Exam information

183 students took the exam. Scores ranged from 1 to 20, with a median of 9.5 and an average of 9.25. There were 17 scores between 15.5 and 20, 54 between 10.5 and 15, 76 between 5.5 and 10, and 36 between 1 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–. The replacement problem that you'll work on on March 5 will presumably improve your score.)

There were two versions of the exam, A and B. (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 (even if the only error is a mistake in adding up your points).

Solutions and grading standards

Problem 0 (1 point)
You lost \( \frac{1}{2} \) point on this problem for each 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 classroom while the exam was being administered.

Problem 1 (7 points)
Both versions involved predicting how much memory was needed for a node, then writing an assembly language function that used the node. Lab 4 and homeworks 2 and 4 provided relevant experience. The node definitions were

```
version A
struct node {
    char name[12];
    int value;
};

version B
struct node {
    char name[20];
    int value;
};
```

One byte is required for each char and four bytes for each int, so the desired answers to part a were 16 and 24 for versions A and B respectively. This part was worth 1 point; no partial credit was awarded. A common error was to think that an N-element character array would take up N+1 bytes. Some students also were unaware that char values on the instructional computers take up one byte each.
In part b, you were to implement one of the following:

version A

```c
void exam1 (struct node **to) {
    exam2 (*to);
    (*(to-1))--;
}
```

version B

```c
void exam1 (struct node **to) {
    exam2 (*to);
    (*(to+1))++;
}
```

Since both involve a call to `exam2` and access to the `to` argument after `exam2` returns, at least two register values must be saved on the stack: `$ra` and either `$a0` or one of the `$s` registers (into which `$a0` would be copied). Here are prologs corresponding to the two alternatives:

```
addi $sp,-8
sw $a0,4($sp)
sw $ra,0($sp)
```

```
addi $sp,-8
sw $s0,4($sp)
sw $ra,0($sp)
move $s0,$a0
```

The order in which register values are stored on the stack is important only insofar as it matches the order in which they are restored from the stack.

Next comes the call to `exam2`. `$a0` contains `to`, but we need `*to`. Here’s how to get it:

```
lw $a0,0($a0)
jal exam2
```

The decrement or increment breaks down into the following steps:

1. Reload from the stack if necessary.
2. Decrement/increment it by 4 (since it’s a pointer to a 4-byte pointer).
3. Load from the resulting address. (We now have `*(to±1)`.)
4. Decrement/increment it by whatever you gave in part a as `sizeof (struct node)`, since `*to` and `*(to±1)` are both pointers to a `struct node`.
5. Update memory.

Here is solution code that assumes `$a0` was saved on the stack rather than in `$s0`.

```
version A
```

```
lw $a0,4($sp)    # restore to
lw $t1,-4($t0)   # get *(to-1)
addi $t1,$t1,-16 # *(to-1))--
sw $t1,-4($t0)   # update *(to-1)
```

```
version B
```

```
lw $a0,4($sp)    # restore to
lw $t1,4($t0)    # get *(to+1)
addi $t1,$t1,24  # *(to+1))++
sw $t1,4($t0)    # update *(to+1)
```

Finally, we restore `$ra` and `$s0` if we used it (restoring `$a0` isn’t necessary), pop the stack, and return:

```
lw $ra,0($sp)
addi $sp,8
jr $ra
```
Solutions could earn up to 2 points for each of the following, for a maximum of 6:

- correct address arithmetic;
- correct loading;
- everything else (the prolog, the epilog, and the call to exam2).

Most errors received a 1-point deduction. Some examples:

**prolog/epilog/call**

- forgetting to save $a0 or $s0
- forgetting to save $ra
- saving but forgetting to restore $ra
- saving $ra in $s0 but not saving $s0
- forgetting that to would be in $a0 on entry to the function

**address arithmetic**

- using an incorrect increment/decrement value
- adding/subtracting in the wrong order

**loads**

- an extra level of indirection
- using `la` instead of `lw`

- forgetting to set up $a0 for the call to exam2
- growing the stack in the wrong direction
- stack pointer off by one
- using “done” instead of `jr`

If you neglected to store the result of the increment/decrement operation, you lost only \( \frac{1}{2} \) point. You were not penalized for using more than two words on the stack. The list above includes the most common errors.

