On Implementing Specifications

Robbert van Renesse

Specifications are often informal and imprecise

- Written out in plain English
- Often ambiguous

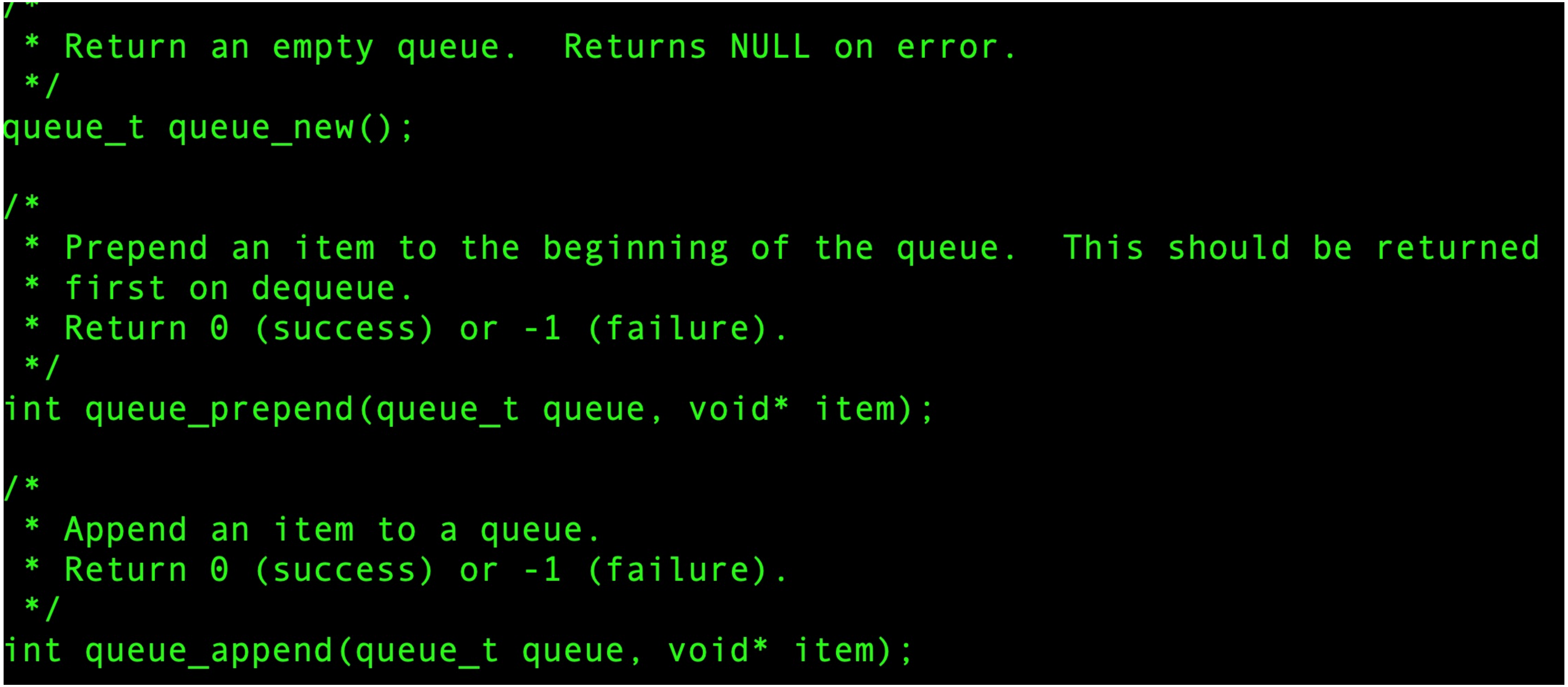
- Should explicitly specify what it should do
  - implicitly: should not do anything else

Example: a queue

- Pseudo-formal spec:
  - The state of a queue is a sequence of values
  - The initial state is the empty sequence
  - Each method involves a state change and a return value
  - A queue supports the following methods:

```
append(x):
        state change: append x to the end of the state
        return value: OK on success, ERROR on failure
dequeue():
        if state is the empty sequence:
                state change: no change
                return value: ERROR
        if state is not the empty sequence
                state change: remove first element from the sequence
                return value: OK(x) where x is first element, ERROR on failure
```

More common queue spec



```
/*
 * Return an empty queue.  Returns NULL on error.
 */
queue_t queue_new();

/*
 * Prepend an item to the beginning of the queue.  This should be returned
 * first on dequeue.
 * Return 0 (success) or -1 (failure).
 */
int queue_prepend(queue_t queue, void* item);

/*
 * Append an item to a queue.
 * Return 0 (success) or -1 (failure).
 */
int queue_append(queue_t queue, void* item);
```

What are examples of “failures”?

- Specified failures:
  - Trying to dequeue a value from an empty queue
  - Trying to free a queue that is non-empty
- Unspecified failures:
  - Trying to allocate memory when there is no memory left
  - Trying to access a disk when the disk has crashed

It’s ok if in your implementation some methods cannot fail and never return an error

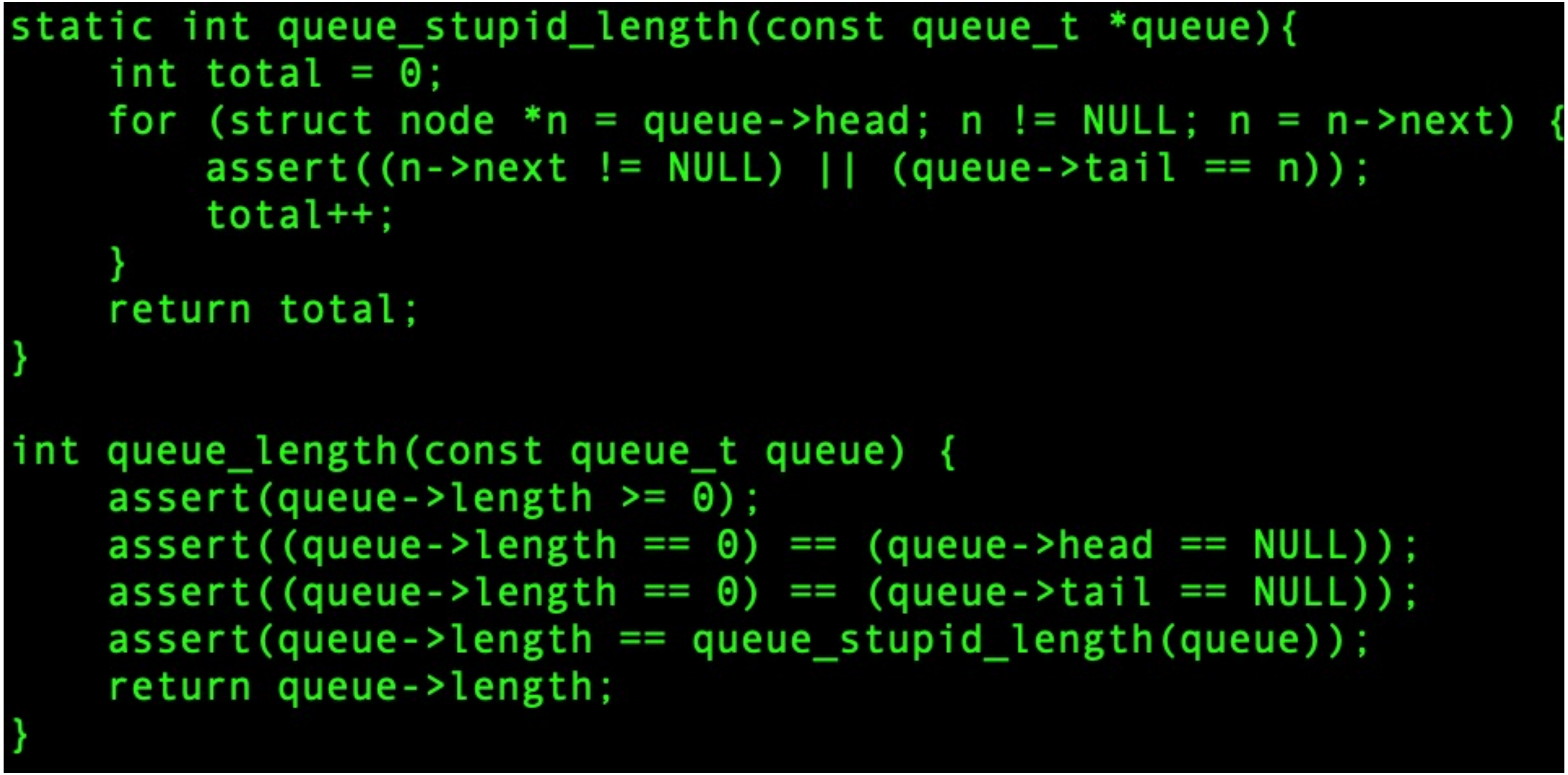
What are not examples of failures

- Somebody tries to use your queue in a way that is not specified
  - That’s a bug, and it is not really your problem
  - Example: queue_length(NULL)
    - behavior is not specified!

