Common JQR / BSLE Mistakes

And how to avoid them!

1LT Tyler Reece
CSD-Tactical
21 FEB 2023

Purpose

Having done many JQR code reviews and several BSLE panels, there are common themes of mistakes that trainees make.

This presentation will discuss several common mistakes to keep an eye out for. It will also provide the reasoning behind why we enforce these standards and explain the potential errors they guard for.

Some of these are very minor or stylistic in nature, and some are more serious.

Overall, avoiding these mistakes will improve your code and make your BSLE submission as clean as possible.

Common Mistakes

- BARR-C-isms
  - Yoda conditionals
  - Conditional parentheses
  - Checking function arguments
  - Variable initialization
  - Descriptive variable names
- Magic numbers
- Macro guards
- Standardized types
- Unsafe operations
- Order of elements in a function
- Multiple exit points
- Assignment inside a conditional check
- Setting to NULL after free()
- Variable Length Arrays (VLAs)
- Checking Return Types
- Comments / Documentation
- Time Management
- Testing
- BSLE Panel Advice

- Parting Thoughts
- Questions & Resources

Yoda Conditionals

Example:

```
// Check divide by zero.
if(op2 == 0)
{
    fprintf(stderr, "division by zero\n");
}
```

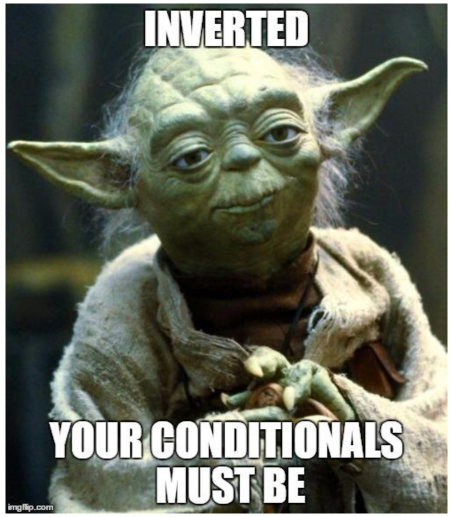
Fix:

```
// Check divide by zero.
if(0 == op2)
{
    fprintf(stderr, "division by zero\n");
}
```

Why do we insist on this?

It prevents the possibility to accidentally do assignment when you meant to check for equality by raising an error at compile time:

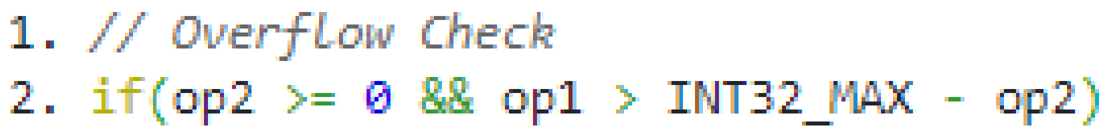
```
if (op2 = 0) // unintended but valid, no error at compile time

if(0 = op2) // error at compile time
```



Constant goes on the left, variable expression on the right.

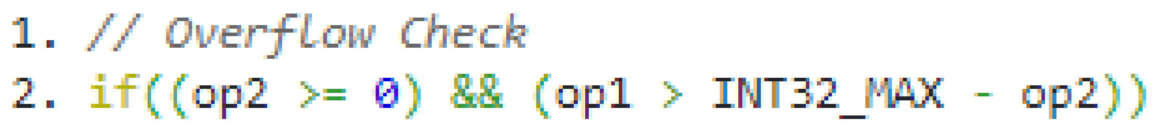
Conditional Parentheses

Example:



```c
// Overflow Check
if(op2 >= 0 && op1 > INT32_MAX - op2)
```

Fix:



```c
// Overflow Check
if((op2 >= 0) && (op1 > INT32_MAX - op2))
```

This improves readability and ensures your conditions are visually separated.

