point $(1, 1)$, you can move to either $(1, 0)$ or $(0, 1)$. Finally, at points $(1, 0)$ or $(0, 1)$, you can only move to the origin $(0, 0)$. As a result, there are 3 different paths from the point $(2, 1)$ to the origin $(0, 0)$.

Implement a recursive function, `countPaths`, that counts and returns the number of different paths from the point (x, y) to the origin $(0, 0)$. As examples,
`countPaths(0, 0)` will return 0; `countPaths(1, 2)` will return 3;
`countPaths(3, 0)` will return 1; `countPaths(3, 3)` will return 20.

Answer:

```
int countPaths(int x, int y)
{
    // TO DO: ADD YOUR CODE HERE




}
```

Problem 5 [4 points] Memory of Static and Dynamic 2D Arrays

For this question, assume the size of `int` is 4 bytes and the size of a memory address is 8 bytes.

(a) Let `a` be a static 2-dimensional array defined as follows: `int a[3][5];`
If the value of `&a[0][0]` is 1000 (decimal), write down the (decimal) value of the expressions in the following table. If the value of an expression cannot be determined from the given information, or if it is compiler-dependent, write "N/A".

Expression	Value
`&a[0][3]`	
`&a[2][0]`	
`&a[0]`	
`a[0]`	

(b) Let `b` be defined as follows: `int** b;`
and dynamic memory allocation is properly performed so that at the end `b` is set up to point to a 3×5 dynamic 2-dimensional array. If the value of `&b[0][0]` is 2000 (decimal), write down the value of the expressions in the following table. Again, if the value of an expression cannot be determined from the given information, or if it is compiler-dependent, write "N/A".

Expression	Value
`&b[0][3]`	
`&b[2][0]`	
`&b[0]`	
`b[0]`	

Problem 6 [5 points] Rotate Rows in a Dynamic 2D Array

Write a function called `rotate_rows` to rotate the rows of a dynamic 2-dimensional array upwards by one row. For example, if a 3-by-4 dynamic 2D array originally has the following values:

	Column 1	Column 2	Column 3	Column 4
Row 1	1	2	3	4
Row 2	5	6	7	8
Row 3	9	10	11	12

After calling the `rotate_rows` function once, it will have the following values:

	Column 1	Column 2	Column 3	Column 4
Row 1	5	6	7	8
Row 2	9	10	11	12
Row 3	1	2	3	4

Your function must have the following prototype

```
void rotate_rows(int** x, int num_rows);
```

where `x` points to a dynamic 2D int array which, you may assume, has been properly constructed; `num_rows` is the number of rows in the 2D array. Your code **MUST** work with **ANY** number of rows and columns.

Answer:

Problem 7 [8 points] Circular Shift in Linked List

Given a singly linked list storing a sequence of characters, implement the function `ll_shift` which circularly shifts its characters by n times to the left or right. That is,

$$\begin{aligned} &\text{if } n > 0, && \text{circularly shift to the left by n times} \\ &\text{if } n < 0, && \text{circularly shift to the right by n times} \\ &\text{if } n = 0, && \text{no shift} \end{aligned}$$

Note that your solution must:

i. perform the shifts **in-place**, without using any additional temporary storage for another linked list.

ii. produce the outputs of the given driver program as shown in the inline comments of the `main()`.

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

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

/* Assume the following functions have been implemented and you may use them */
ll_cnode* ll_create(char c);         // Create a node with the given character
ll_cnode* ll_create(const char s[]); // Create a singly linked list with the
                                     // given string; return the head pointer
int ll_length(const ll_cnode* head); // Return number of nodes in the linked list
void ll_print(const ll_cnode* head); // Print the linked list

int mod(int x, int y)                // Helper function for modular arithmetic
{                                    // e.g., mod(1,8) = 1, mod(-1,8) = 7
    return ((x < 0) ? ((x%y) + y) : x) % y;
}

