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. This examination is closed book and closed notes, aside from the permitted one-page fact sheet. Your fact sheet may contain definitions and examples, but it may not contain questions and/or answers taken from old tests or homework. You may include examples from the course notes.
- 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 7 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.

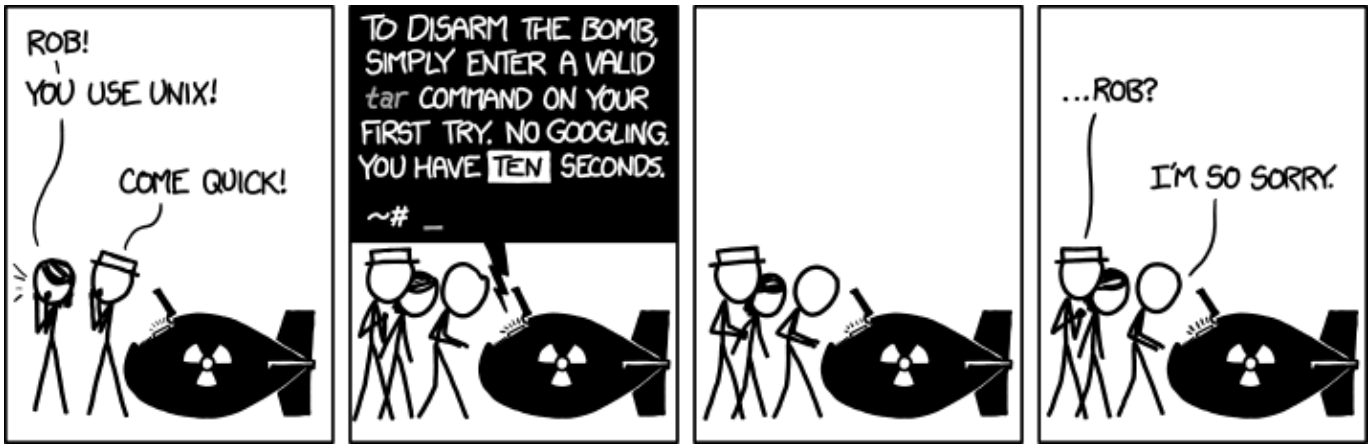
Do not start the test until instructed to do so!

Answers are in blue.
Commentary is in green.

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. A CentOS user executes the following commands in a bash shell:

```
CentOS> rm foo
CentOS> umask 044
CentOS> touch foo
CentOS> ls -l
total 16
-rw-rw-r--. 1 johokie  johokie  126 Mar 25 21:59 a.c
-rw--w--w-. 1 johokie  johokie    0 Jun 15 18:48 foo
```

The user is surprised, because he expected the default permissions for the newly-created file `foo` to be different.

a) **[6 points]** Why did the `umask` command shown above result in those permissions for the file `foo`?

The parameter to umask specifies the permissions that are to be denied, not granted.

So, the user's command specified that, for new files, everyone but the owner would be denied read access.

The question asked "why"... many answers did not give an explanation of umask.

b) **[4 points]** What `umask` command should he have used, if he wanted owner to have all permissions, and group and other to have only read permissions?

umask 033

c) **[4 points]** The user discovers that if he opens a new instance of the bash shell, the default file permissions are the same as they were before the `umask` command shown above was executed. What should he have done in order to make the change to the default file permissions persistent?

He should have placed the umask command into .bash_profile or .bashrc.

---

2. **[10 points]** Suppose you are running a bash shell in a directory below your root directory, so we can assume you have full permissions for your current directory and its parent directory. Write a single command that will create a tar file named `Source.tar`, located in the parent directory, and containing all of the `.c` and `.h` files in your current directory, and nothing else.

tar cf ../Source.tar *.c *.h

Common errors were to omit the "cf" switch, to omit or misplace the "../" that puts the created tar file in the parent directory, and to (disastrously) put Source.tar at the end of the command.

