eNoah Coding Standards - C Language

Table of Contents

---

1. Naming Conventions

1.1 File Names

Use lowercase with underscores for file names.

```c
// Good

file_operations.c

database_manager.h

network_utils.c

// Bad

FileOperations.c    // PascalCase

fileoperations.c   // no underscores

FILE_OPS.C         // uppercase
```

1.2 Function Names

Use snake_case for function names.

```c
// Goodvoid calculate_user_score(void);int validate_input_data
(const char* data);

FILE* open_log_file(const char* filename);

// Badvoid CalculateUserScore(void);   // PascalCasevoid calcu
lateUserScore(void);   // camelCasevoid CALCULATE_USER_SCORE(v
oid); // uppercase
```

1.3 Variable Names

Use snake_case for variables. Prefix pointers with 'p', arrays with 'arr'.

c

```
// Goodint user_count;char* p_filename;int arr_user_ids[MAX_USERS];float temperature_celsius;
```

```
// Badint userCount;           // camelCasechar* Filename;
    // PascalCaseint user_ids_array[];    // redundant
```

1.4 Constant Names

Use UPPER_SNAKE_CASE for constants and macros.

c

```
// Good#define MAX_BUFFER_SIZE 1024#define DEFAULT_TIMEOUT 30const int MAX_USERS = 100;
```

```
// Bad#define maxBufferSize 1024     // camelCase#define DefaultTimeout 30      // PascalCaseconst int max_users = 100;     // snake_case
```

1.5 Type Definitions

Use `typedef` with `_t` suffix for custom types.

c

```
// Goodtypedef unsigned int user_id_t;typedef struct node_s node_t;typedef enum status_e status_t;
```

```
// Badtypedef unsigned int USER_ID;  // uppercasetypedef struct node Node;       // no suffix
```

1.6 Macro Names

Use UPPER_SNAKE_CASE for macros. Parenthesize macro parameters.

c

```
// Good#define MIN(x, y) ((x) < (y) ? (x) : (y))#define ARRAY_SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))


// Bad#define min(x, y) x < y ? x : y         // no parentheses
#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])  // no spacing
```

2. Code Organization

2.1 Header Files (.h)

Header files should contain declarations only.

c

```
// database_manager.h#ifndef DATABASE_MANAGER_H#define DATABASE_MANAGER_H


#include <stdint.h>#include <stdbool.h>


#define DB_MAX_CONNECTIONS 10#define DB_TIMEOUT_MS 5000


typedef struct database_s database_t;


typedef enum {


    DB_SUCCESS = 0,


    DB_ERROR_CONNECTION,


    DB_ERROR_TIMEOUT} db_status_t;


db_status_t database_connect(const char* connection_string);db_status_t database_execute_query(database_t* db, const char* query);void database_disconnect(database_t* db);


#endif /* DATABASE_MANAGER_H */
```

2.2 Source Files (.c)

Source files should contain implementations.

c

```
// database_manager.c#include "database_manager.h"#include <st
dlib.h>#include <string.h>

struct database_s {

    char connection_string[256];

    bool is_connected;

    uint32_t timeout_ms;};

db_status_t database_connect(const char* connection_string){

    if (connection_string == NULL || strlen(connection_string)
 == 0) {

        return DB_ERROR_CONNECTION;

    }



    // Implementation

    return DB_SUCCESS;}
```

2.3 Include Guards

Use `#ifndef` guards with file-specific names.

c

```
// Good#ifndef NETWORK_UTILS_H#define NETWORK_UTILS_H// conten
t#endif /* NETWORK_UTILS_H */
```

```
// Bad#pragma once  // Not all compilers support this#ifndef U
TILS_H // Too generic
```

2.4 Include Order

Group includes in logical order.

c

```
// System headers first#include <stdio.h>#include <stdlib.h>#i
nclude <string.h>

// Third-party headers#include <openssl/ssl.h>

// Project headers#include "database_manager.h"#include "netwo
rk_utils.h"
```

3. Formatting Standards

3.1 Braces Placement

Use K&R style braces.

c

```
// Goodint calculate_sum(int a, int b){

    if (a > 0 && b > 0) {

        return a + b;

    } else {

        return 0;
    }}

// Bad - Allman styleint calculate_sum(int a, int b){

    if (a > 0 && b > 0)

    {

        return a + b;

    }

    else

    {

        return 0;

    }}
```

