5. C Pointers [8 marks]

Give the values of the listed variables following the execution of this block of code.

```
int i = 1;
int j = 10;
int* p = &i;
int* q = &j;
*p = *p + *q;
q = p;
*p = *p + *q;
*q = *q + *p;
p = (int*) 0x1000;
int k = &p[3] - &p[1];
q = p + 2;
```

What is the value of `i` ?

What is the value of `j` ?

What is the value of `k` ?

What is the value of `q` ?

5 **[10 marks]** **C Pointers.** Consider the following global variable declarations.

```
int a[4] = (int[4]){5, 4, 3, 2};
int b[3] = (int[3]){6, 7, 8};
int* c;
```

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 additional code.

```
c = &a[3];
b[*c] = a[1];
c = (b+1);
*c = *c + 1;
*c = *c + 1;
a[3] = c[1];
```

Indicate the value of each of the following expressions (i.e., the values of `a`, `b`, and `c`) below by checking *Unchanged* if the execution does not change the value or by entering its new value in the space provided. Assuming that `int`'s are 4-bytes long. Answer these questions about the value of these variables following the execution of this code.

```
a[0]:  Unchanged ○   or  Changed to Value ____________

a[1]:  Unchanged ○   or  Changed to Value ____________

a[2]:  Unchanged ○   or  Changed to Value ____________

a[3]:  Unchanged ○   or  Changed to Value ____________

b[0]:  Unchanged ○   or  Changed to Value ____________

b[1]:  Unchanged ○   or  Changed to Value ____________

b[2]:  Unchanged ○   or  Changed to Value ____________

c:     Unchanged ○   or  Changed to Value ____________
```

6 **(9 marks)** **C Pointers and Functions** Consider the following C code.

```
void foo(int x, int *p) {
A.     printf("x = %d, *p = %d", x, *p);
       *p = x + 12;
       --x;
       p = 0;
}
int main (void) {
       int   a = 1;
       int   b = 2;
       int  *p = &a;
       int **q = &p;
B.     printf("**q = %d", **q);
       foo(*p, *q);
C.     printf("a = %d", a);
       if (!p)
D.           printf("p is 0");
       else
E.           printf("*p is %d", *p);
       *q = &b;
F.     printf("*p = %d", *p);
       return 0;
}
```

In the above code, what if anything is printed by the `printf` instruction at locations A to F when the program is executed?

6a A

6b B

6c C

6d D

6e E

6f F

4 **(6 marks)** **Static Control Flow.** Answer these questions using the register `r0` for `x` and `r1` for `y`.

4a Write commented assembly code equivalent to the following.

```
if (x <= 0)
    x = x + 1;
else
    x = x - 1;
```

4b Write commented assembly code equivalent to the following.

```
for (x=0; x<y; x++)
   y--;
```

5 **(6 marks)** **C Pointers.** Consider the following C procedure `copy()` and global variable `a`.

```
void copy (char* s, char* d, int n) {
    for (int i=0; i<n; i++)
        d[i] = s[i];
}

char a[9] = {1,2,3,4,5,6,7,8,9};
```

And this procedure call:

```
copy (a, a+3, 6);
```

List the value of the elements of the array `a` (in order), following the execution of this procedure call.

2 **(7 marks)** **C Pointers.** Consider the following C code.

```
int  a[10] = {0,1,2,3,4,5,6,7,8,9};  // i.e., a[i] = i
int* b     = a+4;

int foo (int* x, int* y, int* z) {
    *x = *x + *y;
    *x = *x + *z;

    return *x;
}

int bar () {
    return foo (b - 2, a + (b - a) + (&a[7] - &a[6]), a + 2);
}
```

What value does `bar()` return? Justify your answer (1) by simplifying the description of the arguments to `foo()` as much as possible so that the relationship among them, if any, is clear and (2) by carefully explaining what happens when `foo()` executes.

