**Virginia Tech**
1872

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 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
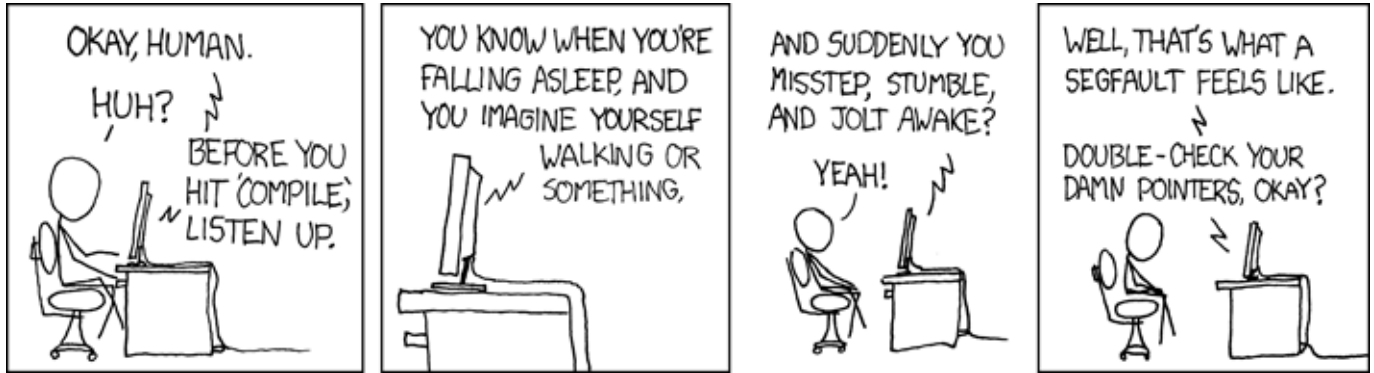
xkcd.com

1. **[12 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.

```c
/**
 *  Create and return an array holding the sums of the runs of odd elements
 *  in the array pointed to by A. A "run of odd elements" is a sequence of
 *  one or more odd values in adjacent array cells.
 *  Return NULL if A contains no runs of odd elements, or the array of sums
 *  cannot be created.
 *
 *  Pre:   A == NULL and Sz == 0, or A[0:Sz-1] have been initialized.
 *  Post:  *Sz equals the number of elements in the array created by the
 *         function (zero if no array is created).
 *  Returns:
 *     If A holds at least two elements, a pointer to an array that holds
 *     the sums of successive elements of A;
 *     The dimension of that array is as small as logically possible;
 *     If A holds fewer than two elements, NULL.
 *
 *  Examples:
 *     {7, 10, 9, 23, 50, 32, 17, 37, 11} yields {7, 32, 65}
 *     {16} yields NULL
 *
 *  Restrictions:  you must use only pointer notation when accessing
 *                 array elements (penalty is half off)
 */
int* SumEvens(const int* const A, int* const Sz) {

   int numRuns = 0;
   int idx = 0;

   int* sums = calloc(*Sz, sizeof(int));

   while ( idx < *Sz ) {

      while ( idx < *Sz && A[idx] % 2 != 1 ) {
         idx++;
      }
      if ( idx == *Sz ) break;
      numRuns++;

      while ( idx < *Sz && A[idx] % 2 == 1 ) {
         sums[numRuns-1] += A[idx];
         idx++;
      }
   }

   if ( numRuns == 0 ) {
      free(sums);
      sums = NULL;
   }
   else {
      sums = realloc(sums, numRuns);
   }

   *Sz = numRuns;
   return sums;
}
```

Some observations:

The preconditions and constraints in the header comments allow A to be NULL, in which case *Sz == 0, and also specify NULL should be returned if *Sz < 2. This implies that it's sufficient, and necessary, to test whether *Sz < 2.

The return specification requires that the function create an array to hold the sums, and return a pointer to that array, so the array must be allocated dynamically; this must be done AFTER the check on *Sz, since otherwise you'll either have a memory leak or an unnecessary allocation/deallocation.

The array of sums is also required to be of minimum size, which means we don't know the size of the array until we know how many "runs" of even values occur in A. There are basically three approaches:
- traverse A to count the runs, then allocate the array for the sums, and traverse a again to compute the sums and store them
- allocate an array for the sums that's sure to be big enough (*Sz would do, as would *Sz - 1), then traverse A computing sums and storing them, then shrink the array of sums to the proper size (ideally using realloc)
- allocate a small array to hold the sums, then traverse A computing the sums, and reallocate the array of sums every time it needs to grow

The third approach is wildly inefficient, and should not be used (but was not penalized). The second approach requires two essentially similar traversals of A, which is also inefficient. I used the first approach.