**Problem 2 (3 points)**

This problem was the same on both versions. You were to write a function that, when called immediately before the buggy swap function, would ensure that swap would crash when dereferencing the uninitialized pointer variable `temp`.

Step 1 is to figure out where `temp` is stored, and how it could get accidentally initialized to a value that would make swap “work”. As you saw in lab 3, space for local variables in a C program is allocated on the system stack. Thus `f` must put an invalid pointer onto the stack in the space that `temp` will occupy to ensure the crash.

Though `gcc` pushes function arguments onto the stack, a different compiler might keep them in registers. Thus the number of arguments to `f` should be the same as the
number passed to swap, and the sizes of the arguments should also match those of
swap, just to make sure f does the same kind of stack manipulation as swap does.
Here is our solution:

```c
void f (int *a, int *b) {
    int *temp;
    temp = 0;  /* put an invalid pointer value where swap's temp will be */
}
```

The pointers could have been declared as ints, which take up the same amount of
space. Any odd number would have worked as an invalid pointer value in place of 0.
We also accepted the following definition, which assumed that arguments all took up
stack space as they did in your lab 3 experiments.

```c
void f (int *a, int *b, int *c) {
    a = 0;  /* put an invalid pointer value where swap's temp will be */
}
```

We accepted this code with c set to 0 instead of a. However, C always pushes argu-
ments in reverse order.
The 3 points for this problem were split 1 for the code and 2 for the explanation: 1 for
saying something reasonable about a position on the stack, and 1 for saying some-
thing reasonable about a value of temp sure to produce a crash when dereferenced.
Few students got this right. Some incorrect answers were the following:

```c
void f (int *a, int *b) {
    a = 0;  /* put an invalid pointer value where swap's temp will be */
}
```

call: f(&x,&y);

We think that in the first example, students were trying to make sure swap crashed
because of dereferencing *a rather than *temp. If that worked, swap would then
 crash even after it was fixed! Note, however, that since arguments in C are passed by
value, the assignment only affects the copy of &x on the stack; its subsequent use in
the call to swap would use the original &x. In the second, they mistakenly identified
f's temp with swap's; however, the fact that two local variables have the same name
says nothing about where they appear in memory.

**Problem 3 (6 points)**
This problem was the same on both versions. You were to write a function that
returned a copy of an array of strings (of which argv is an example). We assumed that
lab 2 and homework assignment 2 would prepare you sufficiently for this problem.
A diagram for how argv is represented appears on page 115 of K&R. The top-level
data structure is an array of pointers, each one pointing to the first character of a C-
style string. (In the diagram, the last pointer is 0. We allowed you to disregard this.)
As you noted in lab 2, calls to malloc were necessary both for the top-level structure
and for each of the individual strings. A call to strcpy was necessary to copy each
argv entry into the new data structure, as you also noted in lab 2. Here is a solution:
char **copyStrArray (int count, char **strArray) {
    /* get space for all the pointers */
    char **copy = (char **) malloc (count * sizeof(char *));
    int k;
    for (k=0; k<count; k++) {
        /* get space for each individual string, then fill it up */
        copy[k] = (char *) malloc((strlen(strArray[k])+1) * sizeof(char));
        strcpy (copy[k], strArray[k]);
    }
    return (copy);
}

One might use pointer expressions instead of array references, substituting *(copy+k) for copy[k] and *(strArray+k) for strArray[k]

Grading started with a count of malloc calls; you lost 2 points for each missing one. If your solution copied only one of the elements of the array, you lost 4 (this included 2 for a missing malloc). Then 1½ points were allocated to each of the following:
• the outer malloc, if supplied;
• the inner malloc, if supplied;
• the call to strcpy (you were allowed to code your own version);
• everything else, in a solution that attempted to copy all elements of the array.

This generally meant that each small error was worth ½ point. Most common were misparenthesizing malloc arguments or pointer expressions, forgetting a *, and forgetting to allocate space for the terminating byte in each string. Use of strdup, a function in nova’sstdio library but not K&R’s, lost you 1 point. (We warned you about this at the start of the exam.) Multiple errors of the same type were generally combined into a 1-point deduction.