Aside: what are assertions?

- They are no-ops!
    - They are part of the specification, not the implementation
    - They are executable comments
- If they are no-ops, why are they useful?
    - Comments are good. Executable comments are even better
    - They help people with understanding your code
    - They can help you find bugs in your code
    - They don’t cost anything because the code is automatically removed from production systems
- Do not use assertions to check for failures!!!!
    - That’s not what they are for (remember: assertions are no-ops)
    - Use **if** (or **try**) statements instead

Good examples of assertions

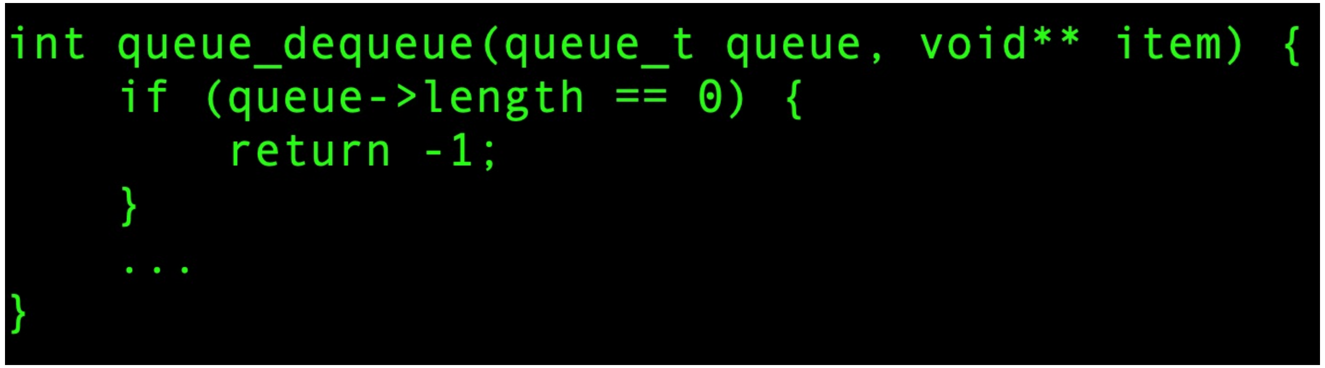


```
static int queue_stupid_length(const queue_t *queue){
    int total = 0;
    for (struct node *n = queue->head; n != NULL; n = n->next) {
        assert((n->next != NULL) || (queue->tail == n));
        total++;
    }
    return total;
}

int queue_length(const queue_t queue) {
    assert(queue->length >= 0);
    assert((queue->length == 0) == (queue->head == NULL));
    assert((queue->length == 0) == (queue->tail == NULL));
    assert(queue->length == queue_stupid_length(queue));
    return queue->length;
}
```

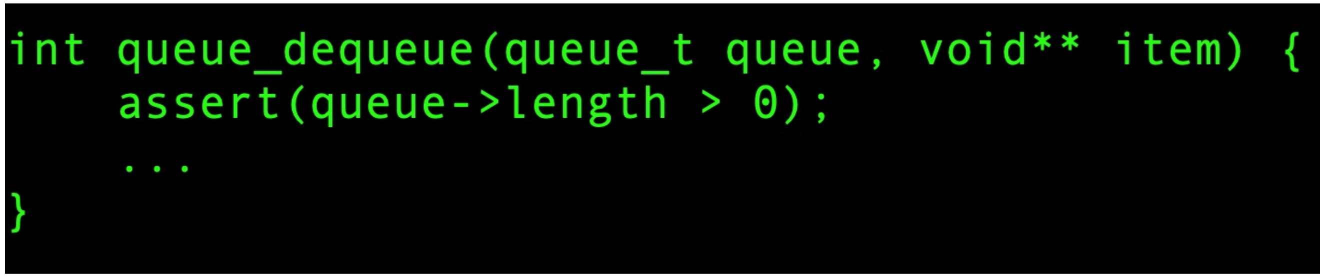
How to deal with failures? Example 1

(A)



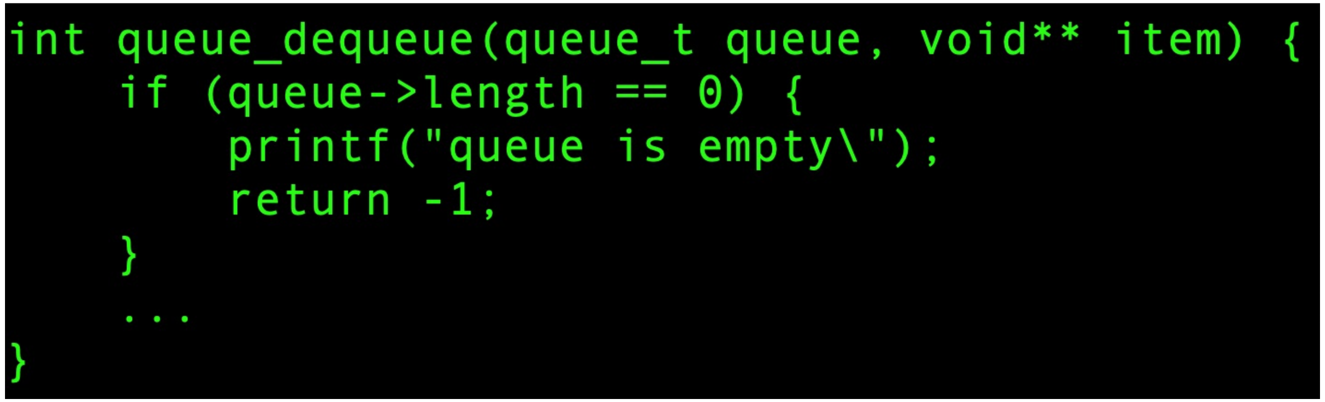
```
int queue_dequeue(queue_t queue, void** item) {
    if (queue->length == 0) {
        return -1;
    }
    ...
}
```

(B)



```
int queue_dequeue(queue_t queue, void** item) {
    assert(queue->length > 0);
    ...
}
```

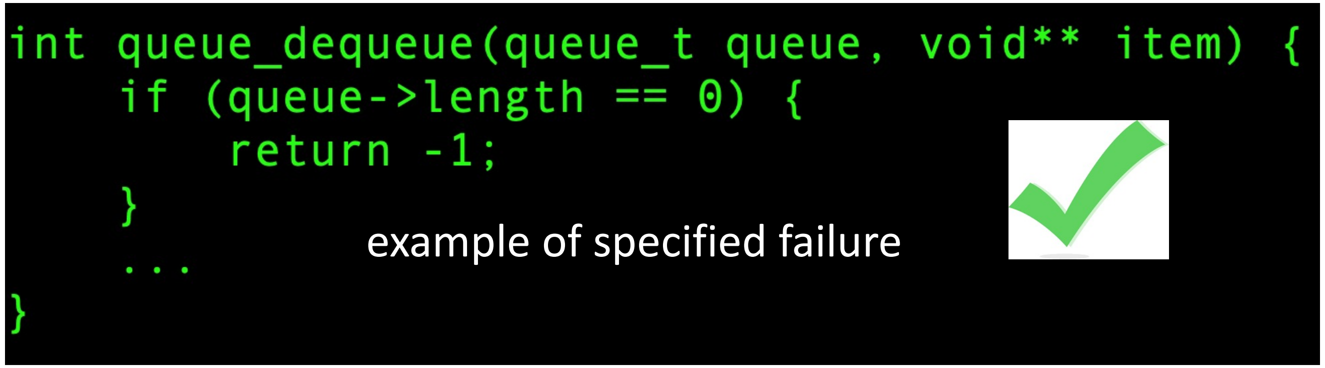
(C)



```
int queue_dequeue(queue_t queue, void** item) {
    if (queue->length == 0) {
        printf("queue is empty\");
        return -1;
    }
    ...
}
```