3.2 Indentation

Use 4 spaces for indentation (no tabs).

c

```
// Goodvoid process_data(const char* data, size_t length){

    if (data != NULL && length > 0) {

        for (size_t i = 0; i < length; i++) {

            if (is_valid_character(data[i])) {

                handle_character(data[i]);

            }
        }

    }}

// Bad - mixed tabs and spacesvoid process_data(const char* da
ta, size_t length){

  if (data != NULL) {  // 2 spaces

    for (int i = 0; i < length; i++) {  // tab

        // inconsistent

    }

  }}
```

3.3 Line Length

Maximum 80 characters per line.

c

```
// Good - break long lines logicallyif (user_is_authenticated
&& user_has_permission(PERM_READ) &&

    data_is_available() && !system_is_locked()) {

    display_content();}

// Bad - too longif (user_is_authenticated && user_has_permiss
ion(PERM_READ) && data_is_available() && !system_is_locked())
{

    display_content();}
```

3.4 Spacing

Use consistent spacing around operators.

c

```c
// Goodint result = (a * b) + (c / d);if (count > MAX_ITEMS ||
 flag == true) {

    process_items();}

// Badint result=(a*b)+(c/d);if (count>MAX_ITEMS||flag==true){

    process_items();}
```

4. Commenting Standards

4.1 File Headers

Include descriptive file headers.

c

```c
/**

 * @file network_utils.c

 * @brief Network utility functions for TCP/IP communication

 *

 * This module provides utility functions for network operatio
ns including

 * socket creation, connection management, and data transmissi
on.

 *

 * @author Developer Name
 * @date 2024-01-15

 * @version 1.0

 *

 * Updated By: Developer Name

 * Updated Date: 2024-02-20

 * Description: Added IPv6 support, fixed memory leak in socke
t creation

 */

#include "network_utils.h"
```

4.2 Function Comments

Use Doxygen-style function comments.

c

```
/**

 * @brief Establishes a TCP connection to the specified host a
nd port

 *

 * This function creates a TCP socket and attempts to connect
to the

 * specified hostname and port. It handles DNS resolution and
timeout.

 *

 * @param hostname The target hostname or IP address
 * @param port The target port number

 * @param timeout_ms Connection timeout in milliseconds

 *

 * @return socket_fd_t File descriptor of the connected socket
 on success,

 *         INVALID_SOCKET on failure

 *

 * @note The caller is responsible for closing the socket usin
g close_socket()

 */socket_fd_t tcp_connect(const char* hostname, uint16_t port,
 uint32_t timeout_ms){

    // Implementation}
```

4.3 Inline Comments

Comment complex logic, but avoid stating the obvious.

c

```
// Good// Calculate CRC32 checksum for data validationuint32_t
 crc = calculate_crc32(buffer, length);if (crc != expected_crc)
 {

    // Data corruption detected, request retransmission

    request_retransmission();}

// Badint x = 5; // Set x to 5

x++; // Increment x by 1
```

5. Data Types and Declarations

5.1 Variable Declarations

Declare variables at the beginning of blocks.

c

```
// Goodvoid process_user_data(user_t* user){

    int error_code = 0;

    char buffer[MAX_BUFFER_SIZE];

    size_t bytes_read = 0;



    // Rest of function}

// Bad - mixed declarations and codevoid process_user_data(use
r_t* user){

    int error_code = 0;

    process_something();

    char buffer[MAX_BUFFER_SIZE];  // Declaration after code}
```

5.2 Type Definitions

Use meaningful type names with `_t` suffix.

c

```
// Goodtypedef uint32_t user_id_t;typedef int64_t timestamp_t;
typedef enum {
    STATE_IDLE,

    STATE_ACTIVE,

    STATE_ERROR} system_state_t;

// Badtypedef uint32_t U32;  // Not descriptivetypedef int ID;
        // Too generic
```

5.3 Structure Definitions

Use forward declarations when needed.

