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 computing 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.

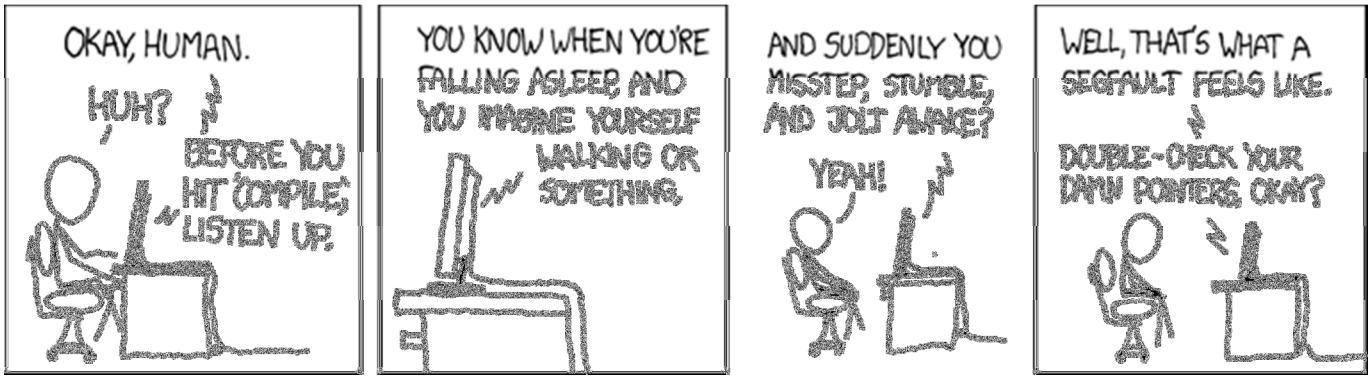
Do not start the test until instructed to do so!

Name Solution

printed

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

signed


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE', LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP AND YOU IMAGINE YOURSELF WALKING OR SOMETHING.
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

xkcd.com

1. a) [10 points] Write an implementation of the C function described below. You must achieve the specified results, and you must conform to the given interface and header comments. Be careful to avoid memory leaks.

```
/**
 *  Create and return an array holding the differences of the successive
 *  elements in the array pointed to by A.  Return NULL if A contains fewer
 *  than two elements, or the array of differences cannot be created.
 *
 *  Pre:     A == NULL and Sz == 0, or A[0:Sz-1] have been initialized.
 *
 *  Examples:
 *    {10, 9, 23, 51, 32} yields {-1, 14, 28, -19}
 *    {74} yields NULL
*/
int* Differences(const int* const A, int Sz) {

   // write the body of your function below:

   if ( A == NULL || Sz < 2 ) return NULL;         // check parameters

   int* Diffs = malloc( (Sz - 1) * sizeof(int) ); // create array for diffs
   if ( Diffs == NULL ) return NULL;               // make sure you got it

   for (int idx = 0; idx < Sz - 1; idx++) {        // Sz - 1 diffs

      Diffs[idx] = A[idx + 1] - A[idx];            // compute diff
   }

   return Diffs;                                   // return array pointer
}
```

b) [4 points] A C programmer wants to design a function `F()` that could be called as follows:

```
int X = ???;  // initial value is set but not shown
if ( F(X) )
   printf("X has been reset to %d\n", X);
```

The function `F()` either resets the value of the user's integer or returns a value indicating it failed to do so.

Write the most appropriate declaration the programmer could use for the function `F()`. DO NOT write an implementation of `F()`. Due to the nature of the question, it is important that you use correct C syntax for your answer.

```
bool F(int* const pX);
```

Return type must be bool or int in order to signal success/failure.
Parameter must be a pointer to the caller's int variable.
F() shouldn't be able to change where that pointer points.

2. Consider the following C struct and its corresponding initialization function.

```
struct _Rational {
     int32_t Top;       // numerator
     int32_t Bottom;    // denominator
};
typedef struct _Rational Rational;

/* Creates and initializes a new Rational object. */
Rational Rational_Construct(int Numerator, int Denominator) {
      Rational R;

      R.Top    = Numerator;
      R.Bottom = Denominator;

      Rational_Normalize(&R);

      return R;
}
```

a) [5 points] Assume you have 2 structures of type `Rational` and that have both been properly initialized using `Rational_Construct()`.

```
Rational R1, R2;

/* initialization */
R1 = Rational_Construct(10, 3);
R2 = Rational_Construct(5, 3);

R2 = R1;
```

Does the assignment above work the way the programmer intended? Are there any undesirable consequences from the code above?

