Instructions:

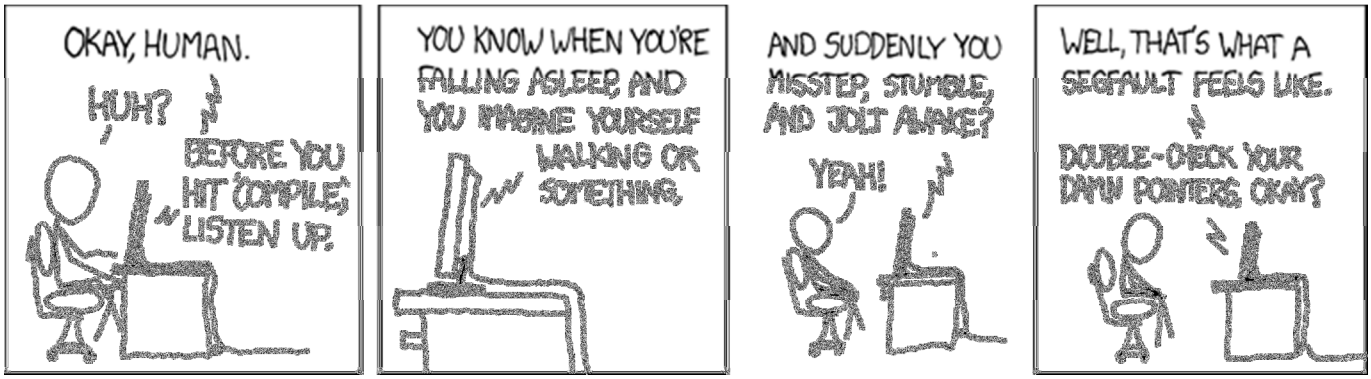
- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula sheet. No calculators or other computing devices may be used. The use of any such device will be interpreted as an indication that you are finished with the test 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 5 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) [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. Recall that:

$$Fibonacci(N) = \begin{cases} 1, \text{ if } N = 0,1 \\ Fibonacci(N-2) + Fibonacci(N-1), \text{ if } N > 1 \end{cases}$$

Be careful to avoid memory leaks.

```
/**
 *  Computes Fibonacci_N, iteratively, using a dynamically-allocated array
 *  to store Fibonacci_0 through Fibonacci_N as they are calculated.
 *
 *  Returns:  if no overflow, the N-th Fibonacci number;
 *            if overflow, probably the incorrect value
 */
uint64_t Fibonacci(uint16_t N) {

   // write the body of your function below:

   if ( N < 2 ) return 1;

   uint64_t Fibo = malloc( (N + 1) * sizeof(uint64_t) );
   if ( Fibo == NULL ) return 0;

   Fibo[0] = Fib0[1] = 1;

   for ( int i = 2; i <= N; i++ ) {

      Fibo[i] = Fibo[i-1] + Fibo[i-2];

   }

   uint64_t FiboN = Fibo[N];

   free(Fibo);

   return FiboN;
}
```

b) [8 points] A C programmer wants to design a function `F()` that satisfies the following requirements:

- `F()` takes two signed 32-bit integer values as parameters. `F()` may nor may not have additional parameters.
- `F()` returns 0 if it succeeds and 1 if it fails.
- `F()` sets a 32-bit integer variable in the calling code to a result computed from two parameters mentioned above, if possible (but it's not always possible).

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.

```
int F(int A, int B, int* const pC);
```

2. Consider the following C struct and its corresponding initialization function. You may assume `Sent` is a correctly formed and properly terminated string.

```
struct _String {
     int len;
     int size;

     char *sentence;
};

typedef struct _String String;

/* Initializes a String with the value of Sent */
String_Init(String * const pSTR, const char * const Sent)
{
     pSTR->len = strlen(Sent);
     pSTR->size = pSTR->len + 1;

     pSTR->sentence = (char*) malloc(sizeof(char) * (pSTR->size));
     strncpy(pSTR->sentence, Sent, pSTR->size);
}
```

a) [8 points] Assume you have two structures of type `String` and that have both been properly initialized using the `String_Init` function.

```
String s1, s2;

/* initialization */
String_Init(&s1, "This is a test");
String_Init(&s2, "Another test");

/* make s2 a copy of s1 */
s2 = s1;
```

This will compile, but in this case will not work in the way the programmer intended. Explain why this doesn't work and any other issues that occur.

