Faculty of Applied Science and Engineering, University of Toronto
CSC181: Introduction to Computer Programming, Fall 2000

First Term Test
October 18, 2000

Duration: 80 minutes
Aids allowed: None

Do NOT turn this page until you have received the signal to start. (In the meantime, please fill out the identification section below, and read the following instructions carefully.)

This term test consists of 9 questions on 8 pages (including this one). When you receive the signal to start, please make sure that your copy of the test is complete.

Answer each question directly on the test paper, in the space provided, and use the reverse side of the pages for rough work. (If you need more space for one of your solutions, use the reverse side of the page and indicate clearly which part of your work should be marked.)

Be aware that concise, well thought-out answers will be rewarded over long rambling ones. Also, unreadable answers will be given zero (0) so write legibly.

Unless otherwise stated, programming questions will be marked for correctness primarily, with efficiency and style as secondary considerations. Comments are not required, but they should accompany any long or tricky bits of code to aid the marker's understanding.

DON'T PANIC. KEEP COOL.

Family name: _____

Given name: _____

Student number: __ | __ | __ | __ | __ | __ | __ | __ | __

1:	____	/ 2
2:	____	/ 6
3:	____	/ 8
4:	____	/ 14
5:	____	/ 5
6:	____	/ 10
7:	____	/ 10
8:	____	/ 15
9:	____	/ 10
Total:	____	/ 80

Question 1. [2 marks]

Write your student number LEGIBLY on the top of every page of this term test.

Question 2. [2 x 3 = 6 marks]

For each of the terms below, concisely define the term and explain its significance (using at least one example).

Dangling pointer:

A dangling pointer refers to memory that has been deallocated. Here is an example of a dangling pointer:

```
int *p = (int*)malloc(sizeof(int));
*p = 2;
free(p); /* p is now dangling */
printf("%d", *p); /* attempt to dereference dangling pointer */
```

Programmers must be vigilant about the possibility of dereferencing a dangling pointer, as this can produce unexpected (and perhaps chaotic) results. Dangling pointers can be hard to spot, since several pointers may point to the same (deallocated) block of memory.

Table lookup:

Table lookup allows us to carry out a mapping function between some *argument* (sometimes called *key*), and some fixed corresponding *value*. This is sometimes implemented using an array:

```
const char* romanNumerals[] = {"", "I", "II", "II"};
printf("%s", romanNumerals[1]); /* prints out "I", the Roman numeral for 1 */
```

Table lookup is a useful technique because it allows us to avoid writing a complicated set of if or switch statements.

Question 3. [4 x 2 = 8 marks]

Consider the following declarations, which compile without error:

```
int a = 1, b = 0, c = -1, d = 2;
```

What is the value of each of the logical expressions listed below?

`a && c && d`	FALSE	**TRUE**
`a >= d && a`	**FALSE**	TRUE
`a || d && b`	FALSE	**TRUE**
`a && !c || !a && !b`	**FALSE**	TRUE

Question 4. [7 x 2 = 14 marks]

Consider the following declarations, which compile without error:

```
int a = 2, b = 3;
int c[] = {1, 2, 3};
int d[] = {4, 5, 6};
int *p = &b;
int *q, *r;
int *s[5];
```

Assume that the assignments listed below directly follow these declarations, and that none of the assignments have an effect on any of the other assignments. For each of these assignments:

- If the assignment is illegal, circle "ILLEGAL" and state why it is illegal.
- If the assignment is legal, circle "LEGAL", give an example of how to get a non-pointer value using the lvalue of the assignment, and state what that non-pointer value is.

`*q = *p;`	**ILLEGAL**	LEGAL	**q is not initialized**
`r = c;`	ILLEGAL	**LEGAL**	***r or r[0] will give us 1, the first element of c**
`d = c;`	**ILLEGAL**	LEGAL	**d is a pointer constant, so it's not a valid lvalue**
`&b = &a;`	**ILLEGAL**	LEGAL	**&b is an address, so it's not a valid lvalue**
`**s = b;`	**ILLEGAL**	LEGAL	***s or s[0] is not initialized**
`s[2] = p;`	ILLEGAL	**LEGAL**	***s[2] will give us 3 (same result as *p)**
`*(s+1) = c;`	ILLEGAL	**LEGAL**	****(s+1) or *s[1] will give us 1 (same result as *c or c[0])**