This is a shallow copy, so the values of Top and Bottom are copied from R1 into R2. Since both Top and Bottom are integers, the shallow copy works as intended and we have two separate, equal copies. Further, there shouldn't be any undesirable side effects.

b) [5 points] Would your answer to part a) change if `R1` and `R2` were pointers initialized using the code below? Explain why or why not.

```
Rational *R1, *R2;

R1 = malloc(sizeof(Rational))
R2 = malloc(sizeof(Rational))

*R1 = Rational_Construct(10, 3);
*R2 = Rational_Construct(5, 3);

R2 = R1;
```

R1 and R2 are pointers, so the assignment overwrites the value (an address) in R2 with the value of R1 (also an address). This is a shallow copy, which is normally not what we want when using pointers. As a result R1 and R2 have the same target (*R1) rather creating than a separate copy. This is would change our answer from before and it most likely not what was intended, as no copy was made, in fact, the code actually results in a memory leak, another undesirable consequence.

c) [6 points] Write an implementation of the C function described below. You must achieve the specified results, and you must conform to the given interface and header comments.

```
/**
*   Compute the arithmetic ceiling of R.
*   Pre:
*        R has been properly initialized.
*   Returns:
*        The smallest integer N such that N >= R.
*/
int Rational_Ceiling(const Rational R) {
      // write the body of your function below:
      assert(R.Bottom != 0);

      int ceiling = 0;
      if ((R.Top % R.Bottom  != 0)&& ((float) R.Top / (R.Bottom > 0))
            ceiling++;

      ceiling += R.Top / R.Bottom;

      return ceiling;
}
```

There are 3 cases we need to handle:
When the Top and Bottom divide evenly.
When the result is positive.
When the result is negative.

3. Consider the x86-32 translation of a short C function shown at right.

Assume that all parameters and local variables are of type `int` or `int*`.

a) [6 points] How many parameters does the function `Q3()` receive?

Two.

State the stack address at which each parameter is stored (e.g., `%ebp - 48`).

`%ebp + 8` # 5
`%epb + 12` # 7

b) [6 points] How many local variables does the function `Q3()` have?

Two.

State the stack address at which each local variable is stored (e.g., `%ebp - 48`).

`%ebp - 12` # 24
`%ebp - 16` # 18

There are other locations in the stack frame, but those are not referred to in the code, and therefore are not actual variables.

```
   . . .
Q3:
   pushl    %ebp                  #  1
   movl     %esp, %ebp            #  2
   pushl    %ebx                  #  3
   subl     $36, %esp             #  4
   cmpl     $0, 8(%ebp)           #  5
   je       .L2                   #  6
   cmpl     $1, 12(%ebp)          #  7
   jg       .L3                   #  8
.L2:                              #  9
   movl     $0, %eax              # 10
   jmp      .L4                   # 11
.L3:                              # 12
   movl     12(%ebp), %eax        # 13
   subl     $1, %eax              # 14
   sall     $2, %eax              # 15
   movl     %eax, (%esp)          # 16
   call     malloc                # 17
   movl     %eax, -16(%ebp)       # 18
   cmpl     $0, -16(%ebp)         # 19
   jne      .L5                   # 20
   movl     $0, %eax              # 21
   jmp      .L4                   # 22
.L5:                              # 23
   movl     $0, -12(%ebp)         # 24
   jmp      .L6                   # 25
.L7:                              # 26
   movl     -12(%ebp), %eax       # 27
   sall     $2, %eax              # 28
   addl     -16(%ebp), %eax       # 29
   movl     -12(%ebp), %edx       # 30
   addl     $1, %edx              # 31
   sall     $2, %edx              # 32
   addl     8(%ebp), %edx         # 33
   movl     (%edx), %ecx          # 34
   movl     -12(%ebp), %edx       # 35
   sall     $2, %edx              # 36
   addl     8(%ebp), %edx         # 37
   movl     (%edx), %edx          # 38
   movl     %ecx, %ebx            # 39
   subl     %edx, %ebx            # 40
   movl     %ebx, %edx            # 41
   movl     %edx, (%eax)          # 42
   addl     $1, -12(%ebp)         # 43
.L6:                              # 44
   movl     12(%ebp), %eax        # 45
   subl     $1, %eax              # 46
   cmpl     -12(%ebp), %eax       # 47
   jg       .L7                   # 48
   movl     -16(%ebp), %eax       # 49
.L4:                              # 50
   addl     $36, %esp             # 51
   popl     %ebx                  # 52
   popl     %ebp                  # 53
   ret                            # 54
   . . .
```

The key to parts c and d is to analyze the patterns formed by the conditional and unconditional jump instructions. The arrows on the previous page illustrate what's going on.

An if construct will involve one conditional jump (going forward) and one label.

An if-else construct will involve two conditional jumps (both going forward) and two labels.

A loop will involve a conditional jump going backwards; remember loops are always transformed to do-while patterns in assembly code. A while loop (in C) will have an unconditional jump to the loop test from a point before the loop body. So, you need at least two labels.

