National University of Singapore

School of Computing

EXAMINATION for Semester 2 AY2011/2012

CS1010E — Programming Methodology

April/May 2012 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This paper contains 19 questions in two parts A and B.
   Part A comprises fifteen (15) questions, and Part B four (4) questions.
   In total, there are twelve (12) printed pages, including this page.

2. The maximum possible mark is 40.

3. Answer *all* questions in the *space provided* in this booklet.

4. This is an *open book* examination.

5. Write your Matriculation Number below.

MATRICULATION NO: | | | | | | | | | |

This portion is for examiner's use only

Question	Marks	Remarks
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

Question	Marks	Remarks
16		
17		
18		
19		

Part A (Each question is worth 1 mark. Total: **15** marks)

For this part A, you are required to write down the output of the given code fragment, or write down that there is an *error*.
An error includes compiler, runtime or infinite loop error.
Your answers are to be brief and to the point.

Do not include any commentary in the answers.

Assume that the relevant `#include` pre-processor statements have been included in the code fragment as appropriate.

Q1.
```
int i, j, count = 0;
for (i = 3; i >= 1; i--)
    for (j = 1; j <= i; j++) count++;
       printf("%d ", count++);
```

Q2.
```
int i = 0, count = 0;
do {
     if (count >= 4) continue;
     count += ++i;
} while (i <= 2);
printf("%d", count);
```

Q3.
```
int i = 0, count = 0;
do {
     if (count >= 4) break;
     count += ++i;
} while (i <= 2);
printf("%d", count);
```

Q4.

```
int c[] = {2,3,4,6};
int j, *p = c, *q = c;
for(j = 0; j < 3; j++) {
    printf(" %d ",*c);
    ++q;
}
for(j = 0; j < 3; j++){
    printf(" %d ",*p);
    ++p;
}
```

Q5.

```
void t(int), u(int), v(int);

int main() {
   t(5);
   return 0;
}

void t(int x) {
  printf("%d ", x--);
  v(x--);
  u(x);
}

void u(int x) {
  printf("%d ", x--);
  v(x--);
}

void v(int x) {
  printf("%d ", x--);
}
```

Q6.
```
void f(int);

int main() {
   f(4); f(5);
   return 0;
}

void f(int x) {
    static int y = 3;
    printf("%d ", x + y++);
}
```

Q7.
```
char a = 'B', b = 3;
printf("%c %c\n", 'a' + b, a + b);
```

Q8.
```
void f(int, int *);

int main() {
    int count = 0;
    f(4, &count);
    printf("%d", count);
    return 0;
}

void f(int x, int *count)
{
    if (x > 0) f(x - 1, count);
    *count++;
}
```

Q9.
```
char a[5], b[5];
strcpy(b, "abc");
strcpy(a, b+1);
printf("%s %s", a, b);
```

Q10.

```
char str[] = {'A', 'B', 'C', '\0', 'D', 'E', 'F', 'G', 'H', '\0', 'I', 'J'};
     printf("%s %s %s", str, str + 3, str + 5);
```

Q11.

```
int a[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
int *b;
    b = *(a + 1);
    printf("%d %d", *(b + 1), **(a + 2));
```

Q12.

```
void f(int), g(int);

int main() {
    f(3);
    return 0;
}

void f(int x) {
    if (x > 0) {
        printf("%d ", x);
        g(x);
    }
}

void g(int x) {
    if (x > 0) {
        f(x - 1);
        printf("%d ", x);
    }
}
```

Q13.

```
char s[] = {'a','b','c','\n','c','\0'};
char *p, *str, *str1;
p = &s[3];
str = p;
str1 = s;
printf("%d",++*p + ++*str1-32);
```

Q14.

```
struct mystruct {  int a[5]; };

void f(struct mystruct);
void g(struct mystruct *);

int main() {
    struct mystruct p1, p2;
    p1.a[0] = p2.a[0] = 0;
    f(p1); g(&p2);
    printf("%d %d\n", p1.a[0], p2.a[0]);
}

void f(struct mystruct p) { p.a[0] = 1; }
void g(struct mystruct *p) { p->a[0] = 1; }
```

Q15.

```
struct mystruct {
    int value;
    struct mystruct *ptr;
};

int dosomething(struct mystruct *);

int main()
{
    int i, j, k;
    struct mystruct p1, p2;

    p1.value = 22; p1.ptr = &p2;
    p2.value = 33; p2.ptr = &p1;
    i = dosomething(&p2);
    j = (p1.ptr == NULL) ? 0 : 1;
    k = (p2.ptr == NULL) ? 0 : 1;
    printf("%d %d %d", i, j, k);
}

int dosomething(struct mystruct *p) {
    int count = 0;
    struct mystruct *q;
    while (p != NULL) {
        count += p->value;
        q = p->ptr;
        p->ptr = NULL;
        p = q;
    }
    return count;
}
```

Part B (Total: **25** marks)

In this part, the required code fragment is only a few lines.
Comments are not necessary; if you wish to provide comments they must be restricted to a few words.
The total marks for each question is stated above each question.

