CSE 333 – Autumn 2016
Midterm

Name: _____

Please do not read beyond this cover page until told to start.

Nothing in this midterm is about C++.

You don't have to explain your answer unless the question asks you to. If you're unsure of your answer, a little explanation could get partial credit if the answer is wrong. (A correct answer with an incorrect explanation will not get full credit.) Long explanations are a mistake, as they will take too much of your time and provide too much opportunity to get something wrong.

We will be using an online system to grade the midterms. That requires that we scan all the midterms. ***Please do not write on the backs of the pages, nor very close to the edge of the pages on the front.*** *More space is allowed that we think should commonly be needed to answer each question so that everyone has adequate space.* ***Please write with sufficient contrast for the scanner to be able to detect your writing.***

If you can't read your handwriting, neither can we.

1. *[6 points]*
(A) Here are the complete contents of file q1.c:

```
int *sub(void *a, int b);
int main(int argc, char *argv[]) {
     int sum = 0;
     return *sub(&sum, argc);
}
```

This file is compiled with the command:

```
gcc -std=c11 -Wall -g -c q1.c
```

(The `-c` switch tells gcc to compile but not link.)

Are any errors or warnings raised by the compiler? If so, explain briefly what they are complaining about.

No errors or warnings.

(B) Now imagine that a file containing the implementation of `sub` is compiled and linked with the code above. No errors or warnings are raised in doing so.
Which of main's local variables (argc, argv, and sum) could possibly have a different value on return from sub than it had when sub was called?

All of them might be changed, if sub() performs some action with undefined behavior. For instance, sub() might simply construct a pointer from an integer and write to the memory pointed at. It might have a bug that causes it to generate and use an out of bounds array index. Unlike Java, C does not provide memory safety.

(C) **If sub contains no bugs,** which of main's local variables (argc, argv, and sum) may have a different value on return from sub than it had when sub was called?

Only sum might be changed – because main() hands a pointer to sum to the called routine, that routine can dereference the pointer and overwrite sum. In contrast, argc can't be overwritten as the called method gets a copy of the value of argc, not access to argc itself.

2. *[4 points]*
I compile my program and run it under `valgrind`. As part of its output, `valgrind` reports the following:
s

```
==6391== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6391==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==6391==    by 0x4005A7: sub3 (in /home/zahorjan/cse333/16au/midterm/code/q2)
==6391==    by 0x4005BF: sub2 (in /home/zahorjan/cse333/16au/midterm/code/q2)
==6391==    by 0x4005E5: sub1 (in /home/zahorjan/cse333/16au/midterm/code/q2)
==6391==    by 0x400614: main (in /home/zahorjan/cse333/16au/midterm/code/q2)
```

(A) Briefly explain what this output means.

100 bytes were allocated on a code path where main called sub1 which called sub2 which called sub3 which invoked malloc. Those bytes were not free'd before the program terminated.

(B) Briefly explain how `valgrind` obtained this information.
Hint: this could be a CSE 351 question.

Each subroutine call puts a new call frame on the stack. The call frames have a fixed format, known to valgrind. Among the items stored in the frame are the return PC and a pointer to the immediately preceding call frame. (The frames "are chained.") Valgrind traverses the call frames. It uses the stored return addresses to determine where the code was executing when the call was made, and then uses symbol table information available in the executable (when compiled with debug info enabled) to determine the name of the method where that return address lies.

3. *[3+2=5 points]*
(A) Write a C function that takes an array of integers and an integer array length as arguments and returns the largest element in the array.

```
//This version is built for speed - no error checking.
// Array must have at least one element in it.
int maxElement(int *pArray, int N) {
   int result = pArray[0];
   for (int i=1; i<N; i++)
      if (pArray[i]>result) result = pArray[i];
   return result;
   }
```

(B) Modify you function so that it can also return a success/failure indication (in addition to the largest array element).

```
// returns 1 if an error occurs, 0 otherwise.
// if no error occurs, returns the max element at *max
int maxElement(int *pArray, int N, int *max) {
     if ( pArray == NULL || N<=0) return 1;
     *max = pArray[0];
     for (int i=1; i<N; i++)
          if (pArray[i]>*max) *max = pArray[i];
     return 0;
}
```

4. *[3 points]*
Here's an interface design for a dictionary implementation in C. What is seriously wrong

with it? (Missing some method you'd like to have, but that isn't essential, isn't a serious failure.)

```
typedef void* dictionary_key_t;
typedef void* dictionary_value_t;
typedef struct dictionary_t *Dictionary;
Dictionary dictionaryCreate();
void dictionaryDestroy(Dictionary d);
void dictionaryAdd(Dictionary d, dictionary_key_t k,
                                        dictionary_value_t v);
dictionary_value_t dictionaryGet(Dictionary d, dictionary_key_t k);
```

dictionaryDestroy should take a callback function that allows the client to decide how to properly destroy any keys and values still in the dictionary when it is destroyed.

