COS 217, Spring 2022
Final Exam

This exam consists of 7 questions, and you have 180 minutes – budget your time wisely. ***Do all of your work on these pages (using provided blank pages for scratch space) and give your answers in the space provided.*** Assume the ArmLab/Linux/C/gcc217 environment unless otherwise stated.

This is a closed-book, closed-note exam, with a 1-page study sheet ***allowed***. Please place items that you will not need out of view in your bag or under your working space at this time. Electronic devices such as cell phones, laptops, music players, etc. may not be used during this exam.

Name: NetID:

Precept (***circle one***):	1: MW 1:30	Christopher Moretti	4:	TTh 1:30 Huihan Li
	2: MW 3:30	Christopher Moretti	7:	TTh 3:30 Cedrick Argueta
	3: TTh 12:30	Maxine Perroni-Scharf		

This examination is administered under the Princeton University Honor Code. Students should sit one seat apart from each other, and refrain from talking to other students during the exam. All suspected violations of the Honor Code must be reported to honor@princeton.edu.

In the box below, write out and sign the Honor Code pledge before turning in the test:

"I pledge my honor that I have not violated the Honor Code during this examination."

Pledge and Signature:

For instructor use only:

Question	**Max Points**	**Points Earned**	**Question**	**Max Points**	**Points Earned**
1	**15**		**5**	**25**	
2	**20**		**6**	**5**	
3	**18**		**7**	**15**	
4	**22**		**TOTAL**	**120**	

(blank page)

Q1. Truthiness and Falsiness (15 points)

Circle **T** (True) or **F** (False) for each statement.

Answer	**Statement**
T **F**	The pre-processor adds the contents of other files to a C file.
T **F**	The compiler determines which variables go into the rodata, data, and bss sections.
T **F**	The compiler can detect memory leak errors in the code.
T **F**	The assembler generates assembly language code.
T **F**	The linker creates relocation records.
T **F**	The linker patches object code with virtual memory addresses.
T **F**	The expression `0xFF – 0xEE` evaluates to `11` in decimal.
T **F**	If `char s[5] = “mom”;` then the expression `*(s+2) – *s` evaluates to `0`.
T **F**	The expression `sizeof(7UL) > sizeof(1L)` evaluates to `1` (true).
T **F**	AArch64 is a *Complex Instruction Set Computer* (CISC) architecture.
T **F**	The *stack memory* can run out of space due to an unbounded number of function calls.
T **F**	*Statement* testing can require more test cases than *Path* testing.
T **F**	*Stress* testing is re-running all test cases after fixing a bug.
T **F**	A *dangling pointer* is when you fail to free some chunk of allocated memory.
T **F**	An optimizing compiler can perform *function inlining* to improve performance.

Q2. Is this in scope? (20 points)

Consider the following C code in `file1.c`:

```
int a;
int b = 0;
static int c = 1;

static int DoubleIt(int input) {
   int d = 2;
   static int e = 3;
   static int f;
   int g = d * input;
   return g;
}
```

(a) Fill in the blanks in the following table: (15 points)

variable	**Scope**	**Linkage**	**Duration**
`a`	file	external	process
`b`			
`c`			
`d`			temporary
`e`			
`f`	block		
`g`		internal	

(b) Which section of memory is `f` in? (2 points)

We now create a second C file, `file2.c`, which includes the following declarations outside any function, plus a main function (not shown):

```
extern int a;
int b = 0;
static int c = 2;
```

(c) When we build `file1.c` and `file2.c` (and no other files) together into a program, which of the three lines from `file2.c` will cause an error? (1 point)

(d) What is the problem? (1 point)

(e) Which stage of the build process will report the error? (1 point)

Q3. Modular Table (18 points)

Consider the following excerpts from an interface (top box) and implementation of that interface (bottom box) for a variant of the symbol table module you wrote in Assignment 3:

Interface:

```
/* A SymTable_T is a set of key-value pairs */
typedef struct SymTable* SymTable_T;

/* instantiate and return a new empty
SymTable_T object with k buckets, or NULL
if there is insufficient memory */
SymTable_T ST_new(size_t k);
```

Implementation:

```
typedef struct binding* Binding_T;
struct binding {
   char* key;
   void* val;
   Binding_T next;
};
struct SymTable {
   Binding_T* buckets;
   size_t count;
   size_t numBuckets;
};
SymTable_T ST_new(size_t k) {
 /* implementation not shown */
}
```

(a) The interface (top box) above violates an ADT design principle from the Modularity lecture. What is the problem found in the interface above? (1 point)

