7. Reference Counting [8 marks]

Modify the following **three independent modules** to eliminate memory leaks and avoid dangling pointers by inserting `free` statements where appropriate and/or changing `malloc`'s to `rc_malloc`'s and inserting `rc_keep_ref` and `rc_free_ref` calls where appropriate, following the best practice for using reference counting. All calls must include correct parameters. The code spans two pages.

```
//////////
// Module A

// private to A and not accessible in any other module
struct A {
  struct S* s;
};
int c = 0;
struct A* a = NULL;

// other modules can call foo
void foo(struct S* s) {



  if (c % 3 == 0) {



    a = malloc(sizeof (struct A));



    a->s = s;



  }



  c = c + 1;



  *s->i = c;



}
```

```
//////////
// Module B

int bar() {



  struct S* s = malloc(sizeof(struct S));



  s->i = malloc(sizeof(int));



  foo(s);



  return *s->i;



}



//////////
// Shared between models A and B

struct S {
  int* i;
};


//////////
// Module C

int main() {
  int s = 0;
  for (int i=0; i<100; i++)
    s = s + bar();
  printf("%d\n", s);
}
```

7. Reference Counting [8 marks]

Modify the following **three independent modules** to eliminate memory leaks and avoid dangling pointers by inserting `free` statements where appropriate and/or changing `malloc`'s to `rc_malloc`'s and inserting `rc_keep_ref` and `rc_free_ref` calls where appropriate, following the best practice for using reference counting. All calls must include correct parameters. The code spans two pages.

```
//////////
// Module A

// private to A and not accessible in any other module
struct A {
  struct S* s;
};
int c = 0;
struct A* a = NULL;

// other modules can call foo
void foo(struct S* s) {
  if (c % 3 == 0) {
    if (a) {
      rc_free_ref(a->s->i);
      rc_free_ref(a->s);
      free(a);
    }
    a = malloc(sizeof (struct A));
    a->s = s;
    rc_keep_ref(a->s->i);
    rc_keep_ref(a->s);
  }
  c = c + 1;
  *s->i = c;
}
```

```
//////////
// Module B

int bar() {
  struct S* s = rc_malloc(sizeof(struct S));
  s->i = rc_malloc(sizeof(int));
  foo(s);
  int t = *s->i;
  rc_free_ref(s->i);
  rc_free_ref(s);
  return t;
}

//////////
// Shared between models A and B

struct S {
  int* i;
};

//////////
// Module C

int main() {
  int s = 0;
  for (int i=0; i<100; i++)
    s = s + bar();
  printf("%d\n", s);
}
```

7. Reference Counting [8 marks]

Modify the following **three independent modules** to eliminate memory leaks and avoid dangling pointers by inserting `free` statements where appropriate and/or changing `malloc`'s to `rc_malloc`'s and inserting `rc_keep_ref` and `rc_free_ref` calls where appropriate, following the best practice for using reference counting. All calls must include correct parameters. The code spans two pages.

```
//////////
// Module A

// private to A and not accessible in any other module
struct A {
  struct S* s;
};
int c = 0;
struct A* a = NULL;

// other modules can call foo
void foo(struct S* s) {
  if (c % 3 == 0) {
    if (a) {
      rc_free_ref(a->s->i);
      rc_free_ref(a->s);
      free(a);
    }
    a = malloc(sizeof (struct A));
    a->s = s;
    rc_keep_ref(a->s->i);
    rc_keep_ref(a->s);
  }
  c = c + 1;
  *s->i = c;
}
```

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;
}
```

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* copy (int* src) {

  int* dst = malloc (sizeof (int));

  *dst = *src;

  return dst;

}


int foo() {

  int  a = 3;

  int* b = copy (&a);

  saveIfMax (b);

  return *b;

}
```

```
int* max;

void saveIfMax (int* x) {

  if (max==NULL || *x > *max) {

    max = x;

  }

}
```

