UNIVERSITY OF DUBLIN

TRINITY COLLEGE

Faculty of Science

school of mathematics

JF Maths/TP/TSM **Trinity Term 2018**

Mathematics 1266: C programming

Thursday, May 3 Goldsmith Hall? 09:30 — 11:30

Prof. Colm Ó Dúnlaing

Attempt 3 questions
Show all work.
Remember to fold down and glue the flap on every answer booklet.

1. (a) Convert -3141 to a short integer, giving the answer in hex, little endian.

   Answer

   bb f3 little endian

   (b) Given

```
char hello[] = "hello";
short *x = (short *) hello;
```

   Convert x to decimal. Note: the ascii codes for $a \ldots z$ are $97 \ldots 122$.

   Answer

```
string hello hex  68 65 6c 6c 6f 00

(*x) is the short int represented by the first
four hex digits: 68 65, little endian.
Big-endian would be: 65 68 (all in hex).

This is a positive number (high-order bit is zero).
Its value is its face value,
6 * 16^3 + 5 * 16^2 + 6 * 16 + 8
which is 25960.
```

```
A slightly different calculation:

(6*16 + 5)*16^2 + (6*16 + 8)
6*16 + 5 = 101, the ascii value of 'e',
and 6*16+8 = 104, the ascii value of 'h':

101*256 + 104 = 25960
```

(c) Given

```
int a[10];
double b[3][3];
char * c = (char*) a;
```

Assume that a begins at address 1000 and b follows a immediately. The address of $b[1][2]$ coincides with the address of $a[i]$ for some i. Calculate i.

Answer

```
Address of b[1][2] = 1000 + 40 + 3 * 8 + 2 * 8 = 1120
Correction: the value is 1080, not 1120.

The address of a[i] is 1000 + 4 * i.
Therefore 4*i = 80 and i = 20.
```

---

2. (a) Write a ***recursive*** routine `void print_binary(int n)` which prints n in binary, at 'face value.'. For example, with `n==5`, the output should be `101`. (You may assume that $n > 0$, and it is unnecessary to print a newline.)

Answer

```
#include <stdio.h>

void print_binary ( int n )
{
  if ( n > 0 )
  {
    int x = n/2, y = n%2;
    print_binary (x);
    printf("%d",y);
  }
```

```
}

main()
{
   print_binary(5); printf("\nGoodbye\n");
}
```

(b) Write an efficient ***recursive*** function

`double power ( int n, double a );`

which returns a^n. You may assume $n \geq 0$. Using recursion rather similar to that in `print_binary()`, the function uses relatively few multiplications.

Answer

```
#include <stdio.h>

double power ( int n, double a )
        // returns the n-th power of a, given n>=0
{
  if ( n == 0 )
    return 1;
  else
  {
    int y = n%2;
    int x = n/2;
    double c = power ( n/2, a );
    if ( y == 0 )
      return c*c;
    else
      return c*c*a;
  }
}
main()
{
  double b = power (5, 3 );
  printf("3^5 is %f\n", b);
}
```

3. (a) Carefully simulate the following program.

```
#include <stdio.h>
int xxx ( int m )
{ if ( m <= 10 )
    return m;
  else
  { int x = m%10, y = m/10;
    return x - xxx ( y );
  }
}
main()
{ int m = 123;
  int z = xxx ( m );
  printf("m is %d, m-xxx(m) is %d\n", m, m-z);
}
```

For your information: z is congruent to m mod 11.

Answer

```
xxx, m == 123
    xxx, m == 12
        xxx, m == 1
                    xxx() returns 1 at line 1
    xxx(12) returns 2 - xxx(1) = 1 at line 3
xxx(123) returns 3 - xxx(12) = 2 at line 3
m is 123, m-xxx(m) is 121
-------------------------
```

(b) Write a complete C program which reads lines from input using `fgets()`, stores copies of these lines in an array `char * string[1000]`, and prints them in reverse order, and separated by blank lines. For example example,

```
          |     should produce
a quick   |    fox
brown     |
fox       |    brown
          |
          |    a quick
```

You can assume that at most 1000 lines will be read.

Answer

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


main()
{
  char * string[1000];
  int n = 0;
  char buffer[200];
  while ( fgets ( buffer, 200, stdin ) != NULL )
  {
    string[n] = malloc ( strlen ( buffer ) + 1 );
    snprintf( string[n], strlen(buffer)+1, "%s", buffer);
    ++n;
  }

  int i;
  for (i=n-1; i >= 0; --i)
  {
    if ( i < n-1 )
      printf("\n");
    printf("%s", string[i]);
  }
}
```

---

4. (a) Write a routine `void transpose( double a[2][2], double b[2][2] )` which copies to `b` the transpose of `a`. You may assume that `a` and `b` are different arrays.

   (b) Use it in a careful simulation of the following (which violates the assumption)

```
main()
{ double a[2][2] = {{1,2},{3,4}};
   transpose (a,a);
   printf("%f %f\n%f %f\n", a[0][0], a[0][1], a[1][0], a[1][1]);
}
```

Answer

```
#include <stdio.h>

void transpose ( double a[2][2], double b[2][2] )
```

```
{
  b[0][0] = a[0][0]; b[1][1] = a[1][1];
  b[0][1] = a[1][0]; b[1][0] = a[0][1];
}

int main() etcetera
gcc...
a.out  (the simulation is not shown here, but this is
        the result).
1.000000 3.000000
3.000000 4.000000
```

(c) Write a routine `void invert( double a[2][2], double b[2][2] )` which stores the inverse of `a` in `b`. You may assume that `a` is invertible and `b` is a different array. Recall

$$\begin{bmatrix} u & v \\ w & x \end{bmatrix}^{-1} = \frac{1}{ux - vw} \begin{bmatrix} x & -v \\ -w & u \end{bmatrix}.$$

Answer

```
void invert ( double a[2][2], double b[2][2] )
{
  double det = a[0][0]*a[1][1] - a[1][0]*a[0][1];
  b[0][0] = a[1][1] / det;
  b[1][1] = a[0][0] / det;
  b[0][1] = - a[0][1] / det;
  b[1][0] = - a[1][0] / det;
}

This has been tested as follows:
A:
      1      2
      3      4
Inverse:
     -2      1
    1.5   -0.5
Product:
      1      0
      0      1
```
