Princeton University
COS 217: Introduction to Programming Systems
Fall 2003 Midterm Exam Answers

Question 1

Here is the corrected program, with boldface comments marking the corrections:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

#define MAX_STRING_LENGTH 10

struct node {
  char *str;
  struct node *next;
};

/* Read a line of stdin into output; return NULL if end-of-file */
char * read_string() {
  size_t n = 0;
  int c;

  char str[MAX_STRING_LENGTH];
  char *output;

  /* read the input into a temporary buffer */

  /* (5) String representation error.
     The str array can store MAX_STRING_LENGTH chars, and so can store a
     string containing at most MAX_STRING_LENGTH – 1 chars.  (Remember that
     a string must be terminated with a null character.)  The termination
     condition of the "for" statement must be changed accordingly.  An
     alternative would be to leave the "for" statement unchanged, and define
     str so it can store MAX_STRING_LENGTH + 1 chars. */

  for (n = 0; n < MAX_STRING_LENGTH - 1; n++) {
    c = getchar();
    if (c == '\n' || c == EOF) break;
    str[n] = (char) c;
  }

  if (c==EOF && n==0) return NULL;

  /* (6) String representation error.
     Must terminate the string with a null character. */

  str[n] = '\0';

  /* Allocate the output buffer, and copy the string there */
  output = malloc(strlen(str) + 1);
  assert(output != NULL);
  strcpy(output, str);

  /* Discard any remaining characters on the line. */
  while ((c != '\n') && (c != EOF)) c = getchar();

  return output;
}

int main(int argc, char **argv) {
  char *line;
  struct node *first_node = NULL, *pn;

  for(line=read_string(); line; line=read_string()) {

     /* (1) Memory management error.
        sizeof(struct node*) evaluates to 4.  That's too little
        memory.  Must use sizeof(struct node) or sizeof(*new_node)
        instead. */

    struct node *new_node = malloc(sizeof(struct node));
    assert(new_node != NULL);

    new_node->str = line;

    if (first_node == NULL) {
       /* case 1, this is the first node */
      new_node->next = NULL;
      first_node = new_node;

      /* (2) Logic error.
         The new node must be the first one in the list if its string
         is *less than or* equal to the string of the current first node. */

    } else if (strcmp(new_node->str, first_node->str) <= 0) {
       /* case 2: new_node should be the first string in the list */
      new_node->next = first_node;
      first_node = new_node;
    } else { /* all other cases */
       for (pn = first_node; ; pn = pn->next) {

          /* (3) Memory management/logic error.
             If pn->next evaluates to NULL, then the expression pn->next->str
             would generate a segmentation fault.  So must reverse the
             expressions. */

          if (pn->next == NULL || strcmp(new_node->str, pn->next->str) <= 0) {
          new_node->next = pn->next;
          pn->next = new_node;
          break;
          }
       }
    }
  }

  /* Now print all strings so far in sorted order. */

  /* (4) Memory management/logic error.
     The body of the loop frees pn.  Subsequently pn is a dangling pointer,
     and so it is erroneous to examine the memory to which pn points.
     But the "for" statement's third expression (pn = pn->next) does exactly
     that.  So the loop must be restructured.  Here's one way... */

  for ( ; first_node; first_node = pn) {
    pn = first_node->next;
    printf("%s\n", first_node->str);
    free(first_node->str);
    free(first_node);
  }

  return 0;
}
```

Question 2 (a)

```
#ifndef BOARDTABLE_INCLUDED
#define BOARDTABLE_INCLUDED
```

```
typedef struct BoardTable *BoardTable_T;

BoardTable_T BoardTable_new(void);
void BoardTable_free(Board_T oBoardTable);
int BoardTable_put(BoardTable_T oBoardTable, Board oBoard, int iValue);

int BoardTable_contains(BoardTable_T oBoardTable, Board oBoard);
int BoardTable_get(BoardTable_T oBoardTable, Board oBoard);
int BoardTable_remove(BoardTable_T oBoardTable, Board oBoard);

#endif
```

Question 2 (b)

```
(1) if (BoardTable_contains(oBoardTable, board))
       return BoardTable_get(oBoardTable, board);

(2) BoardTable_put(oBoardTable, board, b);
```

Question 2 (c)

An efficient way to implement the ADT would be as a hash table. So the data structures should be appropriate for a hash table whose keys are Boards and whose values are integers...