Nothing in the specification implies that the elements in A are nonnegative.

A large issue is to detect the beginning/end of each "run" of even values. Many solutions failed this part because they either depended on the sums being positive, or simply assumed that the detection of an odd (even for form B) value implied the end of a run, which is false.

Aside from that, there are various issues such as resetting *Sz at the end, avoiding memory leaks, etc.

2. **[8 points]** A C programmer wants to design a function `G()` that satisfies the following requirements:

    -   `G()` uses two values supplied by the caller: a signed 32-bit integer value and a value of type double
    -   `G()` returns 0 if it succeeds and 1 if it fails.
    -   `G()` sets a signed 32-bit integer variable in the calling code to a result computed from two values mentioned above, if possible; but it's not always possible, in which case G() does not change the caller's variable

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

```
int32_t G(int32_t* x, double d);
```

The return type must be some integer type. There must be a parameter that supplies a signed 32-bit integer value, and one that supplies a double; there is no reason whatsoever for the double to be passed by pointer, and I penalized that approach unless const was used. There must also be a parameter that allows the function to potentially modify a signed 32-bit integer in the calling code; that can be combined with the need to supply a signed 32-bit value as input, or handled separately, since the description leaves the issue unclear.

3. **[12 points]** Write an implementation of the following C function. You must obey the stated restrictions. Use of forbidden C operators will result in a score of 0.

```
/** Determines whether its parameter is non-negative and odd.
 *
 *  Pre:  N has been initialized.
 *
 *  Returns: 1 if N is non-negative and odd, 0 otherwise
 *
 *  Restrictions:
 *    You may only use the following C operators:
 *       =, &, |, ^, ~, <<, >>
 *
 *    In particular, you may NOT use any arithmetic operators.  You also
 *    may not use any control structures (if, if-else, while, for, do-while).
 *    You may declare constants and local variables as needed.
 */
int16_t isOddNonNegative(int16_t N) {

   // N is odd iff its low bit equals 1, so let's get the low bit
   uint16_t loBit = 0x0001 & N;

   // N is nonnegative iff its high bit equals 0, so let's get the high bit;
   //   and, shift it down because the return value needs to be 0 or 1.
   // However, if we shift a signed value to the right, we might shift in
   //   1's at the top; I'm going to prevent that by using an unsigned
   //   type to hold the result of the bitmask operation.
   uint16_t hiBit = (0x8000 & N) >> 15;

   // N is nonnegative and odd iff loBit is 1 and hiBit is 0;
   //   since I made sure the high bits of hiBit are all 0's, I don't
   //   need to worry about making the high bits of loBit 1's.
   return ( loBit & ~hiBit );
}
```

Some observations:

The solution must examine both the high bit and the low bit of N, which requires different masks.

The return value must be 0 or 1, which requires doing some shifting; it's not acceptable to simply return 0 or a non-0 value.

Dereferencing a uint16_t* yields a 16-bit value. Shifting a 16-bit value by 16 (or more) bits, in either direction, will yield either -1 or 0.

4. Recall the `Untangle` assignment from earlier in the semester. In this question you will write C statements to perform similar tasks. You are given a pointer to a memory block which contains an unsigned 16-bit value storing the address of the first record in the memory block, followed by a sequence of records.

Each record is made up of **two unsigned 8-bit integer values, followed by an unsigned 16-bit value**. In the example memory dump below the 8-bit fields in each record are **bolded** and the 16-bit field is *italicized*. The address of the first record is given by the 16-bit value at the beginning of the memory block. The 16-bit field in each record stores the offset of the next record in the memory block. Here is a sample memory block:

```
Offsets    00 01 02 03 04 05 06 07  08 09 0A 0B 0C 0D 0E 0F
-----------------------------------------------------------
00000000   16 00 23 54 0a 00 fb 7a  0e 00 a2 5d 2a 00 e9 64  |..#T...z...]*..d|
00000010   5e 00 db 82 22 00 16 98  6a 00 a4 20 72 00 cb 57  |^..."...j.. r..W|
00000020   76 00 2d 28 26 00 46 01  32 00 af 3b 06 00 56 66  |v.-(&.F.2..;..Vf|
00000030   46 00 9a a3 36 00 56 c5  62 00 5d bc 2e 00 ba 58  |F...6.V.b.]....X|
00000040   12 00 18 f1 4a 00 09 89  7e 00 35 5b 66 00 6a b6  |....J...~.5[f.j.|
00000050   6e 00 3e a0 4e 00 92 4b  5a 00 2a d9 1a 00 70 c5  |n.>.N..KZ.*...p.|
00000060   3e 00 9b 76 42 00 09 4b  16 00 c8 c2 1e 00 e0 fe  |>..vB..K........|
00000070   56 00 59 fd 02 00 8e 30  7a 00 c6 47 3a 00 5e 38  |V.Y....0z..G:.^8|
00000080   52 00                                             |R.|
```