c

```
// Good - forward declarationtypedef struct list_node_s list_n
ode_t;

struct list_node_s {

    void* data;

    list_node_t* next;

    list_node_t* prev;};

// Self-referential structuretypedef struct tree_node_s {

    int value;

    struct tree_node_s* left;

    struct tree_node_s* right;} tree_node_t;
```

6. Function Standards

6.1 Function Length

Keep functions focused and under 50 lines.

c

```c
// Good - focused functiondb_status_t validate_connection_para
ms(const db_config_t* config){

    if (config == NULL) {

        return DB_ERROR_INVALID_PARAM;

    }



    if (strlen(config->hostname) == 0) {

        return DB_ERROR_INVALID_HOST;

    }



    if (config->port == 0 || config->port > 65535) {

        return DB_ERROR_INVALID_PORT;

    }



    return DB_SUCCESS;}

// Bad - function does too muchdb_status_t setup_database_conn
ection(void){
    // 100+ lines of mixed responsibilities}
```

6.2 Parameter Passing

Pass large structures by pointer, small types by value.

c

```
// Good - large structure by pointervoid update_user_profile(user_profile_t* profile, const update_data_t* update){

    // Modify profile through pointer}

// Good - small types by valueint calculate_sum(int a, int b){

    return a + b;}

// Bad - large structure by valuevoid process_large_data(large_struct_t data)  // Inefficient{

    // Makes copy of entire structure}
```

6.3 Return Values

Use consistent return codes.

c

```
typedef enum {

    SUCCESS = 0,

    ERROR_INVALID_PARAM = -1,

    ERROR_MEMORY_ALLOC = -2,

    ERROR_IO_OPERATION = -3,
    ERROR_NETWORK_TIMEOUT = -4} result_code_t;

result_code_t initialize_system(system_config_t* config){

    if (config == NULL) {

        return ERROR_INVALID_PARAM;

    }



    // Initialization logic



    return SUCCESS;}
```

7. Preprocessor Directives

7.1 Macro Definitions

Parenthesize macro parameters and expressions.

c

```
// Good#define MAX(a, b) ((a) > (b) ? (a) : (b))#define ARRAY_
SIZE(arr) (sizeof(arr) / sizeof((arr)[0]))#define IS_VALID_POI
NTER(ptr) ((ptr) != NULL)

// Bad - missing parentheses#define MAX(a, b) a > b ? a : b#de
fine SQUARE(x) x * x  // SQUARE(2+3) becomes 2+3*2+3 = 11, not
 25
```

7.2 Conditional Compilation

Use `#if` for feature flags, avoid complex nested conditions.

c

```
// Good#ifdef DEBUG

    #define LOG_DEBUG(msg) printf("DEBUG: %s\n", msg)#else

    #define LOG_DEBUG(msg) // Nothing in release#endif

#if VERSION_MAJOR > 2

    #define USE_NEW_API 1#else

    #define USE_NEW_API 0#endif

// Bad - complex nested conditions#ifdef LINUX

    #ifdef DEBUG

        #if VERSION > 2

            // Hard to follow

        #endif

    #endif#endif
```

8. Error Handling

8.1 Return Code Standards

Use consistent error reporting.

c

```
typedef enum {
```

```c
    FILE_OPERATION_SUCCESS = 0,

    FILE_ERROR_NOT_FOUND = -1,

    FILE_ERROR_PERMISSION_DENIED = -2,

    FILE_ERROR_IO = -3,

    FILE_ERROR_INVALID_FORMAT = -4} file_status_t;

file_status_t read_config_file(const char* filename, config_t*
 config){

    if (filename == NULL || config == NULL) {

        return FILE_ERROR_INVALID_PARAM;

    }



    FILE* file = fopen(filename, "r");

    if (file == NULL) {

        if (errno == ENOENT) {

            return FILE_ERROR_NOT_FOUND;

        } else if (errno == EACCES) {

            return FILE_ERROR_PERMISSION_DENIED;

        } else {

            return FILE_ERROR_IO;
        }

    }

    // Read file contents

    fclose(file);

    return FILE_OPERATION_SUCCESS;}
```

8.2 Error Reporting

Use consistent error logging.