Checking Function Arguments

Example:

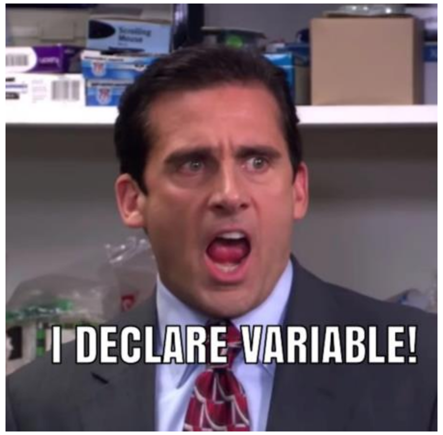
```c
list_node_t *list_peek_tail(list_t *list)
{
    return list->tail;
}
```

Fix:

```c
list_node_t *list_peek_tail(list_t *list)
{
    list_node_t * tail = NULL;

    if (NULL == list || list_emptycheck(list))
    {
        goto end;
    }

    tail = list->tail;

end:
    return tail;
}
```

Checking function arguments ensures passed-in values are valid and guards against attempting to dereference a NULL pointer.

Variable Initialization



Example:

When left uninitialized:

Static variables (file scope and function static) are initialized to zero.

Non-static variables (local variables) are *indeterminate*. Reading them prior to assigning a value results in *undefined behavior*.

Be sure to clean up unused variables as you go.

Fix:

Initialize `ints` to 0, 1, or –1 as appropriate.

```
int idx = 0;
int ret = -1;
```

Initialize pointers to `NULL`.

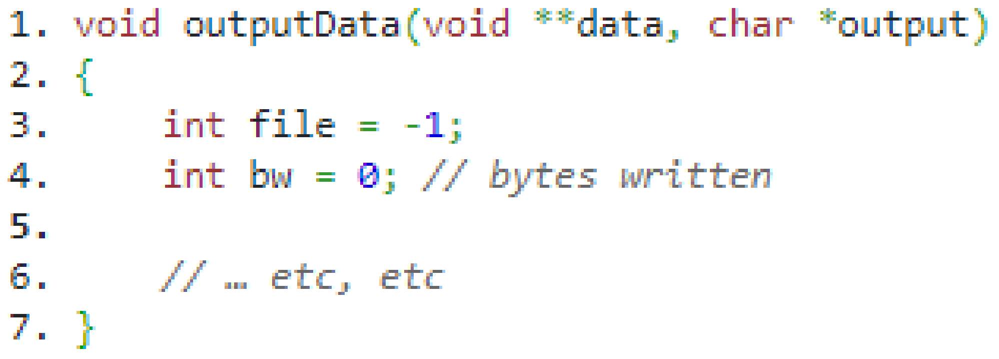
```
char * address = NULL;
proxy_ctx_t * ctx = NULL;
```

Initialize arrays or structs using `{0}`.

```
char buf[1024] = {0};
proxy_ctx_t ctx = {0};
```

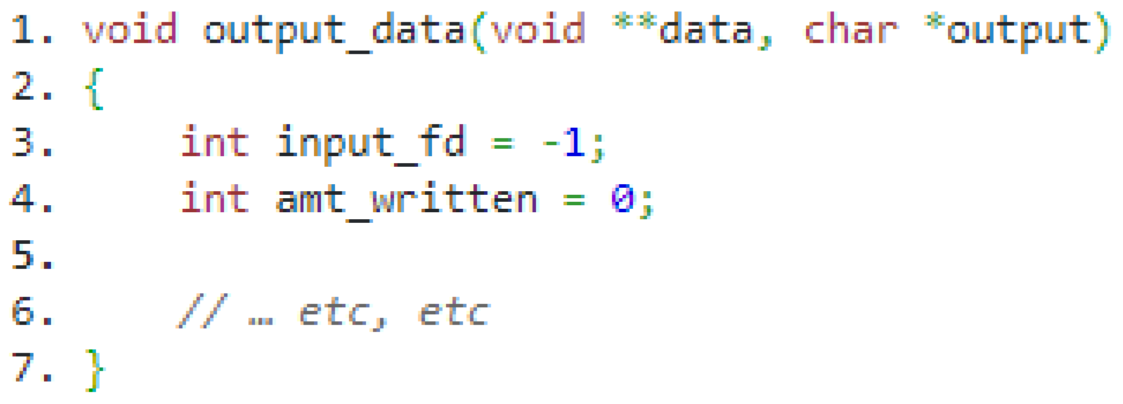
Descriptive Variable Names

Example:



```
void outputData(void **data, char *output)
{
    int file = -1;
    int bw = 0; // bytes written

    // … etc, etc
}
```

Fix:



```
void output_data(void **data, char *output)
{
    int input_fd = -1;
    int amt_written = 0;

    // … etc, etc
}
```

Better variable (and function) names make for more readable, standardized code.

