UNIVERSITY OF TORONTO
FACULTY OF APPLIED SCIENCE AND ENGINEERING

APS 105 — Computer Fundamentals
Final Examination
December 21, 2010
2:00 p.m. – 4:30 p.m.

Examiners: J. Anderson, T. Fairgrieve, B. Li, M. Papagelis

Exam Type A: This is a "closed book" examination; no aids are permitted.

Calculator Type 4: No calculators or other electronic devices are allowed.

All questions are to be answered on the examination paper. If the space provided for a question is insufficient, you may use the last page to complete your answer. If you use the last page, please direct the marker to that page and indicate clearly on that page which question(s) you are answering there.

You must use the C programming language to answer programming questions. You are not required to write `#include` directives in your solutions. You may use any math function you have learned, as necessary.

The examination has 19 pages, including this one.

Circle your lecture section (**one mark deduction** if you do not correctly indicate your section):

L0101	or	**L0102**	or	**L0103**	or	**L0104**	or	**L0105**
Anderson		Papagelis		Li		Fairgrieve		Fairgrieve
Monday 2 PM		Monday 9 AM		Monday 11 AM		Monday 11 AM		Monday 4 PM

Full Name: ____

Student Number: ____________________ ECF Login: ____________________

MARKS

1	2	3	4	5	6	7	8	9	10	11	Total
/28	/6	/6	/6	/6	/8	/8	/8	/8	/8	/8	/100

Question 1 [28 Marks]

1.1 [3 Marks] Write a single C statement that prints the following message followed by a new line character:

```
*/\/\*
```

Solution:

```
printf(‘‘*/\\/\\*\n’’);
```

1.2 [3 Marks] Write a single C statement that declares an `int` array variable named `values` with 5 elements. Each of the 5 elements should be initialized to `99`.

Solution:

```
int values[5] = {99,99,99,99,99};
```

1.3 [3 Marks] Write a single C statement that declares a `char` type variable named `encoded` and initializes it based on the value of another `char` type variable named `c` that has already been declared. The variable `c` has been initialized to an upper case letter. The value of `encoded` should be initialized according to this scheme:

If `c` is `’Z’`, `encoded` should be initialized to `’A’`.
If `c` is `’X’`, `encoded` should be initialized to `’B’`.
...
If `c` is `’A’`, `encoded` should be initialized to `’Z’`.

Solution:

```
char encoded = ’Z’ - (c - ’A’);
```

1.4 [3 marks] Given the following declarations:

```
struct stuRec
{
  int stuNumber;
  char name[100];
  double score;
};

typedef struct stuRec studentRecord;
```

Write a single C statement that uses `malloc` to declare an array of type `studentRecord` with 100 elements. The array should be named `sRecords`.

Solution:

```
studentRecord* sRecords = (studentRecord *)malloc(100*sizeof(studentRecord));
```

1.5 [4 Marks] Consider the following C code fragment:

```
int list[4] = {0};

int *p = list;
p++;
*p = p - list;
p++;
*p = p - list;
p = &list[3];
*p = 4;
```

Give the values the array elements:

Solution:

Array element	Answer
`list[0]`	0
`list[1]`	1
`list[2]`	2
`list[3]`	4

1.6 [4 Marks] The following implementation of quick sort has errors in two statements. Fix the errors in the code to implement quick sort. The parameter `values` is an `int` type array that must be sorted in ascending order.

```
void quickSort (int values[], int low, int high)
{
  int left = low;
  int right = high;

  if (left >= right)
    return;

  int pivot = values[left];

  while (left < right)
  {
    while ((values[right] >= pivot) && (left < right))
      right--;

    values[left] = values[right];

    while ((values[left] <= pivot) && (left < right))
      left++;

    values[right] = values[left];
  }
  values[left] = pivot;

  quickSort(values, low, left+1);
  quickSort(values, high, right+1);
}
```

Solution:
The bugs are in the two recursive call statements, which should be:

```
  quickSort(values, low, left-1); // "left-1" can also be "right-1"
  quickSort(values, left+1, high); // "left+1" can also be "right+1"
```

