# 6. Dynamic Allocation [10 marks]

Consider each of the following code blocks where `add`, `doAdd`, and `doSomethingElse` are in three different modules of a large program. For each block indicate whether the code has a memory leak or dangling pointer and which technique (if any) is ***the single best way (only one)*** to fix the bug or improve the code (even if it doesn't have a bug). Select *none* if the code does not need to be improved or if the needed change isn't listed.

a)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
}
```

Error? ◯ Dangling Pointer ◯ Memory Leak ◯ Both ◯ Neither or can not determine

Change? ◯ Move malloc or free ◯ Add malloc or free ◯ Add reference counting ◯ None of these

b)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
  free(c);
}
```

Error? ◯ Dangling Pointer ◯ Memory Leak ◯ Both ◯ Neither or can not determine

Change? ◯ Move malloc or free ◯ Add malloc or free ◯ Add reference counting ◯ None of these

c)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  doSomethingElse(c);
  free(a);
  free(b);
  free(c);
}
```

Error? ◯ Dangling Pointer ◯ Memory Leak ◯ Both ◯ Neither or can not determine

Change? ◯ Move malloc or free ◯ Add malloc or free ◯ Add reference counting ◯ None of these

d)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  doSomethingElse(*c);
  free(a);
  free(b);
  free(c);
}
```

Error? ◯ Dangling Pointer ◯ Memory Leak ◯ Both ◯ Neither or can not determine

Change? ◯ Move malloc or free ◯ Add malloc or free ◯ Add reference counting ◯ None of these

e)

```
int* add(int *a, int *b, n) {
  int c[n];
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
}
```

Error? ◯ Dangling Pointer ◯ Memory Leak ◯ Both ◯ Neither or can not determine

Change? ◯ Move malloc or free ◯ Add malloc or free ◯ Add reference counting ◯ None of these

**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* copy(int s) {
    int* d = malloc(sizeof(int));
    *d = s;
    return d;
}
```

```
void foo (int s) {
    int* d = copy(s);
    printf("value is %d", *d);
    free(d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

**6b**

```
int* copy(int s) {
    int* d = malloc(sizeof(int));
    *d = s;
    free(d);
    return d;
}
```

```
void foo (int s) {
    int* d = copy(s);
    printf("value is %d", *d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

**6c**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int d = 0;
    copy(s, &d);
    printf("value is %d", d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

**6d**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int* d = malloc (sizeof(int));
    copy(s, d);
    free(d);
    printf("value is %d", *d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

**6e**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int* d = malloc (sizeof(int));
    copy(s, d);
    printf("value is %d", *d);
    process(d);
    free(d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

**6f**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int* d = malloc (sizeof(int));
    copy(s, d);
    printf("value is %d", *d);
    process(*d);
    free(d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

**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.

**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.

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

**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.

**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.

**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.

```
1  struct my_struct {
2      int nNums;
3      int *nums;
4  };
   typedef struct my_struct *my_struct_t;

5  void foo(my_struct_t ptr) {
6      int i;
7      for (i = 0; i <= ptr->nNums; i++) {
8          printf("%d", ptr->nums[i]);
9      }
10     free(ptr);
11 }

12 int main(void)
13 {
14     my_struct_t s = malloc(sizeof(s));
15     s->nNums = 5;
16     s->nums = malloc(5 * sizeof(*s->nums));
17     for (int i = 0; i < s->nNums; i++) {
18         s->nums[i] = i + 4;
19     }
20     foo(s);
21     free(s);
22     return 0;
23  }
```

**8a** Error 1:

line:

type:

fix:

**8b** Error 2:

line:

type:

fix:

**8c** Error 3:

line:

type:

fix:

**8d** Error 4:

line:

type:

fix:

**9 (6 marks)** **Reference Counting.** The following code has a *memory leak* bug. Add calls to `inc(o)` and `dec(o)` to increment and decrement the reference count of object `o` and make any other *small* changes necessary so that the program is free of memory leaks and dangling pointers. Assume that `rc_malloc` calls malloc and initializes the object's reference count to zero.

```
void foo (int i) {
  int* ip = rc_malloc (sizeof (int));

  *ip     = i;
  bar (ip);
  zot (*ip);

}
```

```
int  c = 0;
int* v[2];

void bar (int* ap) {
  if (v[c] == NULL || *ap < *v[c]) {

    v[c] = ap;

  }
  c = (c + 1) % 2;
}

void zot (int a) {
  if (v[c] != NULL && a < *v[c])
    *v[c] = a;
  c = (c + 1) % 2;
}
```

**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* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

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

**6b**

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

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

**6c**

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

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

**6d**

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  *dst = *src;
  free (dst);
  return dst;
}
```

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

**8 (3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

There is a dangerous bug in this code. Carefully describe what it is. Assume that `b` was assigned a value somewhere else in the program.

