CSE 374 Final Exam 06/06/18

Name ________________________________ UW ID# _________________

There are 9 questions worth a total of 100 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam is closed book, closed notes, closed laptops, etc.

If you have a question or would like scratch paper, please raise your hand and stay seated.

Please wait to turn the page until everyone is told to begin.

Score _________________ / 100

1.	______ / 8	preprocessor		6.	______ / 14	memory management
2.	______ / 10	make		7.	______ / 12	C++ programming
3.	______ / 12	debugging + scripting		8.	______ / 10	C++ virtual mystery
4.	______ / 14	trie programming		9.	______ / 8	concurrency
5.	______ / 12	testing				

Note: Please write your answers only on the specified pages.

Question 1 (8 points). Preprocessor. Consider the following program:

```
#include <stdio.h>

#define aardvark 5
#define sum(x,y) x+y
#define times(x,y) x*y

#ifndef aardvark
int magic = 2;
#define DEBUG(s) printf("%s\n", s)
#else
int magic = 3;
#define DEBUG(s)
#endif

void foo(int* a, int* b, int* c) {
  *a = 1 * *c;
  *b = 2 * times(*b, *a);
  *c = 3 * sum(*a, *b);
  DEBUG("in foo");
}

int main() {
  int w = aardvark;
  int x = 1;
  int y = magic;
  int z = 2;
  foo(&x, &y, &w);
  printf("%d %d %d %d\n", w, x, y, z);
  foo(&w, &w, &z);
  printf("%d %d %d %d\n", w, x, y, z);
  return 0;
}
```

Write the output of this program below.

<u>Question 2 (10 points). make.</u> Suppose we have a project with multiple source files that have the `#include` dependencies shown below.

```
***************
* vehicle.h   *
***************
#ifndef VEHICLE_H
#define VEHICLE _H

...
#endif
```

```
***************
* car.h       *
***************
#ifndef CAR_H
#define CAR_H
#include "vehicle.h"
...
#endif
```

```
***************
* vehicle.c   *
***************
#include "vehicle.h"
...
```

```
***************
* car.c       *
***************
#include "car.h"
#include "wheel.h"
...
```

```
***************
* wheel.h     *
***************
#ifndef WHEEL_H
#define WHEEL_H

...
#endif
```

```
***************
* main.c      *
***************
#include "car.h"

int main() {
  ...
}
```

If we're being lazy, we could create an executable program from these files by executing the command `gcc -Wall -g -std=c11 -o main *.c`.

On the next page, create a `Makefile` that builds an executable program named `main` from these files as done by the `gcc` command above, but only recompiles and relinks the minimum number of files needed after any changes are made to the source files. Your answer should be done in two steps:

- First, construct the dependency graph that shows the dependencies between the source files, the compiled `.o` files, and the final executable file `main` that is created by linking the `.o` files.
- After drawing the dependency graph, write the final `Makefile`.

Write your answers on the next page.

(2a) (5 points). Draw the dependency graph (diagram) showing dependencies between the files on the previous page, the `.o` files created by compiling the `.c` files, and the final executable program `main`.

(2b) (5 points). Give the contents of a `Makefile` that will build the program as described by the dependency diagram above. The default target that is built if we just type `make` with no arguments should be the executable program `main`.

<u>Question 3 (12 points). Debugging + scripting.</u> While working on the memory manager for HW6, your partner has discovered that the `bench` program crashes with a segmentation fault when it is run with no arguments.

```
$ ./bench
Segmentation fault
```

You suspect that the problem is because you are accessing the command-line arguments (`argv`) without properly checking that they are set first.

<u>(3a) (5 points).</u> Your partner doesn't know how to use gdb, but you do! Below, describe to your partner the exact commands that they can execute to run gdb on the `bench` executable and see if your hypothesis is true. This includes the commands that you will give to gdb itself.

After the issues that your partner has been having, you decide to write a simple bash script to do some basic validation. On the next page, write a script that does the following three things:

- Runs clint.py on all source files (.c and .h). Assume clint.py is in the same directory.
- Builds the executable with make.
- Runs the bench program with no arguments (`bench` target in the Makefile).

If any of the three commands exits with a status code that is not 0, then you should print an error message to stderr and exit with status code 1. Otherwise, if all three exit successfully with status code 0, exit with status code 0 without printing anything (redirect output for the other commands). Write the script on the following page – some bash hints are provided.

(3b) (5 points). Write a script to perform the three validation steps described on the last page.

Some of the tests that can appear in a [ ] or [[ ]] test command in a bash script:

- string comparisons: `=`, `!=`
- numeric comparisons: `-eq`, `-ne`, `-gt`, `-ge`, `-lt`, `-le`

Shell variables: `$#` (# arguments), `$?` (last command result), `$@`, `$*` (all arguments), `$0`, `$1`, … (specific arguments), `shift` (discard first argument)
Blackhole file: `/dev/null` (used for suppressing output)

```
#!/bin/bash
```

(3c) (2 points). Give the shell commands to put the `validate.sh` script from the previous question into your git repository and share it with your partner.