/* To implement: circularly shift a linked list in-place */
void ll_shift(ll_cnode* &head, int n);

int main()
{
    ll_cnode* ll_string = ll_create("Hello World!");
    cout << ll_length(ll_string) << endl;  // 12
```

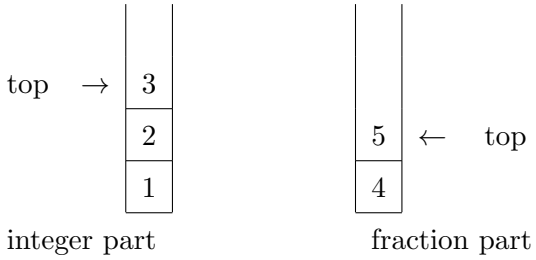
```
    ll_print(ll_string);                    // Hello World!
    ll_shift(ll_string, 0);
    ll_print(ll_string);                    // Hello World!
    ll_shift(ll_string, 2);
    ll_print(ll_string);                    // llo World!He
    ll_shift(ll_string, -2);
    ll_print(ll_string);                    // Hello World!
    ll_shift(ll_string, 12);
    ll_print(ll_string);                    // Hello World!
    ll_shift(ll_string, 13);
    ll_print(ll_string);                    // ello World!H
    ll_shift(ll_string, -1);
    ll_print(ll_string);                    // Hello World!
    ll_shift(ll_string, -14);
    ll_print(ll_string);                    // d!Hello Worl
    return 0;
}
```

Answer:

Problem 8 [23 points] Implementation of PA3 Real Numbers by Stack

```
#include <iostream>     /* File: real.h */
using namespace std;
#include "int-stack.h"

/* Helper functions */
int add_fractions(int_stack& f1, int_stack& f2);
void add_integers(int_stack& i1, int_stack& i2, int carry);

class Real
{
  private:
    int_stack integer;  // Integer part of the Real number
    int_stack fraction; // Fractional part of the Real number

  public:
    Real();              // Default constructor: create 0.0
    Real(const char* s); // Conversion constructor: convert a textual string
                         // representation of real number to this Real number
    Real(const Real& y); // Copy constructor: copy y to this Real number

    /* You DON'T NEED to write the following functions */
    void print();        // print() will modify the Real number/object

    // Add y to the current Real number
    void add(const Real& y)
    {
        Real z(y);
        int carry = add_fractions(fraction, z.fraction);
        add_integers(integer, z.integer, carry);
    }
};
```

This question is modified from your Programming Assignment PA3. The question deals with addition of real numbers to arbitrary degree of precision limited only by their storage in their actual implementation. Unlike your PA3 in which linked list is used to implement the real numbers, this question uses *two* integer stacks, `int_stack` as discussed in our course, to store the integer part and fraction part of a real number *separately* with the most significant bit at the bottom and the least significant bit at the top; the decimal point '.' is not stored. For example, the real number 123.45 is represented by the following 2 stacks:



Above is the header file of the class Real used to implement the real numbers. The header file of `int_stack` is given in Appendix I.

Implement the following 5 functions in a separate file called "real.cpp".

(a) `Real()`
It is the default constructor which will initialize the Real number to represent 0.0.

(b) `Real(const char* s)`
It is often called a conversion constructor as it converts an input textual C string to a Real object. Examples of inputs are: "123.45", "0.2". You may assume that inputs are always valid meaning that the decimal point is always present and there are no extra zeros at the front of the integer part, or at the back of the fraction part. There are always at least one digit before and after the decimal point.

(c) `Real(const Real& y)`
It is often called the copy constructor as it copies the input object `y` to the Real object that is being constructed.

(d) `int add_fractions(int_stack& f1, int_stack& f2)`
It is a helper function used to add the fraction parts of 2 Real numbers namely `f1` and `f2`. The result is put back to the first argument `f1` with the effect of `f1 += f2`. If the sum is over 1.0, the function returns a carry value of 1 otherwise 0.

(e) `void add_integers(int_stack& i1, int_stack& i2, int carry)`
It is a helper function used to add the integer parts of 2 Real numbers namely `i1` and `i2` plus the carry from the addition of their fraction parts. The result is put back to the first argument `i1` with the effect of `i1 += i2 + carry`.

Note that

i. A copy function is added to the implementation of `int_stack` which copies the contents of an input stack to the calling stack without modifying the input stack.

ii. You don't need to implement Real's `print` function. It just prints out the Real number using its 2 internal stacks in the common format as can be seen from the sample outputs below. Also note that since the `print` function will need to `pop` the stacks, the Real number that calls `print` will be modified.

iii. Real's `add` function works similar to += operator and adds the input Real number to the calling Real number.

iv. While the input numbers are always valid, addition of 2 Real numbers may produce extra trailing zeros in the resulting fraction (e.g., 0.99 + 0.01 = 1.00); on the other hand, the sum may have more digits in its integer part than there are in the original 2 numbers (e.g., 9.1 + 1.2 = 10.3).

v. You are NOT allowed to create additional data structure except additional temporary `int_stack`'s in your solution.

vi. You may add as many helper functions as you deem necessary.

Below is a sample driver program and some of its outputs.

