CSE 333 25wi Midterm Exam 2/13/25

Name _____ UW Netid: _____@uw.edu
(please *print* ***legibly***)**

There are 7 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 electronics, closed telepathy, open mind. However, you may have a single 5x8 notecard with any hand-written notes you wish on both sides.

There is a blank sheet of paper at the end with extra space for your answers if you need more room. It is after all the questions but before the detachable pages with reference information.

After the extra blank pages for answers, there is a sheet of paper containing assorted reference information (most of which you probably won't need). You should remove this reference sheet from the exam and use it during the exam. It will not be scanned for grading, so do not write answers on it.

Do not remove any pages from the middle of the exam.

If you do not remember the exact syntax for something, make the best attempt you can. We will make allowances when grading.

Don't be alarmed if there seems to be more space than is needed for some answers – we tried to include enough blank space.

Relax, you are here to learn.

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

Score _____ / 100

1. _____ / 18
2. _____ / 16
3. _____ / 20
4. _____ / 16
5. _____ / 20
6. _____ / 8
7. _____ / 2

Question 1. (18 points) Making things. We're working with our friend Jungkook on a new music player app. So far, we've got the following files with these `#includes` to reference declarations in various header files:

main.c	playlist.h	playlist.c

`#include "playlist.h"`
`#include "audio.h"`
`...`
	`...`	`#include "playlist.h"`
		`...`

audio.h	audio.c

`...`	`#include "playlist.h"`
	`#include "audio.h"`
	`...`

We've been retyping the following `gcc` commands to build the program:

```
gcc -Wall -g --std=c17 -c playlist.c
gcc -Wall -g --std=c17 -c audio.c
gcc -Wall -g --std=c17 -c main.c
gcc -Wall -g --std=c17 -o player audio.o playlist.o main.o
```

Hint: recall that if we compile `foo.c` with the `-c` option and do not specify the output file name (no `-o` option), the output file created will be named `foo.o`, as in the `gcc` commands above.

(a) (6 points) Draw the dependency diagram showing the dependencies between all files used or created during the build process. You should draw an arrow pointing from each file that is built by the `gcc` commands to the file or files that it depends on.

(continued on next page)

Question 1. (cont.) Jungkook has gotten tired of typing `gcc` commands and has written the following `Makefile` to automate compiling and recompiling the `player` program based on the above `gcc` commands, but it doesn't quite work right. It is supposed to build the `player` program by recompiling and relinking only the necessary files.

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

audio.o: audio.c audio.h
      gcc -Wall -g --std=c17 -c audio.c

main.o: main.c
      gcc -Wall -g --std=c17 -c main.c

player: audio.o playlist.o main.o
      gcc -Wall -g --std=c17 -o player audio.o playlist.o main.o
```

Answer the following questions about this `Makefile` and program:

(b) (2 points) Suppose this `Makefile` and all of the source files (`.c` and `.h`), but no compiler or linker output files are in the same directory. What exactly will happen if we run the command `make` in this directory? (Describe the command(s) that are executed and the result(s) they produce.)

(c) (6 points) If running `make` as in part (b) above fails to properly build the `player` program, describe exactly what needs to be done to fix the `Makefile` so that it will work properly. You can either explain your changes below or show the changes needed by writing them on the `Makefile` code given above.

(continued on next page)

Question 1. (cont.) Assume that we have now fixed the `Makefile` so that it recompiles only the necessary source files after any changes and successfully builds the `player` program when we use the `make` command.

(d) (2 points) Assuming that we have fixed all the bugs and that all of the source and output files are up-to-date after running a `make` command, what commands are executed if we modify the file `main.c` and then run `make` again?

(e) (2 points) Assuming that we have fixed all the bugs and that all of the source and output files are up-to-date after running a `make` command, what commands are executed if we modify the file `audio.h` and then run `make` again?

Question 2. (16 points) Preprocessor. Suppose we have the following two C source files:

foo.h

```
#ifndef FOO_H_
#define FOO_H_

#define MAGIC 42
#define SPELL MAGIC + MAGIC
#define LIMIT 333
#define while if

int foo(int n);

#endif // FOO_H_
```

foo.c

```
#include "foo.h"