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

Date: October 2014; Instructor: Mike Feeley

This was a closed book exam. No notes. No electronic calculators. This sample combines questions from multiple exams and so a real exam will have fewer total questions.

Answer in the space provided. Show your work; use the backs of pages if needed. There are **15** questions on **8** pages, totaling **79** marks. You have **50 minutes** to complete the exam.

STUDENT NUMBER: ____________________

NAME: ____

SIGNATURE: ____

Q1	/ 2
Q2	/ 4
Q3	/ 4
Q4	/ 6
Q5	/ 6
Q6	/ 3
Q7	/ 8
Q8	/ 10
Q9	/ 5
Q10	/ 3
Q11	/ 3
Q12	/ 3
Q13	/ 3
Q14	/ 9
Q15	/ 10
Total	/ 79

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.

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) {
```

```
foo: ld $s,     r0          #
     ld 0(r0),  r1          #
     ld 4(r0),  r2          #
     ld 8(r0),  r3          #
     ld $0,     r0          #
     not        r1          #
     inc        r1          #
L0:  bgt        r3, L1      #
     br         L9          #
L1:  ld         (r2), r4    #
     add        r1, r4      #
     beq        r4, L2      #
     br         L3          #
L2:  inc        r0          #
L3:  dec        r3          #
     inca       r2          #
     br         L0          #
L9:  j          (r6)        #
```

8a Carefully comment every line of code above.

8b Give precisely-equivalent C code.

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

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`?

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

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

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

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

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

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).

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).

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).

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

2 **(4 marks)** **Pointers in C**. Consider the following declaration of C global variables.

```
int  a[10] = {0,1,2,3,4,5,6,7,8,9};  // i.e., a[i] = i
int* b     = &a[6];
```

And the following expression that accesses them found in some procedure.

```
*(a + ((&a[9] + 5) - b))
```

When this expression is evaluated at runtime does it cause an error? If not, what value does it compute?

Briefly explain your answer as follows: if there is a runtime error, clearly explain what causes it; if there is not an error, show at least 3 lines of work with intermediate values to explain your answer, step by step.

3 **(4 marks)** **Global Arrays**. Consider the following C global variable declarations.

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

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. Assume the value `0` is in `r0` and the value of `i` is in `r1`. Use labels `a` and `b` for variable addresses.

3a `a[i] = 0;`

3b `b[i] = 0;`

1 **(2 marks)** **Memory Alignment.** Consider the memory address 0x92. List all power-of-two sizes for which aligned memory access is possible and carefully justify your answer.

2 **(8 marks)** **Pointer Arithmetic.** Consider the following lines of C code. For the assignments to `i` , `j`, `k`, and `m` say (a) whether the code generates an error and why or (b) what value the variables have after the code executes. If one line generates an error but a later one does not, give the value of the later ones. Show your work.

```
int a[10] = { 9,8,7,6,5,4,3,2,1,0 };
int i = *(a+4);
int j = &a[3] - &a[1];
int k = *(a+*(a+6));
int m = *(&a[5]-a);
```

2a `i`:

2b `j`:

2c `k`:

2d `m`:

5. C Pointers [8 marks]

Give the values of the listed variables following the execution of this block of code.

```
int i = 1;
int j = 10;
int* p = &i;
int* q = &j;
*p = *p + *q;
q = p;
*p = *p + *q;
*q = *q + *p;
p = (int*) 0x1000;
int k = &p[3] - &p[1];
q = p + 2;
```

What is the value of `i` ?

44

What is the value of `j` ?

10

What is the value of `k` ?

2

What is the value of `q` ?

0x1008

