CPSC 213, Winter 2014, Term 1 — Some More Sample Midterm Questions
Solution

Date: October 2014; Instructor: Mike Feeley

1 **(2 marks)** **Memory Alignment.** The memory address 0x2018 is aligned for certain byte size accesses but not to others. List all power-of-two sizes for which it is aligned and carefully justify your answer.

The lower 3 bits are zero, but the 4th bit is 1 and so its aligned for 2, 4, and 8 byte access, but not for anything more than that.

2 **(4 marks)** **Pointer Arithmetic.** Without using the `[]` array syntax (i.e., using only pointer arithmetic) and without introducing **any** additional variables, finish implementing this function that copies the first `n` integers of array `from` into array `to`.

```
void copy (int* from, int* to, int n) {
    while (n--)
        *to++ = *from++;
}
```

3 **(4 marks)** **Dynamic Allocation.** A *dangling pointer* exits when a program retains a reference to dynamically allocated memory after it has been freed/reclaimed. A *memory leak* exists when a program fails to free/reclaim dynamically allocated memory that it is no longer needed by the program.

3a Carefully explain the most serious symptom of a dangling-pointer bug.

The memory pointed to by the dangling-pointer may be re-allocated for some other use and then the program might use the dangling-pointer to erroneously update this newly allocated thing, thinking its updated the old (but freed) thing.

3b Carefully explain the most serious symptom of a memory-leak bug.

The program could eventually exhaust memory and slow itself, and the entire system it runs on, horribly.

3c Does Java's garbage collector completely eliminate the dangling-pointer bug? Justify your answer carefully.

Yes. It will only free memory when it is unreachable via any pointer in the program.

3d Does Java's garbage collector completely eliminate the memory-leak bug? Justify your answer carefully.

No. Sometimes program's retain references to objects that they never intend to use again. The GC can't free these things since they are still reachable.

4 **(6 marks)** **Global Arrays.** In the context of the following C declarations:

```
int a[10];
int *b;
```

Indicate what, if anything, about the following expressions can not be computed statically. Be precise.

4a `a[3]`

The value of the variable.

4b `&a[3]`

Nothing.

4c `b[3]`

The address and value of the variable.

5 **(6 marks)** **Instance Variables.** In the context of the following C declarations:

```
struct S {
    int i,j;
};
```

```
struct S  a;
struct S* b;
```

Indicate what, if anything, about the following expressions can not be computed statically. Be precise.

5a `&a.i`

Nothing.

5b `&b->i`

The value of the expression; i.e., the address of the variable.

5c `(&b->j)-(&b->i)`

Nothing.

6 (3 marks) **Memory Endianness** Examine this C code.

```
char a[4];
*((int*) (&a[0])) = 1;
```

Carefully explain how this code can be used to determine the *endianness* of the machine on which it runs, recalling that on a *Big Endian* machine, the high-order (most-significant) byte of a number has the lowest address.

The assignment statement stores the integer `0x00000001` in the array `a`. By testing which entry of `a` stores 1 and which store 0, the program can determine whether the least significant byte (i.e., 1) has the lowest or highest address. Thus if `a[0]==1` this is a Little Endian machine and if `a[3]==1` it is Big Endian.

7 (8 marks) Consider the following C declarations.

```
struct S {
  int       i;
  int       j[10];
  struct S* k;
};
```

```
int       a[10];
int*      b;
int       c;
struct S  d;
struct S* e;
```

For each of the following questions indicate the total number of memory references (i.e., distinct loads and/or stores) required to execute the listed statement. Justify your answers carefully.

7a `a[3] = 0;`

One: store value in variable.

7b `a[c] = 0;`

Two: read value of c; store value in variable.

7c `b[c] = 0;`

Three: read value of b; read value of c; store value in variable

7d `d.i = 0;`

One: store value in variable.

7e `d.j[3] = 0;`

One: store value in variable.

7f `e->i = 0;`

Two: read value of e; store value in variable.

7g `d.k->i = 0;`

Two: read value of `d.k`; store value in variable.

7h `e->k->i = 0;`

Three: read value of `e`; read value of `k`; store value in variable.