```
struct Binding {
   Board_T oBoard;
   int iValue;
   struct Binding *psNextBinding;
};

struct BoardTable {
   struct Binding **ppsBuckets;
};
```

Or, assuming that the hash table will never be resized...

```
struct Binding {
   Board_T oBoard;
   int iValue;
   struct Binding *psNextBinding;
};

struct BoardTable {
   struct Binding *ppsBuckets[BUCKET_COUNT];
};
```

Question 3 (a)

A SymTable object “owns” its keys iff it allocates and frees the memory in which they reside.

Question 3 (b)

Protection against corruption. A client cannot corrupt a SymTable object.

Client convenience. (Perhaps multiple) clients need not manage the memory in which the keys reside.

Question 3 (c)

```
strcpy(pcKey, “Ruth”);
SymTable_put(oSymTable, pcKey, pvValue);
strcpy(pcKey, “Gehrig”);  /* oSymTable would be corrupted */
```

Question 3 (d)

Efficiency. A SymTable object would be more time and space efficient.

Question 3 (e)

Iff if the SymTable object knows the type of its values. Only then could it make copies of them.

Question 3 (f)

```
SymTable_T SymTable_new(int (*pfCompare)(const void *pvKey, const void *pvKey));
void *SymTable_get(SymTable_T oSymTable, const void *pvKey);
```

or

```
SymTable_T SymTable_new(void);
void *SymTable_get(SymTable_T oSymTable, const void *pvKey,
          int (*pfCompare)(const void *pvKey, const void *pvKey));
```

Explanation...

The keys would need to be void pointers instead of char pointers.

Moreover, the client would need to provide a pointer to a “compare” function. As shown in the first code fragment, the client could provide that pointer when calling the SymTable_new function; the SymTable_new function could then store that function pointer within the object’s SymTable structure. The SymTable_get, SymTable_put, etc. functions could then use that stored function pointer.

Or the client could provide that pointer when calling the SymTable_get, SymTable_put, etc. functions, as shown in the second code fragment.

The first code fragment is preferable: it is more convenient for the client, and it eliminates the possibility that the client will erroneously call the functions of a SymTable object with different compare functions.

Question 3 (g)

No. The client also would need to provide a pointer to a “hash” function.

Explanation...

Since the SymTable object does not know the types of its keys, it could not compute hash codes for them. Thus the client also would need to provide a pointer to a hash function. The revised interface would look like this:

```
SymTable_T SymTable_new(
   int (*pfCompare)(const void *pvKey, const void *pvKey),
   unsigned int (*pfHhash)(const void *pvKey));
void *SymTable_get(SymTable_T oSymTable, const void *pvKey);
```

Question 4

```
typedef struct {
   Piece square[8][8];
   double value;
   int red_base;
   int black_base;
} * Board;

void applyDelta(Board b, int row, int col, Piece before, Piece after) {
   assert(b->square[row][col] == before);

   if (before != NONE) {
      b->value -= before;
      b->value -= before * protection(b, before, row, col);
      if ((before == RED) && (row == 0))
         b->red_base--;
      if ((before == BLACK) && (row == 7))
         b->black_base--;
   }

   b->square[row][col] = after;

   if (after != NONE) {
      b->value += after;
      b->value += after * protection(b, after, row, col);
      if ((after == RED) && (row == 0))
         b->red_base++;
      if ((after == BLACK) && (row == 7))
         b->black_base++;
   }
}

double eval(Board b) {
   double v = 0.0;
   v = b->value;
   v += .01 * RED * b->red_base * b->red_base;
   v += .01 * BLACK * b->black_base * b->black_base;
   return v;
}
```