This compiles and does make a copy of s1, but it is a shallow copy. This works fine for structures that contain only values (i.e. int x) but doesn't work so well with pointers or arrays.

In this example, s2 receives a copy of the values in s1. So copying the value "14" from s1.len into s2.len works fine, but copying s1.sentence into s2.sentence only copies the pointer address contained s1.sentence. Let's say s1.sentence is an address like 0xFFFF5641, s2.sentence will end up with a copy of that value, rather than a copy of the array of chars. They both point to the same region of memory and changes to one will result in changes to the other.

Further in the process of making the shallow copy the original malloc'd address is lost. This causes a memory leak.

b) [12 points] Write a correct copy function for the `String` structure. You must achieve the specified results, and you must conform to the given interface and header comments.

Be careful to avoid memory leaks.

```
/**
 *  Copies the values of a String struct from src into dest
 *
 *  Returns: NULL if the copy function encountered an error
 *           A pointer to dest if successful
 */
String* StringCopy (String* dest, const String* const src) {

      // write the body of your function below:

      assert(src != NULL);
      assert(dest != NULL);

      /* copy the statically allocated variables */
      dest->size = src->size;
      dest->len = src->len;

      /* if I don't free dest->sentence then we have a memory leak */
      free(dest->sentence);

      /* make space for the copied string in dest */
      dest->sentence = (char*) malloc(sizeof(char) * (src->size));
      strncpy(dest->sentence, src->sentence, src->size);



      return dest;
}

/* I also accepted using the String initialization function */
/* after calling "free" i.e: */

free(dest->sentence);
String_Init (dest, src->sentence);
```

3. Consider the following x86-32 translation of a short C function. Assume that all parameters and local variables are of type `int`.

```
        .file "Q3.c"
        .text
.globl Q1
        .type Q3, @function
Q3:
        pushl   %ebp                    #  1   stack frame setup
        movl    %esp,       %ebp        #  2
        subl    $16,        %esp        #  3
        movl    $0,     -4(%ebp)        #  4   set local V0 to 0
        movl    $0,     -8(%ebp)        #  5   set local V1 to 0
        movl     8(%ebp), %eax          #  6   eax = parameter P0
        cmpl    12(%ebp), %eax          #  7   compare P0 to parameter P1
        jg     .L2                      #  8   goto L2 if P0 > P1
        movl    12(%ebp), %eax          #  9   eax = P1
        movl    %eax,   -4(%ebp)        # 10   V0 = P1
        movl     8(%ebp), %eax          # 11   eax = P0
        movl    %eax,   -8(%ebp)        # 12   V0 = P0
        jmp    .L3                      # 13   goto L3
.L2:
        movl     8(%ebp), %eax          # 14   eax = P0
        movl    %eax,   -4(%ebp)        # 15   V0 = P0
        movl    12(%ebp), %eax          # 16   eax = P1
        movl    %eax,   -8(%ebp)        # 17   V1 = P1
.L3:
        movl    -8(%ebp), %eax          # 18   eax = V1
        movl    -4(%ebp), %edx          # 19   edc = V0
        movl    %edx,       %ecx        # 20   ecx = V0
        subl    %eax,       %ecx        # 21   ecx = V0 - V1
        movl    %ecx,       %eax        # 22   eax = V0 - V1 (return value)
        leave                           # 23   return
        ret                             # 24
```

a) [4 points] How many parameters does the function `Q3()` receive? State the stack address at which each parameter is stored (e.g., `%ebp - 48`).

Parameters are passed on the stack, in the caller's frame, above the return address.

In this case, there are two parameters, at addresses `%ebp + 8` and `%ebp + 12`.

b) [4 points] How many local variables does the function `Q3()` have? State the stack address at which each local variable is stored (e.g., `%ebp - 48`).

Local (automatic) variables are stored within the frame for the called function.