1.7 [4 Marks] Add C statements to the following C function to implement bubble sort. The parameter `values` is an `int` array that must be sorted in ascending order. The parameter `n` is the number of elements in the array `values`.

```
void bubbleSort (int values[], int n)
{
  int top = n;
  int i, j;

  for (j = top; j > 1; j--)
  {
    for (i = 0; i < top - 1; i++)
    {
      if (values[i] > values[i+1])
      {







      }
    }
  }
}
```

Solution:

```
int temp = values[i];
values[i] = values[i+1];
values[i+1] = temp;
```

1.8 [4 Marks] What output does the following program produce?

```
#include <stdio.h>

int factorial (int n)
{
  int val;
  printf("Entering %d\n", n);
  if (n == 1)
    val = 1;
  else
    val = n * factorial(n-1);
  printf("Leaving %d\n", n);
  return val;
}

int main (void)
{
  int i;
  i = factorial(2);
  printf("%d\n", i);
  return 0;
}
```

Solution:

```
Entering 2
Entering 1
Leaving 1
Leaving 2
2
```

Question 2 [6 Marks]

An ancient algorithm for finding prime numbers is called the *Sieve of Eratosthenes*. To use this algorithm to find all prime numbers in a list of consecutive integers less than a given integer, say 100, we start from $p = 2$, the smallest prime number. We eliminate all multiples of p less than 100, $(2p, 3p, 4p, \ldots)$. Since they are multiples of p, they are not prime numbers. We then find the first number after p that has not yet been eliminated, which is the next prime number. We assign this prime number to p, and eliminate its multiples. We repeat this procedure until p^2 is greater than 100, and all the remaining numbers in the list are prime numbers.

Write a C program that uses the *Sieve of Eratosthenes* algorithm to find and print all prime numbers less than 100. Your implementation should not use the `%` operator. The output of your program should be:

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Solution:

```
int main(void)
{
    int sieve[100] = {0}, i, p = 2;
    while (p < sqrt(100)) {
        printf("%d ", p);

        // eliminate multiples of p
        i = 2;
        while (i * p < 100) {
            sieve[i * p] = 1;
            i ++;
        }

        // find the next p
        p++;
        while (sieve[p] == 1)
            p++;
    }
    // print all remaining prime numbers
    for (;p < 100; p++)
        if (sieve[p] == 0)
            printf("%d ", p);
    printf("\n");
}
```

Question 3 [6 Marks]

Assume that the `date` structure contains three members: `month`, `day`, and `year` (all of type `int`). Write a C function named `compareDates`, the prototype of which is given below, that returns `-1` if `d1` is an earlier date than `d2`, `1` if `d1` is a later date than `d2`, and `0` if `d1` and `d2` are the same. Assume `d1` and `d2` contain valid dates

Solution:

```
int compareDates (struct date d1, struct date d2)
{
    int result;
    if (d1.year < d2.year)
        result = -1;
    else if (d1.year > d2.year)
        result = 1;
    else
    {
        if (d1.month < d2.month)
            result = -1;
        else if (d1.month > d2.month)
            result = 1;
        else
        {
            if (d1.day < d2.day)
                result = -1;
            else if (d1.day > d2.day)
                result = 1;
            else
                result = 0;
        }
    }
    return result;
}
```

Question 4 [6 Marks]

Write a C function named `countDistinct`, the prototype of which is given below, that has two parameters. The parameter `list` is an integer array already sorted in ascending (nondecreasing) order, and the parameter `listLength` gives the number of elements in `list`. The function counts and returns the number of distinct elements in `list`.

For example, if `list` is `{1, 1, 3, 3, 3, 3, 6, 8, 8, 9, 9, 9, 9, 9}`, the function returns `5`, as `list` has 5 distinct elements.

Solution:

```
int countDistinct (int list[], int listLength)
{
  int count = 0, i = 0;
  while (i < listLength)
  {
    while (i < listLength - 1 && list[i] == list[i+1])
      i++;
    i++;
    count++;
  }
  return count;
}
```

Question 5 [6 Marks]

Add **one** statement to the `printPermutationsHelper` function below so that the given program prints all permutations of the digits between `1` and `n`. For example, when the user enters the number `3`, the program should output:

```
123 132 213 231 312 321
```

