NAME:
Login name:

Precept (circle one):
1 - Dondero M/W 1:30pm
2 - Dondero M/W 3:30pm
3 - MacDavid M/W 7:30pm
3A - Roberts M/W 7:30pm
4 - Petras T/Th 12:30pm
4A - Wolf T/Th 12:30pm
5 - Petras T/Th 1:30pm
5A - Oda T/Th 1:30pm
6 - Popovych T/Th 3:30pm
7 - Nagree T/Th 7:30pm
7A - Qiu T/Th 7:30pm

Computer Science 217
Final Exam
January 19, 2016
1:30-4:30pm

This test has nine (9) questions and nineteen (19) pages. Put your name (or login-id) on every page, and write out and sign the Honor Code pledge before turning in the test.

“I pledge my honor that I have not violated the Honor Code during this examination.”

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (8 points)</td>
<td></td>
</tr>
<tr>
<td>2 (10 points)</td>
<td></td>
</tr>
<tr>
<td>3 (10 points)</td>
<td></td>
</tr>
<tr>
<td>4 (10 points)</td>
<td></td>
</tr>
<tr>
<td>5 (8 points)</td>
<td></td>
</tr>
<tr>
<td>6 (14 points)</td>
<td></td>
</tr>
<tr>
<td>7 (14 points)</td>
<td></td>
</tr>
<tr>
<td>8 (20 points)</td>
<td></td>
</tr>
<tr>
<td>9 (6 points)</td>
<td></td>
</tr>
</tbody>
</table>

Total (100 points)
Question 1: Grab Bag (8 points)

1a) Suppose p is a character pointer initialized to char *p = “bob”. Then what does printf("%d", *(p+2) - *p) print? (1 point)

1b) What does printf("%d", ~ (15 ^ 12)) print? Show your work. (2 points)

1c) Explain the difference between statement testing and path testing. Which one requires more combinations of test inputs? (2 points)

1d) Suppose a four-byte unsigned integer i is initialized to 0x01020304 and an unsigned character pointer p is assigned as p = (unsigned char *) &i. What does printf("%d", *p) print on a little-endian machine? (1 point)

1e) Suppose the function foo(int *p1, int *p2) performs “*p1 += *p2; *p1 += *p2;”. When would the alternate implementation “*p1 += *p2 * 2;” produce a different final value for *p1? (2 points)
**Question 2: Building Blocks (10 points)**

Building a C program involves four steps: **preprocess**, **compile**, **assemble**, and **link**. For each event below, circle all the steps during which it would occur by circling the appropriate letters (P, C, A, L).

<table>
<thead>
<tr>
<th>Build Step</th>
<th>Event</th>
</tr>
</thead>
</table>
| P C A L    | Generates a warning for this code:  
  ```
  int *pi;  
  pi = 9;
  ``` |
| P C A L    | Creates source code that is human readable |
| P C A L    | Determines which variables go into the rodata, data, and bss sections |
| P C A L    | Traverses relocation records to patch the code |
| P C A L    | Ensures only one definition of each static global variable per module |
| P C A L    | Generates relocation records |
| P C A L    | Generates an error for this code:  
  ```
  int func();
  int main(void){
      return func();
  }
  ``` |
| P C A L    | Inserts code from the contents of other files |
| P C A L    | Generates an error for this code:  
  ```
  char ac[10] = "one";
  ac = "two";
  ``` |
| P C A L    | Creates object code |
Question 3: Totally Alarming (10 points)

Assume the following code includes the necessary include files (e.g., stdio.h, stdlib.h, signal.h, etc.) and has a `#define _GNU_SOURCE` to use the modern style of signal handling. The `alarm(uisec)` system call sends the `14/SIGALRM` signal after `uisec` seconds, and `2/SIGINT` is the signal generated by the user hitting Control-C. The `signal()` system call registers a signal handler for a given signal.

```c
int i = 1;
int done = 0;

static void handleAlarm(int isig) {
    if (i ^= 1)
        printf("hi");
    printf(" hi\n");
    done = 1;
    alarm(2);
}

static void handleControlC(int isig) {
    if (!done)
        printf("Okay, I won't say hello then!\n");
    exit(0);
}

int main(void)
{
    signal(SIGALRM, handleAlarm);
    signal(SIGINT, handleControlC);
    alarm(2);
    for (; ;)
        ;
    return 0;
}
```

