Indian Institute of Technology, Delhi

FALL, 2016

COL 100: INTRODUCTION TO PROGRAMMING

Major Exam

Two Hours

NOTE:

- All answers need to be brief and to the point.
- Please make any assumptions that you deem to be reasonable.
- Follow the *spirit of the question*. Do not immerse yourself in irrelevant details.
- Every answer needs to be written neatly and cleanly in the space provided for it.
- If required, please finalize the code in the space provided for rough work, and then write it cleanly in the space provided for writing your C statements.
- Use proper handwriting, and do not write anything on the margins.
- Note that in some questions, we might not have part marking.
- You are not allowed to carry any electronic gadgets including calculators and mobile phones.
- The closer your answer is to the model answer in terms of the lines of code, the more marks you get.
- The final answer needs to be **written with a pen**.
- There are two additional pages at the end for rough work.
- Even if your final answer is correct, you might not get full marks (or any marks) if the explanation is wrong.
- This question paper **NEEDS TO BE SUBMITTED**. Do not take it with you.

Total Marks: 40
Total Number of Pages : 10

Name:	Group No:	Entry No:

Marks:

1	2	3	4	5	6	7	Total

1. For values of x and y both between $1 \ldots 5$, fill the table below with true/false, based on the value of the following conditional expression: (5 marks)

$(x + y == 6) \,||\, (x - y == 0)$

x	y	value	x	y	value	x	y	value	x	y	value	x	y	value
1	1	T	1	2	F	1	3	F	1	4	F	1	5	T
2	1	F	2	2	T	2	3	F	2	4	T	2	5	F
3	1	F	3	2	F	3	3	T	3	4	F	3	5	F
4	1	F	4	2	T	4	3	F	4	4	T	4	5	F
5	1	T	5	2	F	5	3	F	5	4	F	5	5	T

Marking scheme: One mark per line

2. Given a binary number (a_n, a_{n-1}, $\ldots a_0$) where a_0 is the least significant bit, show that division of the number by 2^k is equivalent to right shifting the binary number by k positions and dropping off the k least significant bits. And, multiplication of the number by 2^k is equivalent to left shifting the number by k positions and filling the new positions with 0s. (6 marks)

The number can be written as $a_n.2^n + a_{n-1}.2^{n-1} + .. + a_1.2^1 + a_0$.

Division by 2^k will give $a_n.2^{n-k} + a_{n-1}.2^{n-1-k} + .. + a_k.2^{k-k} + a_{k-1}.2^{k-1-k} + .. + a_0.2^{0-k}$

The digits $a_{k-1}..a_0$ can be seen to have an exponent portion less than 1, which means that these will fall after the decimal point, and in the case of integers, will get discarded. This only leaves the digits $a_n..a_k$, the least significant k digits having been discarded. This is equivalent to right shifting the binary number by k positions and dropping off the k least significant digits, as required to prove.

Similarly, multiplication by 2^k can be written as
$a_n.2^{n+k} + a_{n-1}.2^{n-1+k} + .. + a_0.2^k + 0.2^{k-1} + 0.2^{k-2} + .. + 0.2^0$

Note that the earlier least significant digit a_0 now has an exponent of 2^k, and we have filled in 0s for the rest of the digits with exponent less than k. This means that the earlier digits now have their exponents incremented by k, which is equivalent to left shifting the digits by k, and the k least significant positions have their digits set to 0, as required to prove.

Marking scheme: 1 mark for writing the number in its expanded digit form, 1 mark for writing the division by 2^k in expanded digit form, 1 mark for writing the multiplication by 2^k in expanded digit form, and 1 mark each for explanation of the proof for division and multiplication respectively.

3. What is the output of the following code snippet? (5 marks)

```
char* line = "lifeisgood";
int *pi;

printf ("Output 1: %c\n",line[3]);
printf ("Output 2: %c\n", * (line + 4));

pi = (int *) (line + 4);
printf ("Output 3: %c\n", * ((char *) (pi + 1)));
```

Fill the following table of the output:

Output 1:	e
Output 2:	i
Output 3:	o

Explain your answers (no marks without proper explanation). Assume that the size of an integer is 4 bytes.

Output 1: line[3] gives the element at position 3 in the array.

Output 2: *(line + 4) is the same as line[4] and gives the element at position 4 in the array.

Output 3: Here the result of (line + 4) is typecast into an int * and assigned to pi. Now, any arithmetic on pi will be done as on int *, where an increment of +1 will actually increment the pointer by 4 bytes. Therefore (pi + 1) will point to the address of line[8]. This is then typecast into a char * for printing a character, which at this position is o.

Marking scheme: 1 mark each for the correct output. 1 mark for an explanation of output 1 and 2, where the student should demonstrate that they know the equivalency between addressing array elements within square brackets or through pointers. 1 mark for an explanation of output 3, where the student should demonstrate that they understand pointer typecasting and the arithmetic on int * is different from that on char *

4. We need to write a function to join two strings, str1 and str2. For example, if str1 is "good", and str2 is "life", then the joined string is "goodlife". Complete the following function. At the end, *strjoined* will contain the joined string. Assume enough space is available in the character arrays. Do not use any inbuilt string functions such as strcpy, strcat, or strlen. (6 marks)

```
void join (char *str1, char *str2, char *strjoined) {
    int i, pos;
    i = 0; pos = 0;
    while(str1[i]) {
        strjoined[pos++] = str1[i++];
    }
    i = 0;
    while(str2[i]) {
        strjoined[pos++] = str2[i++];
    }
    strjoined[pos] = '\0';
}
```

Marking scheme: 1 mark for knowing that strings end with '\0', 1 mark for writing a loop to copy elements character by character, 2 marks for writing the loops correctly (1 for each loop), 1 mark for terminating strjoined by a '\0', 1 mark for overall correctness.

