Question 1. (12 points) Preprocessor. We have the following small program consisting of these two files:

hdr.h

```
#define SIZE 10
typedef char* strptr;
void compute(char a[], int n);
```

main.c

```
#include "hdr.h"
#define NTIMES 2 * SIZE
int main(int argc, char** argv) {
  strptr s;
  char a[SIZE];
  s = a;
  compute(s, NTIMES + 1);
  return 0;
}
```

File `main.c` does compile successfully to produce `main.o` without errors, although this one file by itself is not a complete program since function `compute` is not included.

Show the output produced by the preprocessor command `cpp -P main.c` when it reads and processes this file to generate a source file that can be translated by the compiler to generate `main.o`. Your answer should show all of the output from the preprocessor, exactly as produced by that `cpp` command.

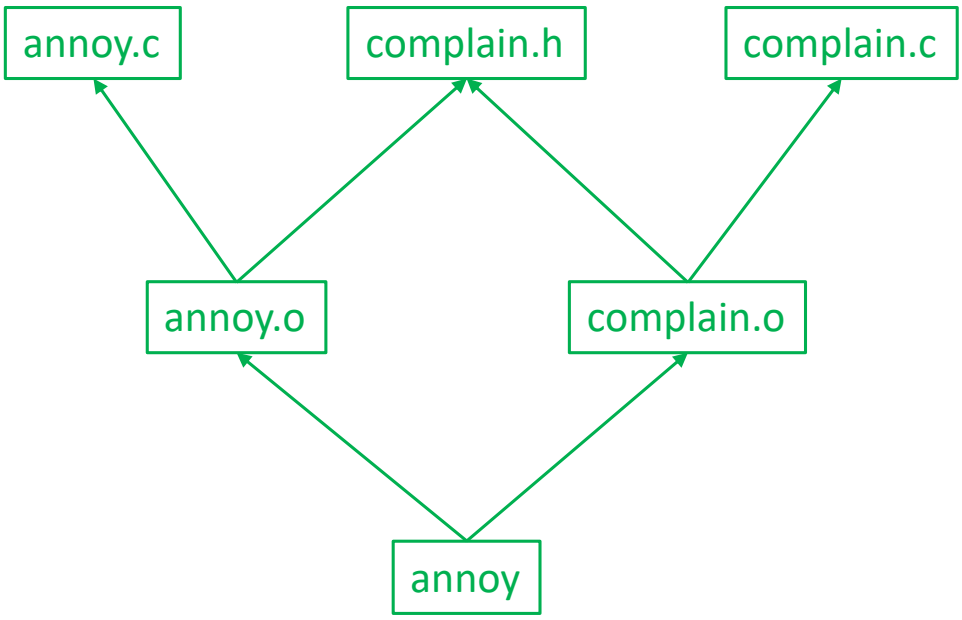
```
typedef char* strptr;

void compute(char a[], int n);

int main(int argc, char** argv) {
  strptr s;
  char a[10];
  s = a;
  compute(s, 2 * 10 + 1);
  return 0;
}
```

Question 2. (20 points) making things and a bit of `git`. Suppose we have the following `Makefile` in the current directory. (Note: recall that if no `-o` option is present on a `gcc` command, the `-c` option will cause the compiler to generate a file named `x.o` if the input is `x.c`.)

```
annoy: annoy.o  complain.o
     gcc -Wall -g -std=c17 -o annoy annoy.o  complain.o
complain.o: complain.h  complain.c
     gcc -Wall -g -std=c17 -c complain.c
annoy.o:  annoy.c  complain.h
     gcc -Wall -g -std=c17 -c annoy.c
```

fie
↑
fum
↑
foo

(a) (7 points) In the space below, draw a dependency graph (diagram) showing the dependencies between the files as specified by the above `Makefile`. Your drawing should have an arrow from each file to the files that it depends on, as in the diagram to the left, where `foo` depends on `fum`, which depends in turn on `fie`.



(continued on next page)

Question 2. (cont.) (b) (7 points) Now suppose we want to add to this program a new source file `irritate.c` along with an associated header file `irritate.h`. File `irritate.c` only includes the `irritate.h` header; it does not use any of the others. Files `annoy.c` and `complain.c` have been modified to use the new functions in these new `irritate` files, with the appropriate `#include` and calls to the new functions.

Write in the modifications needed in the copy of the `Makefile` below so that these new files are included in the program and are correctly compiled (and recompiled) as needed when the program is built.

(changes shown in bold green text)

```
annoy: annoy.o  complain.o  irritate.o

     gcc -Wall -g -std=c17 -o annoy annoy.o complain.o irritate.o

complain.o: complain.h  complain.c  irritate.h

     gcc -Wall -g -std=c17 -c complain.c

annoy.o:  annoy.c  complain.h  irritate.h

     gcc -Wall -g -std=c17 -c annoy.c

irritate.o:  irritate.c  irritate.h

     gcc -Wall -g -std=c17 -c irritate.c
```

(c) (6 points) We've been using `git` and the CSE `gitlab` repository to manage our code for this project. Once the new `irritate.h` and `irritate.c` files have been tested and the `Makefile` and other changed files have been updated in our local copy of the program, we need to be sure to save the new and changed files in our `gitlab` repo. Write the necessary `git` commands below to update the `gitlab` repository to reflect the addition of these new files as well as the other changes made to existing files. You should assume that the current directory in the terminal window is the local git working copy directory containing the source files.