a) **[6 points]** Suppose that `p` is a **`uint8_t`**`*` and that `p` points to the first byte in the given memory region. Write the necessary C code so that `pCurr` points to the starting record:

```
uint8_t* pCurr;               // pointer to starting record (see above)


pCurr = p + *(uint16_t*) p;
```

You must cast p so that dereferencing it will yield a 2-byte value. The result must be added to the base address of the memory block in order to access the specified record.

Remember: shifting an 8-bit value by 8 bits yields either -1 or 0; if you used such a shift to try to combine the individual bytes of the offset field, it will not work unless the high byte is already 0.

b) **[10 points]** Assuming `pCurr` does point to the first byte of a record, write the necessary C code to copy the fields of that record into the variables declared below.

```
uint8_t  x;             // first 8-bit field of record
uint8_t  y;             // second 8-bit field of record
uint16_t offsetNextBlock;     // 16-bit field of record

x = *pCurr;
y = *(pCurr + 1);
offsetNextBlock = *(uint16_t*)(pCurr + 2);
```

or, if you don't mind changing the value of pCurr:

```
x = *pCurr;
pCurr++;
y = *pCurr;
pCurr++;
offsetNextBlock = *(uint16_t*)pCurr;
```

5. Consider the x86-64 translation of a short C function shown at right.

Assume that all parameters and local variables are of type `int64` or `int64*`.

a) [6 points]
Recall that parameters are passed via registers. Call the parameters `Param1`, `Param2`, and so forth, in the order they appear in the parameter list.

How many parameters does the function `oddball()` receive?

4

State the stack address at which each parameter is stored (e.g., `Param5:  rbp – 128`).

Param1: rbp-16
Param2: rbp-24
Param3: rbp-48
Param3: rbp-56

As noted in the question, parameters are passed in my registers, in particular a specific set and ordering of registers. Lines #4-7 in the code to the right show values read from these registers (rdi, rsi, rdx, rcx) and placed into the stack.

b) [4 points]
Call the local variables `LocalXY`, where `XY` is the offset on the stack relative to `$rbp`.
How many local variables does the function `oddBall()` have?
Do not include parameters!

1

State the number of the instruction at which each local variable is first used (e.g., `Local16: #12`).

Local32: #8

Local variables are also stored in the stack with some offset from the base pointer rbp, but have values set in the code itself rather than having values passed in from outside via register. The only other base pointer offset present in the code is rbp-32, first used on line #8 when an initial value of 0 is placed inside.

```
      . . .
oddBall:
    pushq    %rbp              #  1
    movq     %rsp, %rbp        #  2
    subq     $64, %rsp         #  3
    movq     %rdi, -16(%rbp)   #  4
    movq     %rsi, -24(%rbp)   #  5
    movq     %rdx, -48(%rbp)   #  6
    movq     %rcx, -56(%rbp)   #  7
    movq     $0, -32(%rbp)     #  8
    jmp      .L2               #  9
.L4:                           # 10
    movq     -16(%rbp), %rax   # 11
    movq     (%rax), %rax      # 12
    andl     $1, %eax          # 13
    testq    %rax, %rax        # 14
    jne      .L3               # 15
    movq     -16(%rbp), %rax   # 16
    movq     (%rax), %rax      # 17
    addq     %rax, %rax        # 18
    leaq     1(%rax), %rdx     # 19
    movq     -16(%rbp), %rax   # 20
    movq     %rdx, (%rax)      # 21
    movq     -48(%rbp), %rax   # 22
    movq     (%rax), %rax      # 23
    leaq     1(%rax), %rdx     # 24
    movq     -48(%rbp), %rax   # 25
    movq     %rdx, (%rax)      # 26
.L3:                           # 27
    addq     $4, -16(%rbp)     # 28
    addq     $1, -32(%rbp)     # 29
.L2:                           # 30
    movq     -32(%rbp), %rax   # 31
    cmpl     -24(%rbp), %rax   # 32
    jl       .L4               # 33
    leave                      # 34
    ret                        # 35
      . . .
```

c) **[6 points]** Examine the given x86-64 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)?

