# CPSC 213 Midterm

196

TOTAL POINTS

**56.332 / 66**

QUESTION 1

## Memory and Numbers 8 pts

### 1.1 0x2010 LE 1.332 / 1.332

✓ + 0 pts Question Graded or Empty
✓ + 0.333 pts 0x2010 = 0x0d
✓ + 0.333 pts 0x2011 = 0xf0
✓ + 0.333 pts 0x2012 = 0xed
✓ + 0.333 pts 0x2013 = 0xfe
+ 0.666 pts Correct, but wrong endianess

### 1.2 0x2010 BE 1.332 / 1.332

✓ + 0 pts Question Graded or Empty
✓ + 0.333 pts 0x2010 = 0xfe
✓ + 0.333 pts 0x2011 = 0xed
✓ + 0.333 pts 0x2012 = 0xf0
✓ + 0.333 pts 0x2013 = 0x0d
+ 0.666 pts correct, but endianess flipped

### 1.3 0x2014 LE 1.336 / 1.336

✓ + 0 pts Question Graded or Empty
✓ + 0.334 pts 0x2014 = 0xf0
✓ + 0.334 pts 0x2015 = 0xff
✓ + 0.334 pts 0x2016 = 0xff
✓ + 0.334 pts 0x2017 = 0xff
+ 0.334 pts Incorrect: 0x2015-0x2017 = Unknown
+ 0.666 pts Correct, but endianess flipped

### 1.4 0x2014 BE 0.668 / 1.336

✓ + 0 pts Question Graded or Empty
+ 0.334 pts 0x2014 = 0xff
✓ + 0.334 pts 0x2015 = 0xff
✓ + 0.334 pts 0x2016 = 0xff
+ 0.334 pts 0x2017 = UNKNOWN
+ 0.334 pts incorrect 2014-2016 unknown
+ 0.166 pts incorrect: 0x2014 = 0xf0

### 1.5 0x2018 LE 1.332 / 1.332

✓ + 0 pts Question Graded or Empty
✓ + 0.333 pts 0x2018 = 0x14
✓ + 0.333 pts 0x2019 = 0x20
✓ + 0.333 pts 0x201a = 0x00
✓ + 0.333 pts 0x201b = 0x00
+ 0.666 pts correct, but endianess flipped

### 1.6 0x2018 BE 1.332 / 1.332

✓ + 0 pts Question Graded or Empty
✓ + 0.333 pts 0x2018 = 0x00
✓ + 0.333 pts 0x2019 = 0x00
✓ + 0.333 pts 0x201a = 0x20
✓ + 0.333 pts 0x201b = 0x14
+ 0.666 pts correct, but endianess flipped

QUESTION 2

## Global Variables and Arrays 8 pts

### 2.1 Translate Assembly (Global Arrays) 6 / 6

✓ + 0 pts Question Graded or Empty
✓ + 1 pts Load value of i
✓ + 1 pts Load value of a[i]
✓ + 1 pts Load value of b
✓ + 1 pts Load b[a[i]]
✓ + 1 pts Subtract 1 from b[a[i]] (or decrement)
✓ + 1 pts Store b[i]
+ 0.5 pts Optional: Substantial Assembly Comments
+ 0.5 pts Optional: Correct Assembly Comments
- 0.5 pts minor typo

### 2.2 Count Memory Reads 2 / 2

✓ + 0 pts Question Graded or Empty
✓ + 2 pts Answer = 5
+ 1 pts Answer = 4 with work
+ 0.5 pts Answer = 3 with work

+ **0.5 pts** Answer = 6 with work

QUESTION 3

## Structs and Instance Variables 8 pts

### 3.1 Calculate Struct Sizes 1 / 3

+ **0 pts** Completely incorrect or Blank
+ **0.5 pts** sizeof(A) = 20
+ **0.5 pts** sizeof(B) = 28
✓ **+ 0.5 pts** **A.d[1]: 8**
✓ **+ 0.5 pts** **A.e: 16**
+ **0.5 pts** B.a.c[1]: 1
+ **0.5 pts** B.x: 24

