Name: _____

Write your name in the space provided above. Without looking at the test contents, write your initials on the top right corner of every *sheet* of paper.

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

While you are waiting, please read the following information:

There are 9 questions on 15 pages worth a total of 100 points. Please budget your time to get to all the questions. Keep answers brief and to the point.

Some question pages may be detached for your convenience. A stapler is available at the instructor podium if your entire exam falls apart.

The exam is closed book, closed notes, closed electronics, closed Internet, closed neighbor, closed telepathy, etc.

Many of the questions have short solutions even if the question is somewhat long. Don't be alarmed.

If you don't remember the exact syntax of some command or the format of a command's output, make the best attempt you can. We will make allowances when grading. **Write legibly**.

Relax, you are here to learn.

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

Score: _____ / 100

1. _____ / 7

2. _____ / 9

3. _____ / 4

4. _____ / 14

5. _____ / 13

6. _____ / 20

7. _____ / 20

8. _____/ 12

9. _____/ 1

Question 1. (7 points) (debugging) Consider a program with the following functions.

```
receive()
send()
checksum()
write_header()
```

Your program always has the following buggy behavior: after running for a few minutes, it crashes with a *segmentation fault*.

Explain how you could use gdb (include commands) to discover in which function the crash occurs. You cannot make any modifications to the program code, but you may recompile the code as needed.

Question 2. (9 points) (C preprocessor) The following program compiles and runs. What does it print?

```
#include <stdio.h>
#define FAN 12
#define DO_MATH(A, B) A+A+B

#if FAN > 10
#define MYSTERY(x)
#else
#define MYSTERY(x) x
#endif

int num = 0;

int get_id() {
  int r = num;
  num += 1;
  return r;
}

int main() {
  MYSTERY(get_id());
  int one = get_id();
  int two = DO_MATH(get_id(), FAN);

  printf("%d\n", one);
  printf("%d\n", two);
}
```

Answer:

Question 3. (4 points) (gcc errors)

```
gcc -Wall -o talk main.c speak.c shout.c
(.text+0x20): undefined reference to `speak'
collect2: error: ld returned 1 exit status
```

What is the most likely cause for this error? (**circle one**)

A) A source file referenced speak without making its declaration visible with `#include "speak.h"`
B) The linker cannot find a definition for the function speak
C) A source file `#includes "speak.h"` but we did not provide speak.h as an argument to gcc
D) the function speak was called using a reference instead of a pointer
E) The compiler cannot find the source file speak.c

Question 4. (14 points) (Building C programs) Suppose you have the following C implementation, C header, and text files.

```
_________
list.h
_________
#ifndef LIST_H_
#define LIST_H_
…
#endif
```

```
_________
input.txt
_________
…
```

```
_________
parser.h
_________
#ifndef PARSER_H_
#define PARSER_H_

#include “list.h”
…
#endif
```

```
_________
list.c
_________
#include “list.h”
…
```

```
_________
parser.c
_________
#include “parser.h”
#include “grammar.h”
…
```

```
_________
engine.c
_________
#include “parser.h”

int main() { … }
```

These source files are to be used to build an executable program file named `engine`, whose `main` function is in the source file `engine.c`.

The header file `grammar.h` is not written by the programmer, rather it is generated by the file `input.txt` with the following command:

```
generate_grammar input.txt >grammar.h
```

This command should be re-run to rebuild `grammar.h` whenever changes are made to `input.txt`.

Answer the question on the next page using the above information.

Question 4 (cont). Write the contents of your Makefile for building the `engine` executable. You *must* use `gcc -c` for compiling and `gcc` for linking. Your Makefile must be structured properly so that a change to any input causes dependent targets to be rebuilt.

Question 5. (13 points) (version control with git) This question has three parts (a), (b), (c)

Alice makes two changes to the local copy of her project:
(i) created a new file called `Makefile`
(ii) made edits to `foo.c`

Then she runs the following command in her shell:

```
bash-$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
      modified:   foo.c
      modified:   baz.c

Untracked files:
      Makefile
      foo
```

a) Give the sequence of additional git commands to create one new commit that includes *only* the two changes (i) and (ii) above.