Question 4 (14 points). Tries. In HW5 we used a trie to store the words in a dictionary based on the T9 digit sequences that encoded the words. For this problem, assume that a node is defined as follows:

```
typedef struct Node {      // one node in the trie:
  char* word;              //  C-string if this node has a
                           //  word attached, otherwise NULL
  struct Node* next[10];   //  Subtrees. next[2] to next[9]
                           //  are subtrees for digits 2-9;
                           //  next[0] is the synonym ('#') link.
} Node;                    //  For 0<=i<10, next[i]==NULL if
                           //  next[i] is an empty subtree.
```

For this problem, write a function called `isAlphabetized` that takes a `struct Node*` as an argument and determines whether the words in the trie are alphabetized within each T9 digit sequence. The function returns 1 if the trie is alphabetized, 0 if not.

You ONLY need to determine whether the words in each T9 group (those sharing the same T9 digit sequence) are in alphabetical order. You do NOT need to compare ordering across different T9 digit sequences. For instance, you should verify that all words with the sequence 227 are alphabetized relative to each other, but you do not need to check that they are in any particular order relative to words with the sequence 2273.

You should assume that all necessary header files have already been `#include`d and you do not need to add any `#includes`. An empty trie or a NULL trie passed to your function is by definition alphabetized. Duplicate words do not break alphabetization.

Hints: recursion really, *really*, **really** is your friend. Remember that all words with the same T9 digit sequence are represented as a node with a non-NULL word and all nodes that follow in the 0th (#) child position. Alphabetically later words are considered "greater".

A bit of (maybe) useful reference information about strings:

- `char* strncpy`(*dest*, *src*, *n*), copies exactly *n* characters from *src* to *dst*, adding '\0's at end if fewer than *n* characters in *src* so that *n* chars. are copied.
- `char* strcpy`(*dest*, *src*)
- `char* strncat`(*dest*, *src, n*), append up to *n* characters from *src* to the end of *dest*, put '\0' at end, either copy from *src* or added if no '\0' in copied part of *src*.
- `char* strcat`(*dest*, *src*)
- `int    strncmp`(*string1*, *string2*, *n*), <0, =0, >0 if compare s1<s2, s1=s2, s1>s2
- `int    strcmp`(*string1*, *string2*)
- `char* strstr`(*string, search_string*)
- `int    strnlen`(*s, max_length*)
- `int    strlen`(*s*)
- Character tests: `isupper`(*c*), `islower`(*c*), `isdigit`(*c*), `isspace`(*c*)
- Character conversions: `toupper`(*c*), `tolower`(*c*)

Write your answer on the next page.

Question 4 (cont). Write your implementation of the trie `isAlphabetized` function below.

```
// Returns 1 if words in the trie are alphabetized within
// each T9 digit sequence, 0 if they are not. Words are not
// compared across different T9 digit sequences.
int isAlphabetized(struct Node* root) {













}
```

<u>Question 5 (12 points). Testing.</u> Describe three *black-box* tests for the `isAlphabetized` function from the previous problem. For full credit, the three tests must verify different things about the implementation, and must describe the specific input or setup for the test and the expected result. In your description of test inputs/outputs you should specify the characteristics of the trie passed as input to the function. A drawing is not necessary, but we do expect specific examples of word(s) and a reasonable explanation of how they are placed in the trie.

(5a) Test input/setup:

Expected result:

(5b) Test input/setup:

Expected result:

(5c) Test input/setup:

Expected result:

Question 6 (14 points). Memory management. Recall from HW6 that we can represent the free list for the `getmem/freemem` storage allocator as a linked list of blocks. The beginning of each block is described by the following C struct:

```
typedef struct FreeNode {  // node on free list:
  uintptr_t size;          //   number of bytes in this
                           //     block, not including the
                           //     size of this header
  struct FreeNode* next;   //   next block on free list or
                           //     NULL if none
} FreeNode;
```

In HW6, we used `malloc` to request large chunks from the system. Clients then used `getmem/freemem` to reserve smaller pieces of those chunks. However, we did not require you to ever call `free` to release the chunks that we reserved with `malloc`. We will do that now.

(6a) (6 points). Write a function `freeAll()` that `free`s all memory that was allocated by your memory manager with `malloc`. You should assume that all blocks that the client reserved with `getmem` have been released with `freemem`, which means that the free list will store only chunks that were allocated with `malloc`.