3a) What has the program output after eleven seconds have elapsed? Assume the user never enters Control-C. (2 points)
3b) Suppose several other processes run on the same processor. Will the extra processing load introduced by the other processes cause the program’s output at eleven seconds to differ from the answer in 3a? Why or why not? As in question 3a, assume the user never enters Control-C. (2 points)

3c) Explain why and when the program might sometimes print “hi” one or more times, followed by “Okay, I won’t say hello then!” (2 points)

3d) The programmer intends the program to print “Okay, I won’t say hello then!” only if the user uses Control-C to terminate the process before “hi” has ever printed. So, the program has a bug. Describe conceptually how the programmer should fix the bug in the above code. Explaining the necessary change in just a few words is sufficient -- you do not need to write the actual C code. (2 points)

3e) The program overrides the default handler for Control-C. Can the user terminate the program without the program generating any output (i.e., no ‘hi’ or other text)? If so, how? If not, why not? (2 points)
**Question 4: Be Exceptional! (10 points)**

An exception is an abrupt change in the control flow of a process in response to a change in processor state. Classify each of the examples below by the type of exception (*interrupt, trap, fault, and abort*) by circling the appropriate letter (I, T, F, or A) below.

<table>
<thead>
<tr>
<th>Exception Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>I T F A</td>
<td>Application program calls <code>dup()</code> to duplicate a file descriptor</td>
</tr>
<tr>
<td>I T F A</td>
<td>Application program tries to dereference a NULL pointer</td>
</tr>
<tr>
<td>I T F A</td>
<td>Hardware timer expires</td>
</tr>
<tr>
<td>I T F A</td>
<td>Application program tries to run code that is not (yet) in physical memory</td>
</tr>
<tr>
<td>I T F A</td>
<td>Hardware discovers that the physical memory has become corrupted</td>
</tr>
<tr>
<td>I T F A</td>
<td>User clicks a button on the mouse</td>
</tr>
<tr>
<td>I T F A</td>
<td>Application program performs $y/x$ where $x$ is zero</td>
</tr>
<tr>
<td>I T F A</td>
<td>Disk controller finishes reading the contents of a virtual memory page from disk</td>
</tr>
<tr>
<td>I T F A</td>
<td>Exception handler returns control to the <em>current</em> instruction</td>
</tr>
<tr>
<td>I T F A</td>
<td>Application program calls <code>brk()</code> to extend the end of the heap</td>
</tr>
</tbody>
</table>
Question 5: Stick a Fork in It (8 points)

The program snippet below uses `fork()` to create a child process.

```c
int main()
{
    printf("Hello, my pid is %d", getpid());
    if (fork())
        printf("\nA was forked!");
    else
        printf("\nB was forked!");
    return 0;
}
```

5a) Which process (parent or child) prints the string “\nA was forked”? (1 point)

5b) When the program runs, the output to the terminal screen is

```
Hello, my pid is 85037
A was forked!
Hello, my pid is 85037
B was forked!
```

Explain why “Hello, my pid is 85037” prints twice. (1 points)

5c) The programmer did not intend for the “Hello, my pid is” string to print twice. Fix the bug. (1 point)
Consider the following program snippet, which omits the relevant include files and some error checking.

```c
int main(int argc, char *argv[]) {
    for (;;) {
        if (!fork()) {
            char *apcArgv[2];
            apcArgv[0] = argv[1];
            apcArgv[1] = NULL;
            execvp(apcArgv[0], apcArgv);
            perror(argv[0]);
            exit(EXIT_FAILURE);
        }
        wait(NULL);
        sleep(3);
    }
}
```

5d) Suppose the program is compiled to an executable `a.out`. What is the output when running 

```
./a.out date
```

(2 points)

5e) Give one reason for why the call to `execvp()` might fail and return, causing `perror()` to execute? (2 points)

5f) When will the call to `wait()` return? (1 point)
Question 6: Thanks for the Memories (14 points)

Consider a computer with 32-bit virtual addresses, a 16 KB page size, and 1 GB of physical memory.

