CSE220
Fall 2013 – Practice Questions for Midterm #2

1. Consider the following C program.

```
#include <stdio.h>
int mystery (char *c) {
        int n = 0;

        while ( *c >= '0' && *c <= '9') {
                n = n+1;
                *c = *c - '0' + 'A';
                c++;}
        return n;
}

int main(void){
        char str[200];
        printf ("Enter a string, end with return\n");
        scanf ("%s", str);
        printf ("%d\n", mystery(str));       <-- 2nd printf
        printf ("%s\n", str);                <-- 3rd printf
        return 0;
}
```

(a) What does this program print for input string 123-123 ?
(i) Second printf _____ (ii) Third printf _____
(b) In one or two simple sentences, state what does function mystery do?
(c) If we change declaration of parameter to function mystery to int mystery (char c[ ]), and keep everything else as it is, what would be its effect on the entire program?
Syntax Error Execution error Work as before
(d) Explain your answer in (c).

Answer:
(a) (i) 3 (ii) BCD-123
(b) The function mystery, counts digits in the leading (initial) contiguous numeric part of the string. It also changes these digits to upper case alpha characters, where 0-9 are changed to A-J respectively.
(c) It will work as before.
(d) char c[] is an array declaration. Here variable c is treated as a pointer to the first location of this array. Since this is an array of characters, c would point to the first character. So this is same as char *c declaration, and program will work as before.

2. Consider the fetch-decode-execute cycle for the John Von Neumann machine that we studied in class. Note that it does not have a cache memory.
(a) Give an example of an instruction that requires accessing memory during execution of an instruction.
(b) Do we always have to access memory in fetch part of the cycle? Why or why not?

Answer:

(a) LOAD or STORE instruction or something similar. There is no READ or MOVE instruction for John Von Neumann machine.
(b) Yes. We always access memory in fetch part of the cycle. Fetch means reading the next instruction, which is stored in memory. For John von Neumann machine, one word contains two instructions. So a fetch operation gets both left and right instructions. There in no fetch for right instruction. So if someone claims that there is no memory access for right instruction, then it is not valid argument, as there is no fetch either.

3. Consider an array A of integers in memory of MIPS. The array is indexed by 0, 1, 2, ... etc. Also assume that address of A is in register $s1. Suppose we want to copy and store the second element of this array (indexed by 2) to variable i. Address of variable i is in register $s2. Write the exact sequence of instructions MIPS that will accomplish this. You may use a temporary register. (In other words, write MIPS code for statement i = A[2];)

   Answer:
   Second element of an integer array has offset = 8 bytes (4 * Array Index).

   lw $t0, 8($s1) # Load t0, address of A in $s1, offset is 8.
   sw $t0, ($s2) # $s2 has address of i. Store $t0 to i.

   If you used [sw $t0, i] as the second instruction, then 1/2 point would be deducted.
   Since address of i is known, why use sw $t0, i. MIPS assembler allows such instructions, but this would be slow. Also, typically i is a local variable on the stack. Its address is always known. So use the address directly. That is what a compiler would do.

4. Translate the following code into MIPS assembly:
   `A = B – C – 30 + D – E + F`
   Assume that variables A, B, C, D, E, and F are in registers $s0, $s1, $s2, $s3, $s4, $s5. Also, assume that you cannot overwrite any variables (except A) since they are used later in the program.

   Write the code such that it uses the minimal number of registers. How many additional registers (other than $s0-$s5) does your code require?

```
sub $1, $2, $3    # A = B – C
addi $1, $1, -30  # A = B – C - 30
add $1, $4, $1    # A = B – C - 30 + D
sub $1, $1, $5    # A = B – C - 30 + D - E
add $1, $1, $6    # A = B – C – 30 + D – E + F
No additional registers are needed. You can store the
intermediate results in $1.
```

5. MIPS registers
   (i) The MIPS processor has 32 “general purpose” registers. How many bits are needed to refer to them?
   $\log_2(32)$ combinations, 5 bits

   (ii) Where do registers reside, inside or outside of the processor? Why?
   inside. Closer to the CPU means faster operation.

(iii) Usually, more registers imply faster overall operation (throughput). If that is the case, why do modern machines not have Gigabytes of registers?
registers are more expensive (cost, silicon size). More registers means more bus lines to address them. Additionally, extra registers implies higher amount of logic to manage reading/writing.