8 **(10 marks)** **Reading Assembly Code.** Consider the following snippet of SM213 assembly code.

```
foo: ld $s,     r0          # r0 = &s
     ld 0(r0),  r1          # r1 = s.a
     ld 4(r0),  r2          # r2 = s.b
     ld 8(r0),  r3          # r3 = s.c
     ld $0,     r0          # r0 = 0
     not        r1          #
     inc        r1          # r1 = -a
L0:  bgt        r3, L1      # goto L1 if c>0
     br         L9          # goto L9 if c<=0
L1:  ld         (r2), r4    # r4 = *b
     add        r1, r4      # r4 = *b-a
     beq        r4, L2      # goto L2 if *b==a
     br         L3          # goto L3 if *b!=a
L2:  inc        r0          # r0 = r0 +1 if *b==a
L3:  dec        r3          # c--
     inca       r2          # a++
     br         L0          # goto L0
L9:  j          (r6)        # return
```

8a Carefully comment every line of code above.

8b Give precisely-equivalent C code.

```
struct S {
    int  a;
    int* b;
    int  c;
};
S s;
int foo () {
    int i=0;

    while (s.c>0) {
        if (s.a==*s.b)
            i++;
        s.c--;
        b++;
    }
    return i;
}
```

Or

```
int foo () {
    int i=0,j;

    for (j=0; j<s.c; j++)
        if (s.a==s.b[j])
            i++;
    return i;
}
```

8c The code implements a simple function. What is it? Give the simplest, plain English description you can.

It counts the number of elements in the integer array `s.b` whose size is `s.c` that have the value `s.a` and returns this number.

9 **(5 marks)** **Pointers in C** Consider the follow declarations in C.

```
int  a[10] = 0,2,4,6,8,10,12,14,16,18; // a[i] = 2*i;
int* b     = &a[4];
int* c     = a+4;
```

Answer the following questions. Show your work for the last question.

9a What is the *type* of the variable `a`?

```
int*
```

9b What is the value of `b[4]`?

```
16
```

9c What is the value of `c[4]`?

```
16
```

9d What is the value of `*(a+4)`?

```
8
```

9e What is the value of `b-a`?

```
4
```

9f What is the value of `*(&a[3] + *(a+(&a[3]-&a[2])))`?

```
= *((a+3) + *(a+(a+3)-(a+2)))
= *((a+3) + *(a+1))
= *((a+3) + a[1])
= *((a+3) + 2)
= *(a+5)
= a[5]
= 10
```

10 **(3 marks)** **Mystery Variable 1** This code stores 0 in a variable.

```
ld $0, r0
st r0, 8(r5)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Local or argument int.

11 **(3 marks)** **Mystery Variable 2** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Static array of ints.

12 **(3 marks)** **Mystery Variable 3** This code stores 0 in a variable.

```
ld $0, r0
ld $3, r1
ld $0x1000, r2
ld (r2), r2
st r0, (r2, r1, 4)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Dynamic array of ints.

13 **(3 marks)** **Mystery Variable 4** This code stores 0 in a variable.

```
ld $0, r0
ld $0x1000, r2
ld (r2), r2
st r0, 8(r2)
```

Carefully, precisely, and succinctly describe this variable (just the one in which the 0 is stored).

Entry of type int in dynamic, global struct.

14 **(9 marks)** **Dynamic Storage**

14a Carefully explain how a C program can create a *dangling pointer* and what bad thing might happen if it does.

If it retains a pointer to heap-allocated storage after it has been freed and then dereferences this pointer. The program could write to or read from a part of another, unrelated struct, array or variable that is stored in the freed, but pointed-to memory.

14b Carefully explain how a C program can create a *memory leak* and what bad thing might happen if it does.

If it fails to free heap-allocated storage after it is no longer needed by the program. The program's memory size could grow to the point where it no longer fits in available memory on the machine.

14c Can either or both of these two problems occur in a Java program? Briefly explain.

Dangling pointers can not exist, because memory is only freed by the garbage collector when there are not pointers referring to it. Memory leaks can occur when a program inadvertently retains references to objects that it no longer needs.

15 **(10 marks)** Implement the following in SM213 assembly. You can use a register for `c` instead of a local variable. **Comment every line.**

```
int  len;
int* a;
int countNotZero () {
    int c=0;
    while (len>0) {
        len=len-1;
        if (a[len]!=0)
            c=c+1;
    }
    return c;
}
```

```
untZero: ld  $len, r1          # r1 = &len
   ld  0(r1), r1         # r1 = len
   ld  $a, r2            # r2 = &a
         ld  0(r2), r2         # r2 = a
         ld  $0, r0            # r0 = c
op:      bgt r1, cont          # goto cont if len>0
         br  done              # goto done if len<=0
nt:      dec r1                # len = len - 1
   ld  (r2, r1, 4), r3   # r3 = a[len]
   beq r3, loop          # goto skip if a[len]==0
   inc r0                # c=c+1 if a[len]!=0
   br  loop              # goto loop
ne:      j   (r6)              # return c
```