Type mismatches between the parameters of the given call and the header of the copyStrArray function were unfortunately common. Many of you also confused sizeof and strlen, or neglected to use sizeof despite the warning against assuming anything about the size of a pointer or a char, or failed to include the count argument in the outer malloc. Students who omitted the outer malloc generally tried to declare the array as a local variable:

char **copyStrArray (int count, char **strArray) {
    char *copy[count];
    ...
    return copy;
}

This is wrong for two reasons. First, C doesn’t allow a variable to appear between the brackets in an array declaration (it needs to know exactly how much storage to allocate for the array at compile time). Moreover, the copy array is allocated on the stack, and disappears once the function returns.
Problem 4 (3 points)

This problem was based on homework assignment 3, with the additional feature of involving actual memory contents. You were to determine the result of freeing a given block. The storage layouts on the two versions were as follows:

<table>
<thead>
<tr>
<th>Version A</th>
<th>Version B</th>
</tr>
</thead>
<tbody>
<tr>
<td>101C 0000032</td>
<td>101C 0000030</td>
</tr>
<tr>
<td>1020 0001808</td>
<td>1020 0001808</td>
</tr>
<tr>
<td>1024 0001B1C</td>
<td>1024 0001B1C</td>
</tr>
<tr>
<td>1028 0000031</td>
<td>1028 0000032</td>
</tr>
<tr>
<td>102C 0001054</td>
<td>102C 0000E0C</td>
</tr>
<tr>
<td>1030 0001010</td>
<td>1030 0001FF0</td>
</tr>
<tr>
<td>1034 0000032</td>
<td>1034 0000031</td>
</tr>
<tr>
<td>1038 0000FF4</td>
<td>1038 0000FF4</td>
</tr>
<tr>
<td>103C 000200C</td>
<td>103C 000200C</td>
</tr>
<tr>
<td>1040 0000031</td>
<td>1040 0000030</td>
</tr>
<tr>
<td>1044 0001050</td>
<td>1044 0001050</td>
</tr>
<tr>
<td>1048 0001028</td>
<td>1048 0001028</td>
</tr>
<tr>
<td>104C 0000030</td>
<td>104C 0000032</td>
</tr>
<tr>
<td>1050 0001030</td>
<td>1050 0001C08</td>
</tr>
<tr>
<td>1054 000102C</td>
<td>1054 00010AC</td>
</tr>
</tbody>
</table>

A good first step is to determine what the blocks are. You can determine this from the administrative words. They have the following format:

<table>
<thead>
<tr>
<th>size</th>
<th>isFree</th>
<th>precedingIsFree</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 bits</td>
<td>1 bit</td>
<td>1 bit</td>
</tr>
</tbody>
</table>

A value of 30 in an administrative word represents an allocated block of 12 bytes (3 words) whose left neighbor is also allocated. 31 is an allocated block of 12 bytes next to a free block, and 32 is a free block of 12 bytes (whose neighbor blocks must be allocated).

The address given to free points to the word that immediately follows the administrative word. In both versions, the administrative word indicates that the neighboring block on the left is free, so the two blocks must be combined into a block whose size is 6 words. The figure on the next page indicates all necessary changes.

The 3 points for this problem were split evenly among the three changes to make: increasing the size to 6 at address 1034 or 1028, copying the prev field from 103C/1030 to 1048/103C, and turning on the precedingIsFree bit at address 104C or 1040. You lost 1 point for each other change that overwrote crucial information (say, a link in a free block or any other administrative word, or any change to an allocated block). We ignored changes that would have no effect. You lost 1/2 point for an incomplete description of a change, for example by saying “the precedingIsFree bit is turned on” without giving the resulting memory contents. You also lost 1/2 point for a small error that was unrelated to your understanding of memory management, for example, a hexadecimal arithmetic error.
Few of the exams contained intelligible answers. When the page wasn’t blank, it usually contained various scribbles followed by an answer for changing two or three words, unaccompanied by any kind of explanation.

A common mistake was to assume that the linked list of free blocks should be sorted by address as in the K&R storage allocation scheme, and to set the next and prev fields of the free blocks accordingly. Some students misread the administrative block because of hexadecimal arithmetic problems or administrative data ordering misconceptions. Students also seemed confused about the role of the administrative word; some of you seemed to think that everything (including administrative data and user data) was contained in the word at (say) 1044, and that calling precedingBlock(1044) would return 1040. We are unsure of the reason for this error.