CS 61CL (Clancy/Culler) **Solutions and grading standards for exam 1**
Fall 2008

154 students took the exam. The average score was 24.5; the median was 26.5. Scores ranged from 5 to 30. There were 117 scores between 23 and 30, 28 between 15.5 and 22.5, 6 between 8 and 15, and 3 between 5 and 7.5. (Were you to receive grades of 23 out of 30 on your two in-class exams and 46 out of 60 on the final exam, plus good grades on homework and lab, you would receive an A–; similarly, a test grade of 16 may be projected to a B–.)

There were two versions of the test. (The version indicator appears at the bottom of the first page.)

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your lab t.a. or to Mike Clancy. We will regrade the entire exam.

Problem 0 (2 points)

You lost 1 point on this problem if you did any one of the following:

- you earned some credit on a problem and did not put your login name on the page,
- you did not adequately identify your lab section, or
- you failed to put the names of your neighbors on the exam.

The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we want to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned. Finally, we occasionally need to verify where students were sitting in the classroom while the exam was being administered.

Problem 1 (4 points)

In version A, the decimal value to be represented was 244; in version B, it was 253. Here were the answers.

part	*version A*	*version B*
a	244 = 128 + 64 + 32 + 16 + 4 = 11110100 base 2	253 = 128 + 64 + 32 + 16 + 8 + 4 + 1 = 11111101 base 2
b	Bit 7 is on, so it's a negative number. To find out which, we compute its negative by complementing the bits and adding 1: –n = 00001011 + 1 = 00001100 = 12, so n = -12	The same procedure results in n = –3. You could also get this by noting that 253 + n ought to equal 0 mod 256.
c	244 = 15 * 16 + 4 = F4 base 16.	253 = 15 * 16 + 13 = FD base 16.
d	244 = 243 + 1 = 100001 base 3	253 = 243 + 9 + 1 = 100101 base 3.

Each part was worth 1 point; ½ point was deducted on each part where you didn't show work. (That was the most common source of errors.)

Problem 2 (4 points)

You were to implement a function named strchr that, given arguments char *s and char c, returns a pointer to the first occurrence of c in the string pointed to by s. If c does not occur in the string, strchr returns NULL. The terminating null character is considered part of the string; therefore if c is '\0', strchr locates the terminating '\0'. You were told at the exam not to use other functions declared in string.h. Here are two solutions.

```
char *strchr (char *s, char c) {
   char *p = s;
   while (*p) {
      if (*p == c) return p;
      p++;
   }
   if (*p == c) return p;
   return NULL;
}
```

```
char *strchr (char *s, char c) {
   char *p;                    // could just use s
   for (p=s; *p; p++)  if (*p == c) return p;
   if (*p == c) return p;
   return NULL;
}
```

The 4 points for this problem were divided as follows: 2 points for loop control; 1 point for comparison; and 1 point for value returned. This usually worked out to –1 point per error. Some common errors were the following:

- returning NULL instead of a pointer to the terminating '\0' (1-point deduction);
- returning a malloc'd copy of the desired value (1-point deduction);
- used a needless malloc that didn't interfere with the returned value (½-point deduction).

Problem 3 (6 points)

Each part of this problem involved completing the replace function described below. Given an array of strings named table as argument along with an index k into the table and a new string s, replace essentially performs the assignment

```
table[k] = a copy of s;
```

Version B of this problem differed only in the parameter sequence (the second and third parameters were exchanged). Presented below are answers for version A.

Part a was to complete the replace function using array notation, i.e. square brackets, where possible.

```
void replace ( char *table [ ], int k, char s [ ] ) {
   table[k] = (char *) malloc (sizeof(char) * (strlen(s)+1));
   strcpy (table[k], s);
}
```

The reason that char table[ ] [ ] isn't allowed in the function header is that the declaration of any n-dimensional array must include the sizes of the last n–1 dimensions so that the algorithm to access elements can be determined. (We didn't expect you to know this.)