Solution:

```
#include <stdbool.h>
#include <stdio.h>
bool digitDoesNotAppear (int digit, int number)
{
    bool result = true;
    while (number > 0 && result)
    {
        if (number%10 == digit)
            result = false;
        number = number / 10;
    }
    return result;
}
void printPermutationsHelper (int a, int b, int c)
{
    if (c == 0)
        printf("%d ", a);
    else
        for (int i = 1; i <= b; i++)
        {
            if (digitDoesNotAppear(i,a))
                // add one statement in the box below
                --------------------------------------------------------
               |  printPermutationsHelper(10*a+i,b,c-1);                |
                --------------------------------------------------------
        }
}
void printPermutations (int n)
{
    printPermutationsHelper(0,n,n);
}
int main (void)
{
    int n;
    scanf("%d",&n);
    printPermutations(n);
    return 0;
}
```

Question 6 [8 Marks]

Write a C function having prototype `void moveRight (int arr[], int size, int k)` that moves the first `(size - k)` elements in array `arr` `k` places to the right and wraps the last `k` elements in array `arr` around to the left end of the array.

Assume that `0 <= k < size`.

For example, if `arr[] = {11, 22, 33, 44, 55, 66, 77, 88}`,
the call `moveRight(arr, 8, 3)` changes `arr` to `{66, 77, 88, 11, 22, 33, 44, 55}`.
Solution:

```
void moveRight (int arr[], int size, int k)
{
    int *temp = (int *) malloc(size*sizeof(int));

    int i;
    for (i=0; i<size; i++)
        temp[(i+k)%size] = arr[i];

    for (i=0; i< size; i++)
        arr[i] = temp[i];

}
void moveRightAlternate (int arr[], int size, int k)
{

    int j;
    for (j=1; j<=k; j++)
    {
        int i, temp=arr[size-1];
        for (i=size-1; i>0; i--)
            arr[i] = arr[i-1];
        arr[0] = temp;
    }
}
```

Question 7 [8 Marks]

Write a complete C function named `mergeSortedStrings`, the prototype of which is given below, that receives two alphanumerically ordered character strings (`str1` and `str2`) as parameters. The function must create and return a new string that contains the characters from `str1` and `str2` and is alphanumerically ordered as well. For example, if `str1` were `"CEFHIJ"` and `str2` were `"ABDG"` the function must create and return a new string containing `"ABCDEFGHIJ"`. You may assume that characters in the strings `str1` and `str2` are unique (i.e., they cannot appear in both strings).

You may use library functions from `string.h` in your solution. You may assume that the arguments are equal sized strings.

Solution:

```
char * mergeSortedStrings (char str1[], char str2[])
{
  char * newString = (char *) malloc
                     (sizeof(char) * (strlen(str1)+strlen(str2)+1));

  int i = 0;
  while (*str1 != '\0' && *str2 != '\0')
  {
    if(strcmp(str1, str2) < 0)
    {
  newString[i] = *str1;
  i++;
  str1++;
}
else
{
  newString[i] = *str2;
  i++;
  str2++;
}
  }

  // Copy the rest of the non-ended string
  if (*str1 == '\0')
  {
    while (*str2 != '\0')
    {
  newString[i] = *str2;
  i++;
  str2++;
}
  }
  else
  {
    while(*str1 != ’\0’)
    {
  newString[i] = *str1;
  i++;
  str1++;
}
  }
  return newString;
}
```

Question 8 [8 Marks]

Write a complete C function named `minElementSort`, the prototype of which is given below, that receives an integer array `list` and its size `listLength` as its parameters. The function sorts the given array in ascending (nondecreasing) order, following a slight variation of the selection sort algorithm, called the `minimum element sort`. The algorithm works as follows: It goes through the list, finds the smallest element and swaps it with the first element of the list. Then, finds the smallest element of the rest of the list and swaps it with the second element of the list. And so on. An example follows.

If the input integer array `list` is:

```
10 5 20 40 50 12 434 21 9 232
```

then after calling your sorting function, the integer array `list` is:

```
5 9 10 12 20 21 40 50 232 434
```

Solution:

```
void minElementSort (int list[], int listLength)
{
  int bottom, i, temp, minLoc;

  for (bottom = 0; bottom < listLength - 1; bottom ++)
  {
    minLoc = bottom;

    for (i = bottom+1; i <= listLength-1; i++)
      if(list[i] < list[minLoc])
        minLoc = i;

temp = list[bottom];
list[bottom] = list[minLoc];
list[minLoc] = temp;
  }
}
```

Question 9 [8 Marks]

The sequence $2, 6, 18, 54, 162, \ldots$ is said to be *geometric* since the result from dividing each term by the term that precedes it is always the same. In the given sequence, the common ratio is 3. A sequence with fewer than 2 terms is **not** geometric.