How to deal with failures? Example 1

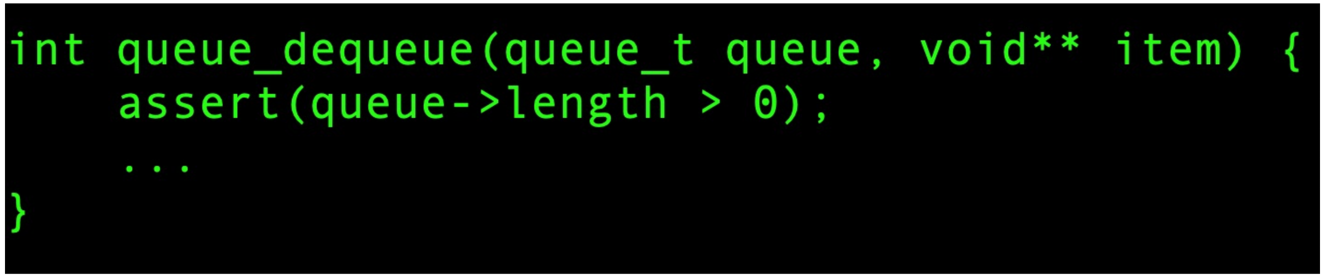
(A)



```
int queue_dequeue(queue_t queue, void** item) {
    if (queue->length == 0) {
        return -1;
    }
    ...
}
```

example of specified failure ✓

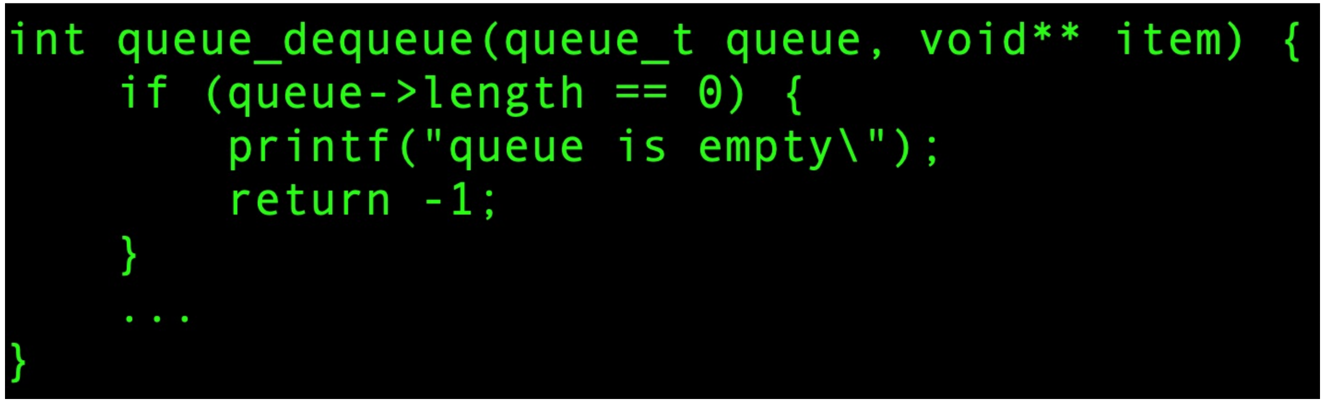
(B)



```
int queue_dequeue(queue_t queue, void** item) {
    assert(queue->length > 0);
    ...
}
```

bug

(C)

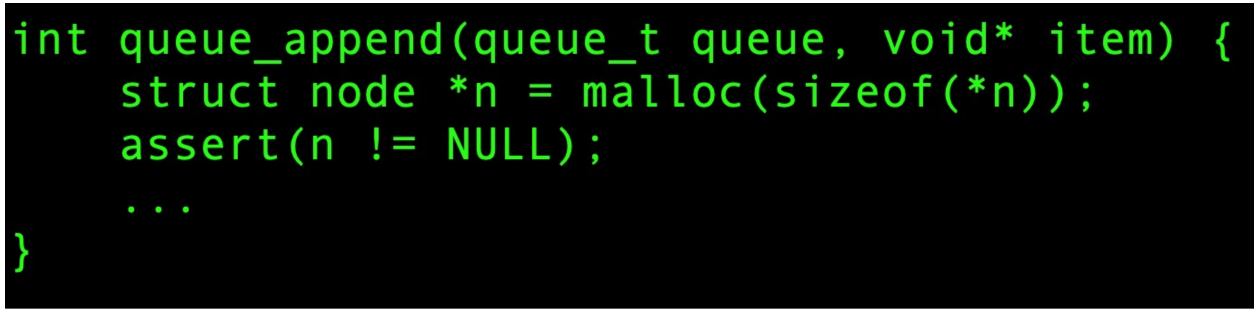


```
int queue_dequeue(queue_t queue, void** item) {
    if (queue->length == 0) {
        printf("queue is empty\");
        return -1;
    }
    ...
}
```

unspecified behavior
(technically a bug as well)

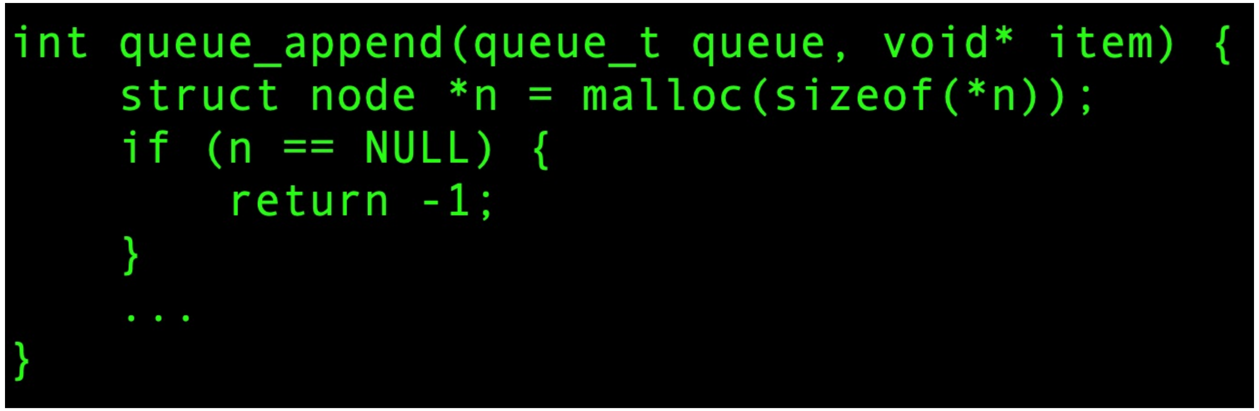
How to deal with failures? Example 2

(A)



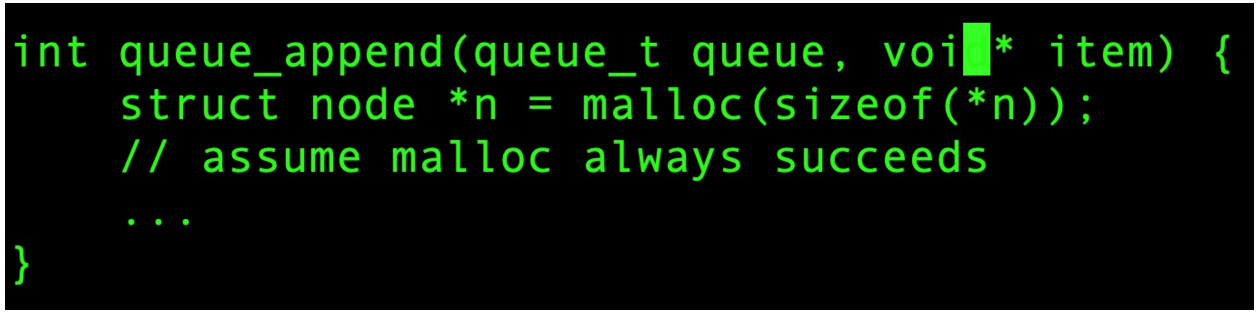
```
int queue_append(queue_t queue, void* item) {
    struct node *n = malloc(sizeof(*n));
    assert(n != NULL);
    ...
}
```

(B)



```
int queue_append(queue_t queue, void* item) {
    struct node *n = malloc(sizeof(*n));
    if (n == NULL) {
        return -1;
    }
    ...
}
```

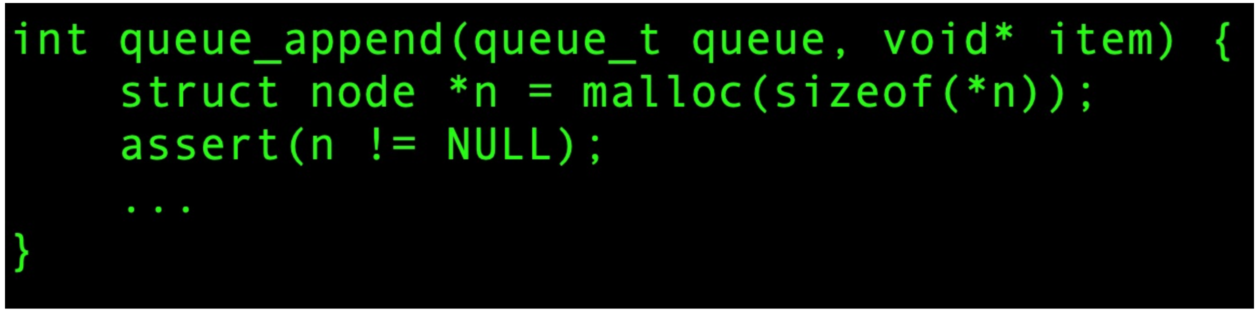
(C)



```
int queue_append(queue_t queue, void* item) {
    struct node *n = malloc(sizeof(*n));
    // assume malloc always succeeds
    ...
}
```