### 3.2 Translate Assembly (Structs) 4.5 / 5

✓ **+ 0 pts** **Question Graded or Empty**
✓ **+ 1 pts** **Load value of a**
✓ **+ 0.5 pts** **Load a->b (any offset)**
✓ **+ 0.5 pts** **Load a->b (correct offset 12)**
✓ **+ 0.5 pts** **Load a->b[1].x (any offset)**
+ **0.5 pts** Load a->b[1].x (correct offset 52)
✓ **+ 1 pts** **Shift left**
✓ **+ 0.5 pts** **Store to b.x (any offset)**
+ **0.5 pts** Store to b.x (correct offset 24)
✓ **+ 0.5 pts** **Optional: Substantial Assembly Comments**
+ **0.5 pts** Optional: Correct Assembly Comments
- **0.5 pts** minor mistake

QUESTION 4

## 4 Static Control Flow 7.5 / 8

✓ **+ 0 pts** **Question Graded or Empty**
✓ **+ 1 pts** **Load x into y (can use temporary for y)**
✓ **+ 1 pts** **Test y against 0 (beq)**
✓ **+ 1 pts** **Load y->a**
✓ **+ 1 pts** **Add 3**
✓ **+ 1 pts** **Test y->a+3 against 0 (bgt or bgt+beq)**
✓ **+ 1 pts** **Call foo**
✓ **+ 1 pts** **Load y->b into y (offset 4) (or temporary y)**
✓ **+ 1 pts** **Correct loop structure**
✓ **- 0.5 pts** **If temporary y used, did not store back to y at end**
+ **0.5 pts** Optional: Substantial Assembly Comments
+ **0.5 pts** Optional: Correct Assembly Comments
- **0.5 pts** minor mistakes

QUESTION 5

## 5 C Pointers 4.5 / 8

✓ **+ 0 pts** **Question Graded or Empty**
✓ **+ 1 pts** **b = 0x3008 or &c[2]**
+ **1.75 pts** c[1] = 7
✓ **+ 1.75 pts** **c[2] = 2**
✓ **+ 1.75 pts** **c[5] = 6**
+ **1.75 pts** a[7] = 8
- **0.75 pts** One extra entry

QUESTION 6

## Dynamic Allocation 6 pts

### 6.1 Dynamic 1 1 / 1

+ **0 pts** Incorrect or Empty
+ **0.5 pts** Correct explanation, but did not write Memory Leak
✓ **+ 1 pts** **Correct: Memory Leak**

### 6.2 Dynamic 2 1 / 1

+ **0 pts** Incorrect or Empty
+ **0.5 pts** Dangling Pointer, incorrect explanation
✓ **+ 1 pts** **Correct: Dangling Pointer/References Freed Memory**

### 6.3 Dynamic 3 1 / 1

+ **0 pts** Incorrect
✓ **+ 1 pts** **Correct: No Error/Nothing Wrong**

### 6.4 Dynamic 4 0.5 / 1

+ **0 pts** Question Graded or Empty
+ **1 pts** Correct: Invalid Free
✓ **+ 0.5 pts** **Partially Correct: Dangling Pointer**
+ **0.5 pts** Partially Correct: *r is not dynamic
+ **0.5 pts** Partially Correct: Identified error type but not the variable that causes it

### 6.5 Dynamic 5 1 / 1

+ 0 pts Incorrect or Empty

✓ + 1 pts **Correct: (Potential) Dangling Pointer**

+ 1 pts Correct: (Potential) Invalid/Double Free

### 6.6 Dynamic 6 1 / 1

+ 0 pts Incorrect or Empty

✓ + 1 pts **Correct: No Error/Nothing Wrong**

QUESTION 7

## Reference Counting 10 pts

### 7.1 Predict Output 2 / 2

+ 0 pts Question Graded or Empty

✓ + 2 pts **Correct: 23 42 42 42**

+ 1 pts Incorrect: 23 23 42 42, OR at least two correct out of the four numbers

- 0.5 pts minor mistake

### 7.2 stack_push Memory Problem 0 / 2

✓ + 0 pts **Question Graded or Empty**

+ 2 pts Identify that this is always a memory leak

+ 1 pts Identify that the capacity of the stack could cause a problem.

