CSE 333 19su Midterm Exam

Date: July 29, 2019 | Instructor: Aaron Johnston

First Name:	
Last Name:	
UW NetID:	@uw.edu
Signature: All work is my own. I did not have advance knowledge of the exam and will not distribute knowledge to classmates who haven't taken it yet. Violating these terms may result in a failing grade.	

Do not turn this page until 10:50 am.

Instructions

- This exam contains 14 pages (7 pieces of paper), including this cover page and the reference pages. Show scratch work for partial credit, but put your final answers in the boxes and blanks provided. If you cannot fit your work or solutions in the space provided, you may use the two "extra space" pages at the end, but you must clearly mark that you have done so at the original question and on the extra page.
- The last page is a reference sheet. Please detach it from the rest of the exam.
- The exam is closed book (no laptops, tablets, calculators, or telepathy). You are allowed one page (US letter, double-sided) of handwritten or typed notes.
- Please silence and put away all cell phones and other mobile or noise-making devices. Remove all hats, sunglasses, and headphones.
- You have 60 minutes to complete this exam.

Advice

- Read questions carefully before starting. Skip questions that are taking a long time.
- Read all questions first and start where you feel the most confident. Note the point breakdown below – the questions are of varying difficulty and point value.
- Relax. You are here to learn. ☺

Point Breakdown

Question	1	2	3	4	5	6
Topic	C Pointers 'n' Structs	Code Organization	C Preprocessor	C++ Classes	C File I/O	Memory and the OS
Total Points	22 pts	18 pts	6 pts	26 pts	14 pts	14 pts

Question 1: C Pointers 'n' Structs [22 pts]

In this problem, you will finish a partially-implemented climate control system written in C for Sieg Hall (which might explain some things…). Use the following struct definitions for buildings with an arbitrary number of classrooms. The `room` struct acts as a linked list node, each pointing to the next `room` via the `next` member (the last node has `next == NULL`).

```
typedef struct room_st {
  int32_t room_number;
  float temp;
  struct room_st *next;
} Room;
```

```
typedef struct building_st {
  char code[3];
  char *long_name;
  room *room_head;
} Building;
```

a) [6 pts] The following code snippet creates a building with 2 classrooms. Complete the code so the building represents "Sieg Hall" (building code "SIG") and contains (in order): classroom 224 with a temperature of 70.0, and classroom 134 with a temperature of -100.0. For this problem, assume the `Verify333` function is sufficient for handling `malloc` failures.

```
Building sieg;
Room *r1 = (Room *) malloc(sizeof(Room)); Verify333(r1 != NULL);
Room *r2 = (Room *) malloc(sizeof(Room)); Verify333(r2 != NULL);
sieg.room_head = r1;
```

b) [8 pts] Draw a memory diagram for `sieg` as described above. All character arrays should have elements drawn out explicitly, and you should clearly indicate what segment of memory (stack, heap, static data, etc.) everything resides in. Don't forget to label variable/field names.

c) [8 pts] Now assume the code from part (a) has been placed in a function `MakeSieg`, and we want to add code after it to return the value to the caller. Assume all code in the program treats buildings and rooms as immutable once created, and that using `Verify333` to check for a `malloc` failure is sufficient. For each of the following options, simply circle **OK** if it works and does no unnecessary operations. Otherwise, circle either **BROKEN** (if it doesn't work) or **UNNECESSARY** (if it works but does unnecessary operations) and then <u>explain your answer</u> in no more than 1 sentence.

```
Building *MakeSieg () {

  // code from (a)

  return &sieg;
}
```

OK BROKEN UNNECESSARY

If not OK, explain:

```
Building MakeSieg() {

  // code from (a)

  return sieg;
}
```

OK BROKEN UNNECESSARY

If not OK, explain:

```
Building *MakeSieg () {

  // code from (a)

  Building *s_copy = (Building *)
      malloc(sizeof(Building));
  Verify333(s_copy != NULL);
  s_copy = &sieg;
  return s_copy;
}
```

OK BROKEN UNNECESSARY

If not OK, explain:

```
Building *MakeSieg () {

  // code from (a)

  Building *s_copy = (Building *)
      malloc(sizeof(Building));
  Verify333(s_copy != NULL);
  *s_copy = sieg;
  return s_copy;
}
```

OK BROKEN UNNECESSARY

If not OK, explain:

Question 2: Code Organization [18 pts]

The following code is the contents of a file called `main.c` (each part of the code has been placed in its own box) for a reading log program that tracks the number of pages read each day. You would like to refactor the code into 3 different modules: `day.c/day.h` for keeping track of a single day, `week.c/week.h` for keeping track of a week, with `main.c` acting as a client of the other code.

a) [10 pts] In the box to the right of each part of the source code below, write the name of the file the code should be placed in, following best practices for style and modularity. You should minimize the number of dependencies that will be created for each file. Some solutions have already been provided for you.

