Princeton University
COS 217: Introduction to Programming Systems
Fall 2000
Midterm 1 Answers

Question 1

```
Assuming an 8-bit word size...

 75 (base 10) = 01001011 (base 2)
 91 (base 10) = 01011011 (base 2)
-91 (base 10) = 10100101 (base 2)

  +75 (base 10) = 01001011 (base 2)
+ -91 (base 10) = 10100101 (base 2)
-----------------------------------
  -16 (base 10) = 11110000 (base 2)

The carry into the sign bit was 0.  The carry out
of the sign bit was 0.  Thus overflow did not occur.

Converted to 16 bits:
1111111111110000
```

Question 2

```
Note:  The given statement is missing a parenthesis.  It should be:
key = key >> (bitoff - ((bitoff >> 3) << 3));

An answer is:
key >>= (bitoff & 0x07);
```

Question 3

```
(a) expands to:
    while (((c=getchar()) >= 'A' && (c=getchar()) <= 'Z'))

(b) expands to:
    while ((c=getchar()) != EOF && ((c) >= 'A' && (c) <= 'Z'))

(b) is better because:
    (1) (a) calls getchar twice; that is probably not the programmer's
        intention.
    (2) If getchar returns EOF, then the test for an upper case
        character is meaningless.  (b) incorporates that logic, but (a)
        does not.
        Also, omitting a test for EOF could cause trouble subsequently
        in the program.
```

Question 4

```
1 : you
2 : have
3 : a
4 : good
5 : break

Explanation:

array[0][2]
= "you"

*(array[1])
= *(array[1] + 0)
= array[1][0]
= "have"

*(*(array + 1) + 1)
= *(array[1] + 1)
= array[1][1]
= "a"

*(q + 2)
= *(*(array + 1) + 2)
= *(array[1] + 2)
= array[1][2]
= "good"

*(*(p + 2) + 1)
= *(*(array + 2) + 1)
= *(array[2] + 1)
= array[2][1]
= "break"
```

Question 5

(a) In the line

```
    b[i] = (CHARBUF)malloc(sizeof(char) * (sizeof(a)+1));
```

the expression sizeof(a) is always 4. That is incorrect. The line should be:

```
    b[i] = (CHARBUF)malloc(sizeof(char) * (strlen(a)+1));
```

(b) Each call to printMatrix creates garbage. That is, printMatrix contains a memory leak.

Question 6

(a) c is a static integer initialized to 1.

(b) i is a constant integer.

(c) pc is a pointer to a constant integer.

(d) cp is a constant pointer to an integer.

(e) a is an array of 10 pointers to integers.

(f) ip is a pointer to a pointer to an integer.

(g) f is a function that takes a void pointer and returns an integer pointer.

(h) f is a pointer to a function that takes a void pointer and returns an integer.

(i)
```
struct person
{
   struct {char *fname, char *lname} *name;
   int dob[3];
   struct person *parent;
};
```

(j)
```
struct person *employees[10];
```

(k)
```
struct person *employees;
int numemps;
...
employees = (struct person*)calloc(numemps, sizeof(struct person));
```

(l)
```
void *p(struct person*);
```

(m)
```
void map(int *(*fp)());
```

(n)
```
printf("%p", (*fp)());
```

Question 7

I've assumed that "depth first traversal" is the same as "inorder traversal." In that case, the visiting order is (1) left subtree, (2) current node, (3) right subtree.

Here's the entire program (in one file):

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

typedef struct tree
{
   int val;
   struct tree *left, *right;
} tree;

typedef tree *ptree;

int size (ptree p);
int * tree2array(ptree p);
ptree insert (ptree p, int nval);

int main()
{
   int * a; /* stores flattened tree */
   int i;
   ptree p = NULL;
   /* insert values into tree */
   p = insert (p, 3);
   p = insert (p, 4);
   p = insert (p, 2);
   p = insert (p, 1);
   p = insert (p, 5);
   a = tree2array(p);
   for (i = 0; i< size(p); i++)
      printf("%d\n",*a++);
   return 0;
}

int size(struct tree *psTree)
{
   if (psTree == NULL)
      return 0;
   else
      return size(psTree->left) + size(psTree->right) + 1;
}

void tree2arrayhelp(struct tree *psTree, int *piArray, int *piIndex)
{
   if (psTree != NULL)
   {
      tree2arrayhelp(psTree->left, piArray, piIndex);
      piArray[*piIndex] = psTree->val;
      ++(*piIndex);
      tree2arrayhelp(psTree->right, piArray, piIndex);
```

```
   }
}

int *tree2array(struct tree *psTree)
{
   int *piArray;
   int iIndex = 0;
   piArray = (int*)calloc(size(psTree), sizeof(int));
   tree2arrayhelp(psTree, piArray, &iIndex);
   return piArray;
}

struct tree *insert(struct tree *psTree, int iVal)
{
   if (psTree == NULL)
   {
      struct tree *psNew = (struct tree*)malloc(sizeof(struct tree));
      psNew->val = iVal;
      psNew->left = NULL;
      psNew->right = NULL;
      return psNew;
   }
   else if (iVal < psTree->val)
   {
      psTree->left = insert(psTree->left, iVal);
      return psTree;
   }
   else
   {
      psTree->right = insert(psTree->right, iVal);
      return psTree;
   }
}
```