#define WIZARD "Gandalf"

int foo(int k) {
  int ans = MAGIC + SPELL;
  while (ans < LIMIT) {
    ans = ans + MAGIC + WIZARD;
  }
  return ans;
}
```

Show the output produced by the C preprocessor when it processes file `foo.c` (i.e., if we were compiling this file, what output would the preprocessor send to the C compiler that actually translates the program to machine code?) Hint: remember that the preprocessor does string substitution and does not analyze the C code it produces for correctness.

Question 3. (20 points) Valgrind and memory. This question concerns the following C program `membugs.c` that copies an array of numbers into a linked list on the heap, prints the contents of the list, and then frees the data.

Warning!! Watch your time and do not get bogged down on this question! Only a small number of fixes are needed once you've found the problem(s).

```
 1    #include <stdio.h>     // for printf
 2    #include <stdlib.h>    // for EXIT_SUCCESS
 3
 4    // Node for linked list of integers
 5    typedef struct Node_st {
 6         int value;
 7         struct Node_st* next;
 8    } Node;
 9
10    // Creates a list of nodes on the heap of numbers
11    // such that the ith node contains values[i].
12    // Caller is responsible for freeing the nodes.
13    // Parameters
14    //   count: the number of elements in the array and
15    //   number of nodes in the returned list
16    //   values: array of numbers to be copied to the list
17    // returns:
18    //   pointer to first node of new list
19    Node* MakeList(int count, int values[]);
20
21    int main(int argc, char** argv) {
22      int values[] = {1, 2, 3, 4};
23      int count = 3;
24
25      // create list
26      Node* head_ptr = MakeList(count, values);
27
28      // print values in list
29      Node* curr_ptr = head_ptr;
30      while (curr_ptr != NULL) {
31        int value_to_print = curr_ptr->value;
32        printf("%i\n", value_to_print);
33        curr_ptr = curr_ptr->next;
34      }
35
36      // free list contents
37      for (int i = 0; i < count; i++) {
38        head_ptr = head_ptr->next;
39        free(curr_ptr);
40        curr_ptr = head_ptr;
41      }
42      return EXIT_SUCCESS;
43    }
```

(continued on next page)

Question 3 (cont.) Code continued below:

```
44
45     Node* MakeList(int count, int values[]) {
46       Node* head = (Node*) malloc(sizeof(Node));
47       // ** imagine null check **
48       Node* curr_node = head;
49       for (int i = 0; i < count; i++) {
50         curr_node->value = values[i];
51         if (i != count) {
52           curr_node->next = (Node*) malloc(sizeof(Node));
53           // ** null check - omitted to save space**
54         }
55         curr_node = curr_node->next;
56       }
57       return head;
58     }
```

Question continues on next page. Remainder of this page left blank to be used as needed while working the problem.

(continued on next page)

Question 3. (cont.) The program compiles without errors and runs without crashing. However, when we used valgrind to look for memory bugs, it reported some trouble.

Your job for this question is to examine the valgrind report and the code and then show where the bugs are in the code and how to fix them. You should show corrections by crossing out, changing, or adding code in the listing on the previous two pages. There is extra blank space at the bottom of the previous page for you to use if you need additional space, or if you need space to draw diagrams or do other work to figure out the answers.

```
$ valgrind --leak-check=full --track-origins=yes ./membugs
==2432== Memcheck, a memory error detector
==2432== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==2432== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==2432== Command: ./membugs
==2432==
1
2
3
==2432== Conditional jump or move depends on uninitialised value(s)
==2432==    at 0x48CB3EB: __vfprintf_internal (in /usr/lib64/libc.so.6)
==2432==    by 0x48C052E: printf (in /usr/lib64/libc.so.6)
==2432==    by 0x4011B3: main (membugs.c:32)
==2432==  Uninitialised value was created by a heap allocation
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432== Use of uninitialised value of size 8
==2432==    at 0x48BF94B: _itoa_word (in /usr/lib64/libc.so.6)
==2432==    by 0x48CAFDB: __vfprintf_internal (in /usr/lib64/libc.so.6)
==2432==    by 0x48C052E: printf (in /usr/lib64/libc.so.6)
==2432==    by 0x4011B3: main (membugs.c:32)
==2432==  Uninitialised value was created by a heap allocation
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432== Conditional jump or move depends on uninitialised value(s)
==2432==    at 0x48BF95C: _itoa_word (in /usr/lib64/libc.so.6)
==2432==    by 0x48CAFDB: __vfprintf_internal (in /usr/lib64/libc.so.6)
==2432==    by 0x48C052E: printf (in /usr/lib64/libc.so.6)
==2432==    by 0x4011B3: main (membugs.c:32)
==2432==  Uninitialised value was created by a heap allocation
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432== Conditional jump or move depends on uninitialised value(s)
==2432==    at 0x48CB8D3: __vfprintf_internal (in /usr/lib64/libc.so.6)
==2432==    by 0x48C052E: printf (in /usr/lib64/libc.so.6)
==2432==    by 0x4011B3: main (membugs.c:32)
==2432==  Uninitialised value was created by a heap allocation
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
```