**9 (3 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}

void three () {
    int* ret = one();
    two();
}
```

There is a dangerous bug in this code. Carefully describe what it is.

Hint: what is the value of `*ret` just before and just after `two()` is called? Look carefully at the implementation of `one()`, what it returns, and when variables are allocated and deallocated.

**10 (4 marks)** **Branch and Jump Instructions**.

**10a** What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

**10b** What is the value of the program-counter register (i.e., the PC) following the execution of this unconditional branch instruction at address `0x500`. Justify your answer.

`0x500: 8005`

```
}
```

**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.

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

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

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

**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]`

```
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).

## 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.

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

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

**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;
}
```

**6** **(4 marks)** **Branch and Jump Instructions**.

**6a** What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

**6b** What is the value of the program-counter register (i.e., the PC) following the execution of this unconditional branch instruction at address `0x500`. Justify your answer.

```
0x500: 8005
```

**7** **(4 marks)** **Loops**. Consider the consequences of eliminating the two conditional branch instructions from SM213 (and not adding any other instructions). Would compilers still be able to compile C programs to run on this modified machine? If so, explain how. If not, carefully explain what feature or features of C would be impossible to execute on the modified machine.

**8** **(4 marks)** **Dynamic Allocation**. The following code contains a procedure that creates a copy of a null-terminated string (the standard representation for strings in C). It contains a serious bug related to dynamic memory allocation.

```
char* copy (char* s) {
  int   i, len = 0;
  char* cpy;

  while (s [len] != 0)
    len++;
  cpy = (char*) malloc (len+1);
  for (i=0, i<len; i++)
    cpy [i] = s [i];
  cpy [len] = 0;
  return cpy;
}
```

```
void doSomething () {
  char* x;
  x = copy ("Hello World");
  printf ("%s", x);
}
```

Explain in plan English what the bug is and how you would fix it (without changing the semantics of `copy`).

**5b** Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

**6 (3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

Is there a bug in this code? If so, carefully describe what it is.

**7 (6 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}

void three () {
    int* ret = one();
    two();
}
```

**7a** Is there a bug in this code? If so, carefully describe what it is.

**7b** What is the value of "`*ret`" at the end of `three`? Explain carefully.

**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.
- Dangling pointers can occur in Java.
- Memory leaks can occur in C.
- Memory leaks can occur in Java.
- Garbage collection is a partial solution to the dangling pointer problem.
- Garbage collection is a partial solution to the memory leak problem.
- Garbage collection is a full solution to the dangling pointer problem.
- Garbage collection is a full solution to the memory leak problem.
- The stack is a partial solution to the dangling pointer problem.
- Having memory allocation and deallocation in separate functions is a partial solution to the dangling pointer problem.

## 6. Dynamic Allocation [10 marks]

Consider each of the following code blocks where `add`, `doAdd`, and `doSomethingElse` are in three different modules of a large program. For each block indicate whether the code has a memory leak or dangling pointer and which technique (if any) is ***the single best way (only one)*** to fix the bug or improve the code (even if it doesn't have a bug). Select *none* if the code does not need to be improved or if the needed change isn't listed.

a)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
}
```

Error? ○ Dangling Pointer ● Memory Leak ○ Both ○ Neither or can not determine

Change? ○ Move malloc or free ● Add malloc or free ○ Add reference counting ○ None of these

b)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
  free(c);
}
```

Error? ○ Dangling Pointer ○ Memory Leak ○ Both ● Neither or can not determine

Change? ● Move malloc or free ○ Add malloc or free ○ Add reference counting ○ None of these

c)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  doSomethingElse(c);
  free(a);
  free(b);
  free(c);
}
```

Error? ○ Dangling Pointer ○ Memory Leak ○ Both ● Neither or can not determine

Change? ○ Move malloc or free ○ Add malloc or free ● Add reference counting ○ None of these

d)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  doSomethingElse(*c);
  free(a);
  free(b);
  free(c);
}
```

Error? ○ Dangling Pointer ○ Memory Leak ○ Both ● Neither or can not determine

Change? ● Move malloc or free ○ Add malloc or free ○ Add reference counting ○ None of these