We accepted three answers as correct for full points: lines 14-27, 11-27, and 15-27.

Lines 11-13 move values around from the stack and into registers in preparation for the boolean test, the comparison is performed on line 14, the jump decision is made on line 15, lines 16-26 run conditionally, and line 27 provides a label to jump to if the conditional code is not executed.

It is incorrect to include lines 28-29 as a part of the if structure, because they run regardless of the decision in lines 14-15. Note that execution passes straight from line 26 to line 28 across the label; there is no jump to skip those lines. We also docked a point for not including line 27, as the question specifies to include relevant labels.

Arguably, lines 31-33 constitute an if decision, but this should more accurately be considered a loop test for the while/for loop. Some points were awarded for noting this decision but not the larger if structure.

d) **[6 points]** Examine the given x86-64 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)?

We accepted three answers as correct for full points: lines 10-33, 9-33, and 8-33.

Lines 31-33 constitute the loop test to determine if the code should continue running the loop or exit. If the code does continue running through the loop, execution jumps to the L4 label on line 10, and everything between 10 and 31 belongs to the loop body. Line 9 could also be considered a part of the loop structure, as that unconditional jump moves execution straight to the loop test to determine whether or not to enter the loop. Line 8 could be considered a part of the loop structure as well, depending on whether you think of the loop as a for loop (initializing the loop counter is a part of the loop) or a while loop (initializing the loop counter happens before the loop).

Skipping line 10 or line 33 lost a point, as these are both important components of the loop. Skipping the loop test on lines 31-33 and only providing the loop body also lost points, as the question specifies that the loop test should be included in your answer. Including lines 34-35 is incorrect, as these instructions executed outside of the loop.

e) **[4 points]** Examine the given x86-64 code. How many bytes are in the stack frame for `oddBall`, including the backed up value for the frame pointer `rbp`? Justify your answer.

72 bytes

We can determine the size of the stack frame for oddBall simply by looking at the function setup code in the first few lines. In line 1, the old rbp location is backed up on the stack, requiring 8 bytes. In line 2, a new value of rbp is set, which does not affect the stack size. In line 3, rsp is moved by 64 bytes to create additional space on the stack for variable storage.

The two most common mistakes were (1) treating the rbp backup as 4 bytes rather than 8 (note that the instruction is pushq), and performing an unnecessary division by 8 to convert bits to bytes (all values in the code are given in bytes).

6. A C programmer is implementing a scatterplot, beginning with using a **struct** called Point for each data item in the scatterplot. Each Point needs to store its x and y position on the screen (which can be represented as an unsigned 16-bit integers screenX and screenY), as well as the underlying x and y data values (which can be represented as standard signed double variables dataX and dataY).

a) **[6 points]** Write a **struct** declaration for Point given the specified variables and formats above. The **struct** should be named _Point and then aliased to Point afterward using **typedef**.

```
struct _Point {
    uint_16_t screenX, screenY;
    double dataX, dataY;
};
typedef struct _Point Point;
```

We were looking for the following 6 items in your code: (1) attempting the typedef instruction, (2) writing a syntactically correct typedef instruction, (3) use of curly braces around the struct, (4) a semicolon at the end of the struct, (5) otherwise using correct struct syntax, and (6) using the datatypes as specified.

The most common mistakes were forgetting the semicolon at the end of the struct, and forgetting the keyword "struct" in the typedef statement. These, and similar issues, resulted in point deductions.

b) **[6 points]** Each Point in the scatterplot needs to be initialized, which will be performed by a function called initPoint(). To do so, the programmer needs to supply only the dataX and dataY data values; the screenX and screenY position in the scatterplot will be computed by another function called calcPosition(), which is called by initPoint(). Write an implementation for initPoint() that conforms to the given interface and header comments. You do not need to provide an implementation for calcPosition().