```
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432== Conditional jump or move depends on uninitialised value(s)
==2432==    at 0x48CB0F7: __vfprintf_internal (in /usr/lib64/libc.so.6)
==2432==    by 0x48C052E: printf (in /usr/lib64/libc.so.6)
==2432==    by 0x4011B3: main (membugs.c:32)
==2432==  Uninitialised value was created by a heap allocation
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
0
==2432== Conditional jump or move depends on uninitialised value(s)
==2432==    at 0x4011C5: main (membugs.c:30)
==2432==  Uninitialised value was created by a heap allocation
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432== Conditional jump or move depends on uninitialised value(s)
==2432==    at 0x4847AFF: free (vg_replace_malloc.c:989)
==2432==    by 0x4011E7: main (membugs.c:39)
==2432==  Uninitialised value was created by a heap allocation
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432==
==2432== HEAP SUMMARY:
==2432==     in use at exit: 32 bytes in 2 blocks
==2432==   total heap usage: 5 allocs, 3 frees, 1,088 bytes allocated
==2432==
==2432== 16 bytes in 1 blocks are definitely lost in loss record 1 of 2
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40121B: MakeList (membugs.c:46)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432== 16 bytes in 1 blocks are definitely lost in loss record 2 of 2
==2432==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==2432==    by 0x40125E: MakeList (membugs.c:52)
==2432==    by 0x401188: main (membugs.c:26)
==2432==
==2432== LEAK SUMMARY:
==2432==    definitely lost: 32 bytes in 2 blocks
==2432==    indirectly lost: 0 bytes in 0 blocks
==2432==      possibly lost: 0 bytes in 0 blocks
==2432==    still reachable: 0 bytes in 0 blocks
==2432==         suppressed: 0 bytes in 0 blocks
==2432==
==2432== For lists of detected and suppressed errors, rerun with: -s
==2432== ERROR SUMMARY: 9 errors from 9 contexts (suppressed: 0 from 0)
$
```

Question 4. (16 points) The function on this page and the next opens two files, one for reading and one for writing, and copies the contents of the first file to the second. Your job is to complete the code by filling in the blanks lines with the correct POSIX I/O function calls to handle the file operatons (open, close, read, write).

Here is a summary of some key POSIX I/O functions for your reference.

int open(const char **name*, int *mode*);
mode is one of O_RDONLY, O_WRONLY, O_RDWR
int creat(const char **name*, int *mode*);
create a new file
int close(int *fd*);
ssize_t read(int *fd*, void **buffer*, size_t *count*);
returns # bytes read or 0 (eof) or -1 (error)
ssize_t write(int *fd*, void **buffer*, size_t *count*);
returns # bytes written or -1 (error)

Below is the code you are to complete. You should assume that all necessary header files have been `#included` and you do not need write any other `#includes`.

```
#define BUFFER_SIZE 1024

void copy_file(const char *source_file, const char *dest_file) {

    int source_fd = ________________________________________;

    if (source_fd < 0) {

        perror("Error opening source file");

        return;

    }

    int dest_fd = _________________________________________;

    if (dest_fd < 0) {

        perror("Error opening destination file");

        ___________________(source_fd);  // Close source file

        return;

    }
```

(continued on next page)

Question 4. (cont.) Below, complete the rest of the function to copy the files.