How to deal with failures? Example 2

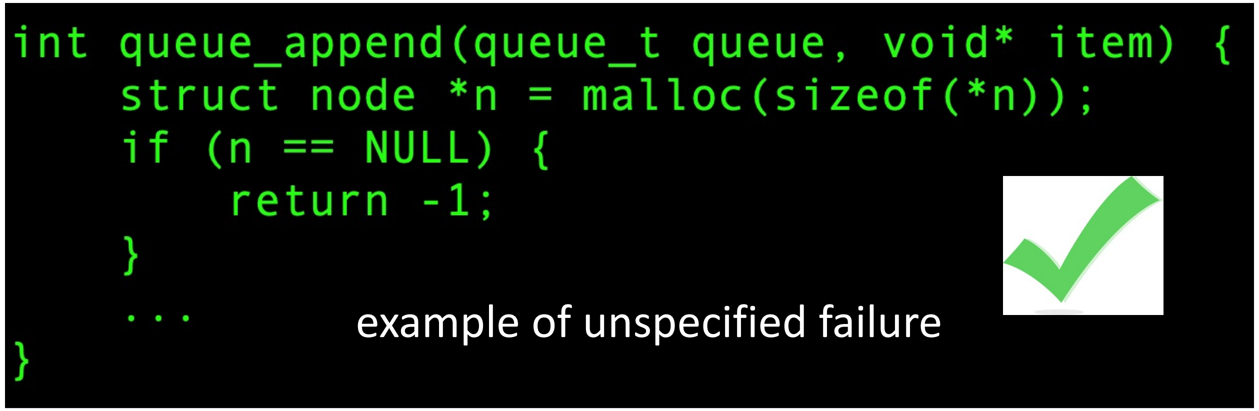
(A)



```
int queue_append(queue_t queue, void* item) {
    struct node *n = malloc(sizeof(*n));
    assert(n != NULL);
    ...
}
```

bug: can’t assume malloc always succeeds

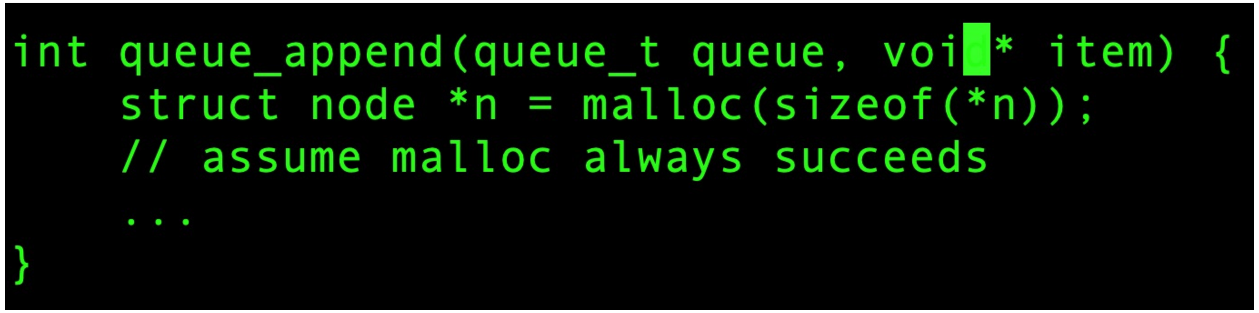
(B)



```
int queue_append(queue_t queue, void* item) {
    struct node *n = malloc(sizeof(*n));
    if (n == NULL) {
        return -1;
    }
    ...
}
```

example of unspecified failure ✓

(C)

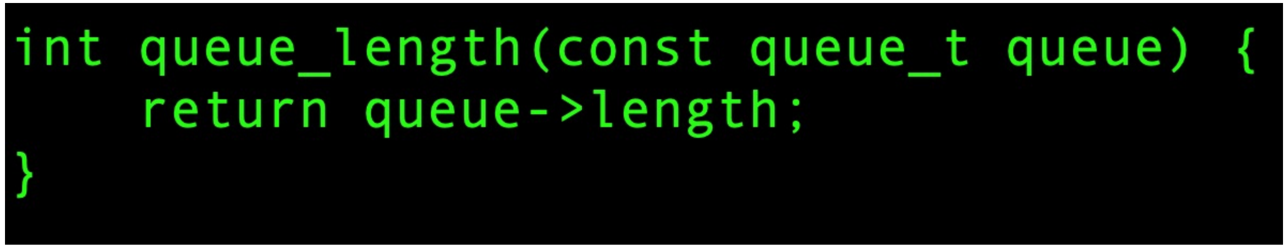


```
int queue_append(queue_t queue, void* item) {
    struct node *n = malloc(sizeof(*n));
    // assume malloc always succeeds
    ...
}
```

(A) and (C) are essentially the same

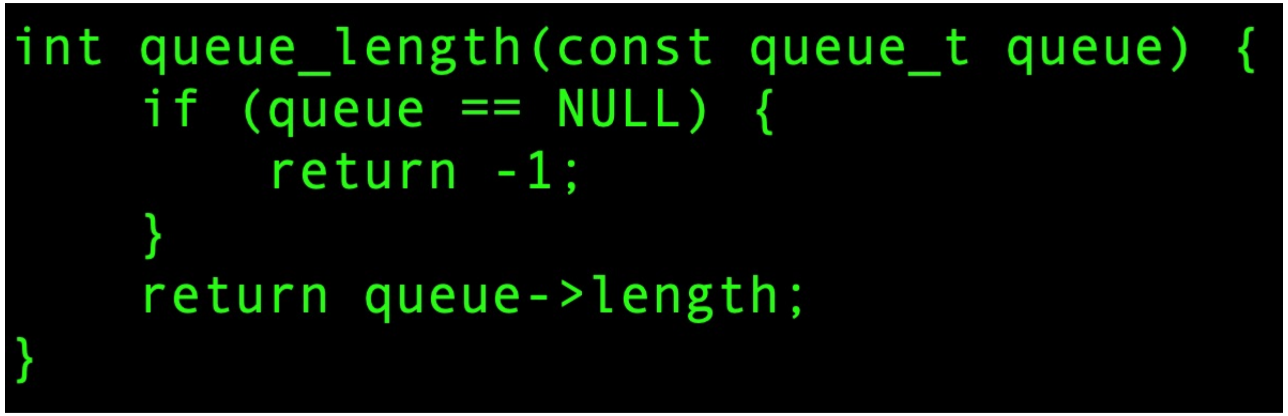
How to deal with failures? Example 3

(A)



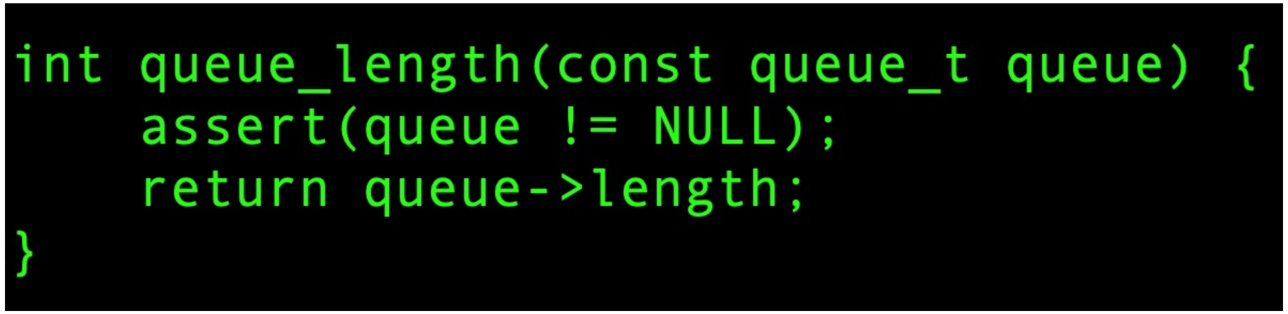
```
int queue_length(const queue_t queue) {
    return queue->length;
}
```

(B)



```
int queue_length(const queue_t queue) {
    if (queue == NULL) {
        return -1;
    }
    return queue->length;
}
```

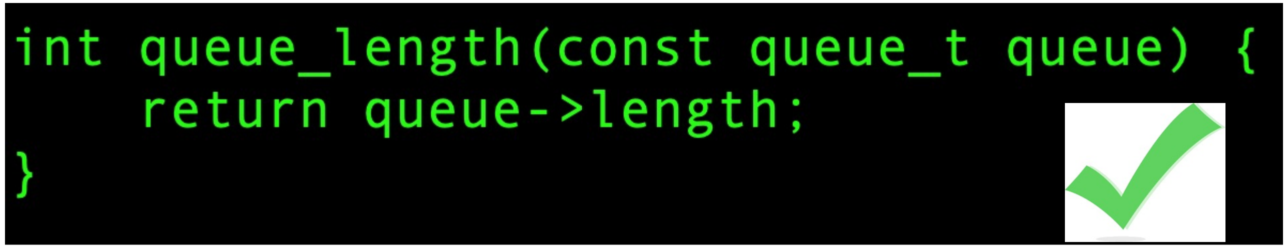
(C)



```
int queue_length(const queue_t queue) {
    assert(queue != NULL);
    return queue->length;
}
```

