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 ________________________________________

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

____________________________________________
signed
nano? REAL PROGRAMMERS USE emacs

HEY, REAL PROGRAMMERS USE vim.

WELL, REAL PROGRAMMERS USE ed.

NO, REAL PROGRAMMERS USE cat.

REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.

EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.

THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.

THE DISTURBANCE Ripples OUTWARD, CHANGING THE FLOW Of THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.

These CAUSE MOMENTARY POCKETS OF HIGHER-PRESSURE AIR TO FORM.

Which act as LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.

NICE. 'COURSE, THERE'S AN EMACS COMMAND TO DO THAT:

OH YEAH! GOOD OL' C.x M.c M-butterfly...

DAMMIT, EMACS.
1. [16 points] Implement the C function described below. Be sure to note the restrictions if any, violating the restrictions will result in a very low score.

```c
// "Doubles" every byte in a bytes_t type.
//
// Pre:
//   bytes - Points to a valid bytes_t variable and is not NULL.
//
// Post:
//   Every byte in *bytes has been duplicated, and bytes->usage has been
//   updated to match the altered number of bytes.
//
// You may assume bytes->dim is always large enough, so you DO NOT need
// to do any dynamic allocation, at least for bytes->data.
//
// Examples:
//   If *bytes contains {0xde} the result would be {0xde, 0xde}.
//   If *bytes contains {0xde, 0xad} the result would be {0xde, 0xde, 0xad, 0xad}.
//
// You may use any function from your bytes_t type (or none).
void bytes_double_bytes(bytes_t * const bytes)
{
    bytes_t copy;
    bytes_init(&copy);

    for (size_t b = 0; b < bytes->usage; b++)
    {
        bytes_insert(&copy, bytes_usage(&copy), bytes.data + b, 1);
        bytes_insert(&copy, bytes_usage(&copy), bytes.data + b, 1);
    }

    bytes_copy(bytes, &copy);
    bytes_free(&copy);
}
```
2. [12 points] Analyze the C function given below.

```c
void mystery(int *a, int *b)
{
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
```

Determine the operation this function performs and justify or explain how you came to your conclusion. The explanation will be worth more than the solution. This should not be a reiteration of each bitwise operation (\(^\) is the bitwise exclusive or operator), and should be specific.

This is a swap function that doesn't use an intermediate temporary variable. As for an explanation, there are many acceptable answers (mathematically, tables, examples). I like to break it down into:

- The bits that are initially the same in both \(*a\) and \(*b\).
- The bits that are initially different in both \(*a\) and \(*b\).

After the first XOR operation, \(*a\) still contains important information. The bits that were the same in both \(*a\) and \(*b\) are 0 in \(*a\), while bits that were different in \(*a\) and \(*b\) are 1 in \(*a\).

When performing the second XOR, for the bits that were the same (now 0 in \(*a\)):

- If bit \(*b_i\) is initially 0, \(*a_i \ ^{\wedge} \ *b_i = 0 \ ^{\wedge} 0 = 0\).
- If bit \(*b_i\) is initially 1, \(*a_i \ ^{\wedge} \ *b_i = 0 \ ^{\wedge} 1 = 1\).

So the bits that were the same in both, stay the same in \(*b\).

For the bits that were different in \(*a\) and \(*b\) (now 1 in \(*a\)):

- If bit \(*b_i\) is initially 0, \(*a_i \ ^{\wedge} \ *b_i = 1 \ ^{\wedge} 0 = 1\).
- If bit \(*b_i\) is initially 1, \(*a_i \ ^{\wedge} \ *b_i = 1 \ ^{\wedge} 1 = 0\).

So the bits that were different in \(*a\) and \(*b\) and are flipped in \(*b\) (to the original value of \(*a\)).

Since the differences have flipped (if \(*a_i\) was 0 and \(*b_i\) was 1, \(*b_i\) is now 0, etc.) and the other bits stayed the same, \(*b\) now contains the original value of \(*a\).