(There are many reasonable solutions for the `git add` command. We allowed wildcards like `git add *.c *.h`. However, commands that would add `.o` files or binaries or all files in the directory like `git add *` did receive a deduction.)

```
git add irritate.c irritate.h annoy.c complain.c Makefile
git commit -m "finished problem 2"
git push
```

Question 3. (18 points) bugs. One of our colleagues has been trying to write a small C function that is supposed to return the number of 0's found in an integer array. Their code does compile without errors, but it doesn't work right. The main program always prints "answer = 0" no matter how many 0's actually appear in the array. In addition, we're suspicious that the program might leak memory and it may have some other pointer or other problems, but, amazingly, it doesn't seem to crash or produce segfaults.

(a) (12 points) On the code listing below, identify all of the problems with the existing code. Your answers can be brief – you don't need to write an extensive essay, but you should identify and explain each place where there is some sort of bug in the code.

After marking the bugs, continue with the rest of this problem on the next page.

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

// return the number of elements that are 0 (zero)
// in the given array which has arr_len elements.
int countZeroes(int arr[], int arr_len) {
  int* count = malloc(sizeof(int));
  for (int i = 0; i < arr_len; i++) {
    if (arr[i] == 0) {
      count += 1;
    }
  }
  return *count;
}

// simple test for countZeroes
int main() {
  int nums[] = { 1, 3, 0, 4, -3, 0, 17 };
  int nzero = countZeroes(nums, 7);
  printf("answer = %d\n", nzero);
  return 0;
}
```

Memory leak – allocated memory is never freed.

No check to be sure `malloc` returned a non-`NULL` pointer

***count never initialized to 0**

Increments (changes) pointer `count` not integer `*count`

Returns contents of unknown memory location because of changes to pointer

Question 3. (cont.) (b) (6 points) Now that you have identified the problems with the code on the previous page, show how to fix the code to work properly by writing in changes on the copy of the code below. You may not change the basic organization of the code or alter the specification of function `countZeroes`. But you are free to modify the implementation to fix the bugs in whatever way seems simplest. Your modified version should produce the correct answers and should execute with no errors, memory leaks, or any other problems.

Write your answers directly on this page. Cross out anything you wish to delete and write your corrections below. If you insert new code, be sure it is clear where the insertions should be made. Hint: it might be possible to fix things by simplifying the existing code rather than patching each bug in the original version separately.

The simplest fix is to change the type of `count` to a simple `int` and count the values directly, instead of allocating an `int` variable on the heap. Other changes that fixed the problems did, of course, receive full credit if done correctly.

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

// return the number of elements that are 0 (zero)
// in the given array which has arr_len elements.
int countZeroes(int arr[], int arr_len) {
  int* count = 0; malloc(sizeof(int));
  for (int i = 0; i < arr_len; i++) {
    if (arr[i] == 0) {
      count += 1;
    }
  }
  return *count;
}

// simple test for countZeroes
int main() {
  int nums[] = { 1, 3, 0, 4, -3, 0, 17 };
  int nzero = countZeroes(nums, 7);
  printf("answer = %d\n", nzero);
  return 0;
}
```

Delete *'s in declaration and use of `count`, and initialize `count` to 0 instead of using `malloc`.

Question 4. (16 points) A little C programming. As you remember from previous courses, a classic data structure is a *binary search tree* (BST). A BST is a binary tree where each node in the tree holds a value. For each node, all of the values in its left subtree are less than the value in that node and all of the values in its right subtree are greater.

For this problem we have a BST containing C string values. The tree nodes are defined by the following struct. All of the node structs and all strings (`char` arrays) referenced by the field `val` have been independently allocated on the heap using `malloc`.

```
struct bstnode {          /* node for a BST of strings:     */
   char * val;            /* string value of this node      */
   struct bstnode *left;  /* left subtree or NULL if empty  */
   struct bstnode *right; /* right subtree or NULL if empty */
};
```

Your job is to implement a function `free_bst(r)` that will free all of the nodes and strings in the BST with root node `r`.

You should assume that all necessary header files have already been `#included` and you do not need to add any `#includes`.

Hints: recursion really, *really*, **really** is your friend.