c

```
#define LOG_ERROR(format, ...) \

    fprintf(stderr, "ERROR [%s:%d]: " format "\n", \

            __FILE__, __LINE__, ##__VA_ARGS__)

#define LOG_WARNING(format, ...) \

    fprintf(stderr, "WARNING [%s:%d]: " format "\n", \

            __FILE__, __LINE__, ##__VA_ARGS__)

result_code_t safe_memory_copy(void* dest, const void* src, si
ze_t size){

    if (dest == NULL || src == NULL) {
        LOG_ERROR("Invalid parameters for memory copy");

        return ERROR_INVALID_PARAM;

    }



    if (size == 0) {

        LOG_WARNING("Zero-size memory copy requested");

        return SUCCESS; // Nothing to do

    }



    memcpy(dest, src, size);

    return SUCCESS;}
```

9. Memory Management

9.1 Dynamic Allocation

Always check allocation results, pair alloc/free calls.

c

```
// Goodint* create_int_array(size_t size){

    if (size == 0) {

        return NULL;
    }



    int* array = (int*)malloc(size * sizeof(int));

    if (array == NULL) {

        LOG_ERROR("Failed to allocate memory for %zu integers",
 size);

        return NULL;

    }



    // Initialize array

    memset(array, 0, size * sizeof(int));

    return array;}

void cleanup_int_array(int** array){

    if (array != NULL && *array != NULL) {

        free(*array);

        *array = NULL; // Prevent dangling pointer

    }}
```

9.2 Pointer Usage

Use `const` appropriately, avoid pointer arithmetic when possible.

```c
// Good - const correctnesssize_t string_length(const char* st
r)  // Input won't be modified{

    if (str == NULL) {

        return 0;

    }



    const char* p = str;  // Pointer to const data

    while (*p != '\0') {

        p++;

    }



    return p - str;}

// Bad - unnecessary pointer arithmeticvoid dangerous_pointer_
use(void){

    int array[10];

    int* p = array;



    // Hard to read and error-prone

    *(p + 5) = 100;  // Instead use array[5] = 100;}
```

10. Security Guidelines

10.1 Buffer Management

Always check buffer bounds.

c

```c
// Good - safe string copyresult_code_t safe_string_copy(char*
 dest, size_t dest_size, const char* src){

    if (dest == NULL || src == NULL || dest_size == 0) {

        return ERROR_INVALID_PARAM;

    }



    size_t src_len = strlen(src);

    if (src_len >= dest_size) {

        // Truncate but ensure null termination

        strncpy(dest, src, dest_size - 1);

        dest[dest_size - 1] = '\0';

        return ERROR_TRUNCATED;

    }



    strcpy(dest, src);
    return SUCCESS;}

// Bad - unsafe buffer operationsvoid unsafe_buffer_operation
(char* input){

    char buffer[64];

    strcpy(buffer, input);  // Potential buffer overflow}
```

10.2 Input Validation

Validate all external inputs.

c

```
result_code_t validate_user_input(const user_input_t* input){

    if (input == NULL) {

        return ERROR_INVALID_PARAM;

    }



    // Validate string inputs

    if (input->username != NULL) {

        size_t username_len = strlen(input->username);

        if (username_len < MIN_USERNAME_LENGTH ||

            username_len > MAX_USERNAME_LENGTH) {

            return ERROR_INVALID_USERNAME;
        }

        // Check for valid characters
        if (!contains_only_alphanumeric(input->username)) {
            return ERROR_INVALID_CHARACTERS;
        }
    }

    // Validate numerical ranges
    if (input->age < MIN_AGE || input->age > MAX_AGE) {
        return ERROR_INVALID_AGE;
    }

    return SUCCESS;}
```

12. Compiler Flags and Warnings

Recommended compiler flags for GCC/Clang:

```makefile
CFLAGS = -Wall -Wextra -Wpedantic -Werror \
         -Wmissing-prototypes -Wstrict-prototypes \
         -Wold-style-definition -Wmissing-declarations \

         -Wredundant-decls -Wnested-externs \

         -Wpointer-arith -Wcast-qual -Wcast-align \

         -Wwrite-strings -Wswitch-default \

         -Wunreachable-code -Wundef -Wshadow \

         -O2 -g -std=c99
```

These C language coding standards ensure consistent, secure, and maintainable code across all C projects at eNoah, following the same professional structure as your PHP and Python standards documents.