It is better to be a little more verbose and clearer, than try to be overly concise. A few more characters doesn't hurt.

Stay consistent between snake_case and camelCase.

Magic Numbers

Example:

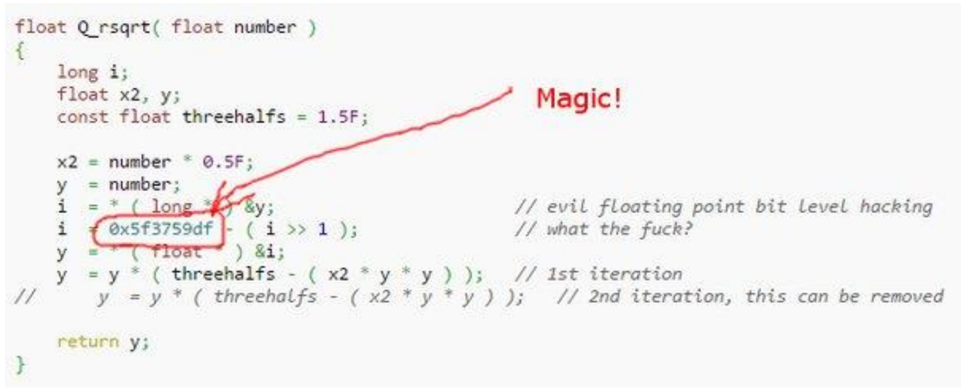
```
switch(operator)
{
    case 0: // +
        printf("%d\n", add( op1, op2));
        break;
    case 1: // -
        printf("%d\n", sub(op1, op2));
        break;
    case 2: // *
        printf("%d\n", mul(op1, op2));
        break;
// etc, etc.
```

Defining magic numbers with a macro (#define) or, if there are several related magic numbers, using an enum, allows for more readable, maintainable, and extendable code.

Fix:

```
// in header file
enum opcodes
{
    ADD,
    SUB,
    MULT,
    // etc, etc
}
```

```
switch(operator)
{
    case ADD:
        printf("%d\n", add( op1, op2));
        break;
    case SUB:
        printf("%d\n", sub(op1, op2));
        break;
    case MULT:
        printf("%d\n", mul(op1, op2));
        break;
```



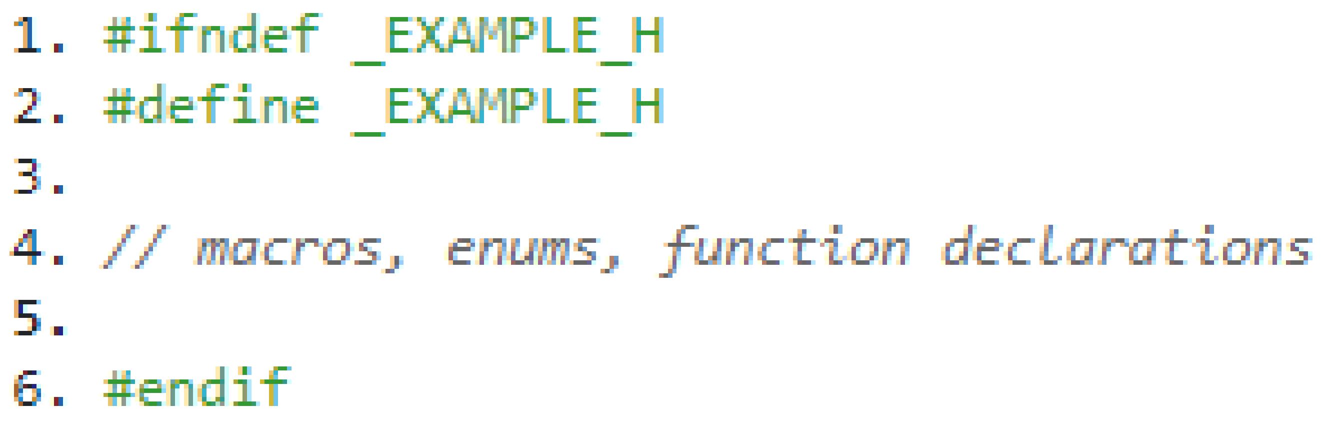
```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;                       // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//      y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration, this can be removed

    return y;
}
```

John Carmack's infamous Quake II square root function

Macro Guards

Macro guards (also called include guards) prevent against double inclusion.