```
#include "real.h"
int main()
{
    char s[1024];
    while (true)
    {
        cout << endl << endl << "Enter the first number: "; cin >> s;
        Real a(s);

        cout << "Enter the second number: "; cin >> s;
        Real b(s);

        a.add(b); a.print();
    }
    return 0;
}
```

```
##################### SAMPLE OUTPUTS BEGIN #####################
Enter the first number: 123.45
Enter the second number: 9.9
133.35

Enter the first number: 0.0000987
Enter the second number: 54.1
54.1000987

Enter the first number: 99.999
Enter the second number: 8888.8888
8988.8878

Enter the first number: 0.0
Enter the second number: 98.765
98.765
##################### SAMPLE OUTPUTS END #####################
```

Answer:
```
// Implement the 5 required functions in the separate file "real.cpp"
```

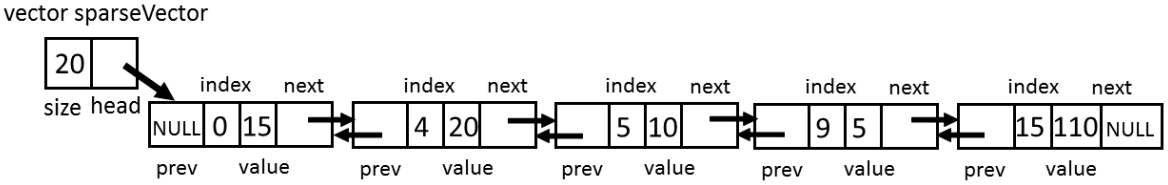
Problem 9 [16 points] Sparse Vector

A vector is a one-dimensional array of elements. When most of the elements of a vector have zero values, it is a sparse vector. Representing a sparse vector using a one-dimensional array is not very efficient. An alternative is to represent the elements of a sparse vector as a linked list of nodes. Each node in the list corresponds to a non-zero element in the vector and contains the index and the value of the element. the nodes in the list are in ascending order of their corresponding indices.

For example, the following is a sparse vector with 20 elements:

$$\begin{bmatrix} 15 & 0 & 0 & 0 & 20 & 10 & 0 & 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 110 & 0 & 0 & 0 & 0 \end{bmatrix}$$

It can be represented as an svector object, sparseVector, by a doubly linked list as follows:



The corresponding structure and function definitions for the node of the doubly linked list and the svector are in the header file "sparse_vector.h" as follows:

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

// A node for storing the non-zero element in the sparse vector
struct nonZeroElement
{
    int index; // The position index of the sparse vector element
    int value; // The value of the sparse vector element
    nonZeroElement* next; // The link to the next node
    nonZeroElement* prev; // The link to the previous node
};

// Sparse vector with a linked list of non-zero elements
struct svector
{
    int size;
    // The number of elements (including the zeros) of a sparse vector,
    // not the number of non-zero elements
    nonZeroElement* head;
       // The head of the linked list with non-zero elements
};

// Insert a non-zero element into the sparse vector's linked list
void insertSparseVectorElement(svector &v, int index, int value);
```

```
// Print the sparse vector by printing the non-zero elements
// in the linked list and the omitted zero elements
void printSparseVector(const svector& v);

/* You DON'T NEED to write the following function */
void deleteSparseVector(svector &v); // Delete the sparse vector
```