Part b was to complete the replace function, *without using* any array notation (square brackets).

```
void replace (char **table, int k, char *s) {
   *(table+k) = (char *) malloc (sizeof(char) * (strlen(s)+1));
   strcpy (*(table+k), s);
}
```

The two parts were each worth 3 points. A bug that occurred in both solutions earned deductions in both parts. Each bug received a 1-point deduction. Common errors included the following:

- k in the wrong place, e.g. *table + k
- off by one in the call to malloc
- use of sizeof instead of strlen in the call to malloc
- lost pointer to malloc'd memory
- didn't set *(table+k) to a string

Each of these lost 1 point.

Problem 4 (7 points)

The versions differed only in the names of the struct fields, and in the substitution of max for min in part b. We present here the solutions to version A.

In part a, you were to provide the type declaration for a struct node that contains an int named val and pointer named ptr to a value of type struct node. Here's the declaration.

```
struct node {
   int val;
   struct node * ptr;
};
```

This was worth 1 point, with no partial credit except that no deduction was made for a missing semicolon at the end. Almost everyone got it right.

Part b was to write a function min that returns the minimum value in its argument list. You were allowed to assume that the argument list was not null and noncircular, and that the last node in the list stored a null ptr pointer. Here's an iterative solution:

```
int min (struct node *p) {
   int rtn;
   if  (p->ptr == NULL) {     /* 1-element list */
      return p->val;
   }
   for (rtn = p->val; p; p = p->ptr) {
      if (p->val < rtn) rtn = p->val;
   }
   return rtn;
}
```

Here's a recursive solution.

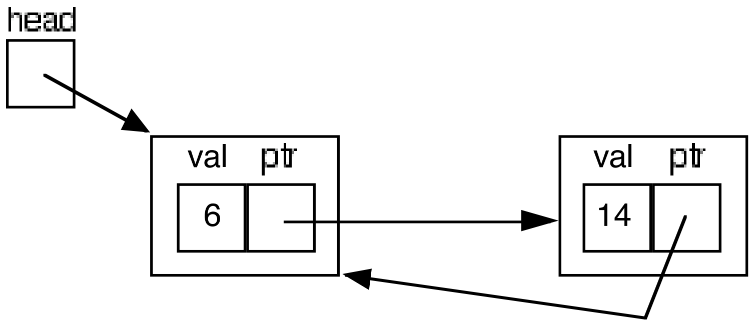
```
int min (struct node *p) {
   return helper (p->ptr, p->val);
}

int helper (struct node *p, int soFar) {
   if (p == NULL) {
      return soFar;
   } else if (p->val < soFar) {
      return helper (p->ptr, p->val);
   } else {
      return helper (p->ptr, soFar);
   }
}
```

This part was worth 3 points: 1 for base case, 1 for loop control, and 1 for comparison and everything else. Common errors included the following:

- assuming that the values in the list were all positive or all negative (1-point deduction);
- overlooking the last element in the list, i.e. `while (p->ptr != NULL)` … (½-point deduction);
- using `p++` as a "following" pointer (1-point deduction).

In part c, you were to use your declaration from part a to provide a program segment that dynamically allocates the structure represented in the diagram below.



Here's the code.

```
struct node *p1;
struct node *head;
p1 = (struct node *) malloc (sizeof (struct node));
head = (struct node *) malloc (sizeof (struct node));
p1->ptr = head;
p1->val = 14;
head->ptr = p1;
head->val = 6;
```

This part was worth 2 points: 1 point for the calls to malloc and ½ point each for scalar initializations and for declarations. Over-mallocing lost 1 point, as did replacing the value just returned by malloc (thus orphaning the malloc'd storage). Incorrect linking lost ½ or 1 point, depending on the number of incorrect links.

Finally, you were to explain in part d what would happen if head in the structure in part c was passed as argument to the function you wrote for part b. An infinite loop would result, since none of the ptr fields are null. This was worth 1 point; there was no partial credit. Almost everyone got it right.

