CS 61C (Clancy) **Solutions and grading standards for exam 1**
Spring 2002

Exam information

285 students took the exam. Scores ranged from 3 to 20, with a median of 15 and an average of 14.3. There were 122 scores between 16 and 20, 130 between 11 and 15, 27 between 5.5 and 10, and 6 between 3 and 5. (Were you to receive a grade of 16 on all your midterm exams, 48 on the final exam, plus good grades on homework, quizzes, and lab, you would receive an A–; similarly, a test grade of 11 may be projected to a B–.)

There were four versions of the exam, A, B, C, and D. (The version indicator appears at the bottom of the first page.) Versions A and C were similar except for the sequence of the problems. Versions B and D were also similar except for the sequence of the problems.

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 (even if the only error is a mistake in adding up your points).

Solutions and grading standards

Problem 0 (1 point)

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

- you earned some credit on a problem and did not put your name on the page,
- you did not indicate your lab section or t.a., 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 would like 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 know where students were sitting in the class room while the exam was being administered.

Problem 1 (3 points)

This was problem 3 on versions C and D. All versions involved initializing a pointer to an int, printing the result of post-incrementing or post-decrementing it, and then printing it directly. Here, for example, is the code for version A:

```
ptr = 0x1050;
printf ("%x\n", ptr--);
printf ("%x\n", ptr);
```

The first printf prints the result of evaluating ptr– –, which is the value of ptr *before the decrementing happens*. (Contrast this to – – ptr, a pre-increment operation in which the decrementing is done and then the *resulting* pointer value is returned.)

Since ptr is a pointer to an int, the result of incrementing or decrementing is an increase or decrease of 4 in the underlying address. Thus the correct answers are

version A	*version B*	*version C*	*version D*
`1050`	`2018`	`6530`	`5498`
`104C`	`201C`	`652C`	`549C`

The 3 points were split as follows:

- 1 for recognizing that a post-increment or post-decrement would return the original value;
- 1 for performing correct pointer arithmetic (adding or subtracting 4, not 1);
- 1 for getting the hexadecimal arithmetic correct.

The most common errors (examples are given for version A) were the following:

- 1050, 1046 (wrong base 16 conversion; 1 point deducted);
- 104C, 104C (wrong post-increment; 1 point deducted);
- 1050, 1050 (wrong post-increment, no evidence of correct pointer arithmetic; 2 points deducted);
- 1050, 104F (wrong pointer arithmetic; 1 point deducted).

Problem 2 (5 points)

This was problem 4 on versions C and D. In versions A and C, you were asked to write a function resultOfInsert that returned the result of inserting a given character into a given string at a given position. In versions B and D, you were to write a function substring that, given a string and two positions, returned the corresponding substring. You were not allowed to change the argument string in either case. Here are solutions.

Versions A and C

```
char *resultOfInsert (char *s, char c, int pos) {
   char *rtn = (char *) malloc (strlen(s)+2);
   int k;
   for (k=0; k<pos; k++) {
      rtn[k] = s[k];
   }
   rtn[pos] = c;
   for (k=pos; k<=strlen(s); k++) {
      rtn[k+1] = s[k];
   }
   return rtn;
}
```

An alternative:

```
char *resultOfInsert (char *s, char c, int pos) {
   char *rtn = (char *) malloc (strlen(s)+2);
   int k, j;
   for (k=0, j=0; k<=strlen(s); k++, j++) {
      if (k==pos) {
         rtn[j] = c;
         j++;
      }
      rtn[j] = s[k];
   }
   return rtn;
}
```

Another alternative:

```
char *resultOfInsert(char *s, char c, int k) {
   char *result = (char *) malloc (strlen(s)+2);
   strncpy (result, s, k);
   result[k] = c;
   result[k+1] = '\0';
   return strcat (result, s+k);
}
```

