UNIVERSITY OF TORONTO
FACULTY OF APPLIED SCIENCE AND ENGINEERING

APS 105 — Computer Fundamentals
Final Examination
December 9, 2011
9:30 a.m. – 12:00 p.m.

Examiners: J. Anderson, T. Fairgrieve, B. Li, G. Steffan, A. Veneris

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 are allowed to write functions that have multiple return statements.

The examination has 22 pages, including this one.

Full Name: _____

Student Number: _____ ECF Login: _____

MARKS

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

Question 1 [28 Marks]

1.1 [4 Marks] Write a single C statement that declares an `int` type variable named `rt`, and initializes it to be a random four-digit integer between 1000 and 9999 (inclusive). You may use library functions.

Solution:

```
int rt = (rand() % 9000) + 1000;
```

Marking Scheme:

```
+ 0.5 for int rt
+ 1 for using rand()
+ 1 for % 9000
+ 1 for +1000
+ 0.5 for using all of the parts together correctly
An answer of int rt = (rand() % 10000) gets 1.5
```

1.2 [4 Marks]
In the box below, place the output that would be printed by an execution of the following C statements:

```
int A[] = {3,1,4,9,7,2,5,8};
int *p = &(A[3]) - 2;
p += *p;
printf("%d\n",*p);
```

Solution:

Output: 4

Marking Scheme:

```
+ 4 marks for answering "4"
alternates ?
```

1.3 [4 Marks] Write a function named `isDivisible` that has two type `int` parameters named `m` and `n`. The `isDivisible` function returns the answer to the question of whether or not `m` is exactly divisible by `n`. The return type of `isDivisible` should be `bool`.
Solution:

```
bool isDivisible (int m, int n)
{
    return ( m%n == 0 );
}
```

Marking Scheme:

```
+ 0.5 mark for prototype with correct return type
+ 0.5 mark total for prototype with 2 correct parameter declarations
+ 1 mark for determining m%n
+ 1 mark for comparing m%n to 0
+ 1 mark for taking correct action based on result of m%n == 0
Give full marks to
 if ( m%n == 0 )
     return true;
 else
     return false;
```

1.4 [4 Marks] The following implementation of quicksort has errors in two statements. Fix the errors in the code so that the implementation of quicksort is correct. The parameter `values` is an `int` type array that must be sorted into ascending (nondescending) order. Place your description of the required changes below the function. When describing your changes to the function, use the numbers that appear to the left of each line in the function.

```
1 void quickSort (int values[], int low, int high)
2 {
3     int left = low;
4     int right = high;

5     if (left >= right)
6         return;

7     int pivot = values[left];

8     while (left < right)
9     {
10        while (values[right] >= pivot)
11            right--;

12        values[left] = values[right];

13        while (values[left] <= pivot)
14            left++;

15        values[right] = values[left];
16    }
17    values[right] = pivot;

18    quickSort(values, low, left - 1);
19    quickSort(values, left + 1, high);
20 }
```

State the line number(s) that need to changed and describe the change(s):

Solution:

The bugs are in line 10 and 13, which should be:

```
    while (values[right] >= pivot && left < right)
```

and

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

Marking Scheme:

```
+ 1 mark for knowing that there is an error in line 10
+ 1 mark for correct fix to line 10
+ 1 mark for knowing that there is an error in line 13
+ 1 mark for correct fix to line 13
```

1.5 [4 Marks] In the box below, place the output that would be printed by an execution of the following C program:

```
int magicNumber (int a[], int n)
{
   if (n == 0)
       return a[0];
   else
       return (a[n] + magicNumber(a,n-1));
}

int main (void)
{
    int i, b[11];

    for(i = 0; i <= 10; i++)
        b[i] = i;

    printf("Magic number is %d \n", magicNumber(b,5));
    return 0;
}
```

Solution:

Output: Magic number is 15

Marking Scheme:

```
+ 1 mark for printing "Magic number is" and a number
+ 3 marks for printing the correct number of 15
If value wrong, + 1 mark for printing the number as 21 or 10
```

1.6 [4 Marks] Write a **single** C statement to complete the recursive function below named `getLength`. The function should count and return the number of characters in the string that is received as a parameter. For example, the function call `getLength("aps105")` should return the `int` value 6.