For a file called example.h:



```
#ifndef _EXAMPLE_H
#define _EXAMPLE_H

// macros, enums, function declarations

#endif
```

What does this do exactly?

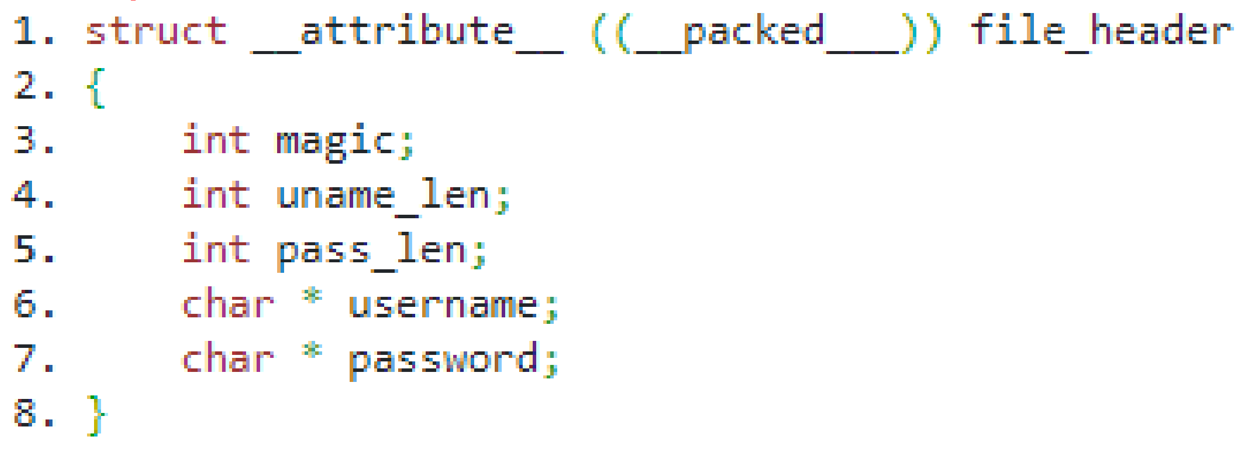
If the preprocessor macro `_EXAMPLE_H` hasn't yet been defined...

...then define it, so that if included later, everything between `#ifndef` and `#endif` (the whole file) won't be included again.

If you define any functions in your header and your header is used by more than one source file, the function will be defined more than once with the same name, leading to a problem when the program is linked.

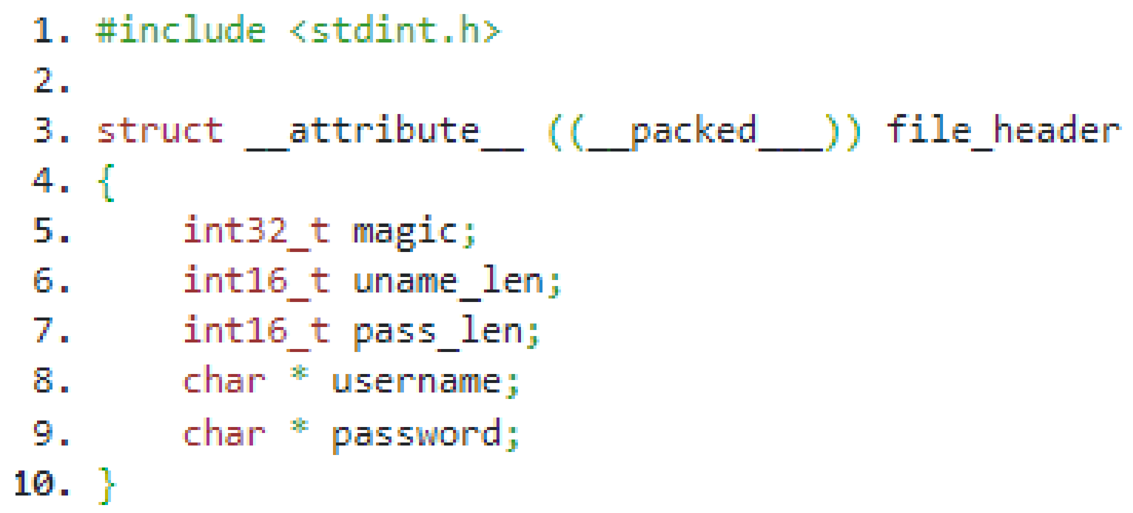
Standardized Types

Example:



```
struct __attribute__ ((__packed__)) file_header
{
    int magic;
    int uname_len;
    int pass_len;
    char * username;
    char * password;
}
```

