Access Softek’s C Proficiency Test

These programming problems are designed to help us evaluate your ability to interpret, solve, and debug programming problems in the ANSI C programming language. In answering the questions, endeavor to write complete, commented code using meaningful identifier names and informative indentation. Reference manuals are available for both C and the standard runtime library. Though we prefer answers in C source code, you may give your solutions in pseudo-code, words, and/or pictures if you prefer.

Here at Access Softek, we are much more concerned with writing readable and portable code than fast or compact code. For instance, constructs such as condensing two statements into one, putting assignments into conditional tests, and using right shifts instead of division are considered substandard. Assumptions about the size of pointers and the like are indications of non-portability.

Most of the following problems have several solutions. Answer with the one you believe is most complete. Thoroughness is much more important than finishing quickly, so take care to answer all parts of each problem. Also, take care to spend your time in proportion to the problems’ weights.

You may assume that all strings terminate with a null character, `'\0'`, and are ASCII-encoded. For those familiar with the various memory models of MS-DOS machines, please do not worry about any 64K limits.

Please write or type your answers in the free room after each question.

Problem 1. Find the Bugs (5 points each)
The following routines each have one bug. Correct each one.

a. *This function is supposed to append* `string_tail` *to the end of* `string_head` *without using the* `strcat()` *function. It assumes that* `string_head` *has enough room to contain the result.*

```
char* AppendString(char* string_head, char* string_tail)
{
   int length;

   length = strlen(string_tail);
   strcpy(string_head + length, string_tail);
   return string_head;
}
```

Solution. `length` variable should contain length of string_head as it’s used as offset to point to destination buffer.

```
char* AppendString(char* string_head, char* string_tail)
{
   int length;

   length = strlen(string_head);
   strcpy(string_head + length, string_tail);
   return string_head;
}
```

Problem 1. Find the Bugs

b. String Look-up

This function returns the look-up table string given a key. Passing the string "tofu" should return a pointer to "soy beans". Passing the string "chickens" should return a `NULL` *pointer.*

```
static char* arrayLookup[] =
{
   "eggs",       "chickens",
   "tofu",       "soy beans",
   "coriander",  "cilantro",
   "yeast",   "wheat"
};

char* LookupString(char* string_key)
{
   char** strIndex;

   for (strIndex = arrayLookup; strIndex; strIndex += 2)
   {
      if (!strcmp(*strIndex, string_key))
      {
         return strIndex[1];
      }
   }
   return NULL;
}
```

Solution. for loop continue condition is malformed. It should check if we've got to the end of table. Hence, I suggest to have a dummy NULL to denote table end and add a special case for string_key being null.

```
static char* arrayLookup[] =
{
   "eggs",       "chickens",
   "tofu",       "soy beans",
   "coriander",  "cilantro",
   "yeast",   "wheat",
   NULL
};

char* LookupString(char* string_key)
{
   char** strIndex;

   if (!string_key)
   {
     return NULL;
   }

   for (strIndex = arrayLookup; *strIndex != NULL; strIndex += 2)
   {
      if (!strcmp(*strIndex, string_key))
      {
         return strIndex[1];
```

```
    }
  }
  return NULL;
}
```

Problem 1. Find the Bugs

c. This function accepts an MS-DOS pathname string, which is of the form `drive_letter:\directory\...\directory\filename.extension`
It returns a pointer to the start of the base filename. For example, given the string “`c:\source\graph\spin.c`*” it returns a pointer to the character “*`s`*” in “*`spin`*”.*

```
char* FindBasename(char* string_filename)
{
  int i;

  /* Look for the beginning of the filename: */
  for (i = strlen(string_filename); i; i--)
  {
    switch (string_filename[i - 1])
    {
    case ':':       /* Look for path separators */
    case '\\':
      break;        /* Quit when one is found */
    }
  }
  /* Return a pointer to the filename: */
  return string_filename + i;
}
```

Solution. Suppose, the string is valid. Then, the only case when it doesn’t have a filename is when it denotes a drive only or a directory, such as “c:” or “c:\xyz\” or “c:\”.
I’m not sure if “c:file.ext” is valid, hence, I suppose it isn’t. Hence, the function shouldn’t check for `:` as path separator. Also, if we’ve got to the beginning of the input string here then there’s no basename present and a NULL can be returned.

```
char* FindBasename(char* string_filename)
{
  int i;

  /* Look for the beginning of the filename: */
  for (i = strlen(string_filename); i; i--)
  {
    switch (string_filename[i - 1])
    {
    case '\\':      /* Look for path separators */
      break;        /* Quit when one is found */
    }
  }

   if (0 == i) {
     return NULL;
   }
  /* Return a pointer to the filename: */
  return string_filename + i;
}
```

Problem 1. Find the Bugs

d. This function initializes each element of an array to contain its own index.

```
void InitArray(int* array, int array_length)
{
   int i = 0;

   while (i < array_length)
   {
      array[i] = i++;
   }
}
```

Solution. The function should check if array isn't null.

```
void InitArray(int* array, int array_length)
{
   int i = 0;

   if (!array)
   {
     return;
   }

   while (i < array_length)
   {
      array[i] = i++;
   }
}
```

Problem 2. High Quality Code (10 points each)
Rewrite the following code fragments to turn them into production-quality code. (Don't forget that good code includes comments.)