5. *[4 points]*
Standard C provides two interfaces to the file system. One interface (we'll call it "interface A") has methods `open`, `read`, and `write`, while the second ("interface B") has `fopen`, `fread`, and `fwrite`.
(A) What is the **main** distinction between the two interfaces?

Interface B provides buffering of files contents in user space to reduce the number of trips into the operating system. Interface A is unbuffered.

(B) Why does it make sense to have two interfaces? Why doesn't one of them dominate the other?

Buffered IO performs much better than unbuffered when many small reads and writes are the natural implementation in the client code. Unbuffered IO makes it easier to flush data to actually put it on the disk, and can perform better when buffering is not useful to the application (and so simply results in an extra copy of the data).

6. *[4 points]*
A user types the following command into a Linux shell (say, bash):

```
$ ./myprog | grep foo
```

Briefly explain what the bash code does to run the executables `./myprog` and `grep` and to connect them by an anonymous pipe.

The parent instance of bash creates the pipe, which has both read and write ends. It then forks child processes P1 and P2. P1 and P2 inherit the open file table entry for the pipe. P1 closes the read end of the pipe, copies the file descriptor for the write end to stdout, and exec's ./myprog. P2 closes the write end of the pipe, copies the read end to stdin, and exec's grep.

7. *[3 points]*
Why are there both named pipes (also called fifo's) and anonymous pipes (as in the previous question)? In what circumstances are named pipes required, because anonymous pipes can't be used?

Anonymous pipes can connect only sibling processes (see Q6). Named pipes (fifo's) can connect any two processes – they must both just know the name of the fifo, and can then open it for reading or writing themselves.

8. *[2 points]*
Write C code that, when run, will allocate 200 bytes or more of memory on the stack.

```
void sub() {
   int A[200];
}
```

9. *[6 points]*
`int myMemcmp(const void *s1, const void *s2, size_t n);`
is a method that compares two regions of memory of size n bytes. It returns 0 if they are equal and -1 if not. Implement `myMemcmp`.

```
int myMemcmp(const void *s1, const void *s2, size_t n) {
   const char *pA = s1;
   const char *pB = s2;
   for (size_t i=0; i<n; i++) {
       if ( pA[i] != pB[i] ) return -1;
   }
   return 0;
}
```

10. *[4 points]*
`typedef struct q11_t { int x; int y; } Q11;`
Write a C method, `createQ11`, that returns a pointer to a new `Q11` that has both `x` and `y` set to 0.

```
Q11 *createQ11() {
   Q11 *result = (Q11*)malloc(sizeof(Q11));
   if ( result == NULL ) return NULL;
   result->x = result->y = 0;
   return result;
}
```

11. *[12 points]*

Write an application that counts the number of times each lower case letter appears twice in a row in a file. The double letter occurrences cannot overlap. For instance, if the file contains

```
aaabbaacbAAa
```

the output would be

```
Results:
aa 2
bb 1
```

(If you don't remember the name of some include file, make your best guess. I know that you know that `man` will tell you the name, so having the name completely right isn't a grading criterion.)

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>

void usage(const char *e) {
   fprintf(stderr, "Usage: %s filename\n", e);
   exit(EXIT_FAILURE);
}
int main(int argc, char *argv[]) {
   if ( argc != 2) usage(argv[0]);

    int count[26];
   bzero(count, sizeof(int)*26);
   FILE *f = fopen(argv[1], "r");
   if ( f == NULL ) usage(argv[0]);

   char lastChar = '\0';
   char c;
   while ( (c = fgetc(f)) != EOF ) {
        if (c == lastChar && c >= 'a' && c <= 'z') {
            count[c-'a']++;
            lastChar = '\0';
        } else {
            lastChar = c;
        }
   }
   if ( ferror(f) ) {
       fprintf(stderr, "Error reading file\n");
       return EXIT_FAILURE;
   }
   fclose(f);
   printf("Results:\n");
   for (c='a'; c<='z'; c++) {
       if ( count[c-'a'] > 0 )
          printf("%c%c\t%d\n", c, c, count[c-'a']);
   }
   return EXIT_SUCCESS;
}
```

Optional bonus question *[2 points]*

You have a file containing only this:

```
main() { printf("bonus question\n"); }
```

You try to build and then run it. What happens? (Circle all that apply.)

(A) It won't compile because you haven't declared the return type for method main().
(B) It won't compile, saying that main should be int (*)(int, char *[])
(C) It won't compile because you don't return anything from main
(D) It won't compile because you forgot to #include <stdio.h>
(E) It compiles, but there are many, many warnings
(F) It compiles but there is only one real warning (about printf)
(G) It compiles with no warnings or errors
(H) It compiles but won't link, saying you have the type of main wrong
(I) It compiles but doesn't link, saying there is no main that it recognizes
(J) It compiles and links but immediately crashes when you run it because you forgot to #include <stdio.h>
(K) It compiles and links but never terminates when it's run because you forgot to return anything from main
(J) It compiles and links and then prints "bonus question\n" and terminates when run

Note: (E) is acceptable, but there aren't really very many warnings (so not (E) is acceptable as well).