Fix:

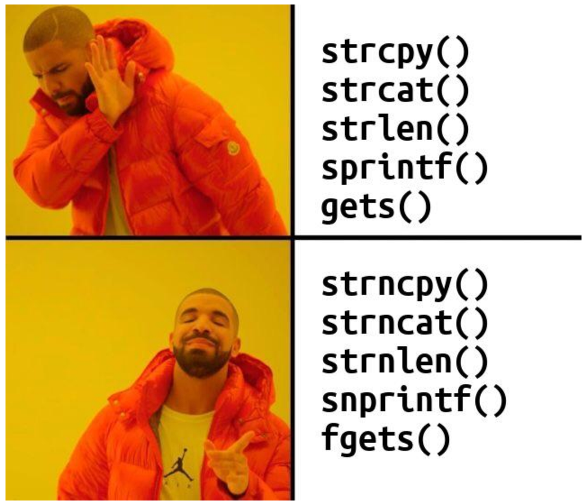


```
#include <stdint.h>

struct __attribute__ ((__packed__)) file_header
{
    int32_t magic;
    int16_t uname_len;
    int16_t pass_len;
    char * username;
    char * password;
}
```

Using well-defined types such as those in `stdint.h` ensures code is easier and safer to port, as you won't get any surprises when, for example, one machine interprets `int` as 16-bit and one as 32-bit.

Unsafe Operations

Unsafe and/or deprecated	Safe(r)
strcat	strncat, strlcat
strcpy	strncpy, strlcpy
strcmp	strncmp
gets	fgets/gets_s
sprintf	snprintf
atoi	strtol
gethostbyname	getaddrinfo



char[]
String
Corporate needs you to find the differences between this picture and this picture.
Ignorant students learning C
They're the same picture.

Order of elements in a function

Nearly every function will follow this order:

1. Declare and initialize **all** variables
2. Check function arguments
3. Allocate memory, if necessary
4. Do function logic
5. Deallocate memory, if necessary
6. Return

This prevents you from forgetting to initialize variables and makes your functions more consistent and readable to others.

Multiple Exit Points

Example:

```c
list_node_t *list_peek_tail(list_t *list)
{
    if (NULL == list || list_emptycheck(list))
    {
        return NULL;
    }
    else
    {
        return list->tail;
    }
}
```

Fix:

```c
list_node_t *list_peek_tail(list_t *list)
{
    list_node_t * tail = NULL;

    if (NULL == list || list_emptycheck(list))
    {
        goto end;
    }

    tail = list->tail;

end:
    return tail;
}
```

A single exit point ensures there is only one path through the function, and you know where to look for the exit.

It also allows you to provide a single place to clean up potential memory leaks before you exit.

Lastly, it may aid in debugging as you can put a breakpoint on the return statement and inspect how the code determines the return value.

Assignment inside a conditional check

Example:

```
if(0 > (file = open(output, O_RDWR | O_CREAT , RW_PERM)))
```

Fix:

```
int open_check = open(output, O_RDWR | O_CREAT, RW_PERM);
if(0 > open_check)
```

This is also seen commonly in a while loop, when using `fgets/fgetc`. It is less error prone to declare a variable and recalculate the variable at the end of the while loop.

Setting to `NULL` after every `free()`

Setting unused pointers to `NULL` is a defensive style, protecting against dangling pointer bugs. If a dangling pointer is accessed after it is freed, you may read or overwrite random memory.

If a null pointer is accessed, you get an immediate crash on most systems, telling you right away what the error is.

To complete the style, you should also initialize pointers to `NULL` before they get assigned a true pointer value.