(iv) Floating point registers of MIPS are used for storing single precision numbers and/or double precision numbers. How many floating point registers does MIPS have? How do we reference them in MIPS (naming convention)?
32 floating point registers, $f0-$f31

If we store 7 single precision numbers, how many double precision numbers can be stored in the remaining floating point registers? 12

6. Write the function declaration for fopen. Explain each of the parameters and return type. Give an example of its use.
FILE *fopen(char *path, char *mode);

The fopen function is used to open the file whose name is the string pointed to by path. The parameter mode points to a string that describes how the file will be used.
For our purposes, there will only be two choices for mode: "w+b" and "r+b". The "w+b" mode causes the specified file to opened in binary mode for reading and writing (it is created if it does not exist and it is truncated to zero length if it does exist).
The "r+b" mode opens an existing file in binary mode for reading and writing (in this case, the file is not truncated).

The return value from fopen has type FILE *; that is, it is a pointer to an object of type FILE. In actuality, FILE is defined (in stdio.h) to be a suitable structure type, but it is not necessary (or desirable) to look at the actual definition, so you should just treat FILE as if it is a predefined type like int or double. If fopen fails to open the specified file, the return value is NULL.

7. Consider the following lines of C code:

```
#define PHRASE_LENGTH 50
struct AcronymNode{
      struct AcronymNode* next;
      char acronym[5];
      double num_phrases;
      struct The_Phrase* phrase_list;
} Dictionary;

struct The_Phrase {
      char phrase[PHRASE_LENGTH];
      int frequency;
      struct The_Phrase* next;
};
```

(a) How many bytes does `Dictionary` take in memory? 32 bytes = 4 + 5 + 3 (alignment to word) + 4 (to align for the double) + 8 + 4 + 4 (to align next structure to a multiple of 8 for the double)

(b) What is the size of an instance of `The_Phrase` structure? 60 bytes

(c) Can either structure be optimized to reduce its space in memory?

```
struct AcronymNode{
      double num_phrases;
      struct AcronymNode* next;
      struct The_Phrase* phrase_list;
      char acronym[5];
} Dictionary;
```

AcronymNode now takes 24 bytes = 8 + 4 + 4 + 5 + 3 (to align to next word boundary)

```
struct The_Phrase {
      int frequency;
      struct The_Phrase* next;
      char phrase[PHRASE_LENGTH];
};
```

The_Phrase still takes 60 bytes, so rearranging it makes no difference in it's size.

(d) Given `Dictionary`, assign the frequency of the head of the `phrase_list` to integer i.

```
int i = Dictionary.phrase_list->frequency;
```

(e) Write the following function `The_Phrase* get(acronymNode* A)` return a pointer to the linked list of all phrases in the `phrase_list` for the `acronymNode` for A.

```
The_Phrase* get(char* A)   {
   char * cur = Dictionary.acronym;
   while (strcmp(cur, A)==0)
   {
         if(cur == NULL)
               return NULL;
         cur = cur->next;
   }

   //match was found
   return cur->phrase_list;
}
```

(f) Write the following function `int add(struct AcronymNode** Head, char* newAcronym, char* newPhrase)`