In this case, there are two local automatic variables, stored at `%ebp - 4` and `%ebp - 8`.

c) [12 points] Analyze the given x86-32 code and write an equivalent C function. Your final answer should not contain any `goto` statements. Name parameters `P0`, `P1`, `P2`, etc., in reverse of the order they occur on the stack (i.e., `P0` is the first parameter in C code, etc.). Name local variables `V0`, `V1`, `V2`, etc., in the order they occur on the stack.

```
int Q3(int P0, int P1) {

   int L0 = 0;
   int L1 = 0;

   if ( P0 <= P1 ) {
      V0 = P1;
      V1 = P0;
   }
   else {
      V0 = P0;
      V1 = P1;
   }
   return (V0 - V1);
}
```

The analysis is in the comments on the previous page.

4. A C programmer has implemented the following function:

```
/**
 *  Compares two zero-terminated C strings.
 *
 *  Pre:  p, q point to zero-terminated strings or are NULL
 *  Returns:  1 if the strings are equal
 *            0 if the strings are not equal
 */
int Q4(const char* p, const char* q);
```

Concerned that the implementation may not be fully correct, the programmer writes a short test program:

```
int main() {

   char *p = "string one";
   char *q = "string two";

   char *s1 = p;
   char *s2 = q;

   if ( Q4(s1, s2) ) {
      printf("'%s' and '%s' are equal.\n", s1, s2);
   }
   else {
      printf("'%s' and '%s' are not equal.\n", s1, s2);
   }

   return 0;
}
```

When the program is compiled and executed, the output is:

```
'string one' and 'string two' are not equal.
```

That is correct, which is moderately comforting. The programmer then edits the `main()` function shown above so that both parameters in the call to `Q4()` are both `s1`. Running that yields the output:

```
'string one' and 'string one' are not equal.
```

That's disturbing. The programmer decides to use `gdb` to analyze what's gone wrong. The session is shown below, with some irrelevant parts omitted:

```
2069: Q4 > gdb Q4
(gdb) break main
Breakpoint 1 at 0x4004fc: file Q4.c, line 7.
(gdb) run
Starting program: /home/johokie/Q4/Q4
Breakpoint 1, main () at Q4.c:7
7                   char *p = "string one";

(gdb) next
8                   char *q = "string two";
(gdb) next
10                       char *s1 = p;
```

```
(gdb) print p
$1 = 0x4006fc "string one"
(gdb) print q
$2 = 0x400707 "string two"
```

That shows that the two strings are indeed pointed to by the variables `p` and `q`, respectively.

```
(gdb) next
11                      char *s2 = p;
(gdb) next
13                  if ( Q4(s1, s2) ) {
(gdb) print s1
$3 = 0x4006fc "string one"
(gdb) print s2
$4 = 0x4006fc "string one"
```

That shows that the two pointers `s1` and `s2` have been set correctly, in preparation for the call to `Q4()`. Now, the programmer steps into the call to `Q4()`:

```
(gdb) step
Q4 (p=0x4006fc "string one", q=0x4006fc "string one") at Q4.c:25
25          if ( p == NULL && q == NULL) return 1;
(gdb) next
26          if ( p == NULL || q == NULL) return 0;
(gdb) next
28           while ( *p != '\0' && *q != '\0' ) {
```

a) [6 points] What do the three commands above tell you about the parameters used in this call to `Q4()`? Be complete.

The first display shows that the parameters to `Q4()` are pointers to the same address, and that the string (starting) at that address is "`string one`".

The next two steps verify that neither `p` nor `q` is `NULL` (which we knew from the earlier display).

Now, the programmer continues, stepping through the execution of `Q4()`:

```
(gdb) next
29                      if ( *p != *q ) {
(gdb) break 29
Breakpoint 2 at 0x4005ac: file Q4.c, line 29.
(gdb) print *p
$5 = 115 's'
(gdb) print *q
$6 = 115 's'
```

b) [6 points] What do the three commands above tell you? Be complete.

The first executes the loop test shown in the previous section, which confirms that neither of the targets of `p` and `q` are equal to the string terminator '`\0`'.

The next two commands show that the targets are both equal to the ASCII code for '`s`'.

