COS126: General Computer Science **Spring 1996**

Second Midterm Exam: Solutions

Date: Monday, April 15, 1996

The following was the distribution of scores.

```
between 0  and 9 :
between 10 and 19: *
between 20 and 29: *******
between 30 and 39: ************
between 40 and 49: *****************
between 50 and 59: *******************
between 60 and 69: ********************
between 70 and 79: *********************
between 80 and 89: ***************************
between 90 and 99: *********
```

Problem 1, 30 points:

1. (*15 points*)

   Identify all syntax errors in the following program. Copy the offending lines in your answerbook and briefly explain for each why it is an error. (You will be penalized if you identify non-errors as errors.)

```c
#include <stdio.h>
#include <stdlib.h>
struct CS126 {int C; struct CS126 *rest};
struct course {struct CS126 good;};
struct course test(struct CS126 y)
{
 struct course *new;
 y.rest = y.C;
 new =malloc(sizeof(course));
 new.good =y;
 return new;
}
void main()
{
 struct CS126 x;
 test(&x);
}
```

There are 6 syntax errors in the given program.

- *Line 3:* There is a semicolon missing[1]. The line should read:
  `struct CS126 {int C; struct CS126 *rest;};`

[1] The instructors didn't mean to be sneaky; this error was unintended!

- *Line 8:* `y.rest` is a `struct CS126` pointer, while `y.C` is an `int`.
- *Line 9:* The line should read: `new=malloc(sizeof(`*struct* `course))`
- *Line 10:* `new` is a pointer to struct course, and not a variable of type struct course. So we cannot use the ".".
- *Line 11:* The function is trying to return `new`, whose type is "pointer to `struct course`". But according to its declaration, `test` must return a value of type `struct course`.
- *Line 16:* The function `test` takes an argument of type `struct CS126`. In this case we are trying to call it with an argument whose type is `struct CS126 *`.

Every syntax error counts for 2.5 points. 1-1.5 points were taken off for wrong explanation.

2. (*15 points*)

What does the following program print? Briefly explain your answer.

```
#include <stdio.h>
struct good{int x; int y;};
void f(int *x)
{
 if (*x >0)
  {
   printf("%d\n", *x);
   *x = *x -1;
   f(x);
  }

}
void main()
{ struct good my= {4, 5};
  f(&(my.y));
  printf("%d, %d\n", my.x, my.y);
}
```

The given program has the following output:

```
5
4
3
2
1
4, 0
```

First notice that the value of `my.x` never changes. When we call `f` for the first time, we pass to it the *address* of `my.y`. Each time `f` calls itself recursively (i.e., as `f(x)` in line 9), it gives the recursive call the same pointer (`x`) that was passed to it as an argument. This means that in all recursive calls, `x` will contain the address of `my.y` (i.e., `& my.y`) all the time, so the statement in line 8 affects `my.y` each time it's executed. That's why `my.y` ends up having the value 0.

- Wrong final value for `my.y` value costs 7 points if it's completely wrong (many students said it was 5 or 4) and 3 points if it was 1 instead of 0.

- Wrong `my.x` value costs 4 points.
- Wrong or non existent explanation costs 1 to 3 points.
- Claim that the program will not work because of compilation errors, but if it worked would do... costs 4 points.

NOTE: Many students included partial work instead of explanations, and credit was given whenever the work depicted the explanation given above.

Problem 2, 35 points:

Let (x_1, y_1) and (x_2, y_2) be two *pairs* of integers. We say that (x_1, y_1) is *less than* (x_2, y_2) if either $x_1 < x_2$ or

$$x_1 = x_2 \text{ and } y_1 < y_2.$$

For example $(2, 4)$ is less than $(4, 2)$ since $2 < 4$. Similarly, $(2, 4)$ is less than $(2, 5)$ since $2 = 2$ and $4 < 5$.

Now suppose you are given a program that begins with the following lines.

```
#include <stdio.h>
#define N 10000
struct pair { int x; int y;};
struct pair A[N];
```

Write a function `sortA` with the declaration

```
void sortA();
```

that sorts the array A. At the end, the pair $A[i]$ should become less than the pair $A[i+1]$ for $i = 0, 1, \ldots, N-2$. (Note that A is initialized in some other function, and you don't need to worry about that.) You may use any sorting technique of your choice.

The following program does the task. Note its similarity to the bubblesort routine presented in lecture 13 (however, that program as it appeared in the handout was buggy; corrections were made in lecture).

```
int compare(struct pair p1, struct pair p2)
{
  if (p1.x > p2.x)
   return 1;
  else if ((p1.x == p2.x) && (p1.y > p2.y))
   return 1;
  else
   return 0;
}

void swap(struct pair *pa, struct pair *pb)
{
  struct pair  temp;

  temp = *pa;
  *pa = *pb;
```

```
  *pb = temp;
}

void sortA()
{
  int   i, j;

  for (i = 0; i < N - 1; i ++)
    for (j = N - 1; j > 0; j --)
      if (compare(A[j-1], A[j]) > 0)
        swap(&A[j-1], &A[j]);
}
```

- If your bubble sort only contains one pass (missing outer loop or inner loop) then there's a 20-point penalty.
- If in bubble sort you make `i` and `j` go the same direction there's 6-point penalty.
- If you write wrong termination conditions in the loops (such as confusing N or N - 1, ¡ or ¡=) then there's 2-point penalty for each error.
- If you write swap() using call-by-value instead of call-by-reference, there's a 5-point penalty.
- If you declare temp as int in swap() but use it as `struct pair variable`, there's a 2-point penalty.
- If you write a wrong compare(), there's a 5-point penalty.
- Other syntax errors (missing semicolon, missing open brace of function etc.) cost you 1 point each.

Problem 3, 35 points:

Suppose the `node` and `list` structures are defined as in the lecture notes.

```
struct node {int value; struct node *next;};
struct list {struct node *head; struct node *tail;};
```

Write a function `everyother` with the declaration

```
struct list *everyother(struct list *li);
```

that takes a pointer to a list as its argument, creates a new list and copies every second element of the existing list into this new list. The value returned by the function should be a pointer to a newly created list.

Thus for example if `li` is a pointer to the list $\{5, 21, 32, 65, 2, 8, 15\}$ then the list created by the function would be $\{5, 32, 2, 15\}$. However, the function should not destroy the first list while creating the second list.

Here's a sample solution.

```
struct list *everyother(struct list *li)
{
  int include_it = 1;
  struct node *node_ptr;
  struct list *new_list = malloc(sizeof(struct list));

  new_list->head = new_list->tail = NULL;

  for(node_ptr = li->head; node_ptr != NULL; node_ptr = node_ptr->next) {
    if (include_it == 1) {
      add_node(new_list, node_ptr->value);
      include_it = 0;
    }
    else
      include_it = 1;
  }

  return new_list;
}


void add_node(struct list *li, int value)
{
  struct node *new_node = malloc(sizeof(struct node));

  new_node->value = value;
  new_node->next = NULL;

  if (li->head == NULL) {
    li->head = new_node;
    li->tail = new_node;
  }
  else {
    li->tail->next = new_node;
    li->tail = new_node;
  }
}
```

Some of the more common errors:

- using a local `struct list` variable instead of `mallocing` one: -2
- dereferencing a `->next->next` pointer when `->next` could be `NULL`: -5
- not creating a new list, just moving pointers around: -10
- not returning a value: -5
- returning a (slightly) wrong value: -2

- general pointer errors: -2..-5
- creating the new list the wrong way round: -5

For people who were *way* off the mark with their answer, I assigned some points (between 1 and 5) for the effort.