
**Instructions:**

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other electronic devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test and your test form will be collected immediately.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 6 questions, some with multiple parts, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- If you brought a fact sheet to the test, write your name on it and turn it in with the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

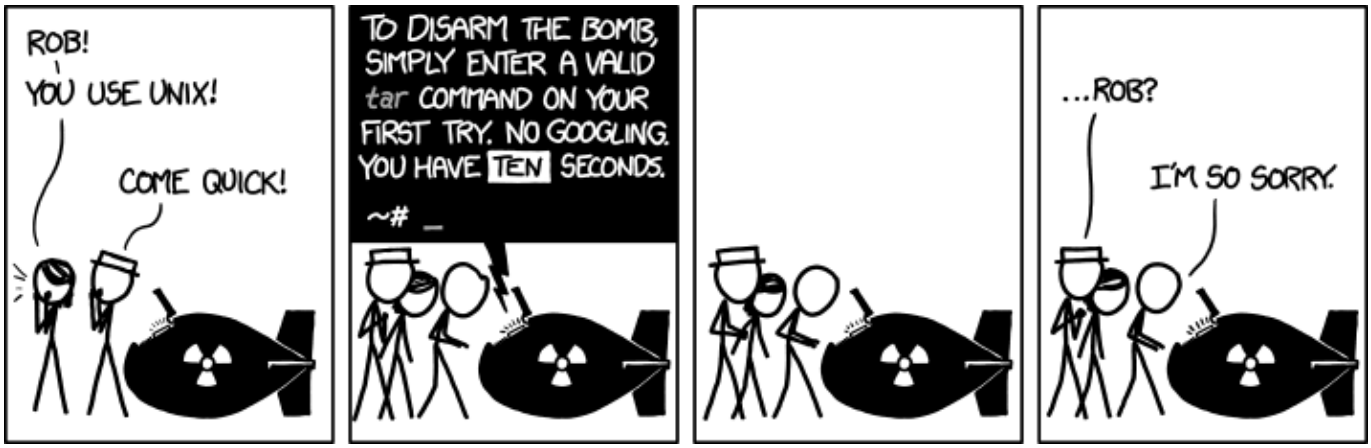
**Do not start the test until instructed to do so!**

**Name** Solution

printed

**Pledge:** On my honor, I have neither given nor received unauthorized aid on this examination.

signed


ROB!
YOU USE UNIX!
COME QUICK!
TO DISARM THE BOMB, SIMPLY ENTER A VALID tar COMMAND ON YOUR FIRST TRY. NO GOOGLING. YOU HAVE TEN SECONDS.
~# _
...ROB?
I'M SO SORRY.

**xkcd.com**

**1.** Executing the command `ls -l` in a directory on my CentOS virtual machine, where my user id is `wdm`, shows me the following information:

```
drw-------. 2 wdm wdm     6 Jun 17 15:25 backup
--w--w----. 1 wdm wdm  2620 Jun 17 15:19 bigMesa.c
-r--r--r--. 1 wdm wdm  1302 Jun 17 15:19 bigMesa.h
-rw-rw----. 1 wdm wdm 13407 Jun 17 15:29 driver
-rw-rw-r--. 1 wdm wdm  5456 Jun 17 15:19 driver.c
```

If I attempt to execute each of the following commands, it fails with an error message from the shell. Explain exactly why each of these commands will fail.

**a)** **[4 points]** `cat bigMesa.c`

The cat command needs to read the contents of the file, and I don't have read permissions for the file bigMesa.c.

**b)** **[4 points]** `cp driver.c backup`

I don't have execute permissions for the directory backup/.

This may be a bit confusing... you need execute permissions in order to execute a command on the directory.

**c)** **[4 points]** `cp driver.c bigMesa.h`

I don't have write permissions for the file bigMesa.h.

The command would overwrite the existing file bigMesa.h with the file driver.c, and that would require having write permissions for bigMesa.h. You might wonder if this would work if you did have write permissions for bigMesa.h, since it seems the result would be two files named driver.c in the same directory...

**d)** **[4 points]** `./driver`

I don't have execute permissions for the file driver.

In Linux, whether it's possible to (attempt to) execute a file is determined solely by whether the user has execute permissions for that file, not by the contents of the file.