Write a function named `isGeometric` that uses **recursion** to determine whether or not the sequence of integer values given in a type `int` array is a geometric sequence.

Your solution **must** use recursion to solve the problem. You **may** use a "helper" function in your solution.
Solution:

```
bool isGeometricHelper (int arr[], int left, int right, int ratio)
{
    int result;
    if (left == right)
        result = true;
    else if (arr[left+1]/arr[left] == ratio)
        result = isGeometricHelper(arr, left+1, right, ratio);
    else
        result = false;

    return result;
}

bool isGeometric (int arr[], int length)
{
    int result;
    if (length <= 1)
        result = false;
    else
    {
        int ratio = arr[1] / arr[0];
        result = isGeometricHelper(arr,1,length-1,ratio);
    }

    return result;
}







}
```

Question 10 [8 Marks]

In this question you are to write a function that examines a linked list of type `Node` variables, where the `Node` type has been defined as follows:

```
    typedef struct node
    {
        char        *studentName;
        int          grade;
        struct node *link;
    } Node;
```

Write a C function having prototype `double averageGrade (Node *head)` that returns the average of the `grade` elements in the linked list. The value `0.0` should be returned if the linked list is empty.

Solution:

```
double averageGrade (Node *head)
{
    double result = 0.0;
    int nodeCount = 0, gradeSum = 0;
    Node *current = head;

    while (current != NULL)
    {
        // process current node
        nodeCount = nodeCount + 1;
        gradeSum = gradeSum + current->grade;

        // prepare for next node
        current = current->link;
    }

    if (gradeSum > 0)
        result = (double) gradeSum / (double) nodeCount;

    return result;
}
```

Question 11 [8 Marks]

Assume that you have implemented two linked lists to maintain in memory information about the players of a soccer team. One list maintains information of the players currently playing, and the other list maintains information of the players currently sitting on the bench, waiting to be replaced (if the coach decides so). In both linked lists a player is represented by a node of type `Node`:

```
typedef struct node
{
  char        *familyName;
  char        *firstName;
  char         position;    // One of 'G', 'D', 'M', 'S'
  int          value;       // in thousands of dollars
  struct node *link;
} Node;
```

Write a complete C function named `replace`, the prototype of which is given below. The function receives three parameters: a pointer to the head of a linked list representing the players currently playing (called `headTeam`), a pointer to the head of a linked list representing the players waiting at the bench (called `headBench`), and a character representing the position of the player (one of `'G', 'D', 'M', 'S'`, indicating the position of a Goalkeeper, a Defender, a Midfielder, or a Striker respectively). Your function should return a pointer to the node in the bench list that consists the best replacement at this moment or `NULL` if there is no need to make a replacement. For a given position, a player with a higher value is preferred to a player with a lower value. You can assume that the values of any two players (either currently playing or at the bench) are different.

Solution:

```
Node * replace (Node *headTeam, Node *headBrench, char position)
{
    Node * bestOut = NULL;
    int bestInValue = 0;
    int bestOutValue = 0;

    // Find best player in this position that is currently playing
    Node * currentIn  = headTeam;
    while (currentIn != NULL)
    {
        if (currentIn -> position == position && currentIn -> value > bestInValue)
        {
            bestInValue = currentIn -> value;
        }
        currentIn = currentIn -> link;
    }

    // Find best player in this position that is currently not playing
    Node * currentOut  = headBench;
    while (currentOut != NULL)
    {
        if (currentOut -> position == position && currentOut -> value > bestOutValu
        {
            bestOutValue = currentOut -> value;
            bestOut = current;
        }
        currentOut = currentOut -> link;
    }

    // compare and return player or NULL
    if(bestOutValue > bestInValue)
        return bestOut;
    else
        return NULL;
}
```

This page has been left blank intentionally. You may use it for your answer to any of the questions in this examination.