Now, the programmer continues, stepping through the execution of `Q4()`, running back to the breakpoint at the test inside the `while` loop:

```
(gdb) continue
Continuing.

Breakpoint 2, Q4 (p=0x4006fe "ring one", q=0x4006fc "string one") at Q4.c:29
29                      if ( *p != *q ) {
(gdb) print *p
$7 = 114 'r'
(gdb) print *q
$8 = 115 's'
(gdb)
```

c) [8 points] At this point, the programmer realized what the error was. What do the three commands above tell you? Be complete.

When the breakpoint is reached, we see the values of the pointers:

```
p = 0x4006fe
q = 0x4006fc
```

Now, previously we had:

```
p = 0x4006fc
q = 0x4006fc
```

So, the value of `p` has increased by 2 and the value of `q` has not changed at all.

(That explains the string displays. `p` is now pointing to the character '`r`' in the string "`string one`" and `q` is still pointing to the character '`s`' in the same string.)

We would expect that the programmer would have stepped each pointer forward one position within the string; it appears the programmer stepped `p` forward twice and failed to step `q` forward at all.

It could be the programmer wrote:

```
p++;
p++;
```

instead of:

```
p++;
q++;
```

5. C bitwise operations

a) [8 points] Complete the C function below that takes a single parameter of type `int` and returns 0 if the parameter is non-negative and 1 if the parameter is negative.

```
/**
 *  Determine if an int is negative.
 *  Returns:  1 if N < 0; 0 if N >= 0
 *  Restrictions:
 *    You may use only the following operations:
 *               ~ & | ^ << >>
 *    You may not use any loops or selection mechanisms.
 *    You may not use literal constants wider than 1 byte, but
 *    you may shift them as much as you like.
 */
int isNegative(int N) {

   // The sign is indicated by whether the high bit is 0 or 1.

   // We can shift to put the high bit in position 0, but we will
   // have shifted in 1's if N < 0.
   // So, we need to apply a mask to guarantee the final result
   // is 0x00000001, not 0xFFFFFFFF.

   int mask  = 0x01;           // 0x00000001
   int hiBit = N >> 31;        // 0x00000001 or 0xFFFFFFFF

   return hiBit & mask;        // 0x00000000 or 0x00000001
}
```

b) [12 points] When adding unsigned integer values, like `uint32_t`, overflow occurs when there is a carry out of the high bit position in the answer. Complete the C function below that takes two parameters of type `uint32_t` and returns 0 if adding them would not result in overflow and 1 if adding them would result in overflow.

Note: this is not the same as any function in the Data Lab assignment.

```
/**
 *  Determine if addition will result in overflow.
 *  Returns:  1 if x + y causes overflow, 0 otherwise
 *  Restrictions:
 *    You may use only the following operations:
 *               ~  &  |  ^  <<  >>  ||  &&  +
 *    You may not use any loops or selection mechanisms.
 *    You may not use literal constants wider than 1 byte, but
 *    you may shift them as much as you like.
 */
uint32_t willOverflow(uint32_t x, uint32_t y) {

   // With unsigned integers, overflow occurs iff there is a carry out of
   // the high position.
   // The problem is that if you just add x and y, that carry (if it occurs)
   // will be lost.
   // One solution is to do the addition with 64-bit values:

   uint64_t sum = (uint64_t) x + (uint64_t) y;
   return ( sum >> 32 );



   // Another solution determines the carry bit logically.  You get a carry
   // from the high position if the high bits of x and y are both 1, or if
   // the high bit of x or y is 1 and there is a carry into the high position.
   // We can isolate the high bits easily enough:

   uint32t hiX = ( x & ( 0x80 << 24) ) >> 31;
   uint32t hiY = ( y & ( 0x80 << 24) ) >> 31;

   // We can determine if there's a carry into the high position by being a
   // bit sneaky (sorry):

   uint32_t mask = (0x7F << 24) | (FF << 16) | (FF << 8) | FF;
                   // 01111111 11111111 11111111 11111111

   x = x & mask;               // high bit is now 0
   y = y & mask;               // high bit is now 0

   uint32_t sum = x + y;       // cannot overflow

   uint32_t hiSum = (sum & (0x80 << 24) ) >> 31;

   return ( ( hiX & hiY ) |                // case 1
            ( hiX | hiY ) & hiSum );       // case 2
}
```