The declaration for calcPosition() is: void calcPosition(Point* **const** input).

```
/**
 *  Create and return a Point that stores the (x,y) scatterplot position
 *  and (x,y) data value of a data item.
 *
 *  Pre:     dX and dY contain the (dataX,dataY) data values for the item
 *  Post:    A Point has been created, using calcPosition() to determine the
 *           (screenX,screenY) scatterplot positions
 *  Returns: A Point that contains (screenX,screenY) scatterplot position and
 *           (dataX,dataY) parameter data values for the item
 *
 *  Example:
 *    Inputs {-16.2, 15.0} yields a Point with (dataX,dataY) data values
 *    {-16.2, 15.0} and (screenX,screenY) position values {118, 76}
 */
Point initPoint(double dX, double dY) {
    Point newPoint;
    newPoint.dataX = dX;
    newPoint.dataY = dY;
    calcPosition(&newPoint);
    return newPoint;
}
```

We looked in your code for this question for the following four items: (1) creating a locally-scoped Point variable, (2) assigning the dX and dY parameters appropriately to the dataX and dataY variables of the Point, (3) correct use of calcPosition (including passing in an address), and (4) returning the Point variable to the calling function.

A number of mistakes were common throughout the exams, each of which resulted in point deductions. These include:
- Use of pointers. Either this was implemented with a statically-allocated pointer that was created and returned (in which case the reference is destroyed when the function exits), or this was implemented using malloc to create a dynamically-allocated pointer (in which case you have caused a memory leak because this pointer never gets deallocated).
- Similar to the above, many exams that used pointers returned a Point* rather than a Point.
- Passing a Point into calcPosition() rather than the address to a Point.
- Passing anything other than a Point* into calcPosition() (such as a double).
- Treating calcPosition as if it has a return value (note that it is a void function).
- No local variable declaration.
- Using "struct Point" rather than "Point" or "struct _Point" in the variable declaration.

One item that did NOT lose points was calling calcPosition() before setting the dataX and dataY values of the Point. This was intended, but not 100% clear in the header comment.

c) **[6 points]** Next, the programmer needs a data structure to store all of the points in the scatterplot, and decides to create a **struct** called Scatterplot that stores a dynamically allocated array of Points, the number of points in the array (as a 16-bit unsigned integer called usage), and the maximum number of Points that can be stored in the array (as a 16-bit unsigned integer called dimension).

Write a **struct** declaration for Scatterplot given the specified variables and formats above. The **struct** should be named _Scatterplot and then aliased to Scatterplot afterward using **typedef**.

```
struct _Scatterplot {
    Point* data;
    uint16_t usage;
    uint16_t dimension;
};
typedef struct _Scatterplot Scatterplot;
```

Grading for this question was nearly identical to that for 6a. Note that the question required dynamic allocation of an array of Points. Using array bracket notation was incorrect; a pointer is required here.

If you missed two or more little things is 6a and made the same mistakes here in 6c, you only lost one point.

d) [8 points] Finally, write an implementation for a `ScatterplotAppend()` function, which will add a new `Point` to the scatterplot and will double the size of the array if the array is full. Your implementation must conform to the given interface and header comments.

```
/**
 *  Appends a new Point to the scatterplot, doubling the size of the array
 *  stored within the Scatterplot struct if the array was full
 *
 *  Pre:     splot is a Scatterplot that has been initialized, and may or may
 *           not already contain Points inside
 *  Post:    newPoint has been appended to splot, the usage variable has been
 *           incremented, and the size of the array and variable dimension
 *           have been updated if the array was full.
 *  Returns: TRUE if the Point was successfully appended to splot, FALSE
 *           otherwise (e.g., if memory reallocation failed)
 */
bool ScatterplotAppend(Scatterplot* const splot, Point newPoint) {
    if (splot->dimension == splot->usage) {
        Point* temp = realloc(splot->data, 2*splot->dimension);
        if (splot->data == NULL) {
            return false;
        }
        splot->data = temp;
        splot->dimension = 2*splot->dimension;
    }
    splot->data[splot->usage] = newPoint;
    splot->usage++;
    return true;
}
```

We looked in your code for the following 8 items: (1) check to see if resize is needed, (2) realloc, (3) check to see if realloc returned NULL and if so return false, (4) copy data back (splot->data = temp), (5) double size of dimension, (6) add new point, (7) increment size of usage, (8) return true. Each of these were one point.

Common mistakes that resulted in point deductions included incorrect use of precedence with pointers [using *var. rather than (*var).], incorrect use of operators (using = rather than == for comparisons), and incorrect overall structure (e.g., all operations inside the dimension==usage check, leaving no code to be executed if usage<dimension).