The third XOR operation behaves the same way, flipping the different bits, by using the information \(*a\) to set \(*a\) to the original value of \(*b\).
3. [16 points] Write a solution for this function and explain the logic that makes it work. The explanation will be worth more than the solution. You may not use any loops (while, for, do-while) or selection (if, switch), and you may not write any constants that are more than one byte wide. Don’t worry about the number of operations. Be sure to note the restrictions, violating the restrictions will result in a very low score.

// rotate_right - "rotate" the bits in *x right. This operation is kind of like
// a right shift, but the bits that are "shifted off" become the high order bits.
//
// Pre:
//   x points to a signed 32-bit integer value and is not NULL.
//   amt >= 1 and amt <= 31
//
// Post:
//   *x is rotated right by amt.
//
// Examples:
//   These rotate an 8 bit number for clarity. *x is a signed 32 bit number.
//   1110 0000 rotated right by 3 would produce 0001 1100.
//   1110 0000 rotated right by 4 would produce 0000 1110.
//
// Legal ops: ! ~ & ^ | + - << >> =

void rotate_right(int *x, int amt)
{
    int mask = 1 << 31;

    // Create a mask where amt high bits are 0.
    // There's already one 1 in mask, so subtract 1.
    int high_mask = ~(mask >> (amt - 1));

    // Create a mask where amt low bits are 1.
    // Minus 1 since already have the 1 in mask, and we want to shift
    // down to, but not including, the bits we want to save. So if
    // amt = 1, we wouldn't overwrite bit 0, etc.
    int low_mask = ~(mask >> (32 - amt - 1));

    // Grab the low bits.
    int bottom_bits = (low_mask & *x);

    // Perform the right shift, getting rid of the low bits.
    // & with high_mask getting rid of any extraneous leading 1s added
    // by the signed right shift.
    *x = (*x >> amt) & high_mask;

    // Shift the low bits up.
    // No minus one this time because we want to overwrite
    // all of those high bits.
    bottom_bits = (bottom_bits << (32 - amt));

    // Finish the rotate, the low bits go into the high position.
    *x = *x | bottom_bits;
}
4. [16 points] Implement the C function described below.

```c
// Counts all of the bits that are set to 0 in an array of 16 bit numbers.
//
// Pre:
//   buf points to an array that contains len numbers.
//
// Returns:
//   The number of bits that are set to 0 across all of the numbers in the
//   array.
//
// Examples:
//   When buf[] = {8, 4, 2, 1} the function would return 60.
//   (8 in binary is 0...1000, 4 is 0...100, 2 is 0...10, and 0...1 is 1,
//    so each number contains 15 0s and a 1, summed together produces 60)
//   When buf[] = {7} the function would return 13, since 7 is 111 (the rest
//    are zeros).

uint64_t count_zeros(uint16_t * buf, size_t len)
{
    uint16_t mask = 0x0001; // 0000 0000 0000 0001
    uint64_t count = 0;

    for (size_t x = 0; x < len; x++)
    {
        uint16_t copy = buf[x];

        for (size_t b = 0; b < 16; b++)
        {
            count += !(mask & copy);
            copy >>= 1;
        }
    }

    return count;
}
```
5. [12 points] Analyze the x86-64 assembly given below.

```
string_length:
  movl $0, %edx       # A local variable.
.L2:
  leaq 1(%rdx), %rax
  movzbl (%rdi,%rdx), %ecx     # %rdi is the parameter (the string).
  movq %rax, %rdx
  testb %cl, %cl
  jne .L2
  rep ret          # I look weird, but am basically a ret.
```

When you are asked to write C code suppose that any parameters are named P1, P2, etc., in the order the parameters would be listed in the C code. Suppose the local variables are called L1, L2, etc., in the order they occur in the x86-64 assembly, i.e. %edx (or %rdx) is the first local variable.

a. [6 points] Write C code for the function that could have yielded the x86-64 assembly code given above for string_length. Your final answer must not include any goto statements!

```
size_t string_length(const char * const str)
{
    size_t L1 = 0;
    do
    {
        L1++;
    }
    while(P1[LI++]  != '\0');
    return L1;
}
```

Or something like this:

```
size_t string_length(const char * const str)
{
    size_t L1 = 0;
    do
    {
        L1++;
    }
    while(P1[LI - 1]  != '\0');
    return L1;
}
```

b. [6 points] This function attempts to calculate the string length of the parameter, but does not work as intended. Using the assembly code above, explain why this function doesn't compute the correct string length. What does this function actually produce?