```
a[0]:  Unchanged ○   or  Changed to Value ______________

a[1]:  Unchanged ○   or  Changed to Value ______________

a[2]:  Unchanged ○   or  Changed to Value ______________

a[3]:  Unchanged ○   or  Changed to Value ______________

b[0]:  Unchanged ○   or  Changed to Value ______________

b[1]:  Unchanged ○   or  Changed to Value ______________

b[2]:  Unchanged ○   or  Changed to Value ______________

c:     Unchanged ○   or  Changed to Value ______________
```

```
Unchanged
Unchanged
Unchanged
4
Unchanged
9
4
0x2004
```

6 **[12 marks]** **Dynamic Allocation.** Consider each of the following pieces of C code to determine whether it contains (or may contain) a memory-related problem. Label each of the following code snippets as one of the following:

A: There is no memory leak or dangling pointer; nothing needs to be changed with malloc or free.
B: There is no memory leak or dangling pointer, but the code would be improved by moving malloc or free.
C: There is a possible memory leak that is best resolved by adding, removing or moving malloc or free.
D: There is a possible memory leak that is best resolved by adding reference counting.
E: There is a possible dangling pointer that is best resolved by adding, removing or moving malloc or free.
F: There is a possible dangling pointer that is best resolved by adding reference counting.

You can assume that the starting point for each snippet of code is a call to `foo`, and that `copy` is in a different module. Do not fix any bugs; for each part, fill in a single multiple choice bubble based off of the options above. Most of the code snippets are very similar. Changes from previous versions and/or key things to look for in **bold** font.

6a

```
int x;
int (*proc) (int);
```

Give the SM213 assembly for the code below (just for this single statement, not for the procedure itself). Assume that the return value is in `r0`. Comments are not required.

```
x = proc (1);
```

```
ld   $1, r0
deca r5
st   r0, (r5)
ld   $proc, r0
gpc  $2, r6
j    *(r0)
inca r5
ld   $x, r1
st   r0, (r1)
```

Rubric: 1 for arg, 2 for call (1 for indirect), 1 for store result. One minor error to gpc value or offsets okay; two is -1.

6 **(9 marks)** **C Pointers and Functions** Consider the following C code.

```
void foo(int x, int *p) {
A.     printf("x = %d, *p = %dn", x, *p);
       *p = x + 12;
       --x;
       p = 0;
}
int main (void) {
       int    a = 1;
       int    b = 2;
       int   *p = &a;
       int **q = &p;
B.     printf("**q = %dn", **q);
       foo(*p, *q);
C.     printf("a = %dn", a);
       if (!p)
D.             printf("p is 0n");
       else
E.             printf("*p is %dn", *p);
       *q = &b;
F.     printf("*p = %dn", *p);
       return 0;
}
```

In the above code, what if anything is printed by the `printf` instruction at locations A to F when the program is executed?

6a A

(2) x = 1, *p = 1

6b B

(1) **q = 1

6c C

(2) a = 13

6d D

(1) nothing

6e E

(1) *p is 13

6f F

(2) *p = 2

7 **(10 marks)** **Allocation in C.** You are giving expert advice to a development team about what type of variable allocation strategy is ideal for various aspects of their system. Indicate which of the following five strategies **best suits** each of the scenarios listed below (a strategy can apply to more than one scenario and some strategies may not be best for any of them). Name the strategy using its letter (i.e., `A-E`) and justify your answer briefly.

Strategies

A. Global variable storing the entire object in question.
B. Local variable storing the entire object in question.
C. Calling `malloc` and `free` in the same procedure.
D. Calling `malloc` and `free` in different procedures.
E. Calling `malloc` and using reference counting instead of calling `free`.

Scenarios

7a An object that is allocated when a procedure is called and deallocated when it returns, where the primary concern is to prevent memory leaks and to simplify the code for allocation and deallocation.

B, object only used in one procedure.

7b An object that is allocated when a procedure is called and deallocated when it returns, where the primary concern is to prevent stack-smash, buffer-overflow attacks.

C, object only used in one procedure, but prevent stack smashing by allocating from heap.

7c A dynamically-allocated object whose lifetime can only be determined by understanding the implementation of multiple procedures in different modules of the program.

E, use reference counting to avoid adding coupling between modules.

7d An object whose size is independent of program inputs and that exists for the entire execution of the program, where the primary concern is to minimize runtime costs (i.e., CPU time) for allocating and accessing the object.

A, use static allocation when possible.

7e An object whose size depends on program inputs and that exists for the entire execution of the program, where the primary concern is to efficiently handle a very wide range of input values.

C or D, dynamic allocation is required, but deallocation is not really necessary to avoid a memory leak since object's lifetime is entire program.

8 **(9 marks)** **C Memory Errors.** There are four memory related errors in the program below. For each error, give the line number on which the error occurred, the type of error, and indicate how to fix it.

And this procedure call:

```
    copy (a, a+3, 6);
```

List the value of the elements of the array a (in order), following the execution of this procedure call.

```
{1,2,3,1,2,3,1,2,3}
```

6 **(6 marks)** **Dynamic Allocation.** The following four pieces of code are identical except for the their use of `free()`. Each of them may be correct or they may have a memory leak, dangling pointer or both. In each case, determine whether these bugs exists and if so, briefly describe the bug(s); do not describe how to fix the bug.

6a

```
int* copy (int* src) {                int foo() {
  int* dst = malloc (sizeof (int));     int  a = 3;
  *dst = *src;                          int* b = copy (&a);
  return dst;                           return *b;
}                                     }
```

Memory leak, because object allocated in `copy` is not freed in the shown code and when `foo` returns it is unreachable.

6b

```
int* copy (int* src) {                int foo() {
  int* dst = malloc (sizeof (int));     int  a = 3;
  *dst = *src;                          int* b = copy (&a);
  free (dst);                           return *b;
  return dst;                         }
}
```

Dangling pointer. After `free` in `copy`, `dst` is a dangling pointer. This value is returned by copy and so `b` in `foo` is also a dangling pointer. The last statement of `foo`, `return *b` dereferences this dangling pointer.

6c

```
int* copy (int* src) {                int foo() {
  int* dst = malloc (sizeof (int));     int  a = 3;
  *dst = *src;                          int* b = copy (&a);
  return dst;                           free (b);
}                                       return *b;
                                      }
```

Dangling pointer. After `free` in `foo`, `b` becomes and dangling pointer and it is then dereferenced in the last statement.

6d

```
int* copy (int* src) {                int foo() {
  int* dst = malloc (sizeof (int));     int  a = 3;
  *dst = *src;                          int* b = copy (&a);
  free (dst);                           free (b);
  return dst;                           return *b;
}                                     }
```

Dangling pointer. After `free` in `copy`, `dst` becomes a dangling pointer. This value is returned by `copy` and so `b` in `foo` is also a dangling pointer. The third statement of `foo` then calls `free` again on this value, attempting to free an object that has already been freed, which results in an error. If the program where to proceed it would then dereference the dandling pointer in the `return` statement.

7 **(8 marks)** **Reference Counting.** The following extends the code from the previous question by adding a procedure `saveIfMax` that is implemented in a separate module. Add calls to `inc_ref` and `dec_ref` to use referencing counting to eliminate all dangling pointers and memory leaks in this code while creating no *coupling* between `saveIfMax` and the rest of the code (i.e., `saveIfMax` can not know about what the rest of the code does and neither can the rest of the code know what `saveIfMax` does). Do not implement reference counting nor worry about storing the reference count itself; just add calls to `inc_ref` and `dec_ref` in the right places, **which may require slightly rewriting portions of the code**.

CPSC 213, Winter 2015, Term 2 — MIDTERM SAMPLE QUESTIONS
Solution

Date: Feb 18, 2016; Instructor: Mike Feeley

1 **(7 marks)** **Variables and Memory.** Consider the following C code with three global variables, `a`, `b`, and `c`, that are stored at addresses `0x1000`, `0x2000`, `0x3000`, respectively, and a procedure `foo()` that accesses them.