```
    char buffer[BUFFER_SIZE];

    ssize_t bytes_read, bytes_written, nbytes;

    // ok to assume that input either works or doesn't, but does
    // not need to be retried if a failure is detected

    while ((bytes_read = _____________(source_fd,
                                buffer, BUFFER_SIZE)) > 0) {

        bytes_written = 0;
        while (bytes_written < bytes_read) {
            // fill in output function, buffer location, & length
            // (OK to define extra variables here if that helps)




            nbytes = ___________________(dest_fd,

                       ___________________________________________ ,

                       ___________________________________________);

            if (nbytes < 0) {
              perror("error writing to destination file");
              break;
            }
            bytes_written += nbytes;
        }
    }
    if (bytes_read < 0) {
        perror("Error reading from source file");
    }
    // close files

    close(source_fd);

    close (dest_fd);

}
```

Question 5. (20 points) Here is one of those slightly maddening C++ programs with a fairly simple class, which is a wrapper for an integer value, and a small program that uses it. The code compiles and executes with no errors. In the box on the right, write the output produced when it runs.

You should assume that all copy constructors, constructors, and destructors are called as specified by the C++ language and not eliminated by possible compiler optimizations

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

class Int {
public:
  Int(): n_(17)      { cout << "default ctr 17" << endl; }
  Int(int n): n_(n)  { cout << "ctr " << n_ << endl; }
  Int(const Int &other): n_(other.n_)
                     {cout << "copy ctr " << n_ << endl;}
  Int &operator=(const Int & other) {
    cout << "op= " << n_ << "=" << other.n_ << endl;
    if (this == &other) return *this;
    n_ = other.n_;
    return *this;
  }
  ~Int() { cout << "dtr " << n_ << endl; }
private:
  int n_;
};

// Return copy of value parameter n
Int cloneval(Int n) {
  return n;
}

// return copy of reference parameter n
Int cloneref(Int &n) {
  return n;
}

int main() {
  Int n1 = 42;
  Int n2 = n1;
  cout << "--1--" << endl;
  Int n3;
  n3 = n1;
  cout << "--2--" << endl;
  n3 = cloneval(n1);
  cout << "--3--" << endl;
  n3 = cloneref(n1);
  cout << "--4--" << endl;
  return EXIT_SUCCESS;
}
```

Write the program output here

Question 6. (8 points) Recall the small C++ string class from lecture. `Str.h` defines the class as follows:

```
class Str {
 public:
  Str();                // default constructor
  Str(const char *s);   // c-string constructor
  Str(const Str &s);    // copy constructor
  ~Str();               // destructor

  // operations
  int length() const;       // return length of this STR string
  Str &operator=(const Str &s);   // assignment

  // stream output
  friend std::ostream &operator<<(std::ostream &out, const Str &s);

 private:
  char *st_;  // c-string on heap with data bytes terminated by '\0'
};
```

One of our colleagues is experimenting with this class to see if they understand how it works, and they decided to add a `*=` operator that would "multiply" the contents of a `Str` by concatenating it with itself the specified number of times. Here is an example:

```
  Str hi("howdy");   // use existing char* constructor to create Str hi
  hi *= 3;
  cout << hi << endl; // writes "howdyhowdyhowdy"
```

(a) (3 points) Give a declaration of the `*=` operator function as it would be written in class `Str` in the `Str.h` header file (i.e., what needs to be added to the above class definition?):

(b) (5 points) Give the implementation of the `*=` operator as it would appear in the `Str.cc` file that implements class `Str`. (To simplify things, you may assume the operand of `*=` is an integer > 0. Hint: the C-string operations summarized on the reference page at the end of the exam may be useful here, especially `strcat`.)

Question 7. (2 free points) (All reasonable answers receive the points. All answers are reasonable as long as there is an answer. ☺)

(a) (1 point) What question were you expecting to appear on this exam that wasn't included?

(b) (1 point) Should we include that question on the final exam? (circle or fill in)

Yes

No

Heck No!!

$!@$^*% No !!!!!

Yes, yes, it *must* be included!!!

No opinion / don't care

None of the above. My answer is _____.

Extra space for answers, if needed. Please be sure to label which question(s) are answered here, and be sure to put a note on the question page so the grader will know to look here.

