NAME:

Login name:

Computer Science 217
Midterm Exam
March 11, 2009
10am-10:50am

This test has **four** (4) questions. You should spend no more than 10 minutes per question (except for the last, 30-point question) for the 50-minute exam. Put your name on ***every page***, and write out and sign the Honor Code pledge before turning in the test.

``I pledge my honor that I have not violated the Honor Code during this examination.''

Question	**Score**
1 (20 pts)	
2 (25 pts)	
3 (25 pts)	
4 (30 pts)	
Total	

QUESTION 1: Arithmetic and Logic Operations *(20 POINTS)*

1a) How is the decimal number **91** represented as an *eight-bit binary number*? What is the representation in *hexadecimal* notation of **91**? What *is two's complement* of **91**? *(3 points)*

1b) What does

```
printf("%d,%d,%d,%d,%d\n", 91/4, 91%4, 91&&4, 91&4, 91&3);
```

print to standard output**?** *(5 points)*

1c) Consider the following code, where **k** is a *signed 16-bit integer*:

```
printf("%d %d\n", (0 > k > 1), (4*(83/4) – 83));
```

What does the code print to standard output**?** Briefly explain your answers. *(6 points)*

1d) Consider the following code, where **k** is an *unsigned 16-bit integer*:

```
printf("%u\n",((k << 4) >> 4) – (k & 0xFFF));
```

What does the code print to standard out? Briefly explain your answer. *(6 points)*

QUESTION 2: Short Answer *(25 POINTS, 5 points each)*

2a) Briefly, *in one phrase each*, explain the meaning of the three ways of using the ampersand ('&') in C. That is, what is the meaning of (i) &x, (ii) x & y, and (iii) x && y?

2b) Briefly, *in one phrase each*, explain the difference between (i) '\0', (ii) '0', and (iii) "0".

2c) Why should a data structure storing key-value pairs, like a hash table or linked list, make *its own copy* of the keys? Why is this *even more important* for a hash table than a linked list?

2d) Why are local variables and function parameters stored on the STACK, instead of (say) in the DATA section of memory?

2e) Give *two* reasons why a modular design place a data-structure definition (e.g., the "struct" type definition) in the .c file rather than the .h file.

QUESTION 3: What Do These String Functions Do? *(25 POINTS)*

State concisely (in one sentence) what each of these four functions do.

3a) What does function q3a(char *s) do? *(6 points)*

```
void q3a(char *s) {
  int i;

  for (i=strlen(s)-1; i>=0; i--)
    putchar(s[i]);
  putchar('\n');
}
```

3b) What does function q3b(char *s) do? *(6 points)*

```
int q3b(char *s) {
  int i, j, yes=1;

  for (i=0, j=strlen(s)-1; i<=j; i++, j--)
    yes &= (s[i] == s[j]);
  return yes;
}
```

3c) What does function q3c(char *s) do? *(6 points)*

```
int q3c(char *s) {
  if (!(*s))
    return 0;

  for ( ; *s; s++)
    if ((*s < '0') || (*s > '9'))
      return 0;

  return 1;
}
```

3d) What does function q3d(char *s1, char *s2) do? *(7 points)*

```
int q3d(char *s1, char *s2) {
  int i, j, len1=strlen(s1), len2=strlen(s2);

  for (i=0; i<=len2-len1; i++) {
    for (j=0; j<len1; j++) {
      if (s1[j] != s2[i+j])
        break;
    }
    if (j == len1)
      return 1;
  }
  return 0;
}
```

QUESTION 4: Abstract Data Types *(30 POINTS)*

Consider the following "expanding array" data structure that supports adding key-value pairs and returning the value associated with a given key. The array grows dynamically as needed. Note, in the interest of brevity, that the code does ***not*** include the typical calls to assert().

```
enum {INITIAL_SIZE = 2};
enum {GROWTH_FACTOR = 2};

struct Pair {
   const char *key;
   int value;
};

struct Table {
   int pairCount;      /* Number of pairs in table */
   int arraySize;      /* Physical size of array */
   struct Pair *array; /* Address of array */
};

struct Table *Table_create(void) {
   struct Table *t;
   t = (struct Table*) malloc(sizeof(struct Table));
   t->pairCount = 0;
   t->arraySize = INITIAL_SIZE;
   t->array = (struct Pair*) calloc(INITIAL_SIZE, sizeof(struct Pair));
   return t;
}

void Table_add(struct Table *t, const char *key, int value) {
   if (t->pairCount == t->arraySize) {
      t->arraySize *= GROWTH_FACTOR;
      t->array = (struct Pair*) realloc(t->array,
            t->arraySize * sizeof(struct Pair));
   }
   t->array[t->pairCount].key = key;
   t->array[t->pairCount].value = value;
   t->pairCount++;
}

int Table_search(struct Table *t, const char *key, int *value) {
   int i;
   for (i = 0; i < t->pairCount; i++) {
      struct Pair p = t->array[i];
      if (strcmp(p.key, key) == 0) {
         *value = p.value;
         return 1;
      }
   }
   return 0;
}

void Table_free(struct Table *t) {
   free(t->array);
   free(t);
}
```

4a) If a client were to call Table_add() with a key that was already in the table, the Table_add() function would store the key a second time. Write a new function Table_unique_add() that adds a <key, value> pair *only if the key is not already in the table* (returning a 1 on success), and otherwise returns a 0 and does *not* insert the <key, value> pair. Feel free to call any of the existing functions listed above in your new Table_unique_add() function. (10 points)

4b) Create a Table_delete() function that, given a key, deletes the Pair associated with that key, returning a 1 on success; the function should return a 0 if the key does not exist in the table. The remaining key-value pairs do not have to stay in the same order. Please write the most efficient code for updating the data structure to reflect the removed entry, and *include the relevant calls to assert(). (10 points)*

4c) Write additional code, to run at the end of your new Table_delete() function, that decreases the space allocated to the array, when appropriate; assume that the "shrink factor" is the same as the "growth factor." *(10 points)*