This function will count one character too many, so the empty string would produce a length of 1 (rather than 0), "a" would have length 2 (rather than 1), etc.

This happens because the length is incremented before the loop test occurs, so we don't find out if the string ended until after the increment (this is a do-while loop, rather than a while loop.)
6. [16 points] Suppose you are analyzing the C function below, but you only have object code for the functions `process_char` and `use_result` that are invoked within the function:

```c
size_t another_mystery(char * buf)
{
    size_t x = 0;
    size_t len = strlen(buf);
    size_t total = 0;

    while (buf[x] != '\0')
    {
        size_t result = process_char(buf[x]);
        total += use_result(result, len - x - 1);
        x++;
    }

    return total;
}
```

Examining the object files with `gdb` (or `objdump`) shows the assembly code for the functions:

```assembly
process_char:
    movsbl %dil, %eax  # %dil is the bottom 8 bits %rdi (the parameter).
    subl $48, %eax    # Conversion to size_t (8 byte value)
    cltq
    ret

use_result:
    movl $0, %ecx      #1 Local variable.
    movl $1, %edx      #2 Local variable.
    jmp .L3            #3
.L4:
    leaq (%rdx,%rdx,4), %rdx  #5 Think about what C operation is
    addq %rdx, %rdx   #6 performed by these 2 lines.
    addq $1, %rcx    #7
.L3:
    cmpq %rsi, %rcx   #9 %rsi is parameter 2
    jb .L4           #10
    movq %rdi, %rax   #11 %rdi is parameter 1
    imullq %rdx, %rax #12
    ret               #13
```

For the questions below, when you are asked to write C code suppose that any parameters are named P1, P2, etc., in the order the parameters would be listed in the C code. Suppose the local variables are called L1, L2, etc., in the order they occur in the x86-64 assembly, i.e. %ecx (or %rcx) is the first local variable.

a. [6 points] Write C code for the function that could have yielded the x86-64 assembly code given above for `process_char()`. This should be a one liner. Your final answer must not include any goto statements!

```c
size_t process_char(char P1)
{
    return P1 - '0';  // return c - 48 is also fine.
}
```
b. [6 points] use_result() contains a while loop or a do-while loop. Which type of loop is shown above? Justify your answer, and state the line numbers of the instructions that make up the loop, including the loop test and everything else necessary to execute the loop.

There is a while loop between lines #3 and #10; arguably #1 is included since that initializes a local variable used in the loop test. We know it's a loop because of the backwards branch in #10; we know it's a while loop because of the unconditional jump in #3 that takes us to the loop test before we enter the body of the loop the first time. The loop test occurs on lines #9 and #10.

c. [10 points] Write C code for the function that could have yielded the x86-64 assembly code given above for use_result(). Your final answer must not include any goto statements!

```c
size_t use_result(size_t P1, size_t P2)
{
    size_t L1 = 0
    size_t L2 = 1;

    while(L1 < P2)
    {
        L2 *= 10;
        L1++;
    }
    return P1 * L2;
}
```

d. [6 points] In plain English, explain what the C function (another_mystery()) accomplishes. This should not be a reiteration of each assembly instruction, and should be specific, i.e. this function sums the numbers from 1 - 100.

another_mystery() converts a string into a decimal number, character by character.

For each character, another_mystery() first converts it to a number using process_char(), so '0' becomes 0, '1' becomes 1, up to '9' turning into 9.

Next the function use_result() uses the newly converted number and the character's position in the string to determine the weight (i.e. whether the character is in the 1s place, the 10s place, the 100s place, etc.) and multiplies the weight and number.

Finally, all the values produced by use_result() are summed together, giving us the final number.