MONTANA
TECHNOLOGICAL UNIVERSITY

CSCI 112

Programming with C

Lecture 15

Fall 2025

Dr. Jakub L. Pach

Outline

- Review
- Unions
- Enums
- Bit-fields
- File Input / Output