*"None of these" also acceptable.*

e)

```
int* add(int *a, int *b, n) {
  int c[n];
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
}
```

Error? ● Dangling Pointer ○ Memory Leak ○ Both ○ Neither or can not determine

Change? ○ Move malloc or free ○ Add malloc or free ○ Add reference counting ● None of these

```
int* copy(int s) {
    int* d = malloc(sizeof(int));
    *d = s;
    return d;
}
```

```
void foo (int s) {
    int* d = copy(s);
    printf("value is %d", *d);
    free(d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

B. There is no memory leak or dangling pointer, but the code would be improved by moving malloc or free

**6b**

```
int* copy(int s) {
    int* d = malloc(sizeof(int));
    *d = s;
    free(d);
    return d;
}
```

```
void foo (int s) {
    int* d = copy(s);
    printf("value is %d", *d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

E: There is a possible dangling pointer that is best resolved by adding, removing or moving malloc and/or free.

**6c**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int d = 0;
    copy(s, &d);
    printf("value is %d", d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

A: There is no memory leak or dangling pointer; nothing needs to be changed with malloc or free.

**6d**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int* d = malloc (sizeof(int));
    copy(s, d);
    free(d);
    printf("value is %d", *d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

E: There is a possible dangling pointer that is best resolved by adding, removing or moving malloc and/or free.

**6e**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int* d = malloc (sizeof(int));
    copy(s, d);
    printf("value is %d", *d);
    process(d);
    free(d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

F: There is a possible dangling pointer that is best resolved by adding reference counting.

**6f**

```
void copy(int s, int* d) {
    *d = s;
}
```

```
void foo (int s) {
    int* d = malloc (sizeof(int));
    copy(s, d);
    printf("value is %d", *d);
    process(*d);
    free(d);
}
```

A ○ B ○ C ○ D ○ E ○ F ○

A: There is no memory leak or dangling pointer; nothing needs to be changed with malloc or free.

**7** **[8 marks]** **Reference Counting.** Consider the following code that is implemented in three independent modules (and a main module) that share dynamically allocated objects that should be managed using reference counting. The call to `rc_malloc` has 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?

20 10

**7b** Add calls to `rc_keep_ref` and `rc_free_ref` to correct implement reference counting for this program (all modules).

**7c** Assuming this program implements reference counting correctly, give the reference counts (number of pointers currently pointing to) the following two objects when `printf` is called from main?

`*p:` 2

`*q:` 4

**7d** Add code at point TODO so that the program is free of memory leaks and dangling pointers. Your code may call any of the procedures shown here as well as `rc_free_ref`. It may **not** directly access the global variable `b_values` (note that this variable is not listed in the "header file contents" section of the code and so it would not be in scope in main if the modules were implemented is separate files).

(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.

```
1  struct my_struct {
2      int nNums;
3      int *nums;
4  };
   typedef struct my_struct *my_struct_t;

5  void foo(my_struct_t ptr) {
6      int i;
7      for (i = 0; i <= ptr->nNums; i++) {
8          printf("%dn", ptr->nums[i]);
9      }
10     free(ptr);
11 }

12 int main(void)
13 {
14     my_struct_t s = malloc(sizeof(s));
15     s->nNums = 5;
16     s->nums = malloc(5 * sizeof(*s->nums));
17     for (int i = 0; i < s->nNums; i++) {
18         s->nums[i] = i + 4;
19     }
20     foo(s);
21     free(s);
22     return 0;
23  }
```

**8a** Error 1:
line:
type:
fix:

**8b** Error 2:
line:
type:
fix:

**8c** Error 3:
line:
type:
fix:

**8d** Error 4:
line:
type:
fix:

1. line 7, out of bounds error, change ”¡=” to ”¡”
2. line 10, memory leak, need to also free ”ptr-¿nums”
3. line 16, incorrect size for malloc, sizeof(int)
4. line 21, dangling pointer (or double free), remove instruction

**9 (6 marks)** **Reference Counting.** The following code has a *memory leak* bug. Add calls to `inc(o)` and `dec(o)` to increment and decrement the reference count of object `o` and make any other *small* changes necessary so that the program is free of memory leaks and dangling pointers. Assume that `rc_malloc` calls malloc and initializes the object's reference count to zero.

```
void foo (int i) {
  int* ip = rc_malloc (sizeof (int));
  inc(ip);
  *ip     = i;
  bar (ip);
  zot (*ip);
  dec(ip);
}
```