+ 1 pts Incorrect potential memory leak

### 7.3 Predict Refcounts 2 / 2

+ 0 pts Question Graded or Empty/ Incorrect answers

✓ + 2 pts **Correct: *e1 = 3, *e2 = 4**

+ 1 pts Incorrect: *e1 = 3, *e2 = 5 (Ignored the stack capacity)

### 7.4 Implement Refcounting 2.5 / 4

✓ + 0 pts **Graded or Blank**

✓ + 1.25 pts **keep_ref new value in stack_push**

- 0.5 pts Error: free_ref in stack_push (unlike arrays, stack entries past index are always unoccupied)

✓ + 1.25 pts **free_ref element in stack_pop or stack_delete**

- 0.5 pts Error: free_ref in stack_pop - this returns a dangling pointer

+ 0.5 pts free_ref stack

+ 0.5 pts free_ref at least one element in main

+ 0.5 pts free_ref all elements in main

- 0.5 pts any other change that produces incorrect refcounts

- 0.5 pts syntax error (no argument to keep/free or argument is not valid)

- 0.5 pts any other change that creates a dangling pointer

QUESTION 8

## Reverse-Engineering Assembly 10 pts

### 8.1 Comment Assembly 3.5 / 4

✓ + 0 pts **Question Graded or Empty**

✓ + 1 pts **Comments correct start..L0**

✓ + 1 pts **Comments correct L0.."not r6"**

✓ + 1 pts **Comments correct "not r6"..L1**

✓ + 1 pts **Comments correct L1..end**

✓ - 0.5 pts **Minor mistake (e.g. not labeling conditions for branches, taking y[0] instead y[i], etc)**

1 r++ doesn't make sense

### 8.2 Reverse to C 6 / 6

✓ + 0 pts **Question Graded or Empty**

✓ + 0.5 pts **Correct definition of global "int x"**

✓ + 0.5 pts **Correct definition of global "int *y"**

✓ + 0.5 pts **Correct definition of global "int z[...]"**

✓ + 1 pts **for or while loop**

✓ + 0.5 pts **loop variables initialized ("i"=0 and "j"=1)**

✓ + 0.5 pts **loop test is correct ("i" != 32)**

✓ + 0.5 pts **loop continuation "i"++**

✓ + 1 pts **correct test y["i"] == z["i"]**

✓ + 0.5 pts **correct addition x = x+"j"**

✓ + 0.5 pts **loop continuation "j" <<= 1 or "j" = "j"*2**

- 0.5 pts minor mistake

### 8.3 Explain C (BONUS) 2 / 0

✓ + 0 pts **Question Graded or Empty**

✓ + 1 pts **Correct explanation**

✓ + 1 pts **Correctly identify that the bits in x are set**

**according to equality**

+ **1 pts** Correct explanation of reverse-engineered code, although reverse-engineered code is wrong

+ **0 pts** Incorrect Answer

**1** (8 marks) **Memory and Numbers.** Consider the execution of the following code. Assume that the compiler has statically allocated `i` to start at address `0x2010`, and consecutively allocates the subsequent variables, inserting padding (wasting memory) only where needed to ensure that each variable is aligned.

Assume too that `char`'s are **signed**, and `int`'s and pointers are **4 bytes** long. Prior to the execution of `foo()`, assume that the memory contents are indeterminate (i.e. anything could hold any value).

```
int  i;
char c[4];
int *j;
```

```
void foo() {
    i = 0xfeedf00d;
    j = (int *)&c[0];
    c[0] = i >> 8;
    *j = (*j) | (int)(*c);
}
```

Assuming that the code above executes on a **little-endian** machine, give the value in hex stored at each memory address below under the "LE" column, or write UNKNOWN if the value cannot be determined completely. Repeat for a big-endian machine in the "BE" column.