(b) The following lines of code make up the body of a new interface function `ST_maxValue`, which takes an instance pointer `st` and a comparison function (with `strcmp` semantics) `comp` and returns the value (not the key!) in the symbol table with the maximal value based on the comparison function. These lines, however, have become scrambled and have lost their indentation. In the 13 blanks below, write the line numbers in their proper order that results in a correct implementation of this function. (13 points)

```
1     } /* end while */
2     if(max) return max->val;
3     size_t i;
4     Binding_T c = st->buckets[i];
5     return NULL;
6     if(!max || (*comp)(c->val, max->val) >0)
7     } /* end for */
8     while(c != NULL) {
9     Binding_T max = NULL;
10    c = c->next;
11    for(i = 0; i < st->numBuckets; i++) {
12    max = c;
13    assert(st != NULL && comp != NULL);
```

___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___ ___

(c) What is the type of the `comp` parameter in the `ST_maxValue` function? (2 points)

(d) Now consider a representation invariant for the symbol table that each bucket is stored in descending order (i.e., for any valid symbol table, the largest value in each bucket is at the head of that list and the smallest value is at the tail of that list). Describe in two sentences or less a simplification that can be made to the `ST_maxValue` function above by taking advantage of this constraint. (2 points)

Q4. Buggy Parents (22 points)

The function `getParentPath` should return a defensive copy of a string representing the *parent* of a `path` parameter like in Assignment 4. If `path` is the root, the string returned should be the empty string.

You should assume that all relevant header files have been `#include`d. You may assume the path passed in as an argument is well formed by the standards of Assignment 4: it is a non-empty `'\0'`-terminated string that neither begins nor ends with a `'/'` character and does not contain consecutive `'/'` delimiters.

Here are some concrete examples of the function argument and resulting return values:
`path`: `"cos"`, i.e. `{'c', 'o', 's', '\0'}` should return a new string with contents: `""`, i.e. `{'\0'}`
`path`: `"cos/217"` should return a new string with contents: `"cos"`
`path`: `"cos/217/forever!/Yay:)"` should return a new string with contents: `"cos/217/forever!"`

Now consider this implementation:

```
1     char* getParentPath(const char* path) {
2        char* parent;
3        char* end = path;
4        assert(path != NULL);
5        end += strlen(path);
6        while(*end != '/') end--;
7        parent = malloc(end - path);
8        strncpy(parent, path, end – path);
9        return parent;
10    }
```

As a reminder, the `string.h` interface function `strncpy(dest, src, n)` copies the first `n` bytes of `src` to `dest`. If `src`'s length is less than `n`, `strncpy` writes null bytes to `dest` until `n` bytes have been written.

(a) Unfortunately, the implementation above is buggy!

For this problem, the following count as bugs: warnings or errors from `gcc217`, runtime crashes, incorrect behavior, inefficiency as judged in Assignments 2 and 3 (i.e., extra traversals beyond what is required), or dynamic memory management issues observable by MemInfo or Valgrind.

In the spaces below, identify 3 *distinct* bugs (there may be more than that in the code, and they could be bugs of any kind, but you should provide only 3). To identify each bug, list the line number(s) where the bug occurs and a one-sentence description of the bug. (9 points)

Bug 1:

Bug 2:

Bug 3:

(b) Write a function comment for `getParentPath` that accords to the COS 217 assignment standards for establishing a contract with clients who will call your function. (5 points)

(c) Define a DFA (in either the visual format from lecture or the textual format from Assignment 1) that accepts only paths that are well formed as specified above. Here is that description, repeated for convenience:

A non-empty `'\0'`-terminated string that neither begins nor ends with a `'/'` character and does not contain consecutive `'/'` characters.

Remember to identify your start state, classify each state as accept or reject, label all state transitions, and name each state with a semantically meaningful name. (8 points)

Q5. Mysterious Assembly (25 pts)

An AArch64 reference is provided at the end of this question. Consider the following AArch64 program:

```
        .section .rodata
scanFormat:
        .string "%d"
printFormat:
        .string "Answer is %d\n"

        .section .text
        .global mystery
mystery:
        sub     sp, sp, 48
        str     x30, [sp]
        str     x19, [sp, 8]
        str     x20, [sp, 16]
        str     x21, [sp, 24]

        mov     w19, wzr
        add     w20, w19, 1
        add     w21, w19, 2
loop1:
        cmp     w20, w0
        bgt     endloop1
        mul     w20, w20, w21
        add     w19, w19, 1
        b       loop1
endloop1:
        sub     w0, w19, 1

        ldr     x19, [sp, 8]
        ldr     x20, [sp, 16]
        ldr     x21, [sp, 24]
        ldr     x30, [sp]
        add     sp, sp, 48
        ret
        .size   mystery, .-mystery

        .global main
main:
        /* prologue */
        sub     sp, sp, 16
        str     x30, [sp]
        /* call to scanf */
        adr     x0, scanFormat
        add     x1, sp, 8
        bl      scanf
        /* call to mystery */
        ldr     w0, [sp, 8]
        bl      mystery
        /* call to printf */
        mov     w1, w0
        adr     x0, printFormat
        bl      printf
        /* return 0 and epilogue */
        mov     w0, 0
        ldr     x30, [sp]
        add     sp, sp, 16
        ret
     .size   main, .-main
```