```
int a[1];    // at address 0x1000
int b[1];    // at address 0x2000
int* c;      // at address 0x3000

void foo() {
    a[0] = 1;
    b[0] = 2;
    c = a;
    c[0] = 3;
    c = b;
    *c = 4;
}
```

Describe what you know about the content of memory following the execution of `foo()` on a 32-bit **Little Endian** processor. List only memory locations whose address and value you know. **List each byte of memory separately** using the form "`byte_address: byte_value`". List all numbers in hex.

```
0x1000: 0x03
0x1001: 0x00
0x1002: 0x00
0x1003: 0x00
0x2000: 0x04
0x2001: 0x00
0x2002: 0x00
0x2003: 0x00
0x3000: 0x00
0x3001: 0x20
0x3002: 0x00
0x3003: 0x00
```

2 **(7 marks)** **C Pointers.** Consider the following C code.

```
int  a[10] = {0,1,2,3,4,5,6,7,8,9};  // i.e., a[i] = i
int* b     = a+4;

int foo (int* x, int* y, int* z) {
    *x = *x + *y;
    *x = *x + *z;

    return *x;
}

int bar () {
    return foo (b - 2, a + (b - a) + (&a[7] - &a[6]), a + 2);
}
```

What value does `bar()` return? Justify your answer (1) by simplifying the description of the arguments to `foo()` as much as possible so that the relationship among them, if any, is clear and (2) by carefully explaining what happens when `foo()` executes.

```
b - 2                           = a + 4 - 2
                                = a + 2
a + (b - a) + (&a[7] - &a[6])   = a + ((a+4) - a) + ((a+7) - (a+6))
                                = a + 4 + 1
                                = a + 5
```
So the call to `foo` simplifies to `foo(a+2, a+5, a+2)`. Thus when `foo()` runs we have:
```
*(a+2) = *(a+2) + *(a+5);
       = 2 + 5
       = 7
*(a+2) = *(a+2) + *(a+2)
       = 7 + 7
       = 14
```
Thus `foo()` returns 14.

3 **(6 marks)** **Global Arrays**. Consider the following C global variable declarations.

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

Give the SM213 assembly code the compiler might generate for the following statements that access these variables. You may use labels `a`, `b`, and `c` for addresses. You may not assume anything about the value of registers. **Comment every line.**

3a `b = a;`

```
ld $a, r0    # r0 = &a
ld $b, r1    # r2 = &b
st r0, (r1)  # b = a
```

3b `a[i] = i;`

```
ld $i, r0              # r0 = &i
ld (r0), r1            # r1 = i
ld $a, r2              # r2 = &a = &a[0]
st r1, (r2, r1, 4)     # a[i] = i
```

4 **(3 marks)** **Instance Variables**. Consider the following C global variable declarations.

```
struct S {
    int   a;
    void* b;
    int   c;
};

struct S* s;
```

Give the SM213 assembly code the compiler might generate for the statement:

```
s->b = &s->c;
```

You may use the label `s`. You may not assume anything about the value of registers. **Comment every line.**

```
ld $s, r0      # r0 = &s
ld (r0), r1    # r1 = s = &s->a
ld $8, r2      # r2 = 8
add r1, r2     # r2 = &s1->c
st r2, 4(r1)   # s1->b = &s1->c
```

5 **(6 marks)** **Count Memory References.** Consider the following C global variable declarations.

```
struct S {
    int a[10];
};

struct S s;
```

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  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
```

CPSC 213, Winter 2014, Term 1 — Sample Midterm (Winter 2013 Term 2)
Solution

Date: October 2014; Instructor: Mike Feeley

1 **(8 marks)** **Memory and Numbers**. Consider the following C code with global variables a and b.

```
int  a[2];
int* b;

void foo() {
    b    = a;
    a[0] = 1;
    b[1] = 2;
}

void checkGlobalVariableAddressesAndSizes() {
    if ((&a==0x2000) && (&b==0x3000) && sizeof(int)==4  && sizeof(int*)==4)
        printf ("OKAY");
}
```

When `checkGlobalVariableAddressesAndSizes()` executes it prints "OKAY". Recall that `sizeof(t)` returns the number of bytes in variables of type `t`.