16. **(5 marks)**

We say that a sequence s_1 of characters $c_1, c_2, \cdots, c_n$ is a *subsequence* of another sequence s_2 of characters if each character c_i in s_1 appears in some position p_i in s_2, and: $p_1 < p_2 < \cdots < p_n$.

For example, suppose s_2 is:

'a' 'b' 'c' 'd' 'f' 'd' 'g'

Then the sequence 'a' 'd' 'd' is a subsequence of s_2.
However, the sequence 'a' 'c' 'c' is not a subsequence of s_2.

Write a function which has two arguments which are strings.
It returns 1 if the the first string is a subsequence of the second string, and 0 otherwise.

Q17. **(5 marks)**

A simplified *Roulette table* comprises 35 squares, representing *numbers* from 1 to 35, and two other squares, representing *even* and *odd* numbers. A *Roulette wheel* comprises 37 numbers: -1 and 0 through 35.

(a) Design a data structure to represent a Roulette table. Each cell of the the data structure will contain a number representing the number of *bets* on that cell. Assume that each bet is for one unit of money.

(b) Write a function that plays *one round.*
This means to use the `rand()` function to "spin" the Roulette wheel and obtain a random value *result* in the range $-1 \leq result \leq 35$.
This function then returns the *outcome* of this round.
(Intuitively, this outcome represents the money the bank makes in one round.) If *result* is -1 or 0, then the outcome is simply the sum of all bets on the table. Otherwise, the outcome is defined to the be the sum of the *individual outcomes* of the indivdual squares, defined as follows.

Suppose the content of a square is *bets.*

If this square represents a number n (1-35), then the outcome is $-35 * bets$ if $result = n$, and *bets* otherwise.

If this square represents *odd*, then the outcome is $-bets$ if *result* is a (positive) odd number, and *bets* otherwise.

If this square represents *even*, then the outcome is $-bets$ if *result* is a (positive) even number, and *bets* otherwise.

Q18. **(5 marks)**

Write a function `calendar` which has three arguments. The first is a two-dimensional array whose purpose is to represent a calendar month. There should be 6 rows (each representing a week) and 7 columns (each representing a day, the first of which is Monday). The content of each array element is a day-date, that is, a number from 1 to 31.

The second and third arguments of your function are two integers, representing a particular month and year respectively.

For example, if the call `calendar(c, 4, 2012)` is made, then your function will produce the follwing array. (The headers "mon", "tue", ... are for illustration only, and are not part of the array.) Note that for the parts of the array corresponding to days *before* and *after* the month in question, you may simply enter the value 0.

mon	tue	wed	thu	fri	sat	sun
0	0	0	0	0	0	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	0	0	0	0	0	0

You may assume the existence of a function `int dayofweek(int month, int year)` which returns the day of the week (a number from 1 thru 7) on which the first day of a given month and year falls. For example, calling `dayofweek(4, 2012)` will return the value 7 (representing Sunday).

You may also assume that every month has exactly 30 days.

Q19. **(5 + 5 = 10 marks)**

You are given a two-dimensional array `maze` whose contents are either 0 or 1. The idea is that this array represents a maze, where a 0 denotes a *vacant* spot, and 1 denotes a *blocked* spot. Assume the array has the dimension $N \times N$ where N is a defined symbolic constant. The start spot is the first cell (or top left cell) of the array, and you may assume that this spot is vacant. The destination spot is the last cell (or bottom right cell) of the array.

Write a function which navigates this maze.
It should return 1 if there is a path, and 0 otherwise.
You do not need to print the path itself.

Do two versions of the function.

Version 1: movement is only possible from one vacant spot A to another vacant spot B if A is directly above B, or A is directly to the left of B.
(That is, move down or right only.)

Version 2: movement is possible from one vacant spot A to another vacant spot B if A is one is directly to the left or right of B, or A is directly on the top or the bottom of B.
(That is, move up, down, left or right.)

Note that there can be more than one distinct path.
Your function needs only to determine if one exists.

Hint: In both versions, write the function recursively.

Version 1:

Version 2:

END of PAPER