Variable Length Arrays (VLAs)

Example:

```
char username[username_len + 1];
char password[password_len + 1];
```

Fix:

```
#define UNAME_MAX 64
#define PASS_MAX 64

char username[UNAME_MAX];
char password[PASS_MAX];
```

"It is significantly safer to use fixed-width arrays. with fixed-width arrays, the null terminators will ensure that all str related functions will function properly and will not lead to any compiler mix-ups. If you are worried about wasting bytes, the difference between 3 and 20 bytes of memory is a drop in the bucket compared to functionality and security." -1LT Causey, a.k.a. the GOAT

Does your dog bite?
No, but it can hurt you in other ways.
Array indexing starts at 1

Checking Return Types

The only thing worse than using the wrong return type...

```
int pass_len = strnlen(password, PASS_MAX);

size_t amt_written = write(file_fd, buf, BUF_LEN);
```

`strnlen` returns a `size_t`, write returns an `ssize_t`

...is not checking the return type at all.

```
int file_fd = open(filename, O_RDONLY);

read(file_fd, &buf, FILE_MAX);
```

Not checking return types lead to bug-prone code. These library functions can fail (file doesn't exist, inadequate permissions, etc. etc). Consult the man pages!

Comments / Documentation

Examples:

- Missing file headers
- Missing function documentation
- Too much commenting (focus on *why*, not what)
- Leaving in commented-out code

Junior devs be like:

Rule 1: Comments should not duplicate the code.
Rule 2: Good comments do not excuse unclear code.
Rule 3: If you can't write a clear comment, there may be a problem with the code.
Rule 4: Comments should dispel confusion, not cause it.
Rule 5: Explain unidiomatic code in comments.
Rule 6: Provide links to the original source of copied code.
Rule 7: Include links to external references where they will be most helpful.
Rule 8: Add comments when fixing bugs.
Rule 9: Use comments to mark incomplete implementations.

"Programs must be written for people to read and, only incidentally, for machines to execute." - Hal Abelson

Time Management

The Training NCO will have a time-blocked plan for you to complete JQR line items and a projected end date for your BSLE. Stick to the plan!

Commit code often. This allows us oversight into how you are progressing and prevent you from losing progress or getting stuck.

Ask for code reviews early, often, and from multiple developers. You don't necessarily need a fully completed project to get feedback.

Code reviews can take a few days based on operational tempo at the time. Have a plan for what you will work on in the meantime.

Give yourself enough time to (1) write the code, (2) polish/refactor the code, (3) test the code yourself and (4) get code review and incorporate feedback.

When you are gonna merge two branches after long time:

Never let your computer know that you are in a hurry.
Computers can smell fear. They slow down if they know that you are running out of time.

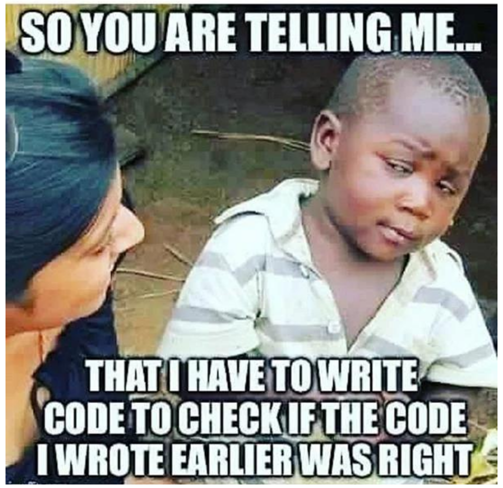
Testing

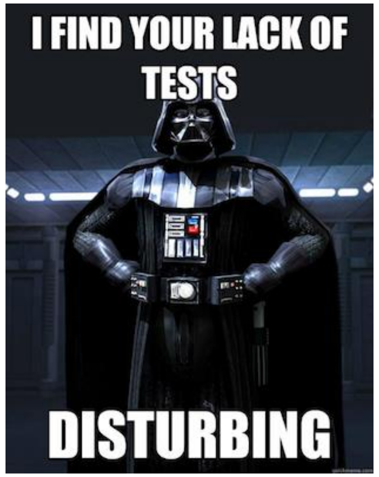
Unit Testing – testing individual methods and functions used by the software. Can easily be integrated into CI/CD pipeline.

Functional Testing – ensure that the software meets product requirements (works correctly).

Integration Testing – ensure that the various modules/services of the application work well together (i.e. client/server interaction).