You may **not** use any library functions in your solution. You **must** use recursion.
Solution:

```
int getLength (char *str)
{
    if (str[0] == '\0')
        return 0;
    else
        // WRITE YOUR SINGLE C STATEMENT HERE:

        return 1 + getLength(str+1);

}
```

Marking Scheme:

```
        + 1 mark a return statement
        + 1 mark for the addition "1 + "
        + 1 mark for calling getLength
        + 1 mark for correct argument to getLength

        If then break this up over more than one single statement,
        deduct 1 mark.
         NOTE:  getLength(&str[1]) is correct too
```

1.7 [4 Marks] Your task is to complete the function below so that it contains a **nonrecursive** implementation of the binary search algorithm. The parameter `values` is an array of `int` type variables. The items in the `values` array have been sorted into **descending** (nonascending) order. Parameter `n` is the number of elements in the `values` array. Parameter `item` is the item being searched for in the `values` array.

The function should return -1 if `item` is not found in the array. Otherwise, the function should return the index position within the array at which `item` is found.

Important: your function should assume that `values` is a sorted array in **descending** (nonascending) order.

Important: Your solution must **not** use recursion.
Solution:

```
int binSearch (int values[], int n, int item)
{
    int left = 0;
    int right = n-1;

    while (left <= right)
    {

      int middle = (left + right) / 2;
      if (item == values[middle])
        return middle;

      // WRITE YOUR CODE HERE:
      if (item < values[middle])
          left = middle + 1;
      else
          right = middle - 1;




    }
```

Marking Scheme:

```
        + 1 mark for an expression comparing item and values[middle]
           (could be < or > depending on how cases below written)
        + 1 mark for having two cases - one updates left, the other right
        + 1 mark for updating left correctly, given the direction
        + 1 mark for updating right correctly, given the direction
```

Question 2 [8 Marks]

Write a function named `rotateLeft`, the prototype of which is given below, that has three parameters: an `int` array named `A` of length `length`, an `int` named `length` and an `int` named `rotateAmount`. The `rotateLeft` function moves each of the elements in array `A` `rotateAmount` positions to the left.

For example, rotating the array `A[] = {0,1,2,3}` to the left by a `rotateAmount` of 2 would modify `A[]` to be `{2,3,0,1}`. Notice that the values `0` and `1` are rotated to the other end of the array. You may assume that the values of both `length` and `rotateAmount` are greater than zero. You may **not** call any library functions, including `malloc`.

Solution:

```
for (int i=0;i<rotateAmount;i++)
{
  int temp = A[0];
  for (int j=0;j<length;j++)
      A[j] = A[(j+1)%length];
  A[length-1] = temp;
}
```

Marking Scheme:

```
      + 2 marks for a proper rotate expression A[j] = A[(j+X)%length];
  (where X could be 1 or rotateAmount)
      + 2 mark for doubly nested loop
      + 2 mark for use of a temporary
      + 2 marks if it seems to work
```

Question 3 [8 Marks]

Write a C function named `stringToInt`, the prototype of which is given below, that has a single string parameter named `str`. The `stringToInt` function determines the numerical value of the integer that is stored in `str`. You may assume that the string only contains digits between `'0'` and `'9'` (plus the required terminating null character), and that it contains at least one digit.

For example, the function call `stringToInt("105")` must return the integer `105`.

