**Carleton University**
**Department of Systems and Computer Engineering**
**SYSC 2006 - Foundations of Imperative Programming - Fall 2013**

**Section A - Midterm Exam**

**Instructions**

1. The TAs will not answer queries about this question paper. Exam questions will not be explained, and no hints will be given. If you think something is unclear or ambiguous, make a reasonable assumption (one that does not contradict the question), write it at the start of your solution, and answer the question.
2. You do not need to write header comments or inline comments for your functions. You do not need to copy any code from the question paper to your answer booklet.

**PART A: Programming**

**Question 1 [10 marks]**

Somewhere in the basement of the Herzberg Laboratories, Mathematics professor Dr. Polly Nomial is attempting to solve a particularly obscure mathematical theorem. She has derived the following formula:

$$silly(x, y) = y! \times e^{x} + (2y + 1)! \times e^{3x} + (3y + 2)! \times e^{5x} + \ldots$$

She has asked you to write a C function named `silly` that has three parameters, `x`, `y` and `n`. Parameter `x` is a real number, and parameters `y` and `n` are integers. This function calculates the first `n` terms of the series and returns the result.

Assume that you have been provided with a function that calculates factorials. The function prototype is:

```
int factorial(int n);
```

For values of `n` that are greater than or equal to 0, this function calculates and returns `n!` Your `silly` function must call `factorial`. Do not write `factorial`.

The constant *e* is Euler's number, which is approximately 2.71828. Recall that the standard C library includes a collection of math functions, which are declared in `math.h`. One of these functions is:

```
double exp(double arg);
```

This function returns *e* raised to the power `arg`.

**Question 2 [6 marks]**

Meanwhile, over in Dunton Tower, English professor Dr. P. Laywright is continuing his research analyzing Shakespeare's sonnets (a sonnet is a type of poem that typically contains 14 lines). He needs a function that can count the number of words in each line.

Write a C function named `count_words` which is passed a character string. and returns the number of words in the string. A word is defined to be a sequence of letters, digits and punctuation symbols, and is separated from other words by one **or more** spaces. For example, "Carleton" is considered to be a word, as is "*%$55!@". The function prototype is:

```
int count_words(char s[]);
```

Here is a summary of two functions from `string.h` in C's standard library, which you may or may not need:

`int strlen(const char str[]);`
Returns the length of string *str*. The length does not include the string's terminating null character.

`int strcmp(const char str1[], const char str2[]);`
Compares strings *str1* and *str2*. A return value of zero value indicates that both strings are equal. A value greater than zero indicates that the first character that does not match has a greater value in *str1* than in *str2*, A value less than zero indicates the opposite.

**Question 3 [6 marks]**

We'll say that a value is "everywhere" in an array of integers if, for every pair of adjacent elements in the array, at least one of the pair is that value. Write a function named `is_everywhere` that returns `true` if the given value is everywhere in the array; otherwise it should return `false`. The function prototype is:

```
_Bool is_everywhere(int a[], int n, int val);
```

Parameter `n` is the number of integers in the array. Your function can assume that `n` will always be at least 2.

Examples:

```
int a1[] = {1, 2, 1, 3};
is_everywhere(a1, 4, 1) returns true   // 1 is everywhere
int a2[] = {1, 2, 1, 3};
is_everywhere(a2, 4, 2) returns false  // 2 isn't everywhere
int a3[] = {1, 2, 1, 3, 4};
is_everywhere(a3, 5, 1) returns false  // 1 isn't everywhere
```

**PART B: Tracing Code/Memory Diagrams**

**Question 4 [13 marks]**

Fibonacci numbers are defined by the following formulas:

$F_1 = 1$
$F_2 = 1$
$F_n = F_{n-1} + F_{n-2}, n > 2$

Also, it is conventional to define $F_0$ as 0. The Fibonacci sequence for $n$ = 0, 1, 2, 3, ... is therefore 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Here is a definition of a C function that is passed $n$ and returns $F_n$, for $n$ >= 0. The function may or may not have bugs (that's something you have to determine).

```
int fibonacci(int n)
{
    if (n == 0)    // fib(0)
        return 0;

    if (n == 1)    // fib(1)
        return 1;

    int temp1 = 0;
    int temp2 = 1;
    int nextfib;

    for (n = n - 2; n >= 0; n = n - 1) {
        nextfib = temp1 + temp2;
        temp1 = temp2;   /* Point A. */
        temp2 = nextfib;
    }

    return nextfib;   /* Point B. */
}
```

Here is the definition of a `main` function that calls `fibonacci`:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int result;
    result = fibonacci(5);
    printf("fib(%d) =  %d\n", 5, result);   /* Point C. */
    exit(0);
}
```

Using the notation presented in lectures, draw three separate memory diagrams, one each for parts (a), (b) and (c). *Do not combine your solutions into a single diagram.* Do not use arrows to depict the flow of data into and out of variables. Remember, arrows are only used to depict pointers.

(a) Draw a memory diagram that depicts the program's stack frame(s) immediately **after** the statement at Point A is executed for **the first time**; that is, immediately after `temp1 = temp2;` is executed during the first iteration of the `for` loop.

(b) Draw a memory diagram that depicts the program's stack frame(s) immediately **before** the statement at Point B is executed; that is, just before the `return` statement is executed.

(c) Draw a memory diagram that depicts the program's stack frame(s) immediately **before** the statement at Point C is executed; that is, just before the `printf` call is executed.