**2. [16 points]** Write an implementation of a C function satisfies the given header comments:

```
/**  Count the number of elements in A[] that are less than a given value.
 *
 *   Pre:   A[0:Sz-1] have been initialized
 *          Ceiling has been initialized
 *
 *   Post:  The contents of A[] have not been modified
 *
 *   Returns: number of elements in A[] that are < Ceiling
 *
 *   You may use either array or pointer notation in your solution.
 */
int countLessThan(int A[], int Sz, int Ceiling) {

   int numLessThan = 0;                  // counter for number of elements
                                         //   that are less than Ceiling

   for (int idx = 0; idx < Sz; idx++) {  // use standard array traversal pattern

      if ( Ceiling > A[idx] ) {          // test for elements less than Ceiling

         numLessThan++;                  // count them
      }
   }

   return numLessThan;                   // return counter
}
```

Here's a pointer-based solution:

```
int countLessThan(int A[], int Sz, int Ceiling) {

   int numLessThan = 0;                  // counter for number of elements
                                         //   that are less than Ceiling

   for (int idx = 0; idx < Sz; idx++) {  // use standard array traversal pattern

      if ( Ceiling > *(A + idx) ) {      // test for elements less than Ceiling

         numLessThan++;                  // count them
      }
   }

   return numLessThan;                   // return counter
}
```

**3.** A developer is testing an implementation of a function, `numUps()`, described in comments in the code below:

```
#include <stdio.h>

int numUps(int A[], int Sz);

int main() {

   int List[20] = { -100, -90, -80, -70, -60,                 // prologue
                    -7, 19, 12, 14, 17, 13, 21, 23, 25, 24,   // array for numUps()
                    100, 110, 120, 130, 140};                 // epilogue

   printf("Number of ups: %d\n", numUps(&List[5], 10) );

   return 0;
}

/**  Returns the number of elements in A[] that are larger than the preceding
 *   element in A[].
 *
 *   Pre:
 *          A[0:Sz-1] are initialized
 *   Post:
 *          A is unchanged
 */
int numUps(int A[], int Sz) {

    int ups = 0;
    for (int idx = 0; idx < Sz; idx++) {
        if ( A[idx] < A[idx + 1] )
           ups++;
    }
    return ups;
}
```

**a)** **[10 points]** When she compiled the code, there were no warning or error messages. When she executed the test she has written above, she expected a return value of 6, but the function `numUps()` returned the value 7. Explain exactly how to fix the logic error in the implementation of `numUps()`.

**The error is that the loop accesses an out-of-bounds element on the last pass.**

**The simple fix is to change the loop test to: `idx < Sz - 1`.**

**b)** **[6 points]** Explain what she did in designing her test that led to discovering the error.

**She embedded the test array within a "buffer" that included values chosen to alter the function's behavior IF it accessed data outside the array; going beyond the array bounds in either direction would cause the function to count additional "ups".**

**4. [16 points]** Write an implementation of a C function satisfies the given header comments:

```
/**  Replaces even values in an array with odd values.
 *   Pre:
 *          pA[0:Sz-1] are initialized
 *          pChanges points to a variable of type int
 *
 *   Post:
 *          If an even value K occurs in the array, K has been replaced by
 *          2K + 1.
 *          *pChanges equals the number of even values that were changed
 *
 *   Restrictions:
 *
 *      You must not use array bracket notation in your solution; you must
 *      use pointer notation for all accesses to the array.
 */
void oddBall(int* pA, int Sz, int* const pChanges) {

   *pChanges = 0;    // *pChanges wasn't guaranteed to be preset to 0

   for (int idx = 0; idx < Sz; idx++) {  // std array traversal logic

      if ( *pA % 2 == 0 ) {          // see if current element is even

         *pA = 2 * (*pA) + 1;        // reset current element, if even
         *pChanges = *pChanges + 1;  // count changes
      }
      pA++;                          // step to next element of array
   }
}
```

Here's a somewhat better solution:

```
void oddBall(int* pA, int Sz, int* const pChanges) {

   int numChanges = 0;     // count changes w/o a pointer dereference
   int* pCurr = pA;        // points to current element of array
   int* pEnd = pA + Sz;    // points to memory location just after end of array

   while ( pCurr < pEnd ) {           // stop when step past end of array

      int currElement = *pCurr;       // copy current element from array;
                                      //   void extra memory accesses

      if ( currElement % 2 == 0 ) {     // see if current element is even

         *pCurr = 2 * currElement + 1;  // reset even element
         numChanges++;                  // count changes
      }
      pCurr++;                          // step to next element
   }
   *pChanges = numChanges;  // set number of changes once, at end
}
```

**5.** Suppose that we have the following variable declarations:

```
uint64_t X = 0x0011223344556677;
uint8_t* q = (uint8_t*) &X;
```