3. [16 points] A novice C programmer writes the following function, which is supposed to perform as described in the header comment:

```
/**  Replaces all the odd values in an array with zero.
 *
 *   Pre:  Data[0:Sz-1] are initialized
 *   Post: Every odd value in Data[0:Sz-1] is now zero
 *   Returns: the number of odd values that were replaced
 */
uint8_t zeroOdds(int32_t Data[], uint32_t Sz) {    // Line  1

   uint8_t nChanged = 0;                           // Line  2

   // Process the array elements from the end down:
   for (uint32_t idx = Sz; idx >= 0; idx--) {      // Line  3

      if ( Data[idx] % 2 == 0 )                    // Line  4
         Data[idx] = 0;                            // Line  5
         nChanged++;                               // Line  6
   }
   return nChanged;                                // Line  7
}
```

He then writes the following code to call the function shown above:

```
#define Dimension 500                                          // Line  8
uint8_t zeroOdds(int32_t Data[], uint32_t Sz);                 // Line  9

int main() {
   // Initialize the array with 400 integer values (some are not shown):
   int32_t List[Dimension] = {23, 31, 22, 53, 44, ..., 11, 10}; // Line 10
   uint8_t N = zeroOdds(List, Dimension);                      // Line 11
   return 0;                                                   // Line 12
}
```

His code compiles, without warnings, but it contains at least 5 logic errors that may affect the program when it's executed. Identify four of them clearly (line numbers are useful). **Your first four answers will be graded; extras will be ignored.**

1. The parameter in the call to zeroOdds() should be the number of elements in the array, not the dimension of the array (unless they are equal).
2. The return type for zeroOdds() should be uint32_t, to match the type of the size parameter.
3. The type of nChanged should be uint32_t for the same reason.
4. The for loop in zeroOdds() will not terminate, since the loop counter cannot be negative.
5. The test for Data[idx] is testing for an even value, not an odd value.
6. The body of the if statement in the loop lacks curly braces, so nChanged will not be correct.
7. The loop counter idx should be initialized to Sz – 1, not Sz.

Some common misconceptions:

- the fact that the dimension of List[] is 500, but only 400 cells are used is NOT an error
- List[400:499] are NOT random garbage; they are initialized to 0
- the function can, indeed, modify the array List[]; passing an array name into a function allows the function to modify the contents of the array (unless const is used), because array names are pointers

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

```
/**  For each element of an array that has a preceding element that's
 *   smaller, replace that element with the smaller predecessor.
 *
 *   For example, given the array
 *      {23, 31, 22, 53, 44, 12, 11, 15}
 *   the function would modify the array to hold
 *      {23, 23, 22, 22, 22, 12, 11, 11}
 *   and return 4.
 *
 *   Restrictions:  you may NOT use array notation in this function;
 *   all accesses to the array elements must use pointer code.
 */
uint8_t capWithMin(int32_t* pA, uint8_t nElems) {

   int32_t currMinimum = *pA;   // smallest so far is first elem

   uint8_t nChanged = 0;        // number of elems replaced

   for (uint8_t pos = 1; pos < nElems; pos++) {

      if ( *(pA + pos) < currMinimum ) {      // found a new minimum

         currMinimum = *(pA + pos);

      }
      else if (*(pA + pos) > currMinimum ) {  // found elem needing replacement

         *(pA + pos) = currMinimum;

         nChanged++;
      }
   }

   return nChanged;
}
```

Here's another solution that hinges on an observation about how this really works:

```
uint8_t capWithMin(int32_t* pA, uint8_t nElems) {

   uint8_t nChanged = 0;        // number of elems replaced

   for (uint8_t pos = 0; pos < nElems - 1; pos++) {

      if ( *pA < *(pA + 1) ) {      // immediate predecessor is smaller

         *(pA+1) = *pA;
         nChanged++;
      }
      pA++;
   }

   return nChanged;
}
```

5. [12 points] Suppose that we have the following variable declarations:

```
uint64_t X = 0x0123456789ABCDEF;
uint8_t* q = (uint8_t*) &X;
```

What will be printed by each of the following statements (when compiled and executed on an x86 system)? Write your answer exactly as the output would look. The code is syntactically correct, and there are no subtle formatting issues. The point of these questions is to analyze the pointer typecasts that are being used, and to think about the in-memory data representation that's used.

a) `printf("%"PRIX8"\n", *q);`

q points to the low byte (x86 uses little-endian order), so EF.

b) `printf("%"PRIX8"\n", *(q+1));`

q+1 points to the second-lowest byte, so CD.

c) `printf("%"PRIX32"\n", *(uint32_t*)q);`

q points to the low byte, but we've cast it so we get 4 bytes, so 89ABCDEF.

d) `printf("%"PRIX16"\n", *(uint16_t*)(q+2));`

q+2 points to the third byte, but we've cast it so we get 2 bytes, so 89AB.