For simplicity, do not worry about memory leaks or `malloc` failures in this problem, and you do not have to specify other file contents such as header guards or `#include` directives.

`typedef struct day_st {`
`  int16_t pages_read;`
`} Day;`	

`typedef struct week_st {`
`  void *days[7];`
`} Week;`	

`Day *AllocateDay();`	
`Week *AllocateWeek();`	
`void ReadPages(Week *wk, int16_t day, int16_t pages);`	
`static Day *GetDay(Week *wk, int16_t day);`	

`int main(int argc, char **argv) {`
`  Week *wk = AllocateWeek();`
`  ReadPages(wk, 0, 25);`
`  return EXIT_SUCCESS;`
`}`
	main.c

`Day *AllocateDay() {`
`  return (Day *) malloc(sizeof(Day));`
`}`
	day.c

`Week *AllocateWeek() {`
`  Week *wk = (Week *) malloc(sizeof(Week));`
`  for (int i = 0; i < 7; i++) {`
`  wk->days[i] = malloc(sizeof(Day));`
`  }`
`  return wk;`
`}`
	week.c

`void ReadPages(Week *wk, int16_t day, int16_t pages) {`
`  GetDay(wk, day)->pages_read = pages;`
`}`	

`static Day *GetDay(Week *wk, int16_t day) {`
`  return (Day *) wk->days[day];`
`}`	

b) [8 pts] The following Makefile was used for the previous version of the code. Update it to be appropriate for the new organization of the code. If you must delete or change the existing contents, cross out the relevant portions and mark clearly what text should be inserted where.

```
CFLAGS = -Wall -std=c11 -g
OBJS = main.o

main: $(OBJS)
     gcc $(CFLAGS) -o main $(OBJS)

main.o: main.c
     gcc $(CFLAGS) -c main.c
```

Question 3: C Preprocessor [6 pts]

Consider the following code in a file called `spooky.c`:

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

#define TRIPLE(x) x * 3
#ifdef DDEBUG
#define CONST 4
#else
#define CONST 2
#endif

int main(int argc, char **argv) {
  int32_t x = TRIPLE(2);
  int32_t y = TRIPLE(10 + CONST);
  printf("%d, %d\n", x, y);
  return EXIT_SUCCESS;
}
```

When compiled with `gcc -Wall -std=c11 -DDEBUG -o spooky spooky.c`, what does the executable `spooky` output when run?

Question 4: C++ Classes [26 pts]

Suppose we have a C++ class that simulates a very simple read-only "file" by storing an array of `char`s and treating it as the "file" contents. For this problem, assume that all data we want to store in a `File` can be stored using `char`s. Each `File` object stores the underlying data in a heap array (pointed to by the field `data_`), and has a field `cursor_` representing the offset of the next `char` to be read. Assume the correct headers are included in all the code for this problem.

```
class File {
  public:
    File(const char data[], const int32_t len);
    File(const File& other);
    File &operator=(const File& other);
    ~File();

  private:
    int32_t cursor_;
    char *data_;
    int32_t data_len_;
};
```

a) [4 pts] Implement the two-argument constructor. You should make space on the heap and copy from the `data` argument to the newly created space on the heap, setting `data_len_` to reflect the size of the heap data. Each `File` starts with its `cursor_` indicating `char` 0. Part of your score will be determined by the efficiency and style of your code.

b) [8 pts] Suppose we add the following implementations of the copy constructor, assignment operator, and destructor (which may not follow good style). Note that the copy constructor and assignment operator make a shallow copy of the `data_` field: they do not copy the array itself.

```
File::File(const File& other) {
  data_ = other.data_;
  data_len_ = other.data_len_;
  cursor_ = 0;
}

File &File::operator=(const File& other) {
  if (this != &other) {
    data_ = other.data_;
    data_len_ = other.data_len_;
    cursor_ = 0;
  }
  return *this;
}