Describe what you know about the content of memory following the execution of `foo()` on a **Little Endian** processor. List only memory locations whose address and value you know. List each byte of memory on a separate line using the form: "`byte_address: byte_value`". List all numbers in hex.

```
0x2000: 0x01
0x2001: 0x00
0x2002: 0x00
0x2003: 0x00
0x2004: 0x02
0x2005: 0x00
0x2006: 0x00
0x2007: 0x00
0x3000: 0x00
0x3001: 0x20
0x3002: 0x00
0x3003: 0x00
```

2 **(4 marks)** **Pointers in C**. Consider the following declaration of C global variables.

```
int  a[10] = {0,1,2,3,4,5,6,7,8,9};  // i.e., a[i] = i
int* b     = &a[6];
```

And the following expression that accesses them found in some procedure.

```
*(a + ((&a[9] + 5) - b))
```

When this expression is evaluated at runtime does it cause an error? If not, what value does it compute?

Briefly explain your answer as follows: if there is a runtime error, clearly explain what causes it; if there is not an error, show at least 3 lines of work with intermediate values to explain your answer, step by step.

It executes without error and computes the value 8.

```
*(a + ((&a[9] + 5) - b)) == *(a + (&a[14] - &a[6]))
                         == *(a + 8)
                         == a[8]
                         == 8
```

3 **(4 marks)** **Global Arrays**. Consider the following C global variable declarations.

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

CPSC 213, Winter 2010, Term 1 — Midterm Exam **Solution**

Date: October 27, 2010; Instructor: Tamara Munzner

1 **(2 marks)** **Memory Alignment.** Consider the memory address 0x92. List all power-of-two sizes for which aligned memory access is possible and carefully justify your answer.

2. The lowest bit is zero, but the 2nd lowest bit is 1 so it is aligned only for 2-byte access. Alternately: the decimal equivalent 146 divides evenly by 2, but not by 4 or any larger power of 2.

2 marks: 1 for correct answer, 1 for correct justification. Thus mark of 1/2 if forgot to convert from hex and did computation for decimal 92 getting answer 2 and 4, or had correct logic but computational mistake.

2 **(8 marks)** **Pointer Arithmetic.** Consider the following lines of C code. For the assignments to `i` , `j`, `k`, and `m` say (a) whether the code generates an error and why or (b) what value the variables have after the code executes. If one line generates an error but a later one does not, give the value of the later ones. Show your work.

```
int a[10] = { 9,8,7,6,5,4,3,2,1,0 };
int i = *(a+4);
int j = &a[3] - &a[1];
int k = *(a+*(a+6));
int m = *(&a[5]-a);
```

2a `i`:

No error. Value of i is 5.

```
int i = *(a+4);
int i = *(&a[4]);
int i = a[4];
int i = 5;
```

2b `j`:

No error. Value of j is 2.

```
int j = &a[3] - &a[1];
int j = 2;
```

2c `k`:

No error. Value of i is 5.

```
int k = *(a+*(a+6);
int k = *(a+*(&a[6]));
int k = *(a+ 3);
int k = *(&a[3]);
int k = 6;
```

2d `m`:

This attempt to de-reference the address `((&a[5]-a) == (&a[5]-&a[0]) == 5`, statement will generate an error (from the compiler or at runtime). The address 5 is not unaligned. It is also a protected location on most architectures.

3 **(5 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. Answer true or false to the following questions:

- Dangling pointers can occur in C. True
- Dangling pointers can occur in Java. False
- Memory leaks can occur in C. True
- Memory leaks can occur in Java. True