Your NAME: _____

EE 312 SP2013 Exam 1:

This exam is worth 37 points

1. (5 pts) Write a function that returns the offset of the smallest string in an array of strings. Specifically, the input to the function is a parameter of type char** named table. This parameter points to an array of pointers (array of char*). Each pointer in the array points to the beginning of a normal C string, except the last array element which is a null pointer. The array contains at least two elements (at least one element in addition to the null pointer). The return value of the function is an integer. The returned value must be the position in the array of the pointer to the smallest string. So, if table[k] points to the smallest string, then your function must return the value k. When comparing two strings to determine which string is smaller, use the function lessThan provided below.

```
int lessThan(char* x, char* y) { // return 1 iff x < y
    while (*x != 0) {
      if (*x < *y) { return 1; }
      if (*y < *x) { return 0; }
      x = x + 1;
      y = y + 1;
    }
    return 0;
}

int smallest(char** table) {
```

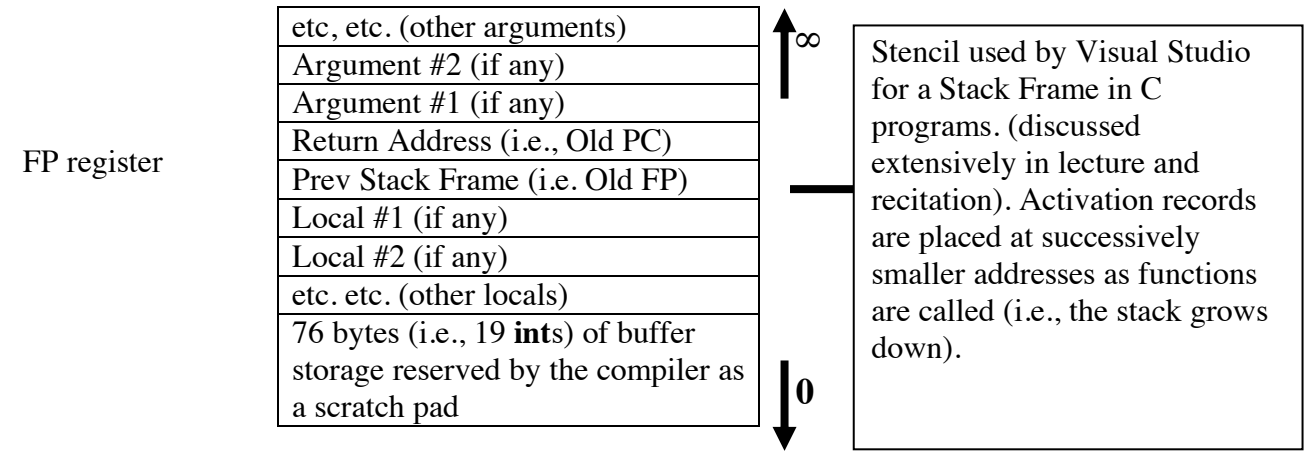
```
int table_len(char** table) {
    int l = 0;
    while (*table != 0) {
        l++;
        table++;
    }
    return l;

}

int smallest(char** table) {
    int min = strlen(table[0]);
    int i = 0;
    int k;
    int l = table_len(table);

    for (i = 1; i < l; i++) {
        int m = strlen(table[i]);
        if (m < min) {
            min = m;
            k = i;
        }
    }
    return k;
}
```

2. (5 pts) Write a function that sorts an array of pointers to strings (see question 1). The function has one parameter of type char** named table. Your sorting algorithm should be the selection sort algorithm and you must call the *smallest* function from question 1. Specifically, you should use *smallest* to find the smallest string in table. Then swap the pointer to this string with the first pointer in the table. Repeat by finding the second-smallest string and swapping with the second pointer in the table, and so forth. Please keep in mind that the primary purpose of this question is to see you invoke the function from question 1 correctly (and to see if you can swap pointers). It is not essential to write a correct implementation of selection sort to receive full credit (although you should be able to at least get very close).

```
void sort(char** table) {
```

Section 2: Program Analysis

For the questions in this section, analyze the program and indicate the output produced when the program is run. There is no partial credit in this section, so please be very careful. NOTE: all of the programs do run (there are no syntax errors or other compiler errors). You should assume that the following stencil is used to organize the information contained in a stack frame.



3. (two points) What is the output of the following program?

```
void doit(int32_t a, int32_t* p) {
  int32_t* q = &a;
  *q = 7;
  *p = 3;
  q[1] = q[1] + 4;
  *p = 1;
}
int main(void) {
  int32_t a = 5;
  int32_t b = 10;
  doit(a, &b);
  printf("%d %d", a, b);
}
```

_____

4. (two points) What is the output of the following program?

```
void doit(char x[], char y[]) {
  x = y;
  x[0] = 'j';
}

int main(void) {
  char x[] = "hello";
  char y[] = "world";
  doit(x, y);
  printf("%s %s", x, y);
}
```

