Q1: Redesign the `tr_identify` function prototype

```
// Define ad position struct
struct ad_position {
    size_t start;          // Ad start position
    size_t end;            // Ad end position
    double correlation;    // Correlation score
    struct ad_position* next;  // Next node in linked list
};
// Define ad position list struct
struct ad_position_list {
    struct ad_position* head;  // Head of position list
    size_t count;              // Number of ads found
};
// New function prototype
struct ad_position_list* tr_identify(
    struct sound_seg* target,    // Target audio track
    struct sound_seg* ad,        // Ad audio track
    double threshold            // Correlation threshold
);
// Free ad position list
void free_ad_positions(struct ad_position_list* list);
```

Fence 1

Advantages:

1. Structured return results, easier to process and traverse
2. Contains more info (correlation scores)
3. Supports custom thresholds
4. Clear memory management interface
5. Avoids string parsing overhead

Q2: Memory management strategy

1. Memory ownership tracking:

```
struct audio_node {
 int16_t* samples;
 bool is_shared;         // Flag for shared data
 struct sound_seg* owner; // Original data owner
 // ... other fields
};
struct sound_seg {
 struct audio_node* head;
 size_t ref_count;       // Reference count
 // ... other fields
};
```

Fence 2

2. Memory deallocation strategy:

```
void tr_destroy(struct sound_seg* track) {
```

```
    if (!track) return;
    // Traverse all nodes
    struct audio_node* curr = track->head;
    while (curr) {
        struct audio_node* next = curr->next;
        // Only free non-shared or owner memory
        if (!curr->is_shared || curr->owner == track) {
            free(curr->samples);
        }
        free(curr);
        curr = next;
    }
    // Update reference count
    if (track->ref_count > 0) {
        track->ref_count--;
    }
    // Clean up parent-child relationships
    free_parent_child_nodes(track->children);
    free_parent_child_nodes(track->parents);
    free(track);
}
```

3. Memory leak detection:

Fence 3

```
// Use memory tracking during development
#ifdef DEBUG
struct mem_tracker {
    void* ptr;
    char* type;
    struct mem_tracker* next;
};
static struct mem_tracker* mem_list = NULL;
void track_allocation(void* ptr, const char* type) {
    // Record memory allocation
}
void track_deallocation(void* ptr) {
    // Record memory deallocation
}
void print_memory_leaks(void) {
    // Print unreleased memory
}
#endif
```

Fence 4

Q3: Time complexity analysis

1. `tr_insert` time complexity:

```
bool tr_insert(struct sound_seg* dest_track, size_t destpos,
               struct sound_seg* src_track, size_t srcpos, size_t len) {
    // 1. Find source node: O(n), n = source track node count
    struct audio_node* src_node = src_track->head;
    while (src_node && src_curr_pos + src_node->length <= srcpos) {
        src_curr_pos += src_node->length;
        src_node = src_node->next;
    }
    // 2. Find target position: O(m), m = destination track node count
```

```
    struct audio_node* dest_curr = dest_track->head;
    while (dest_curr && dest_curr_pos < destpos) {
        dest_curr_pos += dest_curr->length;
        dest_curr = dest_curr->next;
    }
    // 3. Insert operation: O(1)
    // 4. Create relationship node: O(1)
}
// Overall time complexity: O(n + m)
```

2. `tr_read` time complexity:

Fence 5

```
bool tr_read(struct sound_seg* track, size_t pos, size_t len, int16_t*
buffer) {
    // 1. Find starting node: O(n), n = track node count
    struct audio_node* node = track->head;
    while (node && pos >= node->length) {
        pos -= node->length;
        node = node->next;
    }
    // 2. Read data: O(len), len = samples to read
    // May span multiple nodes
}
// Overall time complexity: O(n + len)
```

Fence 6

Q4: Testing strategy

1. Unit test framework:

```
// tests/test_framework.h
#define ASSERT(condition, message) \
    do { \
        if (!(condition)) { \
            printf("FAIL: %s\n", message); \
            return false; \
        } \
    } while (0)
#define RUN_TEST(test) \
    do { \
        printf("Running %s...\n", #test); \
        if (test()) printf("PASS: %s\n\n", #test); \
    } while (0)
```

Fence 7

2. Basic operation tests:

```
// tests/test_basic.c
bool test_tr_init() {
    struct sound_seg* track = tr_init();
    ASSERT(track != NULL, "tr_init should return non-NULL");
    ASSERT(track->head == NULL, "New track should have NULL head");
    ASSERT(track->total_length == 0, "New track should have zero length");
    tr_destroy(track);
    return true;
}
bool test_tr_write() {
```

```
 struct sound_seg* track = tr_init();
 int16_t data[] = {1, 2, 3, 4, 5};
 ASSERT(tr_write(track, 0, 5, data), "Write should succeed");
 ASSERT(tr_length(track) == 5, "Length should be 5");
 tr_destroy(track);
 return true;
}
```

3. Advanced functionality tests:

Fence 8

```
// tests/test_advanced.c
bool test_tr_insert_shared() {
 struct sound_seg* src = tr_init();
 struct sound_seg* dest = tr_init();
 int16_t data[] = {1, 2, 3, 4, 5};
 tr_write(src, 0, 5, data);
 ASSERT(tr_insert(dest, 0, src, 1, 3), "Insert should succeed");
 int16_t buffer[3];
 tr_read(dest, 0, 3, buffer);
 ASSERT(buffer[0] == 2 && buffer[1] == 3 && buffer[2] == 4,
        "Inserted data should match source");
 tr_destroy(src);
 tr_destroy(dest);
 return true;
}
```

Fence 9

4. Test script:

```
#!/bin/bash
# tests/run_tests.sh
gcc -o test_runner tests/*.c ../sound_seg.c -I..
./test_runner
```

Fence 10

5. Memory tests:

```
#!/bin/bash
# tests/memory_test.sh
valgrind --leak-check=full ./test_runner
```

Fence 11