Student Name:

Student Number:

Midterm Examination
308-206B 2001: Introduction to Software Systems

Examiner: Prof. Hans Vangheluwe Thursday, February 15th, 2001

Invigilators: Jan Prawdzik, Atir Syed, Salman Zafar, Viqas Zafar 16:00 – 17:30

INSTRUCTIONS:

1. Answer all questions directly on the examination paper.
2. Use the back of the last page as scrap (it will be ignored during grading).
3. No aids of whatever type are permitted.
4. The exam has 11 questions on 6 pages (including cover page).
5. Attempt all questions: partial marks are given for incomplete but correct answers.
6. Numbers between brackets [] denote the weight of each question. The total is 50 points.

Good luck !

(1) [2]

What gets printed if the following code fragment is run ?

```
int i=5;
int j=6;
if (i = j)
 printf("First\n");
else
 printf("Second\n");
```

```
Solution:
 (i = j) assigns the value of j to i.

 The result of the assignment  i = j
 is the value of the Left Hand Side (LHS).
 (note: this is NOT i == j, the equality test)

 In a C condition, 0 stands for FALSE, any
 non-zero condition stands for TRUE.

 The result is that "First\n" is printed
 (newline at the end).
```

(2) [4]

Given the declaration

```
char s[] = "Good\tluck\\%%\n";
```

- What is the length of string `s` (as returned by `strlen(s)`)?

```
   Solution:
    '\t' is a single character
    '\\' is a single character
         (printed as a single \, the escape character)
    '%'  is a single character
         %% is interpreted by printf() as a single %
 as part of a string, each % counts as a single character however
    '\n' is a single character
    Thus, the lengtsh of s is 13
```

- What is the return value of `strlen(s)` after the next statement has been executed ?

```
s[strlen(s)/2] = '\0';
```

```
Solution:
 strlen(s)/2 = 13/2
 This is an integer division with result 6
 s[6] = '\0' gives a string of length 6
 (s[0] ... s[5])
```

(3) [5]

Assume the variable `name` has been declared as a character vector of appropriate size. Assume a valid C string is present in `name`. Write a code fragment to reverse the string (of any length) *in-place*.

```
Solution (one of many possible):
 #include <stdio.h>
 #include <string.h>

 int  name_size;
 char tmp_store;
 int  index;

 name_size=strlen(name);

 /* loop over half the string
  * note how this works correctly for:
  * 1) empty strings
  * 2) even-length strings
  * 3) odd-length strings (name_size/2 is integer division)
  */
 for (index=0; index < name_size/2; index++)
 {
   tmp_store=name[index];
   name[index]=name[(name_size-1)-index];
   name[(name_size-1)-index]=tmp_store;
   /* (name_size-1) is used due to C array index numbering from 0 */
 }
```

(4) [5]

Write a *recursive* function which prints the reverse of a string. The string `my_text` is a global variable.

```
Solution:
 #include <stdio.h>

 void print_reverse(int position)
 {
  if (my_text[position] != '\0')
  {
    print_reverse(position+1);
    printf("%c", my_text[position]);
  }
```

```
    /* otherwise, end of string, recursion stops */
   }

   /* call this as follows: */
   print_reverse(0);
   printf("\n");
```

(5) [3]

In `header.h`:

```
  #define MAX 10
  extern int size;
```

How does one protect `header.h` from multiple inclusion (`#include "header.h"`) ?

```
  Solution:
   By enclosing it in a

    #ifndef HEADER_H_INCLUDED
    #define HEADER_H_INCLUDED
     ...
    #endif

   pair. Note how the actual name (HEADER_H_INCLUDED) does not matter
   as long as it does not clash with standard defines.
```

(6) [10]

Given the following declarations/initializations

```
  int i=10, j=1, k=5;
  double x = 30.0, y = 40.0;
  char c = ’A’;
  char message[] = "Hello world\n";
```

Write the *value* and *type* of each of the expressions below. Each expression is independent of the others.