In a different project, Carol makes a local commit changing the file gadget.c. Then she runs the following command in her shell:

```
bash-$ git pull
Auto-merging gadget.c
CONFLICT (content): Merge conflict in gadget.c
Automatic merge failed; fix conflicts and then commit the result.
```

b) What was Carol trying to do by running that command? What has happened?

c) What does she need to do to resolve the problem? (be specific)

Question 6. (20 points) (T9) Consider the data structure for the T9 trie.

```
typedef struct Node {    /* T9 trie node */
      char* word;        /* word associated with this node or NULL if no word
                            stored here */
      Node* next[9];    /* Pointers to subtrees of this T9 node.
                         * Pointers to empty subtrees are NULL. */
} Node;
```

Your colleague wrote a new function `remove_word`, which removes a word from the trie. Unfortunately you notice that it is leaving "blank" `Node`s in the trie instead of completely removing them.

A "blank" `Node` is defined as a `Node` that stores no word and that has only empty subtrees.

Your boss won't let you fix `remove_word`, so you must write a new function called `remove_blank_nodes`, which removes **all** blank nodes from a trie. Your solution **must not cause memory leaks**.

Hint: recursion

You should assume that any necessary standard library headers are already `#included` and you do not need to write any `#includes`.

You may define additional helper functions if needed as part of your solution. The next page contains the prototype for `remove_blank_nodes`.

Question 6 (cont). Complete the following function.

```
/* Remove all empty Nodes from the trie rooted at r. */
int remove_blank_nodes(struct Node* r) {
```

Question 7. (20 points) (memory manager) Implementing a `check_list` function.
In the memory manager assignment, the `freemem` function had to search the free list for the proper location to add a returned block to the list and possibly merge it with other free blocks. A different way to handle this is known as the boundary-tag method. Here, in addition to the header at the front of every block, there is additional information in a footer following each block.

The footer contains a pointer back to the beginning of the block and a 1/0 (true/false) value indicating whether the block is currently allocated. Here are definitions for the header and footer structs that surround each block:

```
struct header {        // block header:
  uintptr_t size;      // number of data bytes in the block
                       //    not including the header/footer
  struct header* next; // next block on the free list or NULL
};

struct footer {         // block footer:
  struct header* hdr;   // address of header for this block
  uintptr_t allocated;  // 1 if block allocated, 0 if free
};
```

Here is an illustration of a free block with 160 bytes of user storage whose header is at location 10000 (base 10) and is followed in the freelist by a block at address 10512. The block plus header and footer occupy 160+16+16 = 192 bytes.

```
10000 +--------+--------+
      |size: 160        |  header
      |next: 10512      |
10016 +--------+--------+
      |                 |
      |                 |
      | 160 bytes of    |
      | user storage    |
      |       …         |
      |                 |
10176 +--------+--------+
      | hdr: 10000      |  footer
      | allocated: 0    |
10192 +--------+--------+
```

A common class of bugs in the implementation of `getmem`/`freemem` is for the header or footer to become corrupted. To check for such bugs, your job is to write a `check_list` function for the freelist. This function should check the following properties **for every block in the freelist**, using `assert`:

- the block does not overlap the next block (you may assume the blocks are sorted by increasing address)
- the `hdr` field of the footer is correct
- the `allocated` field of the footer is correct

Question 7 (cont).

Hint: assert works like this:

```
// Stops the program with an assertion error if predicate==0,
// and otherwise just returns
void assert(int predicate);
```

Complete the function below.

```
/* Stops the program with an assertion error if there are overlapping blocks
 * or a footer field is invalid */
void check_list(struct header* freelist) {
```

Question 8. (12 points) (C++ inheritance) The following program prints `howle rorwl`. Add the keyword `virtual` in appropriate places so that the program will print `hello world` instead. You may not make any other changes to the code.

```
#include <iostream>
using namespace std;

class A {
public:
      void m1() {       cout << "o";      }
      void m2() {       cout << "h";      }
      void m3() {       cout << "w";      }
};
class B :    public A {
public:
      void m1() {       cout << "e";      }
      void m2() {       cout << "r";      }
      void m3() {       cout << "l";      }
};
class C :    public B {
public:
      void m1() {      cout << "o";       }
      void m2() {      cout << "w";       }
      void m3() {      cout << "d";       }
};
int main() {
  A* ab =   new    B();
  B* b  =   new    B();
  B* bc =   new    C();
  A* ac =   new    C();

  ac->m2();
  ab->m1();
  ab->m3();
  b->m3();
  bc->m1();
  cout << " ";
  bc->m2();
  ac->m1();
  b->m2();
  ab->m3();
  bc->m3();
  cout << endl;
}
```

Question 9. (1 point) (art)

Draw what **engineering** means to you. For practical purposes, credit requires at least one visible mark, as we cannot distinguish the unsparing use of whitespace from a blank answer (we apologize for the creative constraints).