Note: You may **not** use any library functions in your solution.
Solution:

```
int stringToInt (const char *s)
{
    int result = 0;
    int i = 0;
    do
    {
        result = result*10 + (s[i]-'0');
        i = i + 1;
    } while (s[i] != '\0');

    return result;
}
```

Marking Scheme:

```
   +1 declaring an accumulator variable and initializing to 0
   +2 accumulating the result in a loop, one digit at a time
   +1 knowing that * by 10 shifts numerical to left (pads result with 0)
   +2 knowing to subtract '0' to get int value of single char
   +1 knowing to stop the loop based on reaching a char of '\0'
   +1 returning result

Note: the following is correct too:
    int result = s[0]-'0';
    int i = 1;
    while (s[i] != '\0')
    {
        result = result*10 + (s[i]-'0');
        i = i + 1;
    }

    return result;

Also, the loop condition could be written as:
  '0' <= s[i] && s[i] <= '9'
```

Question 4 [8 Marks]

Write C code for a *recursive* function named `printReverse` that reads a sequence of positive integers and outputs the sequence in reverse order. The input sequence ends when the user enters the number 0.

Assume the user always gives valid input. Your `printReverse` function must be recursive, otherwise your solution will not receive any marks.

You may **not** declare any arrays in your solution.

Here is a sample output from an execution of the program:

```
Enter a positive integer or 0 to exit: 10<enter>
Enter a positive integer or 0 to exit: 5<enter>
Enter a positive integer or 0 to exit: 2<enter>
Enter a positive integer or 0 to exit: 1<enter>
Enter a positive integer or 0 to exit: 0<enter>

Reversed sequence:  0 1 2 5 10
```

Note: No newline character is printed after the last number in the reversed sequence.

Enter your `printReverse` function below.
Solution:

```
void printReverse (void)
{
   int i;
   printf("Enter a positive integer or 0 to exit: ");
   scanf("%d", &i);
   if (i != 0)
       printReverse();
   printf("%d ", i);
}
```

Marking Scheme:

```
0.5 mark for correct prompt
0.5 mark for correctly reading the input
1 mark for a call to printReverse somewhere in the function
1 mark for printing the input number somwhere in the function
1 mark for separating the input == / !=  0 case
4 marks for the correct ordering of the printReverse() and printf()
   function calls
Note: the following is correct too
if (i != 0)
{
    printReverse();
    printf("%d ", i);
}
else
    printf("%d ", i);
```

Question 5 [8 Marks]

Write a C function named `readlines` that has a single type `int` parameter named `n`. The `readlines` function reads `n` lines of input from the user and stores the lines in a dynamically allocated 2-d array of type `char`. The row with index `[i-1]` in the array should be set to hold the i^{th} input line and must be just large enough to hold the line of input as a string. The `readlines` function is to return the dynamically allocated 2-d array to the caller of the function.

You may assume that each line of text contains no more than 1023 characters. You may use the library functions `malloc`, `strlen`, `strcpy` and `gets`, but may not use any other functions.

Solution:

```
char **readlines (int n)
{
    char line[1024];
    char **result = (char **)malloc(n*sizeof(char *));
    int i;
    for (i=0; i<n; i++)
    {
        gets(line);
        result[i] = (char *)malloc((strlen(line)+1)*sizeof(char));
        strcpy(result[i],line);
    }

    return result;
}
```

Marking Scheme:

```
NOTE: there was a small error in the question statement - they are
allowed to use sizeof() - do not deduct marks if they express the
right idea without use of sizeof()

NOTE: they are *not* required to cast to value returned by malloc()

+1 mark for correct return type declaration  (char ** )
+0.5 mark for correct parameter type declaration  (int n)
+0.5 mark for declaring an array for reading input into
+0.5 mark for declaring a char ** variable, named say result
+1 mark for initializing result to point to an allocated array of char *
+1 mark for a loop that executes n times
+1 mark for reading a line of input correctly - they can't use scanf
+1 mark for initializing result[i] correctly by allocating enough space
    for input line + trailing null character
+1 mark for string copying input line to the array they have created
+0.5 mark for returning result
```

Question 6 [8 Marks]

6.1 [5 Marks] Write a C function having prototype
`counting (int input[], int n, int count[], int k)`. Each of the `n` elements in the array `input` is an integer in the range of `0` to `k`, inclusive, and the size of the array `count` is `k + 1`. After a call to the `counting()` function, each element `count[i]` will contain the number of elements in the array `input` that are less than or equal to the index `i`. For example, if the array `input` contains `{2, 5, 3, 0, 2, 3, 0, 3}` and if `k=5`, then the array `count` will be set to `{2, 2, 4, 7, 7, 8}`. There are 4 elements in the array `input` that are less than or equal to 2, and so `count[2]` has the value `4`.