6a) How many virtual pages can a process have? State your answer as a power of 2, rather than computing the number. (2 points)

6b) For the virtual address 0x003AF176, what is the page number and the offset in the page? Please give your answer in hexadecimal. (2 points)

Page number: ____________________ Offset in page: ____________________

6c) How does the hardware detect that a virtual page is not (yet) resident in the physical memory? (1 point)

6d) Suppose only one user process ever runs on the processor, so the operating system never needs to share the physical memory across multiple user processes. Give one reason why having virtual memory is still a valuable abstraction to the user program. (1 point)
Consider the following code snippet:

```c
int computeProduct(int b[], int n) {
    int i, product = 1;
    for (i = 0; i < n; i++)
        product *= b[i];
    return product;
}
```

6e) Give one example of good temporal locality, and one example of good spatial locality, for the data in the function? (2 points)

Spatial locality: ___________________________  Temporal locality: ___________________________

6f) Give one example of good temporal locality, and one example of good spatial locality, for the C instructions in the function. (2 points)

Spatial locality: ___________________________  Temporal locality: ___________________________

6g) Give one advantage and one disadvantage of the “best fit” method for implementing `malloc()` for allocating memory in the heap. (2 points)

6h) Explain the difference between internal and external fragmentation. (2 points)
Question 7: Some Assembly Required (14 points)

Given the assembly language code below (that uses the x86-64 architecture and assembly language), answer the questions on the following pages. Note that the memory is byte-addressable.

```
.section ".rodata"

cPrompt:
  .string "Enter the nth number desired\n"

cScanfformat:
  .string "%d"

cPrintFormat:
  .string "%d\n"

.section ".text"

func:
  subq $4, %rsp
  movl %edi,%rsp
  subq $8, %rsp
  cmpl $2,%r8(%rsp)
  jge else1
  movl 8(%rsp), %eax
  addq $12, %rsp
  ret

  else1:
  movl 8(%rsp), %edi
  decl %edi
  call func
  movl %eax, (%rsp)
  movl 8(%rsp), %edi
  subl $2, %edi
  call func
  addl (%rsp), %eax
  addq $12, %rsp
  ret

.globl main

main:
  subq $4, %rsp
  movq $cPrompt, %rdi
  movl $0, %eax
  call printf
  movq $cScanfformat, %rdi
  leaq (%rsp), %rsi
  movl $0, %eax
  call scanf
  movl (%rsp), %edi
  call func
  movl %eax, %esi
  movq $cPrintFormat, %rdi
  movl $0, %eax
  call printf
  movl $0, %eax
  addq $4, %rsp
  ret
```
7a) Fill in the boxes on the left with the line number(s) from the assembly language code above that correspond to the boxes of C code on the right. Use all line numbers from lines 10-48. (4 points)

<table>
<thead>
<tr>
<th>Assembly Lang. Line Number(s)</th>
<th>C Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>int func(int n) {</td>
</tr>
<tr>
<td></td>
<td>if (n &lt; 2)</td>
</tr>
<tr>
<td></td>
<td>return n;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>return BLANK_1 + BLANK_2</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>int main(void){</td>
</tr>
<tr>
<td></td>
<td>int i;</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;Enter the nth number desired\n&quot;);</td>
</tr>
<tr>
<td></td>
<td>scanf(&quot;%d&quot;, &amp;i);</td>
</tr>
<tr>
<td></td>
<td>printf(&quot;%d\n&quot;, func(i));</td>
</tr>
<tr>
<td></td>
<td>return 0;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

7b) In the C code above that corresponds to the Assembly language code on the previous page, fill in the two blanks below with the correct C code. (2 points)

BLANK_1: 

BLANK_2: 

7c) In ten words or less, briefly describe what the func function does. (2 points)
7d) Fill in the Assembly language code below (one assembly instruction per line) for the `func` function so that it uses registers instead of the stack to store local variables and function parameters. (2 points)

```
.equ VAR1, %rbx

func:
    _______________
    cmp $2, %edi
    jge else1
    mov %edi, %eax
    movq (%rsp), VAR1
    addq $8, %rsp
    ret

else1:
    _______________
    decl %edi
    call func
    _______________
    movq %rax, VAR1
    sub $2, %edi
    call func
    addq VAR1, %rax
    _______________
    ret
```