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* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

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

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

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* dst = malloc (sizeof (int));
  *dst = *src;
  return dst;
}
```

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

```
int foo() {
  int  a = 3;
  int* b = copy (&a);
  free (b);
  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**.

```
int x;

void doit () {
    x = addOne (5);
}

int addOne (int a) {
    return a + 1;
}
```

```
doit:
    deca r5        # allocate space for ra on stack
    st r6, (r5)    # save ra on stack
    deca r5        # make room for argument on stack
    ld $5, r0      # r0 = 5
    st r0, (r5)    # arg0 = 5
    gpc $6, r6     # get return address
    j add          # call addOne (5)
    inca r5        # remove argument area
    ld $x, r1      # r1 = &x
    st r0, (r1)    # x = addOne (5)
    ld (r5), r6    # restore ra from stack
    inca r5        # remove ra space from stack
    j (r6)         # return
addOne:
    ld (r5), r0    # r0 = a
    inc r0         # r0 = a + b
    j (r6)         # return a + b
```

**8** **(3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

There is a dangerous bug in this code. Carefully describe what it is. Assume that `b` was assigned a value somewhere else in the program.

There's a potential array overflow. Need to check that `i` is in range (0 .. size of `b` - 1) before writing to `b[i]` and thus this size, which is dynamically determined, should be a parameter to `set` or a global variable.

**9** **(3 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}
```

```
void three () {
    int* ret = one();
    two();
}
```

There is a dangerous bug in this code. Carefully describe what it is.

Hint: what is the value of `*ret` just before and just after `two()` is called? Look carefully at the implementation of `one()`, what it returns, and when variables are allocated and deallocated.

# 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:

**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.**

Address: `s0.s + 4 (offset to a)`

4. Variable: `s0.s->a[s0.s->b[2]]`

   Address: `s0.s->a + s0.s->b[2] * 4 (size of int)`

**6** **(4 marks)** **Branch and Jump Instructions**.

**6a** What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

smaller instructions

**6b** What is the value of the program-counter register (i.e., the PC) following the execution of this unconditional branch instruction at address `0x500`. Justify your answer.

```
0x500: 8005
```

`0x502 + 5 * 2 == 0x50c`

**7** **(4 marks)** **Loops**. Consider the consequences of eliminating the two conditional branch instructions from SM213 (and not adding any other instructions). Would compilers still be able to compile C programs to run on this modified machine? If so, explain how. If not, carefully explain what feature or features of C would be impossible to execute on the modified machine.

If-then statements whose test condition is a dynamic value and loops that execute a bounded and dynamically determined number of times.

**8** **(4 marks)** **Dynamic Allocation**. The following code contains a procedure that creates a copy of a null-terminated string (the standard representation for strings in C). It contains a serious bug related to dynamic memory allocation.

```
char* copy (char* s) {
  int   i, len = 0;
  char* cpy;

  while (s [len] != 0)
    len++;
  cpy = (char*) malloc (len+1);
  for (i=0, i<len; i++)
    cpy [i] = s [i];
  cpy [len] = 0;
  return cpy;
}
```

```
void doSomething () {
  char* x;
  x = copy ("Hello World");
  printf ("%s", x);
}
```

Explain in plan English what the bug is and how you would fix it (without changing the semantics of `copy`).

The `copy()` procedure allocates memory that is never freed. The simplest fix is to insert a `free(x)` statement as the last line of `doSomething()`; you get full marks for this answer. A better fix is to move the allocation to `doSomething()` and have it pass the target string to `copy()` as a parameter instead of having `copy()` return it; one bonus mark is available for a good explanation of the better solution.

You didn't have to give the code, but here is the code for the better solution.

```
void copy (char* cpy, char* s, int n) {
  int   i, len = 0;

  while (s [len] != 0 && len+1 < n)
    len++;
  for (i=0, i<len; i++)
    cpy [i] = s [i];
  cpy [len] = 0;
  return cpy;
}

void doSomething () {
  char x[1000];
  copy (x, "Hello World", sizeof (x));
  printf ("%s", x);
}
```

**9** **(10 marks)** **Writing Assembly Code**. Write SM213 assembly code that implements the following C program. Use labels for static addresses but do not include variable label declarations (i.e. "`.long`" lines). Show only the code for these two procedures. Do not implement a return from `callReplace()`; simply halt at the end of that procedure. Do not use the stack. Comment every line.

