CSE 110 Final Exam: Dan Welch

M–J. Dominus

21 September, 1992

Due: 2:30 PM, 3 October 1992

Welcome to the exam.

There are 125 points on this exam; answer questions that total at least 100 points. Make sure to make it clear which parts of the things you write are the ones I should grade. Be sure to label each answer with a number of the problem it's the answer to.

The exam is take-home, open-book, and open-notes. You may consult any printed or electronic reference medium, but you may not consult other people and in particular you may not consult your classmates. So the rules are the same as for an in-class open-book test.

1 Data Structures, Dynamic Memory, and Design

Suppose we have a large, complicated program which manages many long linked lists. The program has many routines which manage these lists in various ways. For example, here is one of these functions: it creates a new node and links it to the end of a given list:

```
struct listnode {
  char *data;
  struct listnode *next;
} ;

struct listnode *make_new_listnode(void);

int append(char *new_data, struct listnode *list_head)
{
  struct listnode *tail, *newnode;

  newnode = make_new_listnode();
  if (newnode == NULL) return -1;

  for (tail=list_head; tail->next != NULL; tail=tail->next)
    /* nothing */ ;

  tail->next = newnode;
  newnode->next = NULL;
  newnode->data = new_data;
```

```
  return 0;
}
```

After the program is written, the customer complains that it is too slow. The programming team profiles the program, and discovers that a substantial amount of time is being spent in the `for` loop in this function, and in loops like it, whose only purpose is to find out where the end of the list is. The chief programmer wants to eliminate these loops. He realizes that if information about which node in each list is last is recorded somewhere, then function that need to know thatwill be able to just look it up instead of having to execute a time-consuming loop every time, and the program will run faster.

The problem he encounters is that recording where the end of each list is is not so easy, because the end sometimes moves when we append new nodes to the list (as in this function) or when we delete old nodes from it, so that the records of which nodes are last become out of date quickly. The chief programmer adopts the following compromise solution: He alters the definition of `struct listnode` so that it has a new member, `later`:

```
struct listnode {
  char *data;
  struct listnode *next; /* Pointer to next node in list */
  struct listnode *later; /* Pointer to last (?) node in list */
} ;
```

The idea is that the `later` member in each node of a list will contain a pointer which points to a node that is near the end of that list, and perhaps even points to the last node in the list. `later` members may be `NULL`, which means that no information about the location of the end of the list is available in that node. (See figure.)

1.1 20 points

Functions do not have to use the `later` member to find the end of a list, but they will probably be faster if they do. Speed up the `append` function by using this new information.

1.2 5 Points

Every time the `append` function is called to append a new node to a list, the `later` pointers get more and more out of date, in the sense that they point to nodes in the list that are farther and farther from the end. Ideally, `append` should update these pointers so that they continue to point to the end node, or at least some node that is close to the end.

A beginning programmer presents this version of `append`, which keeps the `later` points up to date:

```
int append(char *new_data, struct listnode *list_head)
{
  struct listnode *tail, *newnode;

  newnode = make_new_listnode();
  if (newnode == NULL) return -1;

  /* Your fast code from problem 1.1 goes here */

  for (p=list_head; p->next; p=p->next)
    p->later = newnode ;        /* Update later pointers */

  tail->next = newnode;
  newnode->next = NULL;
  newnode->data = new_data;

  return 0;
}
```

Discuss this approach.

1.3 20 Points

Write a replacement for `append` that updates the `later` pointers, but which is better than the solution of section 1.2.

1.4 15 Points

Corey Liss, a programmer parallel to you (his boss is your boss), discovers that the chief made a big blunder. Corey was assigned to write a function `delete`, which deletes a node from a linked list. He can write it, but it has to take about three times as long to run as it did before the `struct listnodes` had `later` members. Why is that?

1.5 20 Points

Propose an alternative to the chief programmer's solution, one which is better than the one already described.

1.6 20 Points

Write the code for `append` or for `delete`, using your solution of section 1.5. (To do this, you need to have a solution to problem 1.5. You can supply your own, or you can bag out and open the sealed envelope. To claim credit for section 1.5, you have to give the envelope back still sealed.)

2 Miscellany

2.1 5 Points

Abscissa Eckert-Mauchly, a beginning programmer who knows too many things that are none of her business, is writing a function to allocate a certain structure:

```
struct listnode {
  int province_code;
  int beheadment_count;
  int rioting_index;
  struct listnode *next; /* Pointer to next node in list */
  struct listnode *prev; /* Pointer to previous node in list */
} ;
```

She says, "The structure has three int members, of two bytes each, and two pointers, which are four bytes each, for a total of fourteen bytes." Then she writes

```
newnode = malloc(14);
```

What should she have written instead, and why?

2.2 5 Points

After seeing your solution to the previous question, Abscissa disagrees. "My way is faster that yours," she says, "because the program doesn't have to stop and figure out the value of your argument to `malloc` every time the function is called; it gets the `14` right away without having to compute anything extra."

Is this a legitimate complaint? If it is, whose way is better? If not, why not?

2.3 10 Points

The *Fibonacci function* is defined this way:

$$f(n) = \begin{cases} 0 & \text{if } x = 0 \\ 1 & \text{if } x = 1 \\ f(x-1) + f(x-2) & \text{otherwise} \end{cases}$$

Thus, $f(2) = 1$, $f(3) = 2$, $f(4) = 3$, $f(5) = 5$, $f(6) = 8$, $f(7) = 13$, and soforth. Write a recursive function, `fibo`, which computes the Fibonacci function of its argument.

2.4 10 Points

A graduate student here recently asked this question:

"Consider the following simple program:

```
#include <stdio.h>
int main(void)
{
  long int my_array[3];
  printf("sizeof(my_array) is %d\n", sizeof(my_array));
  return 0;
}
```

"This program returns 12, which is absolutely correct. I was surprised by this, however, because the C books all repeat (like some sort of mantra) that the name of an array is just a pointer to its first element, so I thought it would return `sizeof(long int *)`, or 4. Why exactly can C discriminate between a pointer and an array in this case?"

Answer the question.