---

6. [10 points] What is the important logical difference between the mathematical notion of an integer and the programming language notion of an integer data type, like `int` in C? As a programmer, describe one kind of logical error you may encounter because of this difference.

Integers are unbounded; that is, there is no smallest integer, nor is there a largest integer.

But data types in C are represented using a fixed number of bits. Therefore, there is a fixed number of possible patterns we can form with those bits, and so it is only possible to represent a finite number of different integer values.

Therefore, there is a largest int and a smallest int, and if the correct result for a calculation falls outside of that range, we will store the wrong value.

There are also unsigned int types in C, which cannot be negative. While mathematicians may talk about the natural numbers, that seems not to lead to any serious issues. Unsigned int types in C can, as seen in the loop in the third question on this test.

Integer values within the range of the type are represented exactly, so that is not an issue.

The second issue is not as serious as the bounded range, and I did not award full credit for it. Some other answers lost points for not being sufficiently specific about the possible logical error that could occur.

The fact that integer operations yield integer results, rather than real numbers, is actually NOT a deviation from mathematics (see Number Theory).

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

```
/**  Solves quadratic equations that have integer coefficients.
 *
 *   Pre:  the polynomial to be solved is ax^2 + bx + c
 *   Post: prints the two roots of the polynomial
 */
void solveQ(int a, int b, int c);
```

The plan is to use the quadratic formula to find the two roots: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

If the polynomial has integer coefficients, and the discriminant (the term under the radical) is a perfect square, he can at least hope to get useful solutions even though he'll have to use to floating-point calculations.

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

```
The polynomial is: 2x^2 + 5x + -3
The roots are: -4.0 and -6.0
```

Now, this is puzzling, since it's clear that the actual roots of that polynomial are -3 and 1/2. So, the programmer uses `gdb` to analyze the implementation:

```
CentOS> gdb q7
. . .
(gdb) break solveQ
Breakpoint 1 at 0x400687: file q7.c, line 25.
(gdb) run
Starting program: /home/johokie/Q7/q7

Breakpoint 1, solveQ (a=2, b=5, c=-3) at q7.c:25
25            discriminant  = 4 * a * c;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-106.el7_2.4.x86_64
(gdb) next
26            discriminant += b * b;
(gdb) next
29            numerator1  = -b;
(gdb) print discriminant
$1 = 1
```

a) **[10 points]** The `gdb` session above tells the programmer there is a specific error in the code he has written. What is that error, what specific `gdb` output other than C code that is shown indicates the error, and how can he fix it?

The discriminant is b*b - 4*a*c; in this case, that should equal 25 + 24 = 49, not 1.

It's then clear that the formula used is incorrect; looking at output lines 25 and 26, it seems that the programmer coded this as b*b + 4*a*c.

So, that's easily fixed by adding an arithmetic negation: discriminant = -4 * a * c;

By far, the most common error here (and in the next part) was to ignore the specific question that was asked. Most answers simply cited the displayed C code.

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

```
The polynomial is: 2x^2 + 5x + -3
The roots are: 2.0 and -12.0
```

Well, that is still wrong... the roots are still -3 and 1/2. So, he resorts to gdb again:

```
. . .
(gdb) run
Starting program: /home/wmcquain/2505/Midterm/Q7/q7

Breakpoint 1, solveQ (a=2, b=5, c=-3) at q7.c:25
25            discriminant  = -4 * a * c;
Missing separate debuginfos, use: debuginfo-install glibc-2.17-106.el7_2.4.x86_64
(gdb) next
26            discriminant += b * b;
(gdb) next
29            numerator1  = -b;
(gdb) print discriminant
$1 = 49
```

OK, at least that is correct now... so he proceeds further:

```
(gdb) next
30            numerator1 += sqrt(discriminant);
(gdb) next
33            root1 = numerator1 / 2 * a;
(gdb) print numerator1
$2 = 2
(gdb) next
35            double numerator2 = -b;
(gdb) print root1
$3 = 2
```

b) **[12 points]** The `gdb` session above tells the programmer there is another specific error in the code he has written. Precisely what is that error, what specific `gdb` output other than C code that is shown indicates the error, and how can he fix it?

Here, numerator1 should be -5 + sqrt(49) = 2, and it is...

But root1 should be 2 / 4, since a = 2.

The error is in the assignment: root1 = numerator1 / 2 * a;

But, due to precedence rules, the right side is executed as (numerator1 / 2) * a;

The programmer needs parentheses like so: root1 = numerator1 / (2 * a);