| | | LE | BE |
|---|---|---|---|
| 0x2010: | LE | 0d | BE fe |
| 0x2011: | LE | f0 | BE ed |
| 0x2012: | LE | ed | BE f0 |
| 0x2013: | LE | fe | BE 0d |
| f0 | 0x2014: | LE f0 | ed BE ed |
| ff | 0x2015: | LE ff | ff BE ff |
| ff | 0x2016: | LE ff | ff BE ff |
| ff | 0x2017: | LE ff | ff BE ff |
| 0x2018: | LE | 14 | BE 00 |
| 0x2019: | LE | 20 | BE 00 |
| 0x201a: | LE | 00 | BE 20 |
| 0x201b: | LE | 00 | BE 14 |

**2** (8 marks) **Global Variables and Arrays.** Answer the following questions about these global variables.

```
int    i;
int a[5];
int   *b;
```

**2a** Translate this C statement into assembly: `b[i] = b[a[i]] - 1.`

```
ld $i, r0
ld (r0), r0        # r0 = i
ld $a, r1
ld (r1,r0,4), r1   # r1 = a[i]
ld $b, r2
ld (r2), r2
ld (r2,r1,4), r1   # r1 = b[a[i]]
dec r1             # r1 = r1 - 1
st r1, (r2,r0,4)   # b[i] = r1
```

**2b** What is the *minimum* number of memory reads required to execute this statement? Assume you have enough registers so that no value needs to be read twice. Fill in a single multiple choice option below.

```
b[i] = b[i] + b[a[0]];
```

0○ 1○ 2○ 3○ 4○ 5● 6○ 7○ 8○

b a[0] b[a[0]] i b[i]

i

b

b[i]

b[a[0]]

**3** (8 marks) **Structs and Instance Variables.** Answer the following questions about these structures and variables. Assume that `int`'s and pointers are **4 bytes** long.

```
struct A {
    char c[2];
    int d[2];
    struct B *b;
    char e;
};
```

```
struct B {
    struct A a;
    char y;
    int x;
};
struct A *a;
struct B  b;
```

**3a** Calculate the following values. Recall that `sizeof` obtains the size (in bytes) of the given structure, and `offsetof` obtains the offset (in bytes) of the given field from the start of the structure.

`sizeof(struct A)`: 24

`sizeof(struct B)`: 32

`offsetof(struct A, d[1])`: 8

`offsetof(struct A, e)`: 16

`offsetof(struct B, a.c[1])`: 2

`offsetof(struct B, x)`: 28

**3b** Give assembly code for the C statement: `b.x = a->b[1].x << 2;`.

```
ld $a, r0        # &a
ld (r0), r0      # a
ld 12(r0), r0    # a->b
ld 60(r0), r0    # a->b[1].x
shl $2, r0       # a->b[1].x << 2
ld $b, r1        # &b
st r0, 28(r1)    # b.x = r0
```

**4 (8 marks) Static Control Flow.** Answer the following question on static control flow.

Assume the following global declarations exist.

```
struct X {
    int a;
    struct X *b;
};
struct X *x, *y;
void foo();
```

Give assembly code for the following. Insert a halt at the end of the program. You may assume that `foo()` does not modify any registers.

```
for(y = x; y != NULL; y = y->b) {
    if(y->a + 3 <= 0)
        foo();
}
/* halt */
```

assume typo: has to be x=y→b or can't loop

```
    ld $x, r0       # r0 = &x
    ld (r0), r0     # r0 = x
    mov r0, r1      # r1 = x, assume r1 = y for consistent
loop: beq r1, end   # end if y point to nothing
                    # assume 0 == NULL

    ld (r1), r2     # r2 = y->a
    inc r2
    inc r2
    inc r2          # r2 = y->a + 3
    bgt r2, loopEnd # loop back if !(y->a + 3 <= 0) == (y->a + 3 > 0)
    gpc $6, r6      # rmb return address
    j foo           # jump to foo
    br loopEnd      # loop back
end: halt
loopEnd: ld 4(r1), r2   # r2 = b
    mov r2, r1      # r1 = b
    br loop
```

**5** **(8 marks)** **C Pointers.** Consider the following global variable declarations.

```
int a[8] = {1, 3, 6, 10, 15, 21, 28, 36};
int *b;
int c[7] = {2, 4, 6, 8, 10, 12, 14};
```