Note: The `counting` function should initialize each of the elements in the array `count` to 0.
Solution:

```
void counting (int input[], int n, int count[], int k)
{
  int i, j;

  for (i = 0; i <= k; i++)
    count[i] = 0;

  for (j = 0; j < n; j++)
    count[input[j]]++;

  for (i = 1; i <= k; i++)
    count[i] += count[i - 1];
}
```

6.2 [3 Marks] Complete the following C function with prototype
`void countingSort(int input[], int n, int output[], int k)`
that uses the `counting()` function to sort an array `input` into ascending (nondescending) order. The idea is to first call `counting()`, and then use the information in the `count` array to place each element in `input` directly into its final position in the `output` array. Again, it is assumed that each of the `n` elements in the array `input` is an integer in the range of `0` to `k`, inclusive.

Hint: The information in the `count` array can be used to help find the position of array elements in the sorted array. For example, the value at position `i` of the `count` array, `count[i]`, indicates the starting position for all elements greater than `i` in the sorted array.

As an example, the following `main` program will print `0 0 2 2 3 3 3 5` as its output:

```
int main (void)
{
    int input[] = {2, 5, 3, 0, 2, 3, 0, 3};
    int output[8] = {0};

    countingSort(input, 8, output, 5);
    for (int i = 0; i < 8; i++)
        printf("%d ", output[i]);
    return 0;
}
```

Hint: You only need to fill in two lines of code to complete the solution.
Solution:

```
void countingSort (int input[], int n, int output[], int k)
{
  int *count = (int *) malloc((k + 1) * sizeof(int));
  counting (input, n, count, k);

  for (int i = n - 1; i >= 0; i--)
  {
     output[count[input[i]] - 1] = input[i];
     count[input[i]]--;
  }

  free(count);
}
```

Question 7 [8 Marks]

Write a function named `findStringWithinString`, the prototype of which is given below, that takes strings `s1` and `s2` as parameters, and returns a pointer to the start of the first occurrence of the string `s2` within the string `s1`. The function should return `NULL` if the string `s2` is not found within the string `s1`. You may assume that `s1` and `s2` are strings that each contain at least one character in addition to their terminating null character.

Note: You may **not** use any functions from the `string.h` library in your solution.
Solution1: pointer version:

```
while (*s1)
{
  char *p = s1;
  char *q = s2;
  while (*p == *q)
  {
     q++;
     p++;
     if (!*q)
        return s1;
  }
    s1++;
}
return NULL;
```

Solution2: array version:

```
int x_outer = 0;
while (s1[x_outer])
  {
    int x_inner = x_outer;
    int y = 0;
    while (s1[x_inner] == s2[y])
    {
         x_inner++;
         y++;
         if (!s2[y])
            return &(s1[x_outer]);
    }
    x_outer++;
  }
return NULL;
```

Marking Scheme:

```
 2 marks for any doubly-nested loop attempt
 2 marks if outer-loop traverses s1 properly
 2 marks if inner-loop traverses s2 properly
 2 marks if separate index/pointer for s1 in the inner loop than
      for the outer loop
2 marks if it seems to work
```

Question 8 [8 Marks]

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

```
typedef struct node
{
    int info;
    struct node *link;
} Node;
```

Write a C function named `printDuplicates` that receives a pointer to the first node of a linked list of `Node`s as a parameter. The function should find and print the duplicate integers in the linked list. For example, if the linked list contains the integers: 1, 3, 3, 6, 7, 4, 6, then the `printDuplicates` function should print:

```
3
6
```