7. Reference Counting [8 marks]

Modify the following **three independent modules** to eliminate memory leaks and avoid dangling pointers by inserting `free` statements where appropriate and/or changing `malloc`'s to `rc_malloc`'s and inserting `rc_keep_ref` and `rc_free_ref` calls where appropriate, following the best practice for using reference counting. All calls must include correct parameters. The code spans two pages.

```
//////////
// Module A

// private to A and not accessible in any other module
struct A {
  struct S* s;
};
int c = 0;
struct A* a = NULL;

// other modules can call foo
void foo(struct S* s) {
  if (c % 3 == 0) {
    if (a) {
      rc_free_ref(a->s->i);
      rc_free_ref(a->s);
      free(a);
    }
    a = malloc(sizeof (struct A));
    a->s = s;
    rc_keep_ref(a->s->i);
    rc_keep_ref(a->s);
  }
  c = c + 1;
  *s->i = c;
}
```

```
//////////
// Module B

int bar() {
  struct S* s = rc_malloc(sizeof(struct S));
  s->i = rc_malloc(sizeof(int));
  foo(s);
  int t = *s->i;
  rc_free_ref(s->i);
  rc_free_ref(s);
  return t;
}

//////////
// Shared between models A and B

struct S {
  int* i;
};

//////////
// Module C

int main() {
  int s = 0;
  for (int i=0; i<100; i++)
    s = s + bar();
  printf("%d\n", s);
}
```

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).

```
/***********
 * Module a
 */

int* a_create(int i) {

  int* value = rc_malloc(sizeof(int));



  *value = i;



  return value;
}


/**********
 * Module b
 */

#define B_SIZE 4
int* b_values[A_SIZE];

void b_init() {
  for (int i=0; i<B_SIZE; i++)
    b_values[i] = NULL;
}

void b_put(int index, int* value) {

  if (index>=0 && index<B_SIZE) {

     if (b_values[index])
         rc_free_ref(b_values[index]);



    b_values[index] = value;

     if (value)
         rc_inc_ref(b_values[index]);



  }
}

int* b_get(int index) {
  return b_values[index];
}
```

```
/********
 * Header file content for the three modules - this is all that main can access
 */
int* a_create(int i);
void b_init();
void b_put(int index, int* value);
int* b_get(int index);


/*******
 * main
 */

int main(int arc, char** argv) {
    b_init();
    b_put(0, a_create(10));
    b_put(1, a_create(20));
    b_put(2, b_get(0));
    b_put(3, b_get(2));
    int* p = b_get(1);
    rc_keep_ref(p);
    int* q = b_get(3);
    rc_keep_ref(q);
    printf("%d %d", *p, *q));
    rc_free_ref(p);
    rc_free_ref(q);
    //TODO

     rc_free_ref(p);
     rc_free_ref(q);

     b_put(0, NULL);
     b_put(1, NULL);
     b_put(2, NULL);
     b_put(3, NULL);









}
```

8 **[15 marks]** **Reading Assembly.** Comment the following assembly code and then translate it into C. Assume that the caller prologue was completed as shown in lecture, and that *register 0* is used to return a value. *Use the back of the preceding page for extra space if you need 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);
}
```

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

void bar (int* ap) {
  if (v[c] == NULL || *ap < *v[c]) {
    if (v[c] != NULL)
      dec(v[c]);
    v[c] = ap;
    inc(v[c]);
  }
  c = (c + 1) % 2;
}

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

10 **(6 marks)** **Using Function Pointers.** Write a C procedure named `apply` that returns either the minimum or maximum value of a list of non-negative integers, as determined by one of its parameters, a function pointer. If the list is empty it should return -1. Assume the existence of procedures named `min(a,b)` and `max(a,b)` that compare two integers and returns the min or max integer. No other procedures are allowed and `apply` is not permitted to use an `if` statement. For example, the following statement should compute the maximum value in list `a` of length `n`.