The following main function demonstrates the usage of the above structures and functions:

```
#include "sparse_vector.h"

int main()
{
    // Information of the non-zero elements
    int e[][2] = {{5, 10}, {4, 20}, {0, 15}, {9, 5} , {15, 110}};

    // Create and initialize a svector object
    svector sparseVector;
    sparseVector.size = 20;

    sparseVector.head = NULL;
    cout << "Zero Vector: " << endl;
    printSparseVector(sparseVector);

    // Insert the non-zero elements to the svector object
    for (int i = 0; i < (sizeof(e)/sizeof(e[0])); i++)
        insertSparseVectorElement(sparseVector, e[i][0], e[i][1]);

    // Print the svector object
    cout << "Vector: " << endl;
    printSparseVector(sparseVector);
    deleteSparseVector(sparseVector);
    return 0;
}
```

and the output is:

```
Zero Vector:
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Vector:
[15, 0, 0, 0, 20, 10, 0, 0, 0, 5, 0, 0, 0, 0, 0, 110, 0, 0, 0, 0]
```

(a) Implement the function `insertSparseVectorElement`.

```
#include "sparse_vector.h"

/* Insert a vector element to the linked list of the sparse vector
 * in the appropriate position according to the index.
 * The nodes in the list should be in ascending order of their indices.
 * Parameters:
 *    v: the sparse vector with the linked list
 *    index: the index of the element. You may assume the index
 *           is valid and does not exist in the linked list
 *    value: the value of the non-zero element (i.e. value ≠ 0)
 */
void insertSparseVectorElement(svector &v, int index, int value)
{
    // TO DO: ADD YOUR CODE HERE









}
```

(b) Implement the function `printSparseVector`.

```
#include "sparse_vector.h"

/* Print BOTH the non-zero elements in the linked list of the svector v
 * and the omitted zero elements in the format of: [x_1, x_2, ..., x_n]
 * where n is v.size and x_i is i-th element of the vector
 * Note: If v has an empty linked list, print a vector with n zeros.
 * Paramters:
 *    v: the sparse vector with the linked list of non-zero elements
 */
void printSparseVector(const svector& v)
{
    // TO DO: ADD YOUR CODE HERE









}
```

Problem 10 [17 points] Family Tree with C++ Class

(a) **`Person` Class**

Assume that you are a class developer and you want to develop a C++ class `Person`, which is defined in the header file "person.h" below.

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

class Person
{
  private:
    char* name;
    int age;

    // Below are just links to other Person objects which should have been
    // created before being linked to the current Person
    Person* father;
    Person* mother;
    Person* kid; // Assume only one single child per person

  public:
    // Default constructor
    Person() { name = NULL; age = 0; father = NULL; mother = NULL; kid = NULL;}

    /* You need to implement
     *     a general constructor      .....  Person(.....)
     *     a destructor               ..... ~Person(.....)
     *     a setter mutator function  .....  set(.....)
     * You need to decide their formal parameters and return types.
     */