Note: In your solution, you may assume that a given integer occurs at *most* twice in the linked list.
Solution:

```
void printDuplicates(Node *head)
{
    Node *curr = head;

    while (curr != NULL)
    {
        Node *curr2 = curr->link;
        while (curr2 != NULL)
        {
            if (curr->info == curr2->info)
                printf("%d\n", curr->info);
            curr2 = curr2->link;
        }
        curr = curr->link;
    }
}
```

Marking Scheme:

Question 9 [8 Marks]

In this question, you are to write a function that manipulates a linked list with elements of type `Node`, where the `Node` type is as defined in Question 8.

Write a C function named `reorder` that receives a pointer to the first node of a linked list of `Node`s as a parameter, and returns a pointer to the first node of a *reordered* linked list that contains the *same* nodes as the input list. In the reordered list, all of the nodes that contain an *even* integer must appear before the nodes that contain an *odd* integer.

For example, if the original list contains nodes with the following integers in the given order 9,8,7,6,6,5, a reordered list that meets the requirements is 6,6,8,5,7,9. The even nodes themselves can be in any order. The odd nodes themselves can be in any order. However, the even nodes must appear before the odd nodes. This means that another reordered list that meets the requirements is 8,6,6,9,7,5.

You may assume that the original linked list contains only positive integers. Your function is not allowed to create any new nodes and must only manipulate the nodes in the list given to the function. Solutions that create new nodes using `malloc` will receive 0 marks. **Solution:**

```
Node *reorder(Node *head)
{
    if (head == NULL || head->link == NULL)
        return head; //handle the cases of empty list, or 1-element list

    Node *newHead = NULL;
    Node *newTail = NULL;

    while (head)
    {
        Node *next = head->link;
        if (head->info % 2 == 0)
        {
          // put at front
          head->link = newHead;
          newHead = head;
          if (!newTail)
              newTail = newHead;
        }
        else
        {
            // put at tail
            if (!newTail)
              newHead = newTail = head;
            else
            {
                newTail->link = head;
                newTail = head;
```

```
                }
            }
            head = next;
        }
        newTail->link = NULL;

        return newHead;
    }
```

Marking Scheme:

Question 10 [8 Marks]

Write a recursive C function named `permuteHelper` that prints all possible permutations of the characters in a given string. The `permuteHelper` function is called by the given `permute` function. When the function call `permute("one")` is executed, the output will be the following:

```
one oen noe neo eno eon
```

Note: No newline character is printed after the last permutation.

You may assume the string given to `permuteHelper` contains distinct characters (that is, no character appears more than once).

Hint: Consider an algorithm that works as follows: For each character in the string, `permuteHelper()` exchanges it with the string's first character and then it generates the permutations recursively for the characters that remain.

```
void permuteHelper (char *str, int pos, int length);

void permute (char *str)
{
    permuteHelper(str, 0, strlen(str));
}

void permuteHelper (char *str, int pos, int length)
{









}
```

Solution:

```
void permute(char *s, int pos, int Length);

int main() {

   char *str = (char *) malloc(sizeof(char)*(26+1));

   printf("Enter string to permute: ");
   scanf("%s", str);
   permute(str, 0, (int) strlen(str));
   return;
}


void permute(char *str, int pos, int Length)
{
   char *scopy, temp;
   int i;

   if(pos < Length)
      for(i = pos; i < Length; i++)
      {
          scopy = malloc(Length+1);
          strcpy(scopy, s);
          temp = scopy[i];
          scopy[i] = scopy[pos];
          scopy[pos] = temp;
          permute(scopy, pos + 1, Length);
          free(scopy);
     }
     else
          printf("%s\n", s);
}
```

Marking Scheme:

2 marks given for incorrect coding but correct overall algorithm design
1 mark for malloc as procedure calls itself recursively
1 mark for freeing the space

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