Princeton University
COS 217: Introduction to Programming Systems
Spring 2009 Midterm Exam Answers

The exam was a 50-minute, open-book, open-notes exam.

Question 1 Part A

```
Eight-bit binary number:  01011011
Hexadecimal number:  5B
Two's complement:  10100101
```

Some students omitted the most significant bit '0' for the binary number and as a result, gave the wrong answer when converting it to two's complement.

Question 1 Part B

```
22,3,1,0,3
```

Question 1 Part C

```
0 -3
```

Explanation:
(0 > k > 1) = ((*0 or 1*) > 1) = 0
(4*(83/4) – 83) = ((4*20) – 83) = (80 – 83) = -3

Question 1 Part D

```
0
```

Explanation: The expression ((k << 4) >> 4) zeroes-out the most significant 4 bits of k. The expression (k & 0xFFF) also zeroes-out the most significant 4 bits of k. So the two expressions are equal, and subtracting them yields 0.

Some students thought k&0xFFF is equal to k. They forgot that k is a 16 bit number while 0xFFF is only 12 bits, so the most significant 4 bits of k will be zeroed-out.

Question 2 Part A

"&x" means "address of x". "x & y" means "bitwise-AND x with y". "x && y" means "logical-AND x with y".

Some students thought &x is a pointer.

Question 2 Part B

'\0' is the null character, that is, the character consisting of all zero bits, that is, the string-termination character. '0' is the zero character. "0" is the string containing the zero character.

Question 2 Part C

A data structure storing key-value pairs should make its own copies of its keys to avoid the possibility of the client changing a key, which might violate the client's expectations. In a hash table, changing a key could violate the integrity of the data structure.

Question 2 Part D

Some reasonable answers:

Storing local variables and function parameters on the stack allows the number of local variables and function parameters to expand dynamically as functions are called.

Local variables and function parameters must be created and destroyed in a last-created-first-destroyed manner. So a stack, which is a last-in-first-out data structure, is appropriate.

Question 2 Part E

A modular design places a data structure type definition in the .c file rather than in the .h file for:
(1) safety: to encapsulate the object's data so clients cannot directly access or modify those data, and
(2) maintainability: to allow the module's implementation to change over time without affecting clients.

Some students were only able to give one reason here.

Question 3 Part A

Print the characters of string s in reverse order.

Some students thought the last character of the string was not printed, because of the "strlen(s)-1", which instead is intended to skip printing the '\0' string terminator.

Question 3 Part B

Return 1 if string s is a palindrome, and 0 otherwise.

Some students explained *how* the function worked, rather than *what* it does.

Question 3 Part C

Return 1 if the characters of string s comprise a decimal integer, and 0 otherwise.

Several students gave answers that were not precise or concise, including answers that were not correct for the case of an empty string (""), e.g., "The function returns a 0 if the string contains non-numeric characters, and a 1 otherwise."

Question 3 Part D

Return 1 if string s1 is a substring of string s2, and 0 otherwise.

Some students mistakenly thought the check only applied at the beginning or end of s2.

Question 4 Part A

This is a correct answer:

```
int Table_unique_add(struct Table *t, const char *key, int value) {
   int temp;
   if (Table_search(t, key, &temp))
      return 0;
   Table_add(t, key, value);
   return 1;
}
```

This is a common incorrect answer:

```
int Table_unique_add(struct Table *t, const char *key, int value) {
   int *temp;
   if (Table_search(t, key, temp))
      return 0;
```

```
      Table_add(t, key, value);
      return 1;
   }
```

That code is incorrect because Table_search() might assign an int to the memory pointed to by "temp". But "temp" hasn't been assigned a value, and so points to some arbitrary place in memory. The result could be memory corruption or a segmentation fault.

This is another common incorrect answer:

```
   int Table_unique_add(struct Table *t, const char *key, int *value) {
      if (Table_search(t, key, value))
         return 0;
      Table_add(t, key, *value);
      return 1;
   }
```

That code is incorrect because the "value" formal parameter should be of type int (as with Table_add()), not int*. Moreover, the possibility exists that Table_search() could change the integer to which "value" points, which would affect the caller of Table_unique_add().

Question 4 Part B

This is a correct answer:

```
   int Table_delete(struct Table *t, const char *key) {
      int i;
      assert(t != NULL);
      assert(key != NULL);

      for (i = 0; i < t->pairCount; i++)
         if (strcmp(t->array[i].key, key) == 0)
            break;

      if (i == t->pairCount)
         return 0;

      /* Overwrite element i with the last element. */
      t->array[i] = t->array[t->pairCount-1];

      t->pairCount--;
      return 1;
   }
```

This alternative function definition works, but is an incorrect answer because it doesn't use the most efficient approach:

```
   int Table_delete(struct Table *t, const char *key) {
      int i;
      int j;
      assert(t != NULL);
      assert(key != NULL);

      for (i = 0; i < t->pairCount; i++)
         if (strcmp(t->array[i].key, key) == 0)
            break;

      if (i == t->pairCount)
         return 0;

      /* Move all elements following element i upward one place. */
      for (j = i+1; j < t->pairCount; j++)
         t->array[j-1] = t-array[j];

      t->pairCount--;
      return 1;
   }
```

Question 4 Part C

This is a correct answer:

```
   if ((t->arraySize > INITIAL_SIZE) &&
       ((t->pairCount * GROWTH_FACTOR) <= t->arraySize)) {
      t->arraySize /= GROWTH_FACTOR;
      t->array = (struct Pair*)
         realloc(t->array, t->arraySize * sizeof(struct Pair));
   }
```

The condition (t->arraySize > INITIAL_SIZE) is necessary. Without that condition the value of t->arraySize could become 0, and thereafter *always* would be 0. However, few students included that condition. So we did not deduct points if you did not include that condition. Instead we awarded a bonus point if you *did* include that condition.

The condition (t->pairCount * GROWTH_FACTOR) <= t->arraySize) defines the most aggressive contraction policy. Using it could lead to "oscillations"; imagine the massive inefficiency if expansion and contraction were to occur repeatedly in a loop. So we accepted any reasonable less aggressive contraction policy.