Insert `newAcronym` in the linked list of Acronyms at the tail of the list if the acronym is not in the list already. If the `newAcronym` already exists, check for `newPhrase`. Insert the newPhrase if it doesn't exist, or increment the frequency if it does exists. Remember to create/allocate the linked list elements if they do not already exist. Return 1 if the phrase was inserted into the table, 0 if frequency was incremented.

```
//Iterative approach, inserts at the end of the list if not found
int add(struct AcronymNode** Head, char* newAcronym, char* newPhrase) {
	Struct AcronymNode *curr = *Head;
	if(*curr == NULL) {
		// Attempt to create a new head
		**Head = malloc(sizeof(struct AcronymNode));
		if(Head == NULL) {
			perror("Unable to allocate new Head node.");
			return -1;
		}
		// Create the phrase
		struct The_Phrase *phrase = malloc(sizeof(struct The_Phrase));
		if(phrase == NULL) {
			perror("Unable to allocate phrase node.");
			return -1;
		}
		//
		strcpy(phrase->phrase, newPhrase);
		phrase->next = NULL;
		phrase->frequency = 1;
		//
		curr->next = NULL;
		strcpy(curr->acronym, newAcronym);
		curr->num_phrases = 1.0;
		curr->phrase_list = phrase;
	} else {
		while(curr != NULL) {
			if(strcmp(cur->acronym, newAcronym) == 0) {
				struct The_Phrase *phrase = curr->phrase_list;
				while(phrase != NULL) {
					if(strcmp(phrase->phrase, newPhrase) == 0) {
						phrase->frequency += 1;
						return 0;
					} else {
						if(phrase->next == NULL) {
							struct The_Phrase *p = malloc(sizeof(struct The_Phrase));
							cur->num_phrases++;
							if(p == NULL) {
								perror("Unable to allocate phrase node.");
								return -1;
							}
							strcpy(p->phrase, newPhrase);
							p->next = NULL;
							p->frequency = 1;
							return 0;
						} else {
							phrase = phrase->next;
						}
					}
				}
			} else {
				if(curr->next == NULL) {
					struct AcronymNode *newNode = malloc(sizeof(struct AcronymNode));
					if(newNode == NULL) {
						perror("Unable to allocate new AcronymNode.");
						return -1;
					}
					struct The_Phrase *phrase = malloc(sizeof(struct The_Phrase);
```

```
                                        if(phrase == NULL) {
                                                perror("Unable to allocate phrase node.");
                                                return -1;
                                        }
                                        //
                                        strcpy(phrase->phrase, newPhrase);
                                        phrase->next = NULL;
                                        phrase->frequency = 1;
                                        //
                                        curr->next = NULL;
                                        strcpy(newNode->acronym, newAcronym);
                                        newNode->num_phrases = 1.0;
                                        newNode->phrase_list = phrase;
                                        return 0;
                                } else {
                                        curr = curr->next;
                                }
                        }
                }
        }
        return 0;
}

#########################################################################################################################################################

//Recursive approach, inserted at the end of the list if not found
int add(struct AcronymNode** Head, char* newAcronym, char* newPhrase) {
        if(*Head == NULL) {
                // Attempt to create a new head
                *Head = malloc(sizeof(struct AcronymNode));
                if(*Head == NULL) {
                        return -1;
                }
                // Create the phrase
                struct The_Phrase *phrase = malloc(sizeof(struct The_Phrase));
                if(phrase == NULL) {
                        return -1;
                }
                //
                strcpy(phrase->phrase, newPhrase);
                phrase->next = NULL;
                phrase->frequency = 1;
                //
                *Head->next = NULL;
                strcpy(*Head->acronym, newAcronym);
                *Head->num_phrases = 1.0;
                *Head->phrase_list = phrase;
                return 0;
        }
        return addr(*Head, NULL, newAcronym, newPhrase);
}

int addr(struct AcronymNode* node, struct AcronymNode* prev, char* newAcronym, char* newPhrase) {
        if(node == NULL) {
                struct AcronymNode *newNode = malloc(sizeof(struct AcronymNode));
                if(newNode == NULL) return -1;
                struct The_Phrase *phrase = malloc(sizeof(struct The_Phrase));
                if(phrase == NULL) return -1;
                strcpy(phrase->phrase, newPhrase);
```

```
                phrase->next = NULL;
                phrase->frequency = 1;
                //
                newNode->next = NULL;
                strcpy(newNode->acronym, newAcronym);
                newNode->num_phrases = 1.0;
                newNode->phrase_list = phrase;
                prev->next = newNode;
        } else {
                if(strcmp(node->acronym, newAcronym)==0) {
                        struct The_Phrase *currp = node->phrase_list;
                        while(currp != NULL) {
                                if(strcmp(currp->phrase, newPhrase)==0) {
                                        currp->frequency++;
                                        return 1;
                                } else {
                                        if(currp->next == NULL) {
                                                struct The_Phrase *phrase = malloc(sizeof(struct The_Phrase));
                                                if(phrase == NULL) return -1;
                                                strcpy(phrase->phrase, newPhrase);
                                                phrase->next = NULL;
                                                phrase->frequency = 1;
                                                currp->next = phrase;
                                                node->num_phrases++;
                                                return 1;
                                        } else {
                                                currp = currp->next;
                                        }
                                }
                        }
                } else {
                        return addr(node->next, node, newAcronym, newPhrase);
                }
        }
        return 1;
}
```

(g) Declare one instance of the variables of the structure `AcronymNode` (the member variables) as variables in the .data section of a MIPS program.

```
.data
next:         .word
acronym:      .space 5   # or .byte 0,0,0,0,0
num_phrases:  .double
phrase_list:  .word
```

8. Write a function in C to determine if a string is a palindrome. A string is a palindrome if its reverse is the same as the original string (EX: '12ABCB21' and '229922' are palindromes). Assume the function palin has 2 input parameters, a pointer to the first character of the string and a pointer to the last character of the string. The function returns 1 if it is a palindrome, 0 otherwise.

```
int palin (char *first, char *last){
     while(first<last){
          if(*first!=*last)
               return 0;
          first++;
          last--;
     }
     return 1;
}
```