7e) Give one example of a callee-saved register and one example of a caller-saved register in the code above. This can be deduced from the code alone, without requiring memorization. (2 points)

callee-saved: ___________________________ caller-saved: ___________________________

7f) Why can’t we use registers to store the local variable in our main function? (1 point)

7g) What two things happen when the instruction `call func` is executed? (1 point)
Question 8: Going Modular: Mind your C and Queues (20 points)

Consider the **Queue** interface:

```c
/* queue.h */
/* A Queue is a first-in-first-out data structure. */

/* The state of the Queue */
/* First node of the queue*/
struct QueueNode *first;
/* The last node of the queue*/
struct QueueNode *last;
/* The number of elements in the queue */
int count;

/* Initialize the Queue */
void Queue_init(void);

/* Free the resources consumed by the Queue */
void Queue_free(void);

/* Return the number of items in the Queue */
int Queue_getCount(void);

/* Add item e to the end of the Queue. Return 1 (TRUE) if successful and 0 (FALSE) if memory is exhausted. */
int Queue_enqueue(void *e);

/* Remove the item at the front of the queue and return it. */
void *Queue_dequeue(void);
```

8a) Circle one. (1 point)
This module is:

- an Abstract Object
- an Abstract Data Type
- a Stateless Module

8b) Briefly describe *two* design problems with this code (i.e., two ways the .h file violates standard practice for modular software development) and how they should be fixed? (2 points)
The `quoter.c` client code on the next page does the following:

1. Continuously reads lines from stdin until the EOF is reached or has been simulated by Ctrl-D.
2. Each line with the format “author: quote\n” is parsed and the author and quote strings are stored in a `Quote` data structure which is then placed in a `Queue`.
3. Once it is done reading from stdin (either by reaching EOF or via Ctrl-D), it dequeues each `Quote` from the `Queue` and prints them in a new format to stdout.

Use the `quoter.c` code on the next page to answer the questions that follow. Below are some snippets from the Linux man pages which may help in understanding `quoter.c`.

```c
char* strtok( char* str, const char *delim)
```

The `strtok()` function parses a string into a sequence of tokens. On the first call to `strtok()` the string to be parsed should be specified in `str`. In each subsequent call that should parse the same string, `str` should be NULL. The `delim` argument specifies a set of bytes that delimit the tokens in the parse string. The caller may specify different strings in `delim` in successive calls that parse the same strings. Each call to `strtok()` returns a pointer to a null-terminated string containing the next token. This string does not include the delimiting byte. If no more tokens are found, `strtok()` returns NULL. A sequence of two or more contiguous delimiter bytes in the parsed string is considered to be a singular delimiter. Delimiter bytes at the start or end of the string are ignored. Put another way: the tokens returned by `strtok()` are always nonempty.

```c
char* strstr( const char* haystack, const char *needle)
```

The `strstr()` function finds the first occurrence of the substring `needle` in the string `haystack`. The terminating null bytes (`'\0'`) are not compared. The function returns a pointer to the beginning of the located substring, or NULL if the substring is not found.

```c
char* strncpy( char* dest, const char *src, size_t n)
```

The `strncpy()` function is similar to `strcpy`, except that at most `n` bytes of `src` are copied. If there is no null byte among the first `n` bytes of `src` the string placed in `dest` will not be null-terminated. If the length of `src` is less than `n`, `strncpy()` writes additional null bytes to `dest` to ensure a total of `n` bytes are written.
/* quoter.c */
...other #include Statements....
#include "queue.h"
enum {MAX_LINE_SIZE = 1024};

struct Quote{
    char *quote;
    char *author;
};

int stopPrint = 0;