Assume that the address of `a` is `0x1000`, the address of `b` is `0x2000`, and the address of `c` is `0x3000`. Now, consider the the execution of the following code.

```
b = &c[4];
b[-2] = a[0];
b = b - a[1];
c[b[1]] = 7;
a[c[1]] = b[2];
*(b + 4) = 6;
b = b + 1;
*b = *b + 1;
```

Following the execution of the code, list the value of `b`, followed by the name and value of all array entries that have changed. Leave any extra spaces blank.

b = _____

_____ [ _____ ] = _____

_____ [ _____ ] = _____

_____ [ _____ ] = _____

_____ [ _____ ] = _____

_____ [ _____ ] = _____

_____ [ _____ ] = _____

**6** (6 marks) **Dynamic Allocation.** Consider each of the following pieces of C code to determine whether it contains a memory-related problem. Identify which error(s) the snippet might exhibit: memory leak, dangling pointer, or other memory error.

Assume that array indices are always within bounds. If there are no errors, simply say so. If an error might only occur under certain conditions (e.g. certain behaviour of external functions), briefly describe such a condition. In all snippets, assume that any function called `magic` is defined elsewhere and has unknown behaviour. Don't fix any bugs.

Because the snippets are all quite similar, code **in bold** denotes code that was changed or added compared to the previous snippet.

**6a**

```
int *extract(int *a, int i) {
    int *r = malloc(sizeof(int));
    *r = a[i];
    return r;
}
```

```
int process(int *a, int s) {
    int *r = extract(a, s-1);
    return *r;
}
```

Mem leak. *r not freed in process

**6b**

```
int *extract(int *a, int i) {
    int *r = malloc(sizeof(int));
    *r = a[i];
    return r;
}
```

```
int process(int *a, int s) {
    int *r = extract(a, s-1);
    int *t = r;
    free(t);
    return *r;
}
```

Dangling pointer. *t is freed, so *r dangle and point to possibly occupied mem. Return garbage from process.

**6c**

```
int *extract(int *a, int i) {
    int *r = malloc(sizeof(int));
    *r = a[i];
    return r;
}
```

```
int process(int *a, int s) {
    int *r = extract(a, s-1);
    int r2 = *r;
    free(r);
    return r2;
}
```

no error

**6d**

```
int *extract(int *a, int i) {
    int *r = &a[i];
    return r;
}
```

```
int process(int *a, int s) {
    int *r = extract(a, s-1);
    int r2 = *r;
    free(r);
    return r2;
}
```

dangling pointer since a[i] is free as *r is freed in process.

**6e**

```
int *extract(int *a, int i) {
    int *r = malloc(sizeof(int));
    *r = a[i];
    return r;
}
```

```
void process(int *a, int s) {
    int *r = extract(a, s-1);
    magic(r);
    free(r);
}
```

possible dangling pointer since magic(r) might make & keep copy of *r

**6f**

```
int *extract(int *a, int i) {
    int *r = malloc(sizeof(int));
    *r = a[i];
    return r;
}
```

```
void process(int *a, int s) {
    int *r = extract(a, s-1);
    magic(r[0]);
    free(r);
}
```

no error

**7** (10 marks) **Reference Counting.** Consider the following program, implementing a stack which contains dynamically allocated objects that should be managed using reference counting. Calls to `rc_malloc` have been added for you; recall that `rc_malloc` sets the allocated object's reference count to **1**.

**7a** What does this program print when it executes?

23 42 42 42

**7b** The following line can result in a memory problem due to our specification of the stack functions (see comments). What problem does it result in, and under what circumstances could that occur?

```
stack_push(s, element_new(16));
```

stack_push doesn't do rc_keep_ref when keeping ref in data. Might cause dangling pointer. Error if normal case where input stored.