Use the provided testing scripts for the JQR, but also know that you will be expected to implement a testing plan on the BSLE, so writing additional tests throughout the JQR can be good practice.


SO YOU ARE TELLING ME...
THAT I HAVE TO WRITE CODE TO CHECK IF THE CODE I WROTE EARLIER WAS RIGHT



Advice for BSLE Panel

Your BSLE panel is a formal review of your BSLE code submission. At least 1 senior developer will lead the panel, and at least 2 basic developers will also vote. There may be 1-2 other basic developers acting as non-voting members.

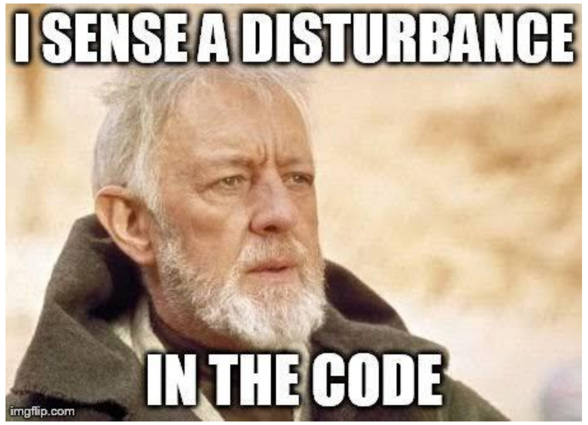
Panelists tend to focus on different areas of your code (Python, concurrency, networking, BARR-C, etc).

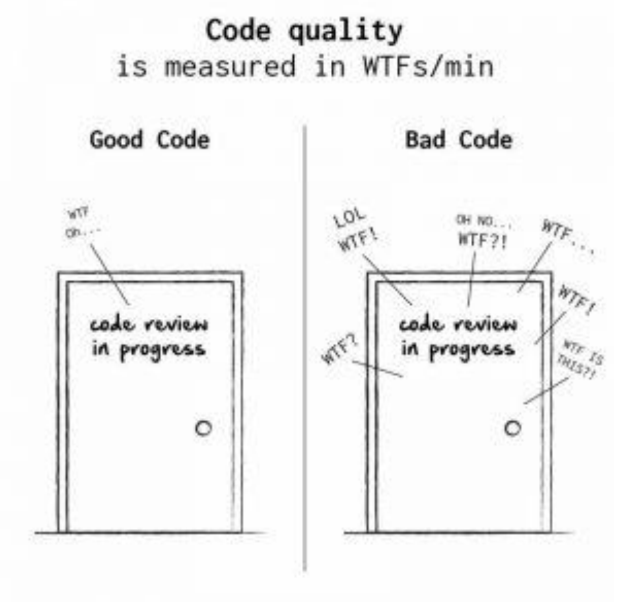
Avoid "code smells" - if something looks a little whacky, it will draw attention and cause further investigation.

Be prepared to justify your overall design and specific implementation decisions.

If you use C keywords, be prepared to explain your conceptual understanding of the implementation as well (atomics, volatile, static, etc).

Your repo will open 1 hour prior to panel start – use this time to review comments and prepare your defense.





Parting Thoughts

Many of these mistakes can be found by incorporating static analysis tools into your build process (i.e. adding clang-tidy checks to CMake, using AddressSanitizer and/or Valgrind).

However, there is no substitute for a thorough self-code review:

- Scrutinize every function you write, especially after taking a break and coming back
- Ask yourself if a function can be refactored into smaller components (< 100 lines)
- Remember common programming principles (DRY, KISS)
- Spend ample time designing and writing pseudocode
- Get code review from other developers early & often

Following CSD-T coding standards are there to ensure your code is:

- Free of bugs
- Easily read and understood
- Maintainable by other developers

"Build it, build it better, build it fast." Functionality, Simplicity, Speed

Resources

Best practices for writing code comments: https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/

The danger of VLAs: https://www.reidatcheson.com/c/programming/2015/12/07/vlas.html

Dangerous functions in C: https://dwheeler.com/secure-programs/Secure-Programs-HOWTO/dangers-c.html

Macro guards: https://stackoverflow.com/questions/27810115/what-exactly-do-c-include-guards-do

Uninitialized variables and undefined behavior: https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/

Setting to NULL after free(): https://stackoverflow.com/questions/1025589/setting-variable-to-null-after-free