File::~File() {}
```

After writing the code above, we would like to try adding code to clean up `File`'s heap memory. The following are three potential ways we could change the `File` class to try and accomplish that. For each, circle **YES** or **NO** to indicate whether or not it would have the correct result, and in no more than 1 sentence, explain why or why not. Assume that no other changes are made beyond what is described – in particular, that no other fields or functions are added to `File`.

Add code to the `File` destructor given above that deallocates the `data_` field.

YES **NO**

Add code to the `File` assignment operator given above that deallocates the old heap memory pointed to by `data_` before shallow copying the new value of `data_`.

YES **NO**

Documenting in the function and class comments of `File` that a client using the class must take responsibility for deallocating its heap memory.

YES **NO**

c) [6 pts] Write the declaration and corresponding comments of a member function (do NOT implement it) called `ReadChar` as they would appear in the `public` section of the above class declaration. The function should read the next `char` from the `File` and return it, advancing the internal `cursor_` by one. Choose appropriate parameters and returns for the function, keeping in mind any error conditions. You may not use C++ exceptions in this problem.

d) [8 pts] Consider the execution of the following code, and assume `File` has been included properly. Write (in order) every member function that is invoked on any `File` object, along with the name of the local variable it is invoked on. Include the output that is printed in the appropriate spot among the list of member functions. You may abbreviate constructor as ctor, copy constructor as cctor, assignment operator as op=, and destructor as dtor. Assume no compiler optimizations.

```
#include <cstdlib>
#include <iostream>

char arr1[] = {'I','<','3','C'};

void DoVeryLittle(File &in) {
  File c(in);
}

void DoQuiteLittle(File &in){
  File d(arr1, 4);
  d = in;
  in = d;
}

int main(int argc, char **argv) {
  File a(arr1, 4);
  File &b = a;
  std::cout << "very" << std::endl;
  DoVeryLittle(a);
  std::cout << "quite" << std::endl;
  DoQuiteLittle(b);
  return EXIT_SUCCESS;
}
```

Operations performed, in order (the first is provided for you):

ctor a

Question 5: C File I/O [14 pts]

a) [6 pts] Suppose you are using the C standard library to write 1024 total bytes to disk. To do so, you invoke the fwrite() function repeatedly, writing N bytes each time until all 1024 bytes are written.

Give a value of N such that the code would be more efficient with buffering turned on than turned off (assume a buffer size of 512 bytes). Briefly explain why.

Give a value of N such that the code would be more efficient with buffering turned off than turned on. Briefly explain why.

b) [8 pts] Suppose you have a file on disk called `midterm_soln.txt` with a guaranteed size of exactly 50 bytes. The following is a partially-implemented POSIX read loop that reads in those 50 bytes from the file and copies them into a buffer called `buf`. Complete the code so that after the last line, `buf` contains those 50 bytes. Make sure to clean up the open file descriptor.

```
#include <fctnl.h>
#include <unistd.h>
#define BUFFER_SIZE 50

char buf[BUFFER_SIZE];
int bytes_left = BUFFER_SIZE;
int fd = open("midterm_soln.txt", O_RDONLY);

while (1) {
  ssize_t res = read(fd, _____________________________, _____________________);
  if (res == 0) {
    break;
  } else if (res == -1) {
    if (errno != _____________) {
      perror("read error");
      exit(1);
    }
  } else {

    ____________________________________________________________________;
  }
}
_____________________________________; // Clean up the fd
```

Question 6: Memory and the OS [14 pts]

a) [10 pts] Consider the following code:

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

int v;
int main(int argc, char **argv) {
  holler(1);
  return EXIT_SUCCESS;
}

void holler(int w) {
  char *x = "yeehaw %d\n";
  printf(x, w);
  int *y = (int *) malloc(sizeof(int));
  int z = 10;
}
```

For each variable, mark two "X"s in the following tables to indicate *when* space for it is allocated and deallocated. For the pointers x and y, you should answer based on when the memory they point to is allocated and deallocated. When you are done, each column should contain 2 "X"s.

	VARIABLE:	v	w	*x	*y	z
ALLOCATED:	During the compilation process					
	When the program starts running					
	At the beginning of `holler()`					
	After the `printf` in `holler()`					
DEALLOCATED:	When the compiler finishes running					
	At the end of `main()`					
	At the end of `holler()`					
	When the computer is powered off					

b) [4 pts] Rather than have all code on the computer run in the same context, there is a divide between the operating system and user code. To invoke functionality implemented in the operating system, user code uses a "system call". Compared to having everything in user code, briefly describe two advantages to dividing the OS/user code in this way, and one disadvantage.

Advantage:
Advantage:
Disadvantage:

Optional Extra Space for Work/Solutions

If you write any answers here, you MUST clearly indicate on the original problem where to find your answer and clearly indicate on this page which problem is being answered.

Optional Extra Space for Work/Solutions

If you write any answers here, you MUST clearly indicate on the original problem where to find your answer and clearly indicate on this page which problem is being answered.