1. `i*(k-j) + sizeof(long int);`

```
     Solution:
      value: 44
      type: int
```

2. `x/i;`

```
Solution:
 value: 3.0
 type: double
```

3. `i % k;`

```
Solution:
 value: 0
 type: int
```

4. `(’F’ - c)/ 2;`

```
Solution:
 value: 2 (namely 5/2, ’F’-’A’ == 5)
 type: int
```

5. `(’e’ - ’b’)/ 2.0;`

```
Solution:
 value: 1.5
 type: double
```

6. `++i*j++;`

```
Solution:
 value: 11 (j is only incremented after the expression is evaluated)
 type: int
```

7. `message[2] += 2;`

```
Solution:
 value: ’n’ (the result of an assignment is the LHS value)
 type: char (may be promoted to int)
```

8. `j && k;`

```
Solution:
 value: 1 (both j and k are != 0, so TRUE && TRUE is TRUE,
        represented as 1 in C)
 type: int
```

9. `message[2] - message[1];`

```
Solution:
 value: 7
 type: int
```

10. `message[0] - message[strlen(message)] - c;`

```
Solution:
 value: 7 ('H' - 'A', message[strlen(message)] is '\0' == 0)
 type: int
```

(7) [4]

Write a function `to_upper` as well as its prototype/signature (separately) which converts any lower case ASCII character into its upper case counterpart. The converted (upper case) character is returned by the function. Return `'\0'` if the function argument is not a lower case ASCII character.

```
Solution:
 char to_upper(char c);

 char
 to_upper(char c)
 {
  /* check if this is a lower case letter */
  if ( (c < 'a') || (c > 'z') )
   return '\0';
  else
   return (c + 'A' - 'a');
 }
```

(8) [5]

Given a 3D matrix of integers (such as `int mat[ISIZE][JSIZE][KSIZE]`). Write a code fragment which determines `int flat_layout_pos`, the position of `mat[i][j][k]`. The indices `i, j, k` are given and valid ($0 \leq i < ISIZE, \ldots$) integers. `flat_layout_pos` should be the linear position in memory when *column-major ordering* (as in Fortran) is used.

```
Solution:
 flat_layout_pos = ISIZE*(JSIZE*k + j) + i;
 /*  i index varies fastest
  *  j index varies less fast
  *  k index varies slowest
  *
  * this is the reverse of the C memory layout of arrays:
  *  k index varies fastest, ..., i index varies slowest
  */
```

(9) [3]

What is the output of the following code fragment ?

```
char selector = ’x’;
int out = 10;
while (1)
{
 ++selector;
 switch (selector)
 {
  case ’x’:
   printf("%o ", out);
  case ’y’:
   printf("%d ", out);
  default:
   printf("default\n");
  case ’z’:
   printf("%x ", out);
 }
 out++;
 if (selector == ’z’)
  break;
}
printf("\n%c\n", selector);
```

```
Solution (note the absence of breaks -> fall-through):

 10 default
 a b
 z
```

(10) [6]

1. All variables in C are implicitly of what type ?

   ```
   Solution:
    int
   ```

2. Does this code (inside a `main()`)

   ```
   char input;
   if (input == "q")
    return (0);
   ```

   (a) Compile ? If not, why ?

```
Solution:
 A compiler will detect this
 as the type of input (char) is not the same
 as the type of "q" (char *)
```

(b) Run ? If not, why ?

```
Solution:
 If per chance this got past a dumb compiler,
 results would be unpredictable.
```

3. What is the largest number of typed in characters that can be read from `stdin` with `fgets(buffer, 6, stdin)` if `buffer` has been appropriately declared ?

```
Solution:
 5 characters as fgets() reserves one byte for the end-of-string
 '\0' character. If the user also pressed newline after less than
 5 characters, '\n'  will be the last character in the buffer.
```

(11) [3]

What will the following program output ?

```
#include <stdio.h>

#define CAT1(X, Y, Z) X Y , 1
#define CAT2(X, Y, Z) X#Y , 2
#define CAT3(X, Y, Z) X##Y , 3

int main(void)
{
   char AABB[] = "CCDD";
   printf("%s %d\n", CAT1("AA", "BB", "CC"));
   printf("%s %d\n", CAT2("AA", BB, "CC"));
   printf("%s %d\n", CAT3(AA, BB, CC));

   return (0);
}
```

```
Solution:
 CAT1("AA", "BB", "CC") is expanded by the C preprocessor to
      "AA" "BB", 1
 C concatenates two adjacent strings "AA" "BB" into one "AABB"
 "AABB", 1 are now the two last arguments of printf()

 CAT2("AA", BB, "CC") is expanded by the C preprocessor to
      "AA" "BB", 2
```

```
This, as #Y means "take the Y argument and put it between quotes".
As before, C concatenates two adjacent strings "AA" "BB" into one "AABB"
"AABB", 2 are now the two last arguments of printf()

CAT3(AA, BB, CC) is expanded by the C preprocessor to
     AABB, 3
This, as X##Y means "take X and Y literally, and concatenate them"
The result is not a string, but a variable name.
AABB, 3 are now the two last arguments of printf()
Note how the variable AABB is declared and has the value "CCDD"

Putting all this together, the following gets printed:

AABB 1
AABB 2
CCDD 3
```