    void print_age() const { cout << age; }
    void print_name() const { cout << name; }
    Person* get_father() { return father; }
    Person* get_mother() { return mother; }
    Person* get_kid() { return kid; }
};
```

Implement the following missing functions of the `Person` class; they are supposed to be written in a separate source file called "person.cpp".

i. a (general) constructor.

It should support all the following usages:

- `Person p("Alice");`
  will create a `Person` object with name = "Alice", age = 0, father = NULL, mother = NULL and kid = NULL.
- `Person y("Bob", 30, NULL, NULL, &p);`
  will create a `Person` object with name = "Bob", age = 30, father = NULL, mother = NULL, and his kid is Alice.

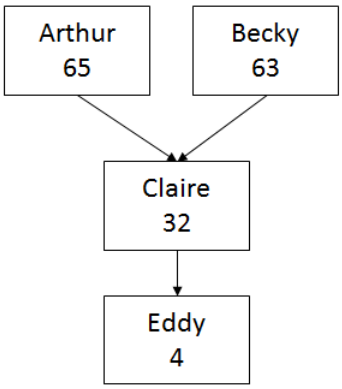
ii. the destructor which should release any dynamic storage of its data member(s).

iii. a mutator member function called `set`. It should support all the following usages:

```
Person x, y, z;
x.set("David", 42, NULL, NULL, &z);  // Person z is David's kid
y.set("Carmen", 40, NULL, NULL, &z); // Person z is Carmen's kid
z.set("Elsa", 14, &x, &y, NULL );    // David is Elsa's father, and
                                     // Carmen is her mother
```

Answer: `/* File: person.cpp */`

(b) **Build a Family Tree with the use of Person Class**



Assume that your role is now an application programmer and you want to use the `Person` class defined in part (a) to build a family tree. Complete the following program to create the family tree in the figure above by setting up the personal information and relative links for the 4 Person objects in the family array. The 4 Person objects should be initialized in the following order (from array index 0 to 3): Arthur (male), Becky (female), Claire (female) and Eddy (male). Assign NULL to any missing/unknown relationships, e.g., father of Arthur is NULL. Your code is supposed to be written in a file called "family-tree.cpp".

Answer:

```
#include "person.h"     /* File: family-tree.cpp */

int main()
{
    Person* family  = new Person[4];
    /* TO DO: ADD YOUR CODE HERE
     *    to initialize the 4 Person objects in the family array
     *    according to the family tree shown above
     */









    // Assume codes here which will release the dynamic Person array
    // You DON'T need to write these codes
    return 0;
}
```

(c) **Print Immediate Parents and Grandparents**

Again assume that you are still an application programmer. Given a family tree that is set up in the way illustrated in part (b) (but it can be a different family tree), implement a function `print_family(Person* p)` which prints out the names of the parents and grandparents of a person given its `Person` pointer. The code is also in the file called "family-tree.cpp".

The output format for the names should be like this:

```
Name: <name>
Father: <name>
Mother: <name>
Grand Fathers: <name of father's side>, <name of mother's side>
Grand Mothers: <name of father's side>, <name of mother's side>
```

Print "unknown" for any missing information, For example, the function outputs for Claire and Eddy should be:

```
Name: Claire
Father: Arthur
Mother: Becky
Grand Fathers: unknown, unknown
Grand Mothers: unknown, unknown

Name: Eddy
Father: unknown
Mother: Claire
Grand Fathers: unknown, Arthur
Grand Mothers: unknown, Becky
```

Answer: `/* File: family-tree.cpp */`

Appendix I

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

class int_stack
{
  private:
    /*
     * The actual data members and their storage should be immaterial to your
     * solution. You should not assume any particular int_stack implementation,
     * but just use its following interface (member functions) to implement
     * your solution.
     */

  public:
    // CONSTRUCTOR member functions
    int_stack();            // Default constructor gives an empty stack

    // ACCESSOR member functions: const => won't modify data members
    bool empty() const;     // Check if the stack is empty
    bool full() const;      // Check if the stack is full
    int size() const;       // Give the number of data currently stored
    int top() const;        // Retrieve the value of the top item

    // MUTATOR member functions
    void push(int);         // Add a new item to the top of the stack
    void pop();             // Remove the top item from the stack

    // Added function copy()
    // It replaces the calling int_stack with a copy of the input int_stack y
    void copy(const int_stack& y);
};
```

Appendix II

```
// strlen() calculates the length of the string pointed to by s, not including
// the terminating '\\0' 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);
```

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

/* Rough work — DO NOT detach this page */

/* Rough work — You may detach this page */

/* Rough work — You may detach this page */