Question 5. [5 marks]

Complete the function `reverse` below, which takes as arguments an array of integers, `a`, and the size of the array, `n`, and rearranges the integers of `a` in reverse order.

```
void reverse(int a[], int n) {
      int i, j;
      for (i = 0, j = n-1; i < j; i++, j--) {
            int temp = a[j];
            a[j] = a[i];
            a[i] = temp;
      }
}
```

Question 6. [1 + 2 + 7 = 10 marks]

a) [1 mark]

Vancouver native Carrie-Anne Moss was one of the actresses in the 1999 action film *The Matrix*. Name one of the actresses in *The Matrix*.

Carrie-Anne Moss.

b) [2 marks]

A square matrix A with dimension n (that is, an nxn matrix), is *symmetric* if for 0 <= i < n and 0 <= j < n, $A_{i,j} = A_{j,i}$.

Suppose we want to write a C function `isSymmetric`, which would take as arguments a square array `A` (representing the square matrix) and an integer `n` (representing the dimension of the matrix), and return true if `A` is symmetric, false otherwise. We might be tempted to use the following prototype:

```
int isSymmetric(int A[][], int n);
```

Why won't this prototype compile?

The second and subsequent dimensions of multi-dimensional array parameters must be specified.

c) [7 marks]

Consider the following code fragments, which belong to a program that compiles without error:

```
/* Represents a square matrix. */
typedef struct squarematrix {
      int n; /* The dimension of this matrix. */
      /* Rest of implementation omitted. */
} SquareMatrix;

/* Returns the element A_i,j. */
int get(const SquareMatrix A, const int i, const int j) {
      /* Implementation omitted. */
}
```

Using those fragments, complete the function `isSymmetric` below, which takes as arguments a square matrix `A` and returns true if `A` is symmetric, false otherwise.

```
int isSymmetric(const SquareMatrix A) {
      int i, j;
      for (i = 0; i != A.n; i++) {
            for (j = i+1; j < A.n; j++) {
                  if (get(A, i, j) != get(A, j, i)) {
                        return 0;
                  }
            }
      }
      return 1;
}
```

Question 7. [3 + 7 = 10 marks]

a) [3 marks]

Consider the following function, which compiles without error:

```
int g(const int n) {
      int i, j;
      int x = 0;
      for (i = 1, j = n; i < n; i++, j--) {
            x += i+j;
      }
      return x/2;
}
```

Write a mathematical expression for what this function calculates and returns, in terms of its parameter n.

((n-1)(n+1))/2

b) [7 marks]

Consider the following program, which compiles without error:

```
int h1(const int k, const int m) { return k % m; }

int h2(const int k, const int m) { return 1 + (k % (m-2)); }

int h(const int k, const int i, const int m) {
      return (h1(k,m) + i*h2(k,m)) % m;
}

int main(void) {
      int a[10] = {0};
      int b[5] = {32, 15, 20, 5, 11};
      int j;

      for (j = 0; j != 5; j++) {
            int i;
            for (i = 0; i != 10; i++) {
                  int x = h(b[j], i, 10);
                  if (a[x] == 0) {
                        a[x] = b[j];
                        break;
                  }
            }
      }

      return 0;
}
```

Using the table below, show what are the contents of the array a just before the program exits.

20	**5**	**32**	**0**	**0**	**15**	**0**	**0**	**0**	**11**
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Question 8. [15 marks]

A set is a collection of distinguishable objects (that is, it contains no duplicate elements). For the purposes of this question, consider an integer set, for which the following related operations can be performed:

- **init():** Returns an empty integer set.
- **add(s, i):** Adds integer `i` to set `s`.
- **contains(s, i):** Returns true if set `s` contains integer `i`, false otherwise.