Then we have the following memory situation (numeric values are all given in hexadecimal):

```
Address   byte
--------------
EC9BD6E8:  77    <--- q points to THIS byte
EC9BD6E9:  66
EC9BD6EA:  55
EC9BD6EB:  44
EC9BD6EC:  33
EC9BD6ED:  22
EC9BD6EE:  11
EC9BD6EF:  00
```

**a)** **[6 points]** Why does `q` point to that particular byte of `X`? Be very specific.

**Because multibyte integers are stored with the least-significant byte at the lowest address (i.e., in little-endian order).**

**The fact that the pointer has a one-byte target is irrelevant; any pointer initialized to &X would store the address of the same byte.**

**b)** **[15 points]** What value (expressed in human-friendly, big-endian hexadecimal) would each of the following statements assign to the variable on the left side of the assignment operation? No justification is needed.

```
uint16_t v1 = *(uint16_t*) q;
```

**0x6677**

```
uint32_t v2 = *(uint16_t*) q;
```

**0x00006677 (The leading zeros are unnecessary, but this IS a 32-bit value.)**

```
uint32_t v3 = *(uint32_t*) q;
```

**0x44556677**

```
uint16_t v4 = *(uint16_t*) (q + 4);
```

**0x2233**

```
uint16_t v5 = *(uint16_t*) q + 4;
```

**0x6677 + 4 = 0x667B**

**6.** A programmer is testing an implementation of a function that's supposed to conform to the description in the header comment below:

```
/**  Returns the square of A % B.  For example F(23 % 10) == 9.
 */
int F(int A, int B);
```

The programmer writes some simple test code to call the function and print the results; he gets the following output:

```
The square of 7834 mod 47 is 45
```

Now, this is puzzling, since it's clear that the result is not even a perfect square. A little arithmetic reveals that 7834 % 47 is actually 32. So, the programmer uses `gdb` to analyze the implementation:

```
CentOS> gcc -o Q6 -std=c99 -Wall -ggdb3 Q6.c
CentOS> gdb Q6
. . .
(gdb) break main
Breakpoint 1 at 0x400538: file Q6.c, line 8.
(gdb) break F
Breakpoint 2 at 0x400583: file Q6.c, line 20.
(gdb) run
Starting program: /home/wdm/2505/Midterm/Q6/Q6

Breakpoint 1, main () at Q6.c:8
8         int N = 7834;

(gdb) next
9         int D = 47;

(gdb) next
. . .

(gdb) print N
$1 = 7834
(gdb) print D
$2 = 47
(gdb) continue
Continuing.

Breakpoint 2, F (A=47, B=7834) at Q6.c:20
20        int R = A % B;
(gdb)
```

**a)** **[5 points]** The `gdb` session above tells the programmer there is a specific error in the code he has written. What is that error?

**The line that begins "Breakpoint 2" shows that the parameters were passed to the function in reverse order (a fairly common programming error, and one the compiler cannot detect since both parameters are of the same type).**

**It's true that the computation shown in the initialization of R is incorrect, but that is entirely due to the fact that the parameters are reversed.**

The programmer fixes the rather embarrassing error mentioned above, recompiles the code and executes it again; this time he gets:

```
The square of 7834 mod 47 is 34
```

Well, that is still wrong... 34 isn't a perfect square either. So, he resorts to gdb again:

```
. . .
Breakpoint 2, F (A=7834, B=47) at Q6.c:20
20          int R = A % B;
(gdb) next
22          R = R ^ 2;
(gdb) print R
$3 = 32
(gdb) next
24          return R;
(gdb) print R
$4 = 34
```

**b)** **[6 points]** The `gdb` session above tells the programmer there is another specific error in the code he has written. Precisely what is that error?

**There are two things to notice. The less obvious one is that he has used '^' to square R; but, of course, it really means XOR in C. The more obvious thing is that the value of $R^2$ is clearly incorrect.**

**The essential point was that the XOR operator was used.**

**c)** **[4 points]** Given that he now understands the second error, and he understands data representation and simple bitwise operations, the programmer also now understands why the function produced the value 34 when he ran it (the second time).

Explain exactly why the value of `R` changed from 32 to 34 near the end of the execution of the function `F()`.

**R was set to 32 when A % B was evaluated.**

**R will be represented in 2's complement form as 0...0010 0000.**

**Then R ^ 2 will be 0...0010 0000 ^ 0...0000 0010.**

**Since 0 XOR 0 == 0 and 0 XOR 1 == 1, R ^ 2 == 0...0010 0010 or 34.**

**A complete explanation needed to explain exactly what's going on at the bit level.**