How to deal with failures? Example 3

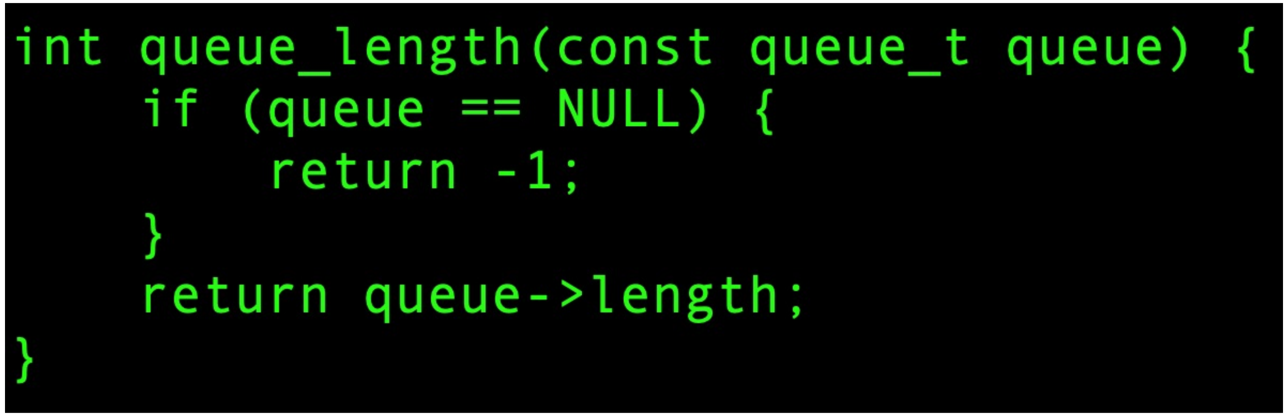
(A)



```
int queue_length(const queue_t queue) {
    return queue->length;
}
```

✓

(B)

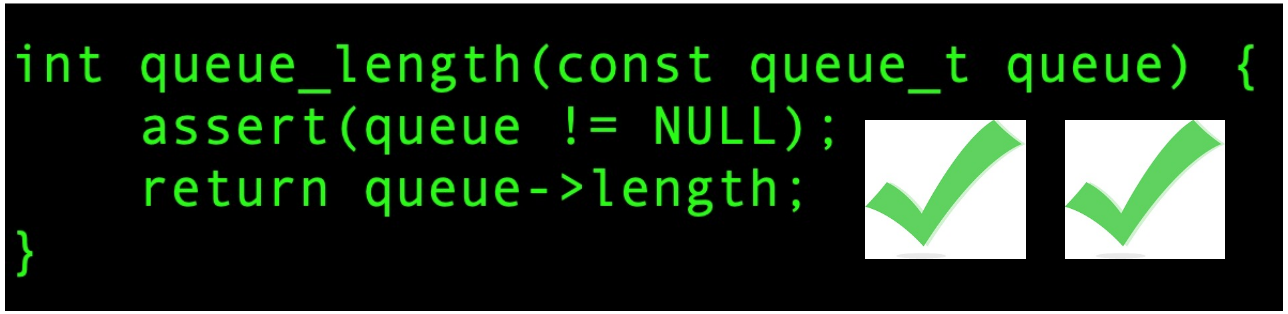


```
int queue_length(const queue_t queue) {
    if (queue == NULL) {
        return -1;
    }
    return queue->length;
}
```

unspecified behavior

- unnecessary overhead
- complicates bug finding

(C)



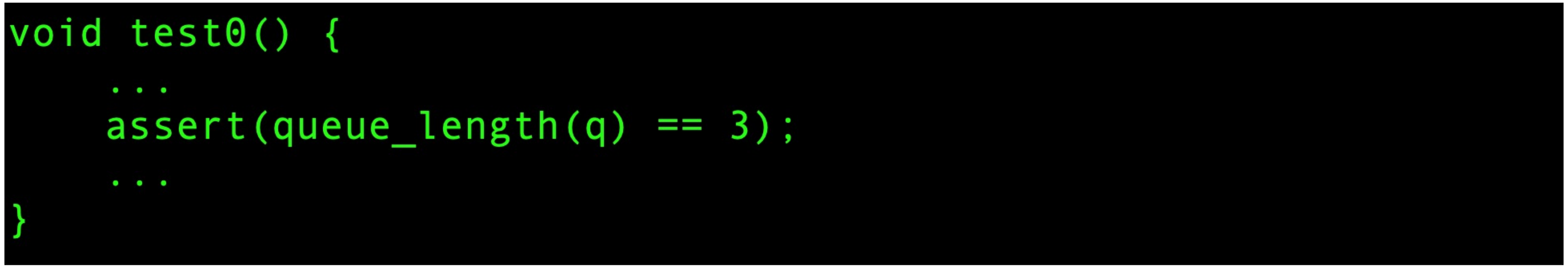
```
int queue_length(const queue_t queue) {
    assert(queue != NULL);
    return queue->length;
}
```

✓ ✓

defensive programming

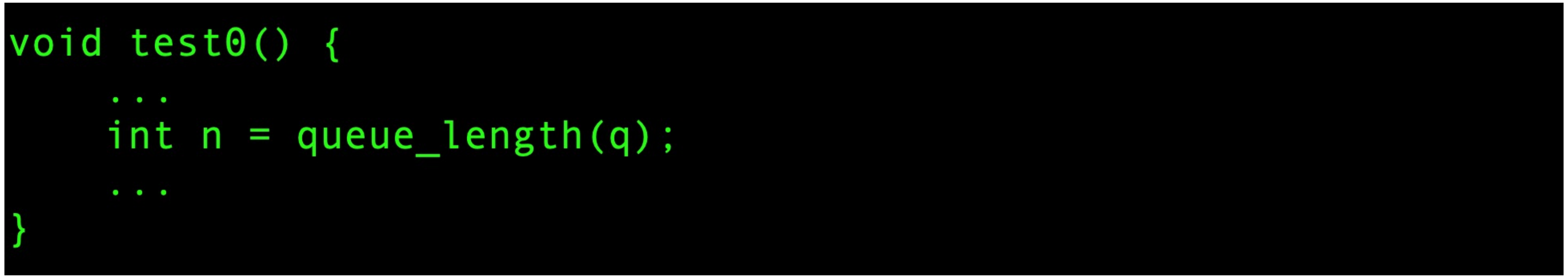
How to deal with failures? Example 4

(A)



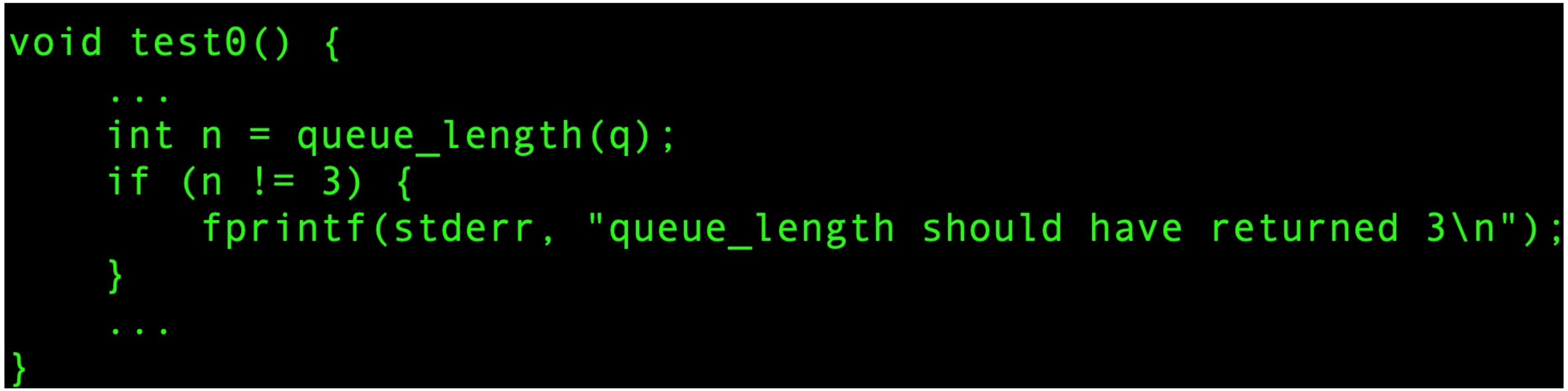
```
void test0() {
    ...
    assert(queue_length(q) == 3);
    ...
}
```