**7c** Assuming this program implements reference counting correctly, give the reference counts for the following two objects when `printf` is called from main:

`*e1:` 3 (e1, data(0), data(1))

`*e2:` 4 (e2, e3, e4, data(2))

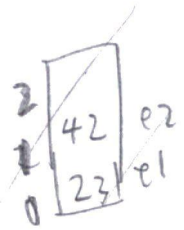
**7d** Add calls to `rc_keep_ref` and `rc_free_ref` to correctly implement reference counting for this program, such that **all** objects are freed by the time `main` completes. Do not add or remove any other code. Use the comments as a guideline to determine how functions should handle reference counts.

```
/* Stack structure. Don't modify. */
struct stack {
  int capacity;
  int index;
  int **data;
};

/* Create a new reference-counted element (an integer). Don't modify. */
int *element_new(int value) {
  int *e = rc_malloc(sizeof(int));
  *e = value;
  return e;
}

/* Create a new stack capable of holding a fixed number of elements. Don't modify. */
struct stack *stack_new(int capacity) {
  struct stack *s = rc_malloc(sizeof(struct stack));
  s->data = rc_malloc(sizeof(int *) * capacity);
  s->capacity = capacity;
  s->index = 0;
  return s;
}

/* Push an element onto the stack, adding a reference to the passed-in value
   if the element can be added */
void stack_push(struct stack *s, int *value) {

  if(s->index < s->capacity) {

    s->data[s->index] = value;

    s->index++;
    rc_keep_ref(value);
  }

}

/* Pop an element off the stack, transferring its reference to the caller. */
int *stack_pop(struct stack *s) {

  if(s->index > 0) {

    int *res = s->data[s->index - 1];
    s->data[s->index - 1] = NULL;
    s->index--;

    return res;

  }

  return NULL;
}
```

ref from stack (ex: do free)

00196

```
/* Delete the stack and all its elements */
void stack_delete(struct stack *s) {

  while(s->index > 0) {

int *c = stack_pop(s);
    rc_free_ref(c);
  }

}

int main() {

  struct stack *s = stack_new(3);

  int *e1 = element_new(23);

  int *e2 = element_new(42);

  stack_push(s, e1);

  stack_push(s, e2);

  int *e3 = stack_pop(s);

  stack_push(s, e1);

  stack_push(s, e2);

  int *e4 = stack_pop(s);

  stack_push(s, e3);

  stack_push(s, e4);

  printf("%d %d %d %d\n", *e1, *e2, *e3, *e4);

  stack_delete(s);

}
```




e3 = e2


e4 = e2

**8** (10 marks) (+2 bonus) **Reverse-Engineering Assembly.** Comment the following assembly code and then reverse-engineer it into C. *Use the back of the preceding page for extra space if you need it.*

```
        ld $0, r0           # r0 = 0 = condition counter = j
        ld $0, r1           # r1 = 0 = loop counter = i
        ld $1, r2           # r2 = 1 = incrementor = k
        ld $y, r3           # r3 = &y
        ld $z, r4           # r4 = &z → z = static array
        ld (r3), r3         # r3 = y  → y = pointer to array
L0:     ld $-32, r5         # r5 = 32
        add r1, r5          # r5 = r1 - 32
        beq r5, L3          # if (r1 == 32) L3, else L00
L00:    ld (r3,r1,4), r5    # r5 = y[i]
        ld (r4,r1,4), r6    # r6 = z[i]
        not r6              # r6 = ~r6
        inc r6              # r6 = -z[i]
        add r5, r6          # r6 = y[i] - z[i]
        beq r6, L1          # if (y[i] == z[i]) L1, else L01
L01:    br L2               # go to L2
L1:     add r2, r0          # r0 = r0 + r2
L2:     inc r1              # r1 = i++
        shl $1, r2          # r2 = r2 * 2
        br L0               # restart loop
L3:     ld $x, r3           # r3 = &x
        st r0, (r3)         # x = r0
        halt                # end fn
```

**8a** Translate into C. Include definitions of all global variables.

```
int i;
int j = 0;
int k = 1;
int *y;
int z[32]; // made up number for size
for (i=0; !(i==32); i++){
  if (y[i] == z[i]){
    j = j + k;
  }
  k = k*2;
}
```

**8b** BONUS (+2): Explain in one sentence what the code computes into the global variable x.

```
int x = j;
```

save bitwise and of y & z to x

compare 2 array & construct 32 bit int where bit at position i is 1 when y[i] == z[i], else bit is zero

*[This page has been left intentionally blank. If you write any answer you want graded on this page, YOU MUST indicate in the answer area for that question that you have work on this page, and indicate on this page what question you are answering.]*