Versions B and D

```
char *substring (char *s, int start, int finish) {
   char *rtn = (char *) malloc (finish-start+2);
   int j, k;
   for (j=0, k=start; k<=finish; j++, k++) {
      rtn[j] = s[k];
   }
   rtn[j] = '\0';
   return rtn;
}
```

In both versions of the problem, the malloc was worth 2 points (1 for recognizing that it was necessary, the other for doing it correctly) and the rest of the code was worth 3 points. Typical errors and corresponding deductions included the following:

- incorrect malloc argument—strlen(s) or sizeof(s) in versions A and C and finish–start+1 in versions B and D were quite common—lost 1 point;
- forgetting to null-terminate the return string lost 1 point;
- freeing the argument string lost 1 point;
- otherwise modifying the argument string lost 2 points;
- modifying the return pointer lost 1 point, for example by

```
for (i = start ; i < finish ; i++) {
   *newPtr = *s;
   newPtr++;
   s++;
}
```

- in versions A and C, replacing rather than inserting a character or otherwise overwriting data incorrectly lost 2 points;
- in versions B and D, ignoring the finish argument, essentially viewing the call as

  ```
  substring (s, start, strlen(s)–1)
  ```

  lost 2 points.

Many students used a loop instead of pointer arithmetic:

```
int i = 0;                          instead of      s += start;
for ( ; i < start ; i++) {
   s++;
}
```

This code received no deductions, but is asking for trouble (more opportunity for errors, especially off by one).

Problem 3 (3 points)

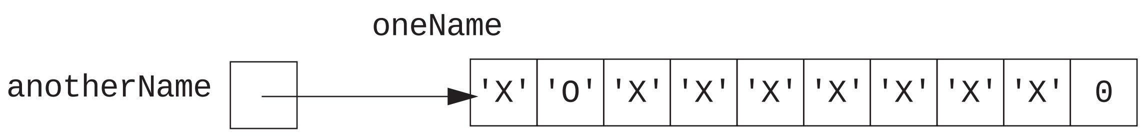
This was problem 5 on versions C and D. All versions of this problem involved drawing a box-and-pointer diagram showing the result of the following code:

```
char oneName[10] = some 9-character string;
char* anotherName;

anotherName = oneName;
strcpy (anotherName, oneName);
change the value of one of the characters in oneName;
```

For example, in version A, oneName was initialized to "xxxxxxxxx" and oneName[1] was set to 'o'. This code copies oneName (interpreted as a pointer to a char) into anotherName. It then copies successive characters in the string addressed by oneName into corresponding locations in the string addressed by anotherName; this operation has no effect since it copies characters to where they already are. (Students who noted in the exam that the result of using strcpy on overlapping strings were referred to the version on page 105 of K&R.)

Here's a full-credit answer for version A:



It displays all ten bytes of the string (the nine ASCII characters plus the terminating zero byte) and shows that anotherName contains a pointer to the first character in the oneName array. You were allowed to omit quotes from the characters as long as you distinguished the 0 byte in some way. You were also allowed to draw oneName as a pointer.

Common errors included the following:

1 point deducted (structure correct)
  length of oneName array is off by one (this was quite common);
  character assignment is off by one;
  the trailing null byte isn't shown, or isn't distinguished from ASCII '0'.

2 points deducted (structure incorrect, but with some shared structure displayed)
  anotherName contains a pointer to the last character of the string represented by oneName;
  the string is drawn as a linked list;
  anotherName points to oneName, which points to the first character of the string.

3 points deducted (no shared structure)
  two copies of the string are drawn;
  anotherName contains the null pointer.