You should also assume that there is a global pointer variable that stores the address of the first block on the free list; your code can access that variable:

```
    FreeNode* freeList;  // free list blocks; NULL
                         // if freeList is empty
```

Write your function below.

```
void freeAll() {










}
```

(6b) (2 points). It's actually not a good idea to assume that all blocks that the client reserved with `getmem` have been released with `freemem`, since a client program can behave incorrectly. Describe in detail what might go wrong with the calls to `free` in `freeAll` in this case – why would `free` fail? Hint: `malloc/free` are similar to `getmem/freemem` from HW6; you can assume that `free` acts like `freemem`.

(6c) (6 points). One way to solve this problem is to do "reference counting." We maintain a global variable `blocksAllocated`, which starts at 0. Every time `getmem` is called and returns a valid pointer (not NULL), we increase `blocksAllocated` by 1. Every time `freemem` is called with a valid pointer (not NULL), we decrease `blocksAllocated` by 1. In `freeAll`, we can then assert that `blocksAllocated == 0` before `free`ing any blocks.

Define an integer `blocksAllocated` to accomplish this, then implement `getmemWrapper` and `freememWrapper` to perform the reference counting (incrementing or decrementing of `blocksAllocated`). These functions should call `getmem` and `freemem` to actually accomplish the allocation/release of the blocks.

```
// Define blocksAllocated here:


void* getmemWrapper(uintptr_t size) {







}
void freememWrapper(void* ptr) {







}
```

<u>Question 7 (12 points). C++.</u> To explore a bit more C++, we've designed a simple class to represent a tuple. A tuple is an ordered sequence of values – like a C array, except that we'll do bounds checking to make sure that we don't access elements that aren't valid. We refer to the number of elements in the tuple as the number of "dimensions".

The class declaration in file `Tuple.h` looks like this:

```
#ifndef TUPLE_H
#define TUPLE_H

class Tuple {
 public:
  // Construct a tuple with "dim" dimensions.
  explicit Tuple(size_t dim);

  // Destructor
  virtual ~Tuple();

  // accessor - returns value at the index
  int get(size_t index) const;

  // setter - sets the value at the index.
  void set(size_t index, int value);

private:
  // Array storing the tuple's value in each dimension
  int* point_;
  size_t dimensions_;
};

#endif   // TUPLE_H
```

On the next page, give implementations for the four functions (constructor, destructor, `get` accessor and `set` setter) that are declared but not implemented above.

Reminder: you'll need to declare an array on the heap since we don't know how big it is until the constructor. You can use either `new` or `malloc` to achieve this. If the requested index is not valid for `get` or `set` (it is larger than the largest valid dimension), then `get` should return 0 and `set` should do nothing.

Write your answer on the next page.

<u>Question 7 (cont).</u> Provide your implementation of the Tuple class declared on the previous page as it would be written in the implementation file Tuple.cpp.

```
#include "Tuple.h"

// write your implementation of class Tuple below.
```

Question 8 (10 points). C++ virtual mystery. Consider the following program, which compiles and executes without error.

```
class B {
 public:
                B() {        cout << "B()"    << endl; }
           void foo() {        cout << "B::foo" << endl; }
  virtual void bar() { foo(); cout << "B::bar" << endl; }
  virtual void baz() { bar(); cout << "B::baz" << endl; }
};

class Bob : public B {
 public:
              Bob() {  cout << "Bob()"    << endl; }
           void foo() {  cout << "Bob::foo" << endl; }
  virtual void baz() {  cout << "Bob::baz" << endl; }
};



int main() {
  B* m = new B();
  m->baz();
  Bob* n = new Bob();
  n->baz();
  B* p = (B*) n;
  p->bar();
  p->foo();

  return 0;
}
```

In the box to the right, write the output that is produced when this program is executed.

Output:

Question 9 (8 points). Concurrency. The following code in C++ manages a chef who is cooking in a shared kitchen. The function will boil a pot for a given number of minutes. Suppose we use this code in a program with two threads (T1/T2), both of which boil many pots.

(9a) (4 points). This code is not thread-safe. Give a brief explanation of what can go wrong if the two threads are using these functions concurrently. You can write out a bad interleaving to the right of the function above, circle sections of the code that have a data race, or write a descriptive paragraph to the right of the function.

```
int burnersAvailable = 4;


bool boilPot(Pot* pot,
             int minutes) {


  if (burnersAvailable > 0) {


    burnersAvailable--;


    placeOnBurner(pot);


    sleep(minutes);


    removeFromBurner(pot);


    burnersAvailable++;


    return true;


  }


  return false;
}
```

(9a)

(9b) (4 points). We can solve this issue with locks (`std::mutex`). Assuming that there is a preexisting mutex called `m`, add lock and unlock calls (ie `m.lock()` and `m.unlock()`) to the function above to make the function completely thread-safe. Do not lock a larger section of code than necessary, so as to allow parallelism when possible.