a. Machine-Dependence and Non-Portability:
This function finds the centroid, or average, of two two-dimensional points A *and* B:

```
long Centroid(long A, long B)
{
   int x, y;

   x = ((int) (A >> 16) + (int) (B >> 16)) / 2;
   y = ((int) A + (int) B) / 2;
   return ((long) x << 16) + y;
}
```

Solution. The straight-forward solution here is to have a structure for each point rather than have them packed into a `long` variable. One might use pointers instead of passing the structures to the function, but I stick to passing by-value here for simplicity.

```
struct Point
{
 int X;
 int Y;
};

struct Point Centroind (struct Point A, struct Point B)
{
 struct Point Result;
 Result.X = (A.X + B.X) / 2;
 Result.Y = (A.Y + B.Y) / 2;
 return Result;
}
```

Problem 2. High Quality Code

b. Error Checking:
This function reads a string, preceded by a two-byte length, from a file into a C-style string in memory.

```
char *ReadStringFromFile(FILE *fp)
{
   char  *str;
   int   length;

   fread(&length, 2, 1, fp);
   str = malloc(length + 1);
   fread(str, 1, length, fp);
   str[length] = 0;
   return str;
}
```

Solution. Should've checked if both freads and malloc succeeded. Also, length should have been checked for being a positive number. Hence, it should be an unsigned integer. Basic change to function contract is return a null if case of error.

```
char *ReadStringFromFile(FILE *fp)
{
   char  *str;
   unsigned int  length;
   int  read_ret; // holds result of read

   read_ret = fread(&length, 2, 1, fp);

   if (read_ret < 1) // less than one chunk of 2 bytes long was read
   {
     return NULL;
   }
   if (length) // nothing to read, hence bail out
   {
     return NULL;
   };

   str = malloc(length + 1);
   if (!str) // memory allocation error
   {
     return NULL;
   }

   read_ret = fread(str, 1, length, fp);
   if (read_ret < length) // fail to read, deallocate memory, bail out
   {
     free(str);
     return NULL;
   }

   str[length] = 0;
   return str;
}
```

Problem 3. Linked-List enhancement (30 points)
Given the data structure `item` *as an item in a linked list, write the function* `AddKey()` *to add a node to the list. Maintain the list sorted by* `key`*.*

Include a comment describing how the function behaves, your assumptions, and the return value from the function. Be sure to think about special cases and possible errors.

```
typedef struct _NODE
{
   struct _NODE  *node_next;
   int           key;
   char          *data;
} ITEM, *NODE, *LIST;
```

LIST AddKey (LIST list, int key, char* data);

Solution. Suppose, the list is sorted by ascending key i.e. the lowest key is in head and the largest one is in tail of the list. The function should find appropriate place for new element first and put it in there with proper changes to node_next of previous element. Suppose, node_next of tail of the list is NULL. Suppose, the list is NULL if it's an empty list, not an error. The list is single-linked, hence inserting into tail or body of the list are the same use-cases.

```
LIST AddKey (LIST list, int key, char* data)
{
  // previous and current nodes when iterating over the list
  NODE Prev;
  NODE Cur;
  NODE Inserted;

  Inserted = malloc(sizeof(ITEM));
  if (!Inserted) // mem alloc failed
  {
    return NULL;
  }

  Inserted->node_next = NULL; // just init
  Inserted->key = key;
  Inserted->data = data;

  if (!list) // Add the very first element of the list
  {
    return Inserted;
  }

  for (Prev = NULL, Cur = list; Cur; Prev = Cur, Cur = Prev-
>node_next)
  {
    if (key < Cur->key)
    {
      break;
    }
  }

  if (!Prev) // insert in head
  {
    Inserted->node_next = list;
  }
  else // insert in tail, Prev contains tail
  {
    Prev->node_next = Inserted;
  }

  return Inserted;
}
```

Problem 4. Loop Optimization (Bonus — 10 points. Do the other problems first.) Rewrite the function for fast execution. Assume that the compiler makes no optimizations and generates code literally from the source. Do not use `register` *variables.*

```
char* SlowFunction(char* output, char* input)
{
   int i, j;

   for (i = 0, j = 0; input[i]; ++i)
   {
      if (input[i] >= ' ' && input[i] < 'a')
      {
         output[j++] = input[i];
      }
      if (input[i] >= 'a' && input[i] <= 'z')
      {
         output[j++] = input[i] - ('a' - 'A');
      }
      if (input[i] > 'z' && input[i] < 0x7f)
      {
         output[j++] = input[i];
      }
```

```
    }
    output[j] = '\0';
    return output;
}
```

Solution. One can increase the pointers instead of only integer indices first. The if statements sequence can be changed to if-else. Hold dereferenced input char in a variable.

```
char* AnotherSlowFunction(char* output, char* input)
{
    char inputVar;
    for (*input; ++input)
    {
        inputVar = *input;
        if (inputVar >= ' ' && inputVar < 'a')
        {
            *output = inputVar;
            ++output;
        }
        else if ( inputVar >= 'a' && *input <= 'z')
        {
            *output = outputVar - ('a' – 'A');
            ++output;
        }
        else if (inputVar > 'z' && inputVar < 0x7f)
        {
            *output = inputVar;
            ++output
        }
    }
    *output = '\0';
    return output;
}
```