Implement an integer set in C; that is, define a suitable struct (with accompanying typedef) to represent the set and implement functions for the operations described above. While doing this, you must take the following three points into consideration:

- The operations have been described abstractly so that you can use either structs or pointers to structs as arguments and return values. For full marks, you should use pointers to structs. If you do not know how to do this, you may use structs instead for partial credit.
- The set should be able to accommodate any number of integers. To facilitate this, your struct should have a field for a pointer to integer--which you can manipulate to point to any integer array space of your choosing--rather than a field for an array of fixed size. If you do not know how to do this, you may use a fixed-size array implementation instead for partial credit.
- If you do use pointers, you will need to sensibly use the memory management function `malloc` and handle null pointer situations. Do not worry about avoiding memory leaks (using `free`).

You may define any constants or helper functions you feel are necessary. Clearly state any assumptions you feel are necessary for your code to work properly.

If you find yourself needing more space for your solution, use the next page.

Solution without pointers

```
#define NUM_ELEMENTS (100)

typedef struct set { int size; int elements[NUM_ELEMENTS]; } Set;

Set setInit(void) {
      Set result;
      result.size = 0;
      return result;
}

Set setAdd(Set s, const int e) {
      if (!setContains(s, e)) s.elements[s.size++] = e;
      return s;
}

int setContains(const Set s, const int e) {
      int i;
      for (i = 0; i != s.size; i++)
            if (s.elements[i] == e)
                  return 1;
      return 0;
}
```

Solution with pointers

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

#define CAPACITY_INCREMENT (10)

struct set {
      int size;
      int capacity;
      int *elements;
};
typedef struct set Set;
typedef Set* SetPtr;

SetPtr setInit(void) {
      SetPtr result;
      result = (SetPtr)malloc(sizeof(Set));
      assert(result != NULL);
      result->size = 0;
      result->capacity = CAPACITY_INCREMENT;
      result->elements = (int*)malloc(result->capacity * sizeof(int));
      assert(result->elements != NULL);
      return result;
}

void setAdd(SetPtr sptr, const int e) {
      assert(sptr != NULL);
      if (!setContains(sptr, e)) {
            if (sptr->size == sptr->capacity) {
                  int *tempElements;
                  int i;
                  sptr->capacity += CAPACITY_INCREMENT;
                  tempElements = (int*)malloc(sptr->capacity * sizeof(int));
                  assert(tempElements != NULL);
                  for (i = 0; i != sptr->size; i++) {
                        tempElements[i] = sptr->elements[i];
                  }
                  free(sptr->elements);
                  sptr->elements = tempElements;
            }
            sptr->elements[sptr->size] = e;
            sptr->size++;
      }
}

int setContains(SetPtr sptr, const int e) {
      int i;
      assert(sptr != NULL);
      for (i = 0; i != sptr->size; i++) {
            if (sptr->elements[i] == e) {
                  return 1;
            }
      }
      return 0;
}
```

Question 9. [1 + 9 = 10 marks]

a) [1 mark]

What is your favourite Scrabble tile? (If it's the blank tile, write down "Blank"--do NOT leave this question blank. If you don't have a favourite tile, then just write down: "Don't have one".)

Any tile is fine.

b) [9 marks]

Complete the function `bringQXToFront` below, which rearranges the elements of a character array `t` so that all the q's and x's (just the lowercase ones) are at the front, and the leftover characters follow.

Specifically, if at the start of the function `t[0..n-1]` contains characters in some arbitrary sequence S, then just before the function returns, `t[0..q-1]` should contain all the q's from S, `t[q..x-1]` should contain all the x's from S, and `t[x..n-1]` should contain the remaning characters from S (which can be arranged in an arbitrary order).

(**Note:** If x is an array, then `x[a..b]` refers to the range of x from a to b inclusive, that is, the subarray of x consisting of elements `x[a], ..., x[b]`. If `a == b`, then `x[a..b]` contains one element `x[a]`. If `a == b+1`, then `x[a..b]` does not contain any elements.)

```
void bringQXToFront(char t[], const int n, int *qp, int *xp) {
      int q, x;

      int j = n;
      q = 0;
      x = 0;

      while (x != j) {
            if (t[x] == 'x') {
                  x++;
            }
            else if (t[x] == 'q') {
                  t[x] = t[q];
                  t[q] = 'q';
                  x++;
                  q++;
            }
            else {
                  char temp;
                  j--;
                  temp = t[j];
                  t[j] = t[x];
                  t[x] = temp;
            }
      }

      *qp = q;
      *xp = x;
}
```

[END OF TEST]