Problem 4 (5 points)

This was problem 2 on versions C and D. All versions of this problem involves finding (in part a) the storage blocks represented in a given layout, and pointing out (in part b) what structural problems the layout displayed that a correctly implemented boundary tags system would not exhibit. Correct answers to part a are given below.

```
A:| 3  0  0 -2  3|-4 -7 -7 -7 -7  -4 | -3 -13 -13 -13 -3| 2   0   0  2|
B:| 4 16  0 -2 -2  4|-1 -8 -1|-4 -12 -12 -12 -12  -4| 3  0   1 -17  3|
C:| 2  0  0  2|-3 -6 -6 -6 -3|-4 -11 -11 -11 -11  -4| 3  0   0 -17  3|
D:| 3 15  0 -2  3|-4 -7 -7 -7 -7  -4 | -1 -13  -1 | 4  0  1 -17 -17  4|
```

In all versions, the first and fourth blocks (reading left to right) were free, and the second and third blocks were allocated. (The sign of the first value in the block indicates the allocated status: > 0 means free, < 0 means allocated.) You were asked to specify the start and end addresses of each block so as to include the boundary tags information. Thus for version A, the start and end addresses are

block	*start*	*end*
1	1	5
2	6	11
3	12	16
4	17	20

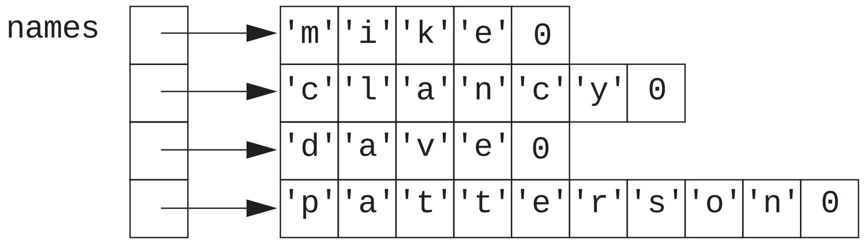
The problem with the layout in versions A and C is that the two free blocks aren't linked (both have prev = next = 0). The problem in versions B and D is the block of length 1, which when freed will not be large enough to hold boundary tag information.

Part a was worth 3 points, with points deducted as follows. You lost 1 point for not including the boundary tags in the start and end addresses (this point was deducted only once). You lost 1 point for addresses being all off by one. You also lost 1 point for providing a correct sequence of commands to the lab 3 program instead of starting and ending block addresses. You lost 2 points for combining the two allocated blocks into one. Where we couldn't figure out a rationale for your errors, you lost 1 point for each incorrect block.

Part b was worth 2 points. Some students had the right answer in versions A and C, but provided incorrect values for the missing links; this lost 1 point. In versions B and D, many of you noted that the contents of one of the blocks (the third block in version B, the fourth in version D) were not what the lab 3 program would have produced. This, however, is not a *structural* problem with the layout since a user might have written anything to his/her allocated block. If this was the only flaw you described, you lost 1 point. An answer that was possibly correct but too vague for us to tell also lost 1 point.

Problem 3 (3 points)

This was problem 1 in versions C and D. In all versions, it involved loading a register with a character from an array of strings (the same array and the same strings in all versions). A diagram of the storage layout appears below.


names
'm' 'i' 'k' 'e' 0
'c' 'l' 'a' 'n' 'c' 'y' 0
'd' 'a' 'v' 'e' 0
'p' 'a' 't' 't' 'e' 'r' 's' 'o' 'n' 0

A solution first involved getting the correct string address into a register. In versions A and C, you were to access the fourth character of the second string. The address of the second string could be loaded in several ways, for example:

```
la $t1,names            lw $t1,names+4          addi $t1,$0,4
lw $t1,4($t1)                                   lw $t1,names($t1)
```

In versions B and D, you were to access the eighth character in the fourth string, so instead of an offset of 4, you would use 12.

Once the proper string address is loaded, you use lb to load one of its characters:

`lb $t0,3($t1)` in versions A and C
`lb $t0,7($t1)` in versions B and D

You lost 1 point for each error. Examples included the wrong load instruction (mixing up la, lw, and lb), the wrong offset (typically off by one or off by a factor of 4), not enough loads, no offset used, and bad syntax. Offset errors were deducted for each instruction they occurred in.

A surprising number of students coded these accesses as loops, thereby maximizing their chances for making mistakes.