Problem 5 (7 points)

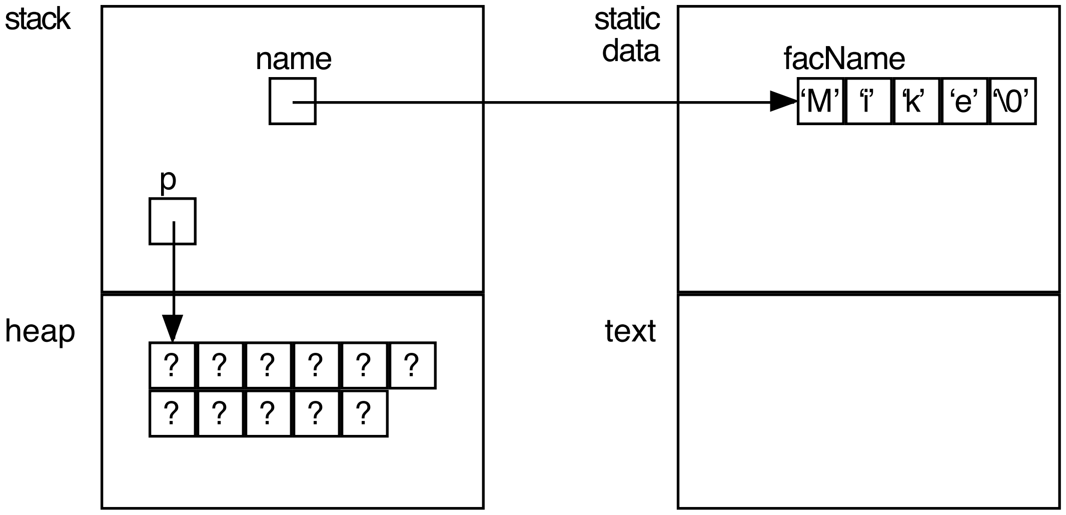
Both parts of this problem involved the following code. The versions differed in the sequence of regions in the memory diagram in part a.

```
char facName [ ] = "Mike";

char *response (char *name) {
   int k, helloLen;
   char *p;
   helloLen = strlen ("Hello, ");
   p = (char *) malloc (sizeof(char) * (helloLen + strlen (name)));
   p = "Hello, ";
   for (k=0; k<=strlen (name); k++) {
      p[k+helloLen] = name[k];
   }
}

int main ( ) {
   printf ("%s\n", response (facName));
   return 0;
}
```

In part a, you were to indicate (using box-and-pointer diagrams) the location and contents of the variables facName, name, and p. Your answer for p was to reflect its state immediately after the call to malloc. If the contents of a variable could not be determined, you were to put a question mark in the corresponding box. Here's a solution.



This part was worth 4 points. You lost 1 point for each variable you put in the wrong place, and 1 point for each contents you put in the wrong place. For example, the most common error by far was to put the box labeled p in the heap rather than the stack, and another common error was to point name at storage on the stack.

The length and contents (i.e. ?) of the storage pointed by p also had to be specified; you lost ½ point for not specifying length *or* contents, 1 point for specifying *neither* length nor contents, and 1 point for listing contents as, say, "Hello …" rather than as eleven un-initialized characters.

Finally, you lost ½ point for describing facName as a pointer (if it were a pointer, you could assigned to it), and ½ point for not including the terminating null character in its contents.

Part b involved debugging the response function and explaining your fixes. The code contains three bugs:

- It doesn't return anything; the last statement should be

  ```
  return p;
  ```
- Space is allocated for the characters in the two strings but not for the terminating null; the `malloc` statement should be

  ```
  p = malloc (1 + helloLen + strlen(name));
  ```
- Constant string space is overwritten by the `for` loop; in fact, the second assignment statement to p completely undoes the `malloc`, orphaning the allocated storage. It should be rewritten to use `strcpy`:

  ```
  strcpy (p, "Hello, ");
  ```

This part was worth 3 points, 1 for each bug. The three most common errors were the following:

- thinking that the loop didn't handle the terminating null character correctly, e.g. by setting p[k+helloLen] = '\0' after the loop;
- setting *p = "Hello, " (thereby losing the value returned by malloc) instead of copying;
- returning the wrong type argument, e.g. &p or *p.

All of these received 1-point deductions.

If you described a "bug" that was in fact correct, you may have lost 2 points: one for omitting the correct fix to the bug, the other for describing correct code as incorrect.