5. Write a recursive function to compute $f(n)$. $f(n)$ is defined as follows: (6 marks)

- $f(n) = f(n/2)$, if n is even
- $f(n) = f((n{-}1)/2)$, if n is odd
- $f(1) = 1$

Note that a *recursive function* calls itself. If **recursion** is not used in the answer, then no marks will be awarded.

```
int f(int n) {
    if(n == 1) {
        return 1;
    }
    else if(n % 2) {
        return f((n-1)/2);
    }
    else {
        return f(n/2);
    }
}
```

Marking scheme: 1 mark for checking for the termination condition, 1 mark for correctly distinguishing between odd and even cases, 1 mark for overall correctness.

How many recursions happen to compute $f(n)$? Write a generic expression in terms of n.

Let $t(n)$ be the number of recursions to compute $f(n)$. We can write:

$t(n) = 1 + t(\lfloor n/2 \rfloor)$

$t(1) = 0$

This recurrence relation can be solved by applying the equation repeatedly;

$t(n) = 1 + t(\lfloor n/2 \rfloor)$

$= 1 + 1 + t(\lfloor \lfloor n/2 \rfloor /2 \rfloor)$

$= 1 + 1 + .. + t(1)$

This will take k steps to get to the terminating condition, where $k = \lfloor log_2 n \rfloor$

Marking scheme: 1 mark for writing the recurrence relation, 1 mark for the terminating condition, 1 mark for solving.

6. Given an array (*numbers*) of N positive (> 0) elements, where $N > 10$, we want to find the top five largest elements. To get you started, here is the *main* function. You need to write the code for the *insert* function that inserts an element into an array, *topv*, that is supposed to contain the top five largest elements. (6 marks)

```
int numbers[N];

int main(){
    int i; int topv[5];

    /* initialize all the elements of topv to -1 */
    for (i=0; i < 5; i++) topv[i] = -1;

    /* scan through the numbers array, just once */
    for (i=0; i < N; i++) insert (topv, numbers[i]);

    /* print the elements */
    for (i=0; i < 5; i++) printf ("%d\n",topv[i]);
}

void insert (int topv[5], int num) {
    int i;
    for(i = 0; i < 5; i++) {
        if(topv[i] == -1 || topv[i] < num)
            topv[i] = num;
    }
}
```

Marking scheme: 1 mark for having understood the question, that topv is to maintain the top 5 elements, and the insert call is meant to update the array with the new top 5 elements. 1 mark for looping through the elements to check if num needs to be inserted. 3 marks for the correct insertion condition, that the array has some elements with -1 to be able to accommodate new elements, or there are elements less than num which can be overwritten. Some students may maintain the array in sorted order, in which case they would need to just check for the smallest element in the array, but the insertion step will be more complex. Count marks for such cases in this same component of 3 marks. 1 mark for overall correctness.

7. You are given an implementation of an integer stack library with the following functions:

```
mystack *create_stack();            // returns an instance of a stack
void push(mystack *stack, int num); // pushes num into the stack
int pop(mystack *stack);            // pops the top element of the stack
int peek(mystack *stack);           // returns the element at the top of the stack
int is_empty(mystack *stack);       // return 1 if the stack is empty, 0 otherwise
```

The behavior of *pop* and *peek* (*peek* is also referred to as *top*) methods, when the stack is empty, is undefined.

It is possible to implement a queue using two stacks as follows. A stack as you know pushes new elements at the top, but a queue enqueues new elements at the bottom. Therefore you cannot get a stack to work as a queue directly, but you can instead first empty the stack by popping all of its elements one by one and push them into the second stack, then push the new element into the first stack, and then pop the elements from the second stack and push them into the first stack. This way you would have effectively used the second stack as a temporary storage, and you will be able to get a queue's behavior from the arrangement.

We want you to use two stacks to implement a queue, ie. the *enqueue*, *dequeue*, and *is_empty* functions. The following is given to get you started. (6 marks)

```
typedef struct {
        mystack *firststack;
        mystack *secondstack;
} myqueue;


void enque(myqueue *queue, int num) {
    while(!is_empty(queue->firststack)) {
        push(queue->secondstack, pop(queue->firststack));
    }
    push(queue->firststack, num);
    while(!is_empty(queue->secondstack)) {
        push(queue->firststack, pop(queue->secondstack));
    }
}
```

Marking scheme: 1 mark for having understood the question, that the student has to implement a queue data structure using stack data structures, and not a raw implementation of a queue. 1 mark for emptying the first stack into the second, 1 mark for pushing the new element into the first stack, 1 mark for emptying the second stack into the first. The correctness condition should be that at the end of the operation, the new element should be at the bottom of the first stack, the rest of the elements should accordingly be in the same order in the first stack, and the second stack should be empty.

```
int dequeue(myqueue *queue) {
    return pop(queue->firststack);
}
```

Marking scheme: 1 mark for the simple correct answer.

```
int is_empty(myqueue *queue) {
    return is_empty(queue->firststack);
}
```

Marking scheme: 1 mark for the simple correct answer.