int main(void){
    char line[MAX_LINE_SIZE];
    char *line_token;
    const char *newLineDelimiter = "\n";
    Queue_init();
    while (fgets(line, MAX_LINE_SIZE, stdin) != NULL){
        line_token = strtok(line, newLineDelimiter);
        while (line_token != NULL){
            struct Quote * quote;
            size_t sizeToCopy;
            char *token, *start_token = line_token;
            char *end_token = strstr(line_token, ": ");
            quote = (struct Quote*)malloc(sizeof(struct Quote));
            quote->author = (char *)malloc(end_token - start_token + 1);
            for(token=quote->author; line_token != end_token; line_token++, token++)
                *token = *line_token;
            *token = '\0';
            line_token++;
            line_token++;
            sizeToCopy = strlen(start_token) - strlen(quote->author) - 1;
            quote->quote = (char *)malloc(sizeToCopy);
            strncpy(quote->quote, line_token, sizeToCopy);
            Queue_enqueue(quote);
            line_token = strtok(NULL, newLineDelimiter);
        }
    }
    while (Queue_getCount() > stopPrint) {
        struct Quote * quote = (struct Quote *)Queue_dequeue();
        printf("%s\n--%s\n", quote->quote, quote->author);
        free(quote->quote);
        free(quote->author);
        free(quote);
    }
    Queue_free();
    return 0;
}
8c) Why does the program need to include “+1” in the call to malloc on line 26? (1 point)

8d) Assume that the string returned from `strtok` at line 19 was represented in memory by the following diagram where each box represents one byte. Draw arrows to which bytes the following pointers would be pointing to after the first time that line 31 is executed: `line_token`, `start_token`, `end_token`. (3 points)

8e) Why does the program use `strstr` at line 24 instead of another call to `strtok` to parse out `line_token` into `author` and `quote`? (2 points)

8f) For the following code from `quoter.c` on the left, circle where the variable’s value is physically stored in memory on the right. (5 points)

<table>
<thead>
<tr>
<th>Line 11:</th>
<th>DATA</th>
<th>RODATA</th>
<th>BSS</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int stopPrint = 0;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line 14:</th>
<th>DATA</th>
<th>RODATA</th>
<th>BSS</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char line[MAX_LINE_SIZE];</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line 16:</th>
<th>DATA</th>
<th>RODATA</th>
<th>BSS</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>const char *newLineDelimiter = &quot;\n&quot;;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line 22:</th>
<th>DATA</th>
<th>RODATA</th>
<th>BSS</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>size_t sizeToCopy;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line 26:</th>
<th>DATA</th>
<th>RODATA</th>
<th>BSS</th>
<th>Stack</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>quote-&gt;author = (char *)malloc(end_token - start_token + 1);</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
8g) Create a separate Quote ADT module (Quote.c and Quote.h) with three functions based on the quoter.c code above. Do not worry about including any standard C header (.h) files, but make sure to have the proper inclusions for any local files. (6 points)

Quote.h
Use proper style when composing the header file. You should not write more than 8 lines.

Quote.c
You do not need to write any new code for this, except for: include statements, function headings and return statements. The rest can be done by simply writing which line numbers from the quoter.c code would belong inside the functions. You do not need to include comments. You should not write more than 8 lines.
Question 9: More Assembly Required: Fun with Structs! (6 points)

Consider the following C structure `point` and a function prototype `fun` that uses the point structure:

```c
struct point {
    long val;
    struct point* po;
};
long fun(struct point *p);
```

9a) What are the offsets (in bytes) of the two fields `val` and `po`? (1 point)

val: ___________________  po: ___________________

9b) In ten words or less describe the data structure that the `point` struct implements, taking into account that the function `fun()` should not run indefinitely (i.e., it should return at some point). (2 points)

fun:
    pushq %rdi
    pushq $1
    jmp L2
L1:
    movq 8(%rsp), %rax
    movq (%rax), %rax
    movq (%rsp), %rdx
    imulq %rdx, %rax
    movq %rax, (%rsp)
    movq 8(%rsp), %rax
    movq 8(%rax), %rax
    movq %rax, 8(%rsp)
L2:
    cmpq $0, 8(%rsp)
    jne L1
    movq (%rsp), %rax
    addq $16, %rsp
    ret
```

9c) In ten words or less, describe the operation performed by `fun`. (3 points)