(a) Consider first the function `mystery`. How many parameters does it take as input? (1 point)

(b) Does it return a value? (Yes, or No?) (1 point)

(c) Write flattened-C corresponding to the function `mystery`. Assume that registers `w19, w20, w21` correspond to local variables `r, p, m,` respectively. Use the same label names as the assembly language code. (12 points)

C code:

(d) Now consider the function `main`. Note that it has calls to `scanf, mystery`, and `printf`. What does this program print if the user provides input 16? (1 point)

(e) What does this program print if the user provides input 15? (1 point)

(f) What does this program print if the user provides an input that is 0 or a negative number? (1 point)

(g) What does the function `mystery` do? (*Hint*: Try out the program on a few other positive numbers if you like.) Show its specification in the form of a function comment that meets the requirements from your programming assignments in this course. (4 points)

(h) List *two distinct* optimizations (i.e., not the same optimization on two different variables) that you can perform on the assembly code for `mystery` to make the program run faster, and state how they would help. (4 points)

You *don't need to show* the optimized assembly code, just briefly describe the optimizations for the specific given code (not just a general technique).

AArch64 quick reference	
`add/sub/mul  dst, src1, src2` Add/subtract/multiply src1 and src2, put result in dst. **`mov  dst, src`** Copy src to dst **`ldr  dst, [src]`** Load word or quad from memory pointed to by src into dst **`str  src, [dst]`** Store word or quad from src to memory pointed to by dst **`cmp  src1, src2`** Compare two registers, set condition flags **`b label`** Unconditional branch to label **`bgt  label`** Branch to label if greater than (signed) **`bl  label`** Branch to label and save return address in x30 **`adr dst, var`** Put address of var in dst **`ret`** Return, i.e., branch to address in x30	**`x0..x30, xzr`** 64-bit (quad) registers **`w0..w30, wzr`** 32-bit (word) registers **`r0..r7`** Parameters, scratch space, caller-saved **`r0`** Return value from function **`r9..r15`** Scratch, caller-saved **`r19..r28`** Scratch, callee-saved **`x30`** Link register (return address) **`zr`** Always holds zero

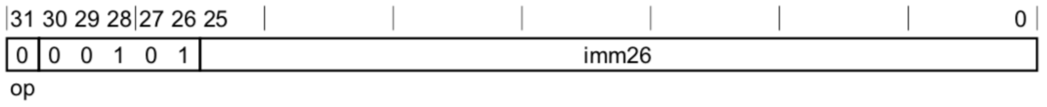
Q6. Missing Branch (5 points)

Consider again the assembly language program from the previous question. This time we will focus on a small part of the `objdump` output shown below, which shows the disassembled instructions for the loop in the `mystery` function.

```
…

0000000000000020 <loop1>:
  20:6b00029f   cmp  w20, w0
  24:5400008c   b.gt 34 <endloop1>
  28:1b157e94   mul  w20, w20, w21
  2c:11000673   add  w19, w19, #0x1
  30:MISSING    b    20 <loop1>

0000000000000034 <endloop1>:
  34:51000660   sub  w0, w19, #0x1

…
```

MISSING:

In the box above on the right, provide the machine language encoding which is **MISSING** for only the branch (`b`) instruction highlighted in red, for which the machine language format is shown here:



26-bit signed PC-relative branch offset variant

```
B <label>
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler symbols

`<label>`	Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
BranchTo(PC[] + offset, BranchType_JMP);
```

- Note that the first column lists the offset of each instruction (in hexadecimal), which you can use to compute the displacement that is needed to fill the correct value in the instruction format.
- Recall also that each machine language instruction is 32-bits, so a machine language instruction can be written as 8 hexadecimal digits (hexits), as shown in the `objdump` output above. You will likely want to work in binary, however your answer should be in hex.

(blank page)

Q7. Finally, some fun! (15 points)

For each code snippet, give a short (at most one line) description of what the program does. Assume all necessary header files are included.

(a) What does the following function print to `stdout` for a non-negative number `a`? (*5 points*)

```
void funA(unsigned int a) {
   if (a & (~1))
      funA(a>>1);
   putchar('0'+(a & 1));
}
```

(b) How does `funB()` change the string passed to it? Recall that the function `toupper` converts a lower-case letter to the corresponding upper-case letter. (*5 points*)

```
char* funB(char *pStr) {
  int i, n = strlen(pStr);

  for (i = 3; i < n; i += 3)
    pStr[i] = toupper(pStr[i]);

  return pStr;
}
```

(c) For what kind of non-negative numbers n does the following function return 1? (*5 points*)

```
int funC(unsigned int n) {
  if (n & (n-1))
     return 0;
  if (n & 0x55555555)
     return 1;
  else
     return 0;
}
```

(blank page)