(B)



```
void test0() {
    ...
    int n = queue_length(q);
    ...
}
```

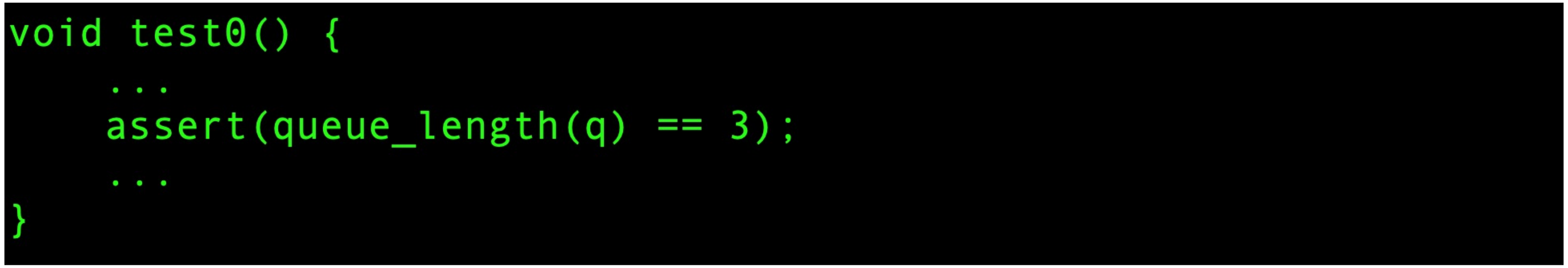
(C)



```
void test0() {
    ...
    int n = queue_length(q);
    if (n != 3) {
        fprintf(stderr, "queue_length should have returned 3\n");
    }
    ...
}
```

How to deal with failures? Example 4

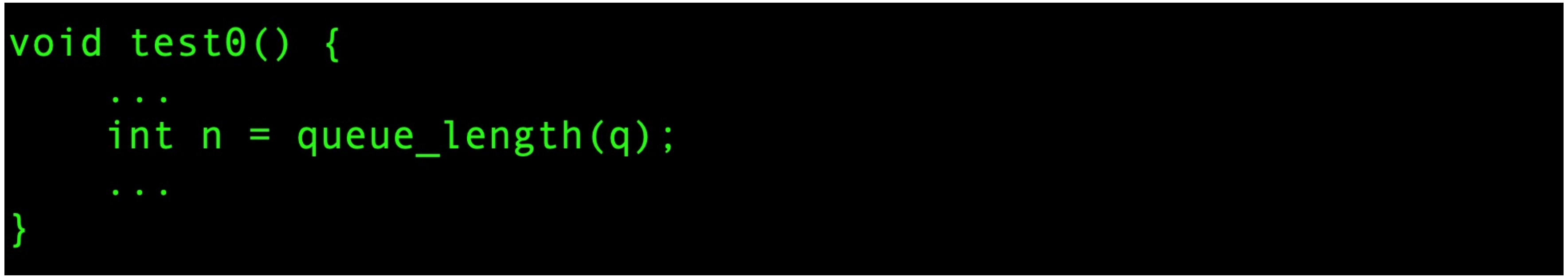
(A)



```
void test0() {
    ...
    assert(queue_length(q) == 3);
    ...
}
```

Doesn’t test anything

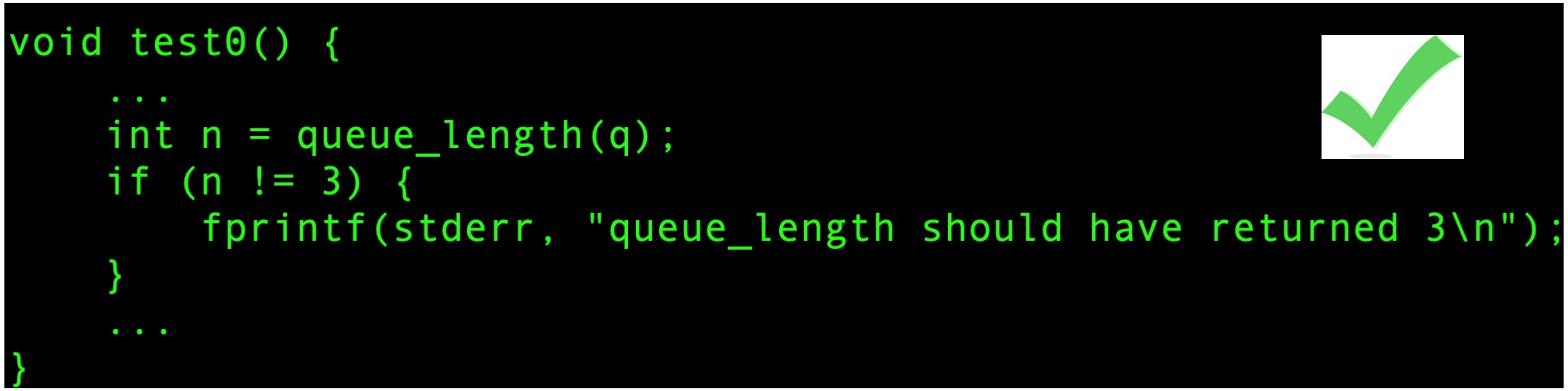
(B)



```
void test0() {
    ...
    int n = queue_length(q);
    ...
}
```

Doesn’t check result

(C)

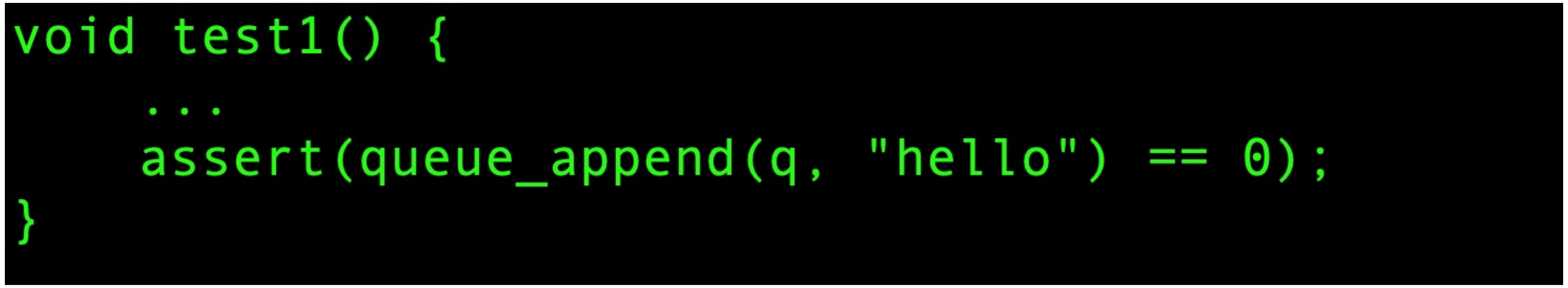


```
void test0() {
    ...
    int n = queue_length(q);
    if (n != 3) {
        fprintf(stderr, "queue_length should have returned 3\n");
    }
    ...
}
```

✓

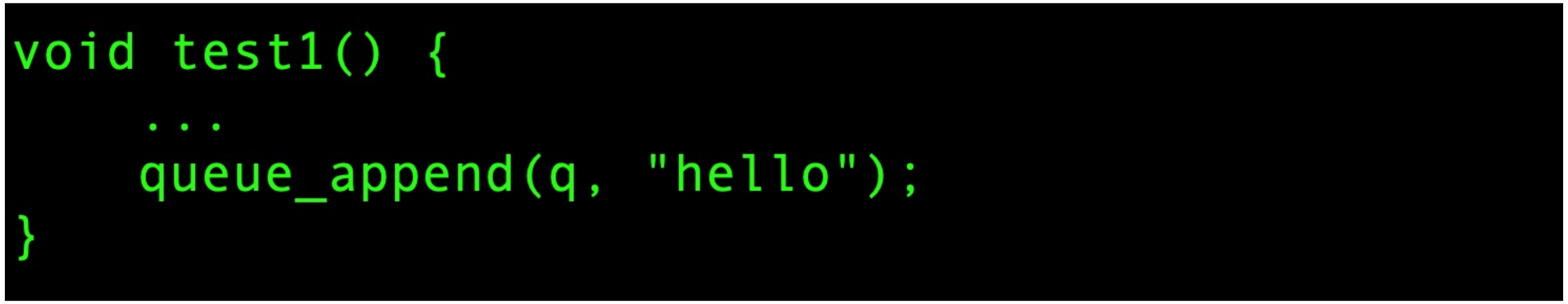
How to deal with failures? Example 5

(A)