```
int m = apply (max, a, n);
```

Write the procedure `apply()`:

```
int apply (int (*f)(int, int), int* a, int n) {
    if (n<=0)
        return -1;   // for 2016W2 this isn't necessary.
    else {
        int v = a[0];
        for (int i=1; i<n; i++)
            v = f(v, a[i]);
        return v;
    }
}
```

11 **(6 marks)** **IO Devices.** For each of the following, (a) explain what it is and (b) state whether the CPU or IO controller determines when it occurs (i.e., initiates it).

11a Programmed IO (PIO):

11b DMA:

11c Interrupts:

12 **(8 marks)** **Threads and Scheduling.** Answer the following questions about threads.

12a Explain briefly (without giving any code) how threads can be used to simplify code that performs asynchronous operations such as communicating with an IO controller.

12b Explain briefly the difference between `uthread_yield` and `uthread_block`.

12c Is it possible for a thread to unblock itself? Explain your answer.

12d Consider a system in which there is at least one thread on the *ready queue* when a thread unblocks. What happens to the unblocked thread? Explain.

13 **(9 marks)** **Synchronization** Consider the following C code that uses mutexes and condition variables. The code is considered to be is correct if both procedures complete successfully when they are called from concurrent threads

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**.

```
//////////
// Module B

int bar() {
  struct S* s = rc_malloc(sizeof(struct S));
  s->i = rc_malloc(sizeof(int));
  foo(s);
  int t = *s->i;
  rc_free_ref(s->i);
  rc_free_ref(s);
  return t;
}

//////////
// Shared between models A and B

struct S {
  int* i;
};

//////////
// Module C

int main() {
  int s = 0;
  for (int i=0; i<100; i++)
    s = s + bar();
  printf("%d\n", s);
}
```

7. Reference Counting [8 marks]

Modify the following **three independent modules** to eliminate memory leaks and avoid dangling pointers by inserting `free` statements where appropriate and/or changing `malloc`'s to `rc_malloc`'s and inserting `rc_keep_ref` and `rc_free_ref` calls where appropriate, following the best practice for using reference counting. All calls must include correct parameters. The code spans two pages.

```
//////////
// Module A

// private to A and not accessible in any other module
struct A {
  struct S* s;
};
int c = 0;
struct A* a = NULL;

// other modules can call foo
void foo(struct S* s) {
  if (c % 3 == 0) {
    if (a) {
      rc_free_ref(a->s->i);
      rc_free_ref(a->s);
      free(a);
    }
    a = malloc(sizeof (struct A));
    a->s = s;
    rc_keep_ref(a->s->i);
    rc_keep_ref(a->s);
  }
  c = c + 1;
  *s->i = c;
}
```

```
//////////
// Module B

int bar() {
  struct S* s = rc_malloc(sizeof(struct S));
  s->i = rc_malloc(sizeof(int));
  foo(s);
  int t = *s->i;
  rc_free_ref(s->i);
  rc_free_ref(s);
  return t;
}

//////////
// Shared between models A and B

struct S {
  int* i;
};

//////////
// Module C

int main() {
  int s = 0;
  for (int i=0; i<100; i++)
    s = s + bar();
  printf("%d\n", s);
}
```

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?

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:`

`*q:`

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).

```
/***********
 * Module a
 */

int* a_create(int i) {

  int* value = rc_malloc(sizeof(int));



  *value = i;



  return value;
}


/**********
 * Module b
 */

#define B_SIZE 4
int* b_values[B_SIZE];

void b_init() {
  for (int i=0; i<B_SIZE; i++)
    b_values[i] = NULL;
}

void b_put(int index, int* value) {

  if (index>=0 && index<B_SIZE) {







    b_values[index] = value;







  }
}

int* b_get(int index) {
  return b_values[index];
}
```

```
/********
 * Header file content for the three modules - this is all that main can access
 */
int* a_create(int i);
void b_init();
void b_put(int index, int* value);
int* b_get(int index);


/*******
 * main
 */

int main(int arc, char** argv) {
    b_init();
    b_put(0, a_create(10));
    b_put(1, a_create(20));
    b_put(2, b_get(0));
    b_put(3, b_get(2));
    int* p = b_get(1);
    rc_keep_ref(p);
    int* q = b_get(3);
    rc_keep_ref(q);
    printf("%d %d", *p, *q));
    rc_free_ref(p);
    rc_free_ref(q);
    //TODO









}
```