(BTW, this code is just the solution to question 1.)

c) [6 points] Examine the given x86-32 code. If there any `if` or `if-else` control structures in the code, for each such structure, what range of statements (e.g., lines 17 through 23) belong to that structure (including any relevant labels and the boolean test)?

if-else in # 5 to # 12

if in # 19 to # 23

d) [6 points] Examine the given x86-32 code. If there any loops in the code, for each loop, what range of statements (e.g., lines 17 through 23) belongs to the body of the loop (including any relevant labels and the loop test)?

while loop in # 25 to # 50

e) [6 points] Examine the given x86-32 code. How many bytes are in the stack frame for `Q3`, including the backed up value for the frame pointer `ebp`? Justify your answer.

The key lines are:

```
pushl %ebp              #  1
. . .
pushl %ebx              #  3
subl  $36, %esp         #  4
```

The first line allocates 4 bytes and stores the old frame pointer there.

The second line allocates 4 bytes and stores the old value of %ebx there.

The third line allocates another 36 bytes.

So the total size of the stack frame is 44 bytes.

4. [15 points] Perform the indicated conversions. You must show work in order to receive credit!

a) Convert from 8-bit 2's complement (i.e., signed) form to base-10: 1010 0101

First, take the negation in 2's complement: 01011011
Then, expand the positional representation: 2^6+2^4+2^3+2^1+2^0 = 64+16+8+2+1 = 91
Finally, take the negation of the result: -91

b) Convert from 8-bit unsigned form to base-10: 1101 0101

Expand the positional representation: 2^7+2^6+2^4+2^2+2^0 = 128+64+16+4+1 = 213

c) Convert 0x20 hexadecimal to base-10:

Expand the positional representation : 2*16^1 = 2*16 = 32

5. Consider the following short C program.

```
#define DATAMAX 256

/*
 * Process chunks of data received from the user.
 * Pre:
 *    The first 2 bytes in *buf are the size of *buf.
 *    The remaining size - 2 bytes are data
 *
 */
void process_user_data(char * buf) {
      char tmpbuf[DATAMAX];

      /* Get the data size. */
      uint16_t size = *((uint16_t *) buf);

      /* Move to the data. */
      buf += 2;

      /* Copy into a temporary location for processing */
      memcpy(tmpbuf, buf, size);

      /* process the data somehow ... not relevant */
}
```

a) [8 points] There is a buffer overflow in this code, analyze the code and determine the conditions where the buffer overflow occurs. You may assume the input buffer is correctly formatted. **Be specific, vague answers will receive no credit. Hint: the implementer has assumed something not guaranteed by the preconditions or the buffer's format.**

There are two:

The intended issue was that the implementer has assumed the size of the data is always less than or equal 256 bytes, despite a potential size of 64k - 2 (uint16_t). This could copy a potentially huge buffer into tmpbuf, which is only 256 bytes long. So you could overwrite important data in your program, and possibly (probably) cause a segmentation fault.

The other issue was unintentional, but buf has been incremented by two, yet `size` bytes are being copied going beyond the end of buf. Only `size - 2` bytes should be copied, so this another buffer overflow. However, `buf +=2` does NOT cause the issue.

Either option was acceptable.

b) [7 points] How could you fix the problem while keeping the same functionality?

There are a number of different ways to solve the issue: comparing the user provided size with DATAMAX and making sure there is enough space in tmpbuf or dynamically allocating space for tmpbuf. If you noticed the other overflow, fixing the memcpy with the correct size was also acceptable. Any reasonable solution received credit here.

6. [10 points] C Attributes. Consider the following short C program.

```
#include <stdio.h>
#include <stdint.h>

int y = 0; /* static storage, file scope */

void mystery(uint32_t num)
{
        static int x = 0; /* static storage, file scope
                             x is initialized only one time !!! */
        int z = 0; /* automatic storage, file scope */

        /* bodiless for loop, still increments x*/
        for(; x < num; ++x, ++y, ++z) { } /* only x is used as the loop end condition */

        printf("x is:  %d, y is:  %d, z is:  %d\n", x, y, z);
}



int main()
{
        mystery(12);
        mystery(7);
        mystery(25);

        return 0;
}
```

What would be the output of running the C program?

x is: 12, y is: 12, z is: 12
x is: 12, y is: 12, z is: 0
x is: 25, y is: 25, z is: 13

/* Note that x is initialized only one time when the function is called for the first time.
Here the value of x is preserved across the function calls, since it has a static storage.
That is, when the function is called with a parameter 7, x still has 12 which is
greater than the parameter. So, the loop immediately terminates. Similarly, when the function
is called with a parameter 25, the previous value of x, i.e., 12, is still there. Here, the loop
just iterates as many as the difference between 12 and the parameter, i.e., 12. */