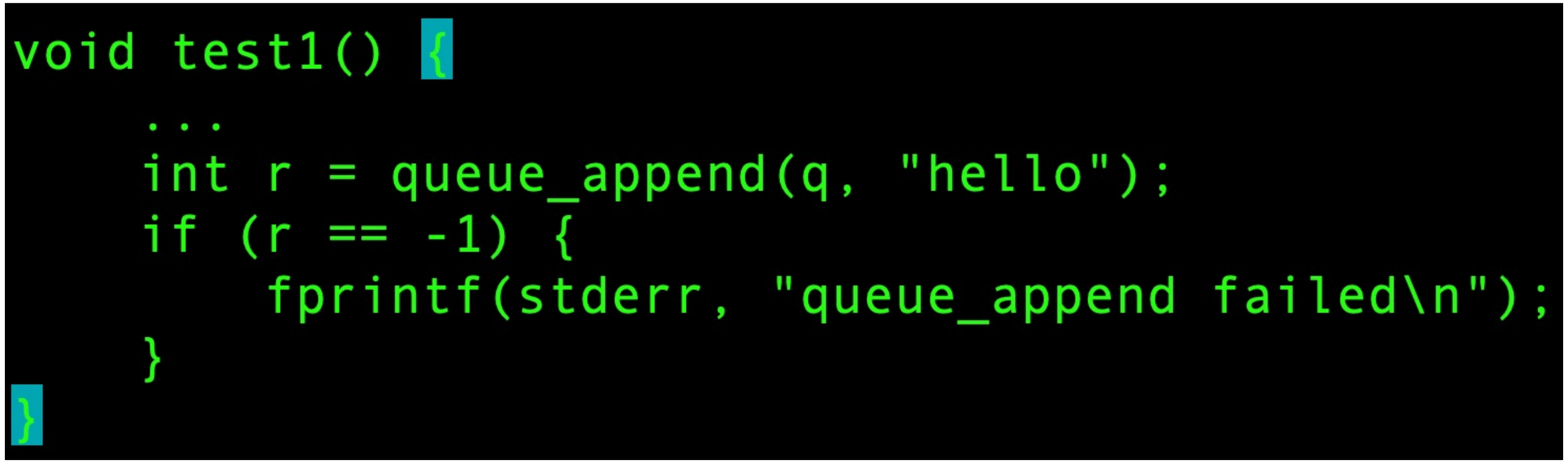
```
void test1() {
    ...
    assert(queue_append(q, "hello") == 0);
}
```

(B)



```
void test1() {
    ...
    queue_append(q, "hello");
}
```

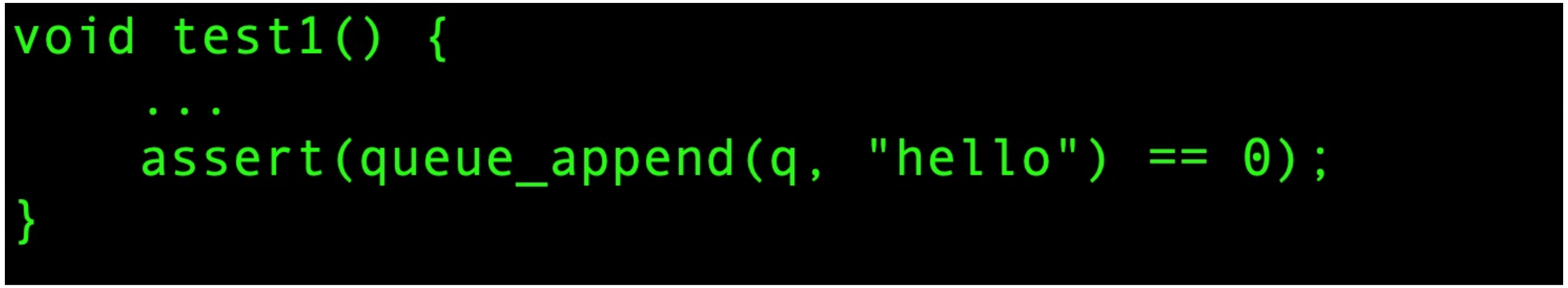
(C)



```
void test1() {
    ...
    int r = queue_append(q, "hello");
    if (r == -1) {
        fprintf(stderr, "queue_append failed\n");
    }
}
```

How to deal with failures? Example 5

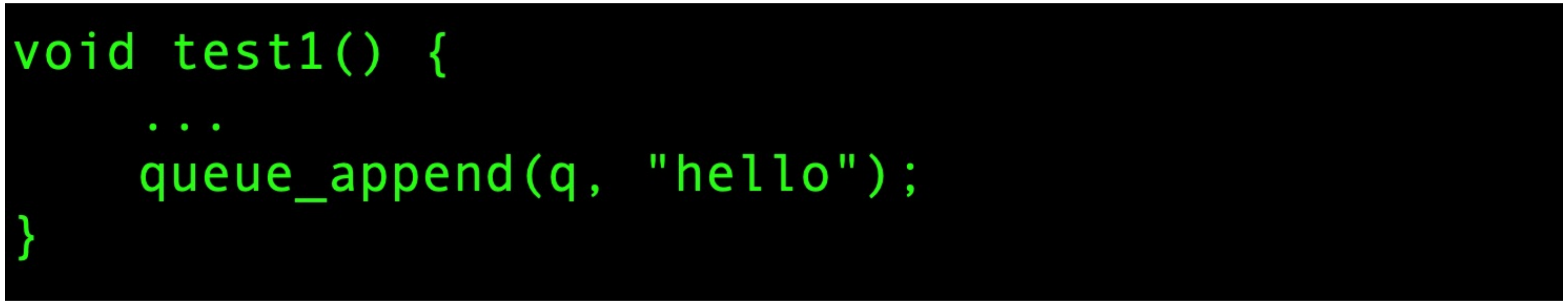
(A)



```
void test1() {
    ...
    assert(queue_append(q, "hello") == 0);
}
```

Assert expression with side-effect is really bad…

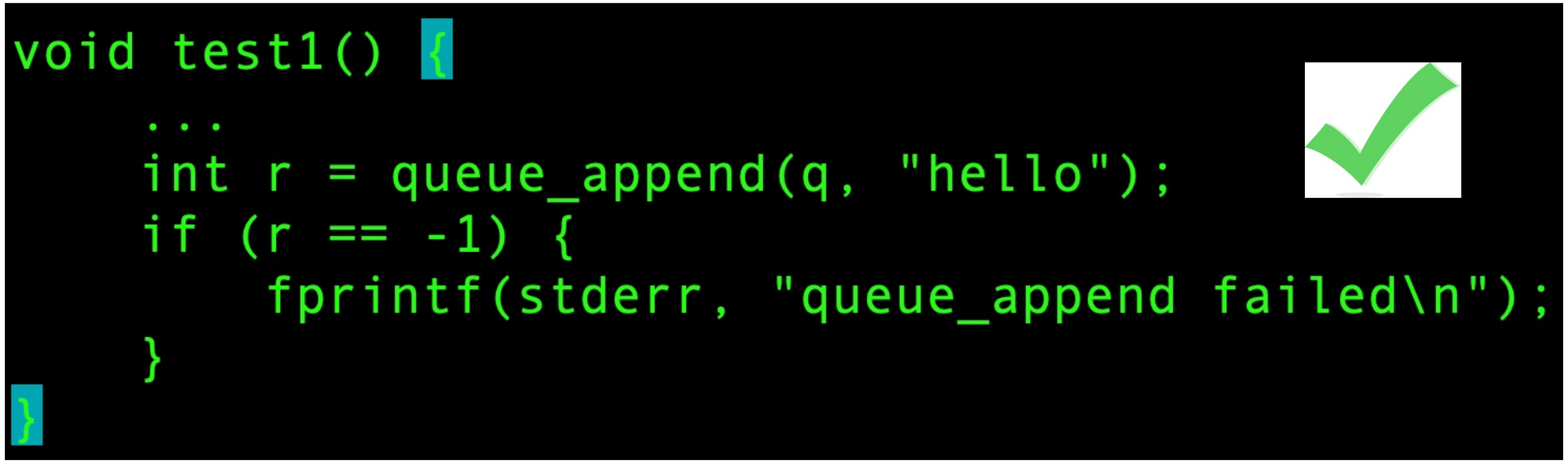
(B)



```
void test1() {
    ...
    queue_append(q, "hello");
}
```

Doesn’t check result

(C)

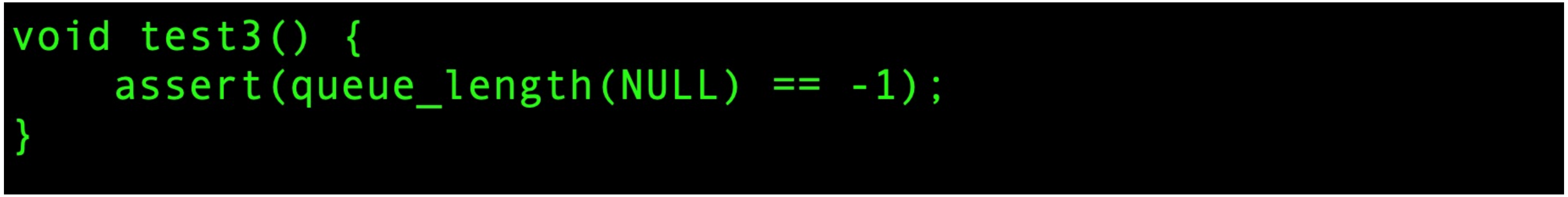


```
void test1() {
    ...
    int r = queue_append(q, "hello");
    if (r == -1) {
        fprintf(stderr, "queue_append failed\n");
    }
}
```