_____

For the following three problems, fill in the table with the final contents of the heap after the statements have executed. You should assume that the heap is an array of 10 **int** locations and is implemented using the metadata and design described in class (i.e., the Knuth Heap). The diagram to the right shows the initial state of the heap. Each question starts with the heap in this state. The "?" indicates that the value stored in that memory location is unknown. Please show only the final state of the heap. No credit will be given for illegible answers. REMEMBER **the argument to malloc is in units of BYTES – two points each.**

the heap

index	contents
9	8
8	?
7	?
6	?
5	?
4	?
3	?
2	?
1	?
0	8

5.
```
int* p = malloc(1);
int* q = p;
free(p);
int* r = malloc(1);
*p = 100;
*q = 200;
*r = 300;
free(q);
```

the heap

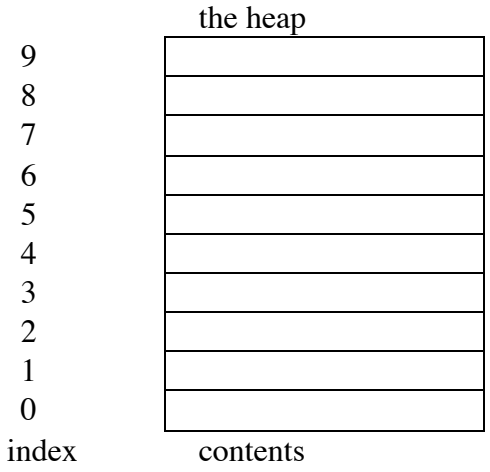
index	contents
9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

6.
```
int* p = malloc(8);
int* q = malloc(10);
```

the heap

index	contents
9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

7.
```
int* p = malloc(1);
*p = 3;
free(p);
int* q = malloc(20);
q[4] = 3;
q = q + 1;
free(q);
```



8. Please assume the following variables are all local variables for some function "doit". If necessary refer back to the stack frame stencil for any issues about the relative positions of these variables in memory. **Also please assume that the address assigned to the variable x is 1000**. Answer each of the questions below (one point each).

```
int x = -1; //  recall these are local variables in a function, and &x == 1000
int* p = &x;
char* q = (char*) p;
```

   a. What is the value of: *p?

   b. What is the value of: &p?

   c. What is the value of: *(p – 1)?

   d. Note that I have two variables, p and q where p == q, but p is of type int* and q is of type char*. I want to be certain that *p != *q. What value can I assign to *p that will guarantee *p != *q. I'm looking for a number here, like "0" or "1". (there's more than one right answer and 0 and 1 are not the right answers).

9. (2 pts each) Give an expression in "big-Oh" notation for the amount of time the following functions will take in the *worst case*.

a. O(log(N))

```
void doit(int x[], int N) {
   k = N - 1;
   while (x[k] != x[0]) {
       k = k / 2;
   }
}
```

b. $O(N^2)$

```
void doit(int x[], int N) {
   int last = N;
   while (last > 1) {
      int here = 1;
      for (j = 1; j < last; j += 1) {
          if (x[j] < x[j-1]) {
             here = j;
             int t = x[j];
             x[j] = x[j-1];
             x[j-1] = t;
          }
      }
      last = here;
   }
```

c. O(N)

```
int doit(int N) {
    int j = 0;
    int k = 0;
    while (j < N && k < N) {
       while (j <= k) { j += 1; }
       while (k <= j) { k += 1; }
    }
```

d. O(N)

```
void doit(int N) { // give the time complexity for this function
   int k = 1;
  while(k < fun(N)) {
    k = k + 1;
  }
}

int fun(int x) {
   int j = 0;
   while (j * j < x) {
     j = j + 1;
  }
  return j;
}
```

e. $O(N^2)$ $O(N^2)$

```
int doit(int N) {
   int k = 1;
   for (k = 0; k < N; k += 1)
      if (k == N / 2) {
         k = 0;
         N = N - 1;
      }
    }
}
```

Section 3: Semi-Short Answers

10. One point each
    a. Most experienced C programmers prefer to use pointers when working with structs. Which is the best explanation for this preference?
        i. The -> operator is visually more appealing than the . operator
        ii. Most structs are on the heap and have to be accessed with pointers
        iii. Actually this preference only applies to struct parameters, and the reason is because pointers are faster to place on the stack than whole structs
        iv. Pointers are more flexible since they can be type cast to int* (or other types) allowing experienced programmers the ability to directly access the components inside the struct

    b. Imagine you are a software development manager and facing a critical deadline for releasing a new version of a desktop application and the current implementation contains memory management bugs. Which choice makes the most sense:
        i. Focus the remaining development time on eliminating bugs other than use-after-free bugs, and release the code with use-after-free bugs in it.
        ii. Focus the remaining development time on eliminating bugs other than buffer overflow bugs, and release the code with buffer overflow bugs in it.
        iii. Focus the remaining development time on eliminating bugs other than memory leak bugs, and release the code with memory leak bugs in it.

    c. The "implied contract" for using an abstract data type includes which of the following
        i. Information (i.e., data) stored in the abstract data type will be "abstractions" of the program data (e.g., averages, statistical distributions or similar).
        ii. The components inside the abstract data type will not be accessed directly by the "client" code.
        iii. The abstract data type will provide functions that perform the arithmetic operations for addition, subtraction, multiplication and division, plus comparison functions for equality and ordering (i.e., less than).
        iv. The "client" code will not directly create struct objects (i.e., variables), but will use pointers-to-structs variables and call functions to create the objects on the heap.