```
int* a;
int  searchFor, replaceWith, size, i;

void replace() {
    for (i=0; i<size; i++)
        if (a[i]==searchFor)
            a[i]=replaceWith;
}
```

```
void callReplace() {
    replace();
    // halt; do not return
}
```

```
replace:      ld   $size, r0          # r0 = &size
              ld   0x0(r0), r0        # r0 = size = i
              ld   $a, r1             # r1 = &a
              ld   0x0(r1), r1        # r1 = a
              ld   $searchFor, r2     # r2 = &searchFor
              ld   0x0(r2), r2        # r2 = searchFor
              not  r2                 # r2 = !searchFor
              inc  r2                 # r2 = -searchFor
              ld   $replaceWith, r3   # r3 = &replaceWith
              ld   0x0(r3), r3        # r3 = replaceWith
loop:         beq  r0, done           # goto done if i==0
              dec  r0                 # i--
              ld   (r1, r0, 4), r4    # r4 = a[i]
              add  r2, r4             # r4 = a[i] - searchFor
              beq  r4, match          # goto match   if a[i]==searchFor
              br   nomatch            # goto nomatch if a[i]!=searchFor
match:        st   r3, (r1, r0, 4)    # a[i] = replaceWith
nomatch:      br   loop               # goto loop
done:         j    0x0(r6)            # return
callReplace:  gpc  $0x6, r6           # ra = pc + 6
              j    replace            # replace()
              halt
```

**5b** Give an equivalent C procedure (i.e., a procedure that may have compiled to this assembly code).

```
int X(int (*a)(int), int* b, int c) {
  int x = 0;
  for (int i = 0; i < c; i++) {
    if (a(b[i]) & 1)
        x += a(b[i]);
  }
  return x;
}
```

Pay attention to how the loop guard `i < c` was derived from the assembly code - the assembly code said `goto L2 if i >= c` meaning "end the loop if `i >= c`". The opposite of this is then "only enter the loop if `i < c`". This is one possible way the loop could have been written. If you wanted to translate the assembly code exactly as you saw it, this is what you would get:

```
int i = 0;
while (1) {
  if (i - c == 0) break;
  if (i - c > 0) break;
  if (a(b[i]) & 1)
    x += a(b[i]);
  i++;
}
```

You probably wouldn't lose marks for this, but I think the way it's written in the answer is more likely to be the original C code. Learn to identify common looping patterns like iterating from 0 to some number. At this point, you may want to even rename your variables to show that you really understand what the code is doing. Something like this:

```
int X(int (*fn)(int), int* array, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    if (fn(array[i]) & 1)
      sum += fn(array[i]);
  }
  return sum;
}
```

You don't have to name every variable like this in an exam. I do this here just to show you exactly what the code is doing. However if you do choose to use good variable names, it will help you convince yourself that you've translated the code correctly. It should be clear now that this code is applying the given function to all odd elements in the array and summing the results. (`n & 1 == 1` implies `n` is odd).

**6 (3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

Is there a bug in this code? If so, carefully describe what it is.

Yes there is a bug in this code. The function does not performing bounds checking on the array `b` and we are not given its size, so this could modify anything in memory.

**7** **(6 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}

void three () {
    int* ret = one();
    two();
}
```

**7a** Is there a bug in this code? If so, carefully describe what it is.

Yes. `one()` returns a dangling pointer to a local variable. Local variables have undefined values after the function returns, because other functions will use the same spot on the stack when they're called.

**7b** What is the value of "`*ret`" at the end of `three`? Explain carefully.

`*ret = 2` because when `two()` is called, it allocates `zot` at the same place where `loc` used to be allocated and overwrites its value.

This question requires you to make an assumption that all local variables are stored on the stack (as opposed to registers, or optimized away by the compiler). Now you must be comfortable with visualizing the stack and how it changes over time.

Below I have drawn what the stack might look like after just entering the function `three()`. Unknown values like return addresses I denote as ... but of course they would actually have some numeric value: the caller's memory address to return to.

| address | value | meaning |
|---|---|---|
| 0x1000 | 0 | unallocated |
| 0x1004 | 0 | unallocated |
| 0x1008 | 0 | unallocated |
| 0x100C | 0 | local var `ret` |
| 0x1010 | ... | return address to `three()`'s caller |

Now `three()` calls `one()`. Before the line `return &loc;` executes, this is what the stack would look like. Pay attention to the address of `loc` because that is what `one()` will return.