9. Translate the following code into MIPS assembly: `A[i] = B[i-1] + B[i] + B[i+1]` Assume that A and B are byte arrays. Both arrays are stored in memory. Register $s1 contains the initial address for array A; register $s2 contains the initial address for array B; and register $s3 contains the value of i. (Hint: use lb & sb. Also, this is not a loop, only a statement).

```
add $t7, $s2, $s3 # Calculate address of B[i]
lb $t0, -1($t7) # Load byte of B[i-1]
lb $t1, 0 ($t7) # Load byte of B[i]
lb $t2, 1 ($t7) # Load byte of B[i+1]
add $t0, $t0, $t1 # add B[i-1] + B[i]
add $t0, $t0, $t2 # add B[i+1] + ( B[i-1] + B[i])
add $t6, $s1, $s3 # Calculate address of A[i]
sb $t0, ($t6) # A[i] = B[i+1] + ( B[i-1] + B[i])
```

10. Add a for loop around the statement in Question 9. Iterate on variable i for values 1 to 8.

In C:

```
int i;
for(i= 1; i<= 8; i++)
{
     A[i] = B[i-1] + B[i] + B[i+1];
}
```

In MIPS:

```
      li $s3, 1 #$s3 is i counter
      li $t9, 8 # $t9 is 8
FOR:  bgt $s3, $t9, ENDFOR #branch on greater than 8
      add $t7, $s2, $s3 # Calculate address of B[i]
      lb $t0, -1($t7) # Load byte of B[i-1]
      lb $t1, 0 ($t7) # Load byte of B[i]
      lb $t2, 1 ($t7) # Load byte of B[i+1]
      add $t0, $t0, $t1 # add B[i-1] + B[i]
      add $t0, $t0, $t2 # add B[i+1] + ( B[i-1] + B[i]) add $t6,
$s1, $s3 # Calculate address of A[i]
      sb $t0, ($t6) # A[i] = B[i+1] + ( B[i-1] + B[i])
      addi $s3, $s3, 1 # i++
      j FOR # or b FOR
ENDFOR: …….
```

11. Assume the arrays in Question 9 are integer (word) arrays, modify the code.

```
add $t9, $s3, $s3     # 2*i
add $t9, $t9, $t9     # 4*i
add $t7, $s2, $t9     # Calculate address of B[i]
lw $t0, -4($t7)  # Load word of B[i-1]
lw $t1, 0 ($t7)  # Load word of B[i]
lw $t2, 4($t7)   # Load word of B[i+1]
add $t0, $t0, $t1 # add B[i-1] + B[i]
add $t0, $t0, $t2 # add B[i+1] + ( B[i-1] + B[i])
add $t6, $s1, $s3 # Calculate address of A[i]
sw $t0, ($t6) # A[i] = B[i+1] + ( B[i-1] + B[i])
```

12. Complete the following C framework using only pointers! Make sure your code would not cause any warnings or errors when compiled with gcc on sparky.

```
int i;
double A[10];
int B[10];

double * pA = A;
int *pB = B;

for (i = 1; i < 9, i++)
{
//Implement A[i] = B[i-1] + B[i] + B[i+1] here
```

```
pA+(i*sizeof(double)) = (double) (*(pB+(i-1)*sizeof(int)) + *(pB
+ i*sizeof(int)) +  *(pB+(i+1)*sizeof(int)))

//Similar to (&A+(i*sizeof(double)) = (double)(*(&B+(i-
1)*sizeof(int)) + *(&B+i&sizeof(int)) + *(&B+(i+1)*sizeof(int)));

}

OR:
for (i = 1; i < 9, i++)
{
//Implement A[i] = B[i-1] + B[i] + B[i+1] here

      double* p = pA+i;
      int * d = (pB+i);
      p = (double) *d;
      p = p + (double) *(--d);
      p = p + (double) *(++d);
}
```

13. Write the MIPS code to exit a program

```
li $v0,10
syscall
```

14. Write the MIPS code to print the integer value in $t0 to the screen

```
li $v0, 1
mov $a0, $t0     # or add $a0, $t0, $0
syscall
```

15. Write the MIPS code to read in a string of length 30 characters from the user.

```
.data
str: .space 31

.text

li $v0, 8
la $a0, str
li $a1, 30
syscall
```