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

Some basic C memory management functions:

- void * **malloc**(size_t size)
- void **free**(void *ptr)
- void * **calloc**(size_t number, size_t size)
- void * **realloc**(void *ptr, size_t size)

Write

Your

Answer

On

The

Next

Page…

Question 4. (cont.) Write your implementation of the `free_bst` function below. The tree struct is repeated for reference.

```
struct bstnode {          /* node for a BST of strings:     */
   char * val;            /* string value of this node      */
   struct bstnode *left;  /* left subtree or NULL if empty  */
   struct bstnode *right; /* right subtree or NULL if empty */
};
```

```
// Free the nodes and strings in the BST with root r.
// If r == NULL, then r represents an empty tree and
// nothing needs to be done.
void free_bst(struct bstnode *r) {

  if (r == NULL) {
    return;
  }

  // recursively free subtrees, then free string and node
  // (note: must free node last, else dangling pointers)
  free_bst(r->left);
  free_bst(r->right);
  free(r->val);
  free(r);

}
```

Question 5. (18 points) A bit of memory management. Recall from HW6 that we stored the free list for the `getmem`/`freemem` memory manager using a linked list of free blocks. The beginning of each freelist block is described by the following C struct:

```
struct free_node {          // node (block) on free list:
  uintptr_t size;           //   number of bytes in this
                            //     block, not including the
                            //     size of this header
  struct free_node *next;   //   next block on free list or
                            //     NULL if this is the last
};                          //     block on the free list
```

(For this problem, you should assume that the `size` field in the `free_node` struct gives only the number of bytes of data following the header struct, and does not include the size of the 16-byte header itself.)

When blocks are returned to the free list by `freemem`, the memory manager is supposed to check whether the returned block occupies locations in storage that are immediately adjacent to an existing block on the freelist and, if so, the adjacent (touching) blocks are supposed to be merged into a single larger block on the free list

For this problem we want to write a function that checks the free list to discover if there are any adjacent blocks on the free list that should have been merged previously. The function `merge_ok(p)` should search the free list starting at free list node `p`. It should check each node in the free list to verify that the next node on the free list is not immediately adjacent to the current one, i.e., for each node on the list starting at `p`, check that the end of that node does not have the same address as the `free_node` header of the next block.

If `merge_ok` finds any adjacent blocks that should have been merged, it should return false (0). If it does not find any problems, it should return true (1).

You should assume that the blocks on the free list are correctly stored in order of ascending addresses, and that no two blocks on the free list overlap (i.e., the end of one block won't be somewhere in the middle of the next block).

You should assume that all necessary header files have already been `#included` and you do not need to add any `#includes`.

Write your answer on the next page.

(continued on next page)

Question 5. (cont.) Write your implementation of `merge_ok` below. The heading of the function is written for you, and the definition of the `free_node` struct is repeated for convenience.

```
struct free_node {          // node on free list:
  uintptr_t size;           //   number of bytes in this
                            //     block, not including the
                            //     size of this header
  struct free_node *next;   //   next block on free list or
                            //     NULL if this is the last
};                          //     block on the free list

// Return 1 (true) if there are no touching adjacent blocks
// on the free list starting at p.  Return 0 (false) if any
// two adjacent blocks are touching.
int merge_ok(struct free_node *p) {

  // if list is empty, there are no adjacent blocks
  // (checked as separate case to simplify following code)
  if (p == NULL) {
    return 1;
  }

  // p!=NULL at top of loop so it is safe to dereference
  while (p->next != NULL) {

    // check that block *p and block p->next do not touch
    // and return false if they do.  This solution compares
    // addresses as uintptr_t values, but solutions that
    // check correctly computed pointer values are also ok.
    // (Note: 16 is the size of the header block. Ok to use
    // a constant in this solution.)

    uintptr_t p_end = 16 + p->size + (uintptr_t)p;
    uintptr_t p_next_start = (uintptr_t)(p->next);
    if (p_end == p_next_start) {
      return 0;
    }

    // advance to next freelist block
    p = p->next;
  }

  // no errors found - return true
  return 1;

}
```

Question 6. (14 points) A little C++. To explore a bit more of C++, we are creating a simple class to represent 3-D vectors. (Don't worry if you are not familiar with vectors – this is a programming problem and the details are all described here.) So far, we've created this file `Vector.h` to declare the class, its representation, a constructor, and an addition function.

```
#ifndef VECTOR_H_
#define VECTOR_H_

class Vector {
 public:
  // Construct Vector with x,y,z coordinates (a,b,c)
  Vector(double a, double b, double c);

  // arithmetic: return a new Vector that is the sum of this and
  // other.  If this Vector has components with values (a,b,c)
  // and other has components with values (p,q,r), the new
  // returned vector has components (a+p, b+q, c+r)
  Vector plus(Vector other) const;

 private:
  // Representation of a Vector: x, y, z coordinates
  double x, y, z;
};

#endif  // VECTOR_H_
```

Below, give the implementations of the constructor and function `plus` as they would appear in the associated implementation file `Vector.cc`. The `#include` at the beginning of `Vector.cc` is written for you. Hint: the answers are not long, but additional space is provided on the next page for you to use if needed.

```
#include "./Vector.h"

// Construct Vector with x,y,z coordinates (a,b,c)
Vector::Vector(double a, double b, double c) {
  x = a;
  y = b;
  z = c;
}


// return a new Vector that is the sum of this and other.
Vector Vector::plus(Vector other) const {
  return Vector(x+other.x, y+other.y, z+other.z);
}
```

Question 7. (2 free points – all answers get the free points) Draw a picture of something you plan to do this summer!

Congratulations on lots of great work this quarter!!
Have a great summer!
The CSE 374 staff