| address | value | meaning |
|---|---|---|
| 0x1000 | 0 | unallocated |
| 0x1004 | 1 | local var `loc` |
| 0x1008 | ... | return address to `three()` |
| 0x100C | 0 | local var `ret` |
| 0x1010 | ... | return address to `three()`'s caller |

Now when `one()` returns and before we call `two()`, the stack will be as follows. **Notice how elements on the stack are unallocated but retain their values**. What would the value of `*ret` be at this point?

| address | value | meaning |
|---|---|---|
| 0x1000 | 0 | unallocated |
| 0x1004 | 1 | unallocated |
| 0x1008 | ... | unallocated |
| 0x100C | 0x1004 | local var `ret` |
| 0x1010 | ... | return address to `three()`'s caller |

Below is the stack when we enter `two()`. It shows you why `*ret = 2` when we return to `three()`.

| address | value | meaning |
|---|---|---|
| 0x1000 | 0 | unallocated |
| 0x1004 | 2 | local var `zot` |
| 0x1008 | ... | return address to `three()` |
| 0x100C | 0x1004 | local var `ret` |
| 0x1010 | ... | return address to `three()`'s caller |

# 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

- Garbage collection is a partial solution to the dangling pointer problem. False
- Garbage collection is a partial solution to the memory leak problem. True
- Garbage collection is a full solution to the dangling pointer problem. True
- Garbage collection is a full solution to the memory leak problem. False
- The stack is a partial solution to the dangling pointer problem. False
- Having memory allocation and deallocation in separate functions is a partial solution to the dangling pointer problem. False

**4** **(12 marks)** **Global Arrays and Writing Assembly Code.** In the context of the following C declarations:

```
int *b;
int a[10];
int i = 3;
a[i] = 2;
b = &a[5];
b[i] = 4;
```

Provide the SM213 assembly code for the C code above, with comments. Be as concise as possible. You may assume labels `$i`, `$a`, `$b` have been created pointing to appropriate memory locations for storage, so there is no need to write any assembly for the first two lines of C code.

```
      ld $i, r0            # r0 = &i
      ld $0x3, r1          # r1 = 3
      st r1, 0x0(r0)       # i = 3
      ld $a, r2            # r2 = &a
      ld $0x2, r3          # r3 = 2
      st r3, (r2, r1, 4)   # a[i] = 2
      ld $b, r4            # r4 = &b
      ld $20, r6           # r6 = 20 (5*4)
      add r2, r6       # r6 = &a[5]
      st r6, 0x0(r4)       # b = &a[5]
      ld $0x4, r7          # r7 = 4
      st r7, (r6, r1, 4)   # b[i] = 4
```

(12 marks, one per line. 1/2 mark off for verbose/unnecessary instructions. 1/2 mark off for manual add instead of using an indexed instruction. No penalty for leaving off the 0 offset when using base/displacement load or store.)

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

```
struct S {
    int i[3];
    int j[4];
    int k;
};
struct S  a;
struct S* b;
```

**5a** Indicate which of the following the compiler knows statically and which is determined dynamically.

- `&(a.i[2])` static
- `&(b->j[2])` dynamic
- `&(b->k)` dynamic
- `(&(b->j[1]) - &(b->j[2]))` static

## 6. Dynamic Allocation [10 marks]

Consider each of the following code blocks where `add`, `doAdd`, and `doSomethingElse` are in three different modules of a large program. For each block indicate whether the code has a memory leak or dangling pointer and which technique (if any) is ***the single best way (only one)*** to fix the bug or improve the code (even if it doesn't have a bug). Select *none* if the code does not need to be improved or if the needed change isn't listed.

a)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
}
```

Error? ○ Dangling Pointer ● Memory Leak ○ Both ○ Neither or can not determine

Change? ○ Move malloc or free ● Add malloc or free ○ Add reference counting ○ None of these

b)

```
int* add(int *a, int *b, n) {
  int* c = malloc(n * sizeof(int));
  for (int i = 0; i<n; i++)
    c[i] = a[i] + b[i];
  return c;
}
```

```
void doAdd() {
  int* a = malloc(1 * sizeof(int));
  int* b = malloc(1 * sizeof(int));
  a[0] = 1;  b[0] = 2;
  int* c = add(a, b, 1);
  printf("%d\n", c[0]);
  free(a);
  free(b);
  free(c);
}
```

Error? ○ Dangling Pointer ○ Memory Leak ○ Both ● Neither or can not determine

Change? ● Move malloc or free ○ Add malloc or free ○ Add reference counting ○ None of these