✓

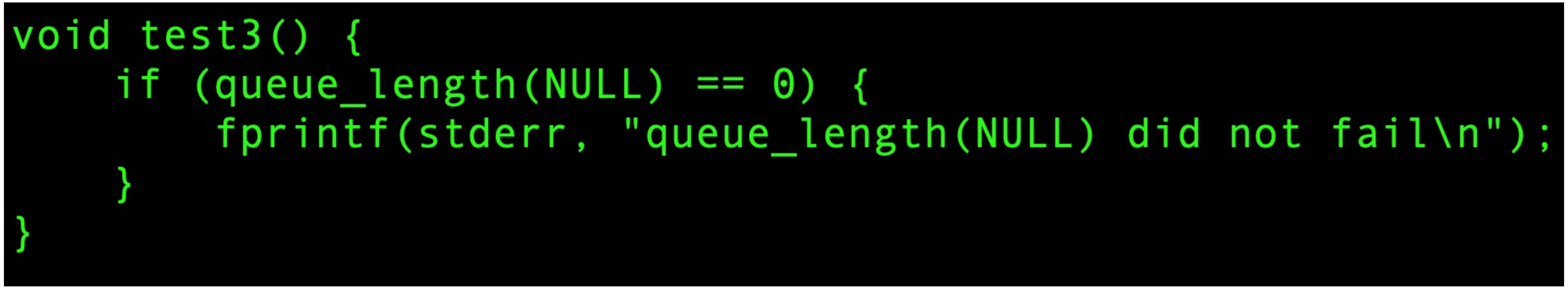
How to deal with failures? Example 6

(A)



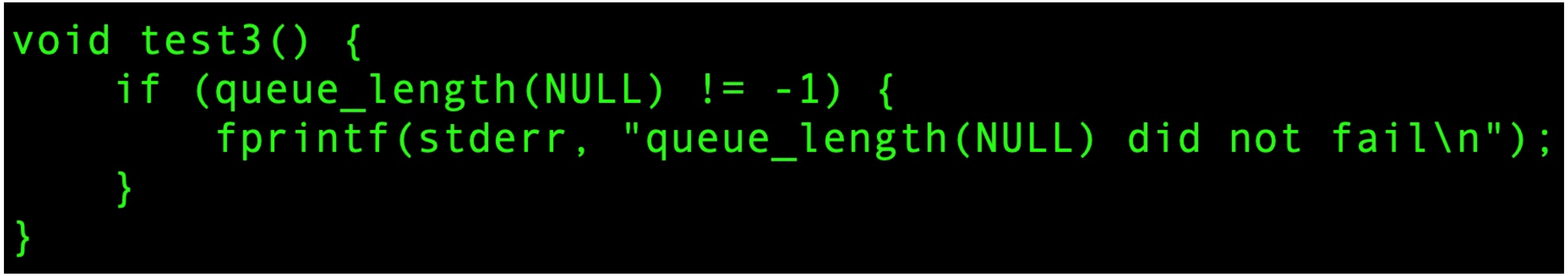
```
void test3() {
    assert(queue_length(NULL) == -1);
}
```

(B)



```
void test3() {
    if (queue_length(NULL) == 0) {
        fprintf(stderr, "queue_length(NULL) did not fail\n");
    }
}
```

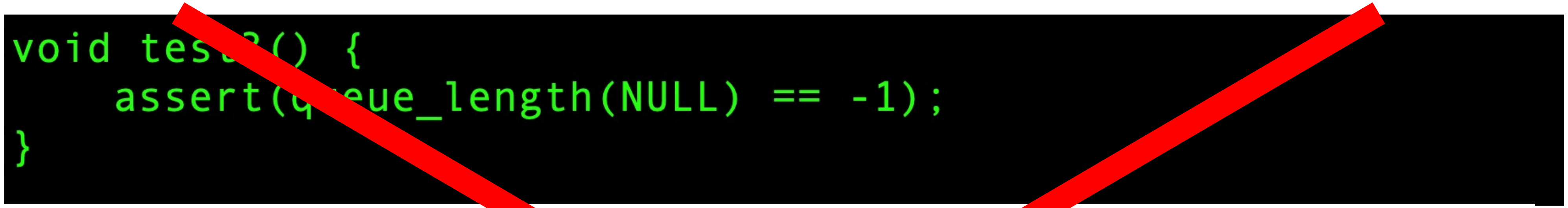
(C)



```
void test3() {
    if (queue_length(NULL) != -1) {
        fprintf(stderr, "queue_length(NULL) did not fail\n");
    }
}
```

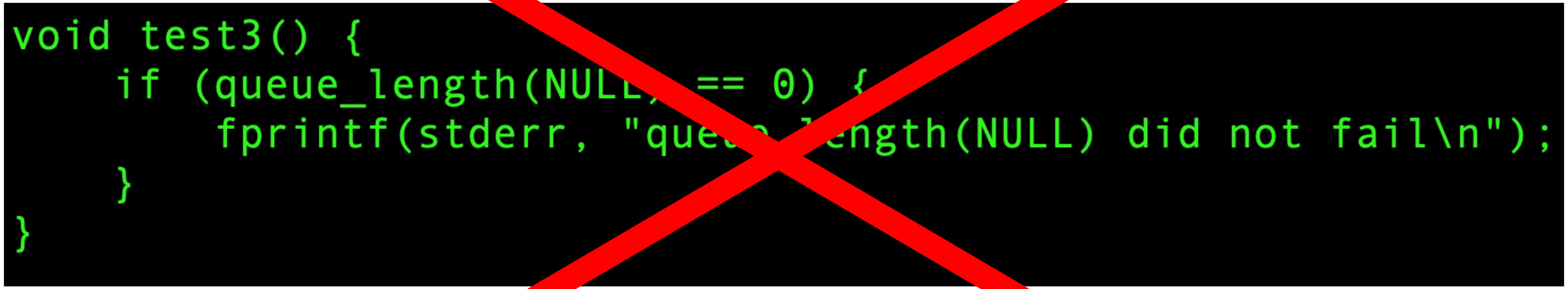
How to deal with failures? Example 6

(A)



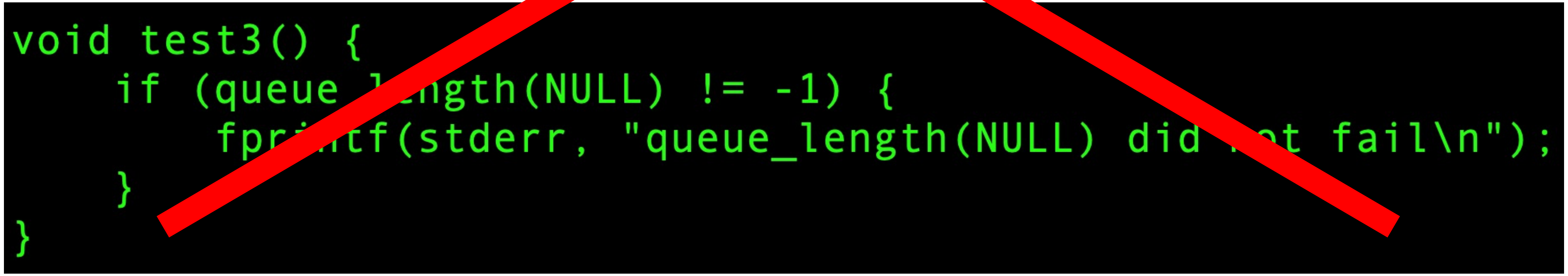
```c
void test3() {
    assert(queue_length(NULL) == -1);
}
```

(B)



```c
void test3() {
    if (queue_length(NULL) == 0) {
        fprintf(stderr, "queue_length(NULL) did not fail\n");
    }
}
```

(C)



```c
void test3() {
    if (queue_length(NULL) != -1) {
        fprintf(stderr, "queue_length(NULL) did not fail\n");
    }
}
```

bug in test program. Not in queue. Queue spec doesn't say what should happen if you pass in a NULL pointer

These rules are general

- This is not about C programming specifically
    - Most programming languages support null references, exceptions, assert statements
    - Same rules apply

Tips (as opposed to rules)

- Use meaningful variable and constant names
- Use comments (especially assert statements)
- Maintain a consistent code layout / format
- Turn **assert A && B** into two assert statements
  - simplifies debugging and reading code
  - but not **assert A || B**
- KISS principle (keep it simple, stupid)