Extra space for answers, if needed. Please be sure to label which question(s) are answered here, and be sure to put a note on the question page so the grader will know to look here.

Reference information. Here is a collection of information that might, or might not, be useful while taking the test. You should **remove this page** from the exam. **Do not write on this page. It will not be scanned for grading.**

Memory management (<stdlib.h>)

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

Strings and characters (<string.h>, <ctype.h>)

Some of the string library functions:

- char* strncpy(*dest*, *src*, *n*), copies exactly *n* characters from *src* to *dst*, adding '\0's at end if the '\0' at the end of the string *src* is found before *n* chars copied.
- char* strcpy(*dest*, *src*), same as strncpy but with no length check
- char* strncat(*dest*, *src, n*), Appends the first *n* characters of *src* to *dst*, plus a terminating null-character. If the length of the C string in *src* is less than *n*, only the content up to the terminating null-character is copied.
- char* strcat(*dest*, *src*), same as strncat but with no length check
- int strncmp(*string1*, *string2*, *n*), <0, =0, >0 if compare <, =, >
- int strcmp(*string1*, *string2*)
- char* strstr(*string, search_string*)
- int strnlen(*s, max_length*), # characters in *s* not including terminating '\0'
- int strlen(*s*)
- Character tests: isupper(*c*), islower(*c*), isalpha(*c*), isdigit(*c*), isspace(*c*)
- Character conversions: toupper(*c*), tolower(*c*)

Files (<stdio.h>)

Some file functions and information:

- Default streams: stdin, stdout, and stderr.
- FILE* fopen(*filename*, *mode*), modes include "r" and "w"
- char* fgets(*line, max_length, file*), returns NULL if eof or error, otherwise reads up to max-1 characters into buffer, including any \n, and adds a \0 at the end
- size_t fread(buf, 1, count, FILE* f)
- size_t fwrite(buf, 1, count, FILE* f)
- int fprintf(format_string, data…, FILE *f)
- int feof(*file*), returns non-zero if end of *file* has been reached
- int ferror(FILE* f), returns non-zero if the error indicator associated with f is set
- int fputs(*line, file*)
- int fclose(*file*)

A few printf format codes: %d (integer), %c (char), %s (char*)

More reference information, C++ this time. **Do not write on this page. It will not be scanned for grading.**

C++ strings

If `s` is a string, `s.length()` and `s.size()` return the number of characters in it. Subscripts (`s[i]`) can be used to access individual characters. The usual comparison operators can be used to compare strings, and the operator + can be used to concatenate strings.

C++ STL

- If `lst` is a STL `vector`, then `lst.begin()` and `lst.end()` return iterator values of type `vector<...>::iterator`. STL `lists` and `sets` are similar.
- A STL `map` is a collection of `Pair` objects. If p is a `Pair`, then `p.first` and `p.second` denote its two components. If the `Pair` is stored in a `map`, then `p.first` is the key and `p.second` is the associated value.
- If `m` is a `map`, `m.begin()` and `m.end()` return iterator values. For a `map`, these iterators refer to the `Pair` objects in the `map`.
- If `it` is an iterator, then `*it` can be used to reference the item it currently points to, and `++it` will advance `it` to the next item, if any.
- Some useful operations on STL containers (lists, maps, sets, etc.):
  - `c.clear()` – remove all elements from c
  - `c.size()` – return number of elements in c
  - `c.empty()` – true if number of elements in c is 0, otherwise false
- Additional operations on `vectors`:
  - `c.push_back(x)` – copy x to end of c
- Some additional operations on `maps`:
  - `m.insert(x)` – add copy of `x` to `m` (a key-value pair for a `map`)
  - `m.count(x)` – number of elements with key `x` in `m` (0 or 1)
  - `m[k]` can be used to access the value associated with key `k`. If `m[k]` is read and has never been accessed before, then a <key,value> `Pair` is added to the map with `k` as the key and with a value created by the default constructor for the value type (0 or `nullptr` for primitive types).
- Some additional operations on `sets`
  - `s.insert(x)` – add `x` to `s` if not already present
  - `s.count(x)` – number of copies of `x` in `s` (0 or 1)
- You may use the C++11 `auto` keyword, C++11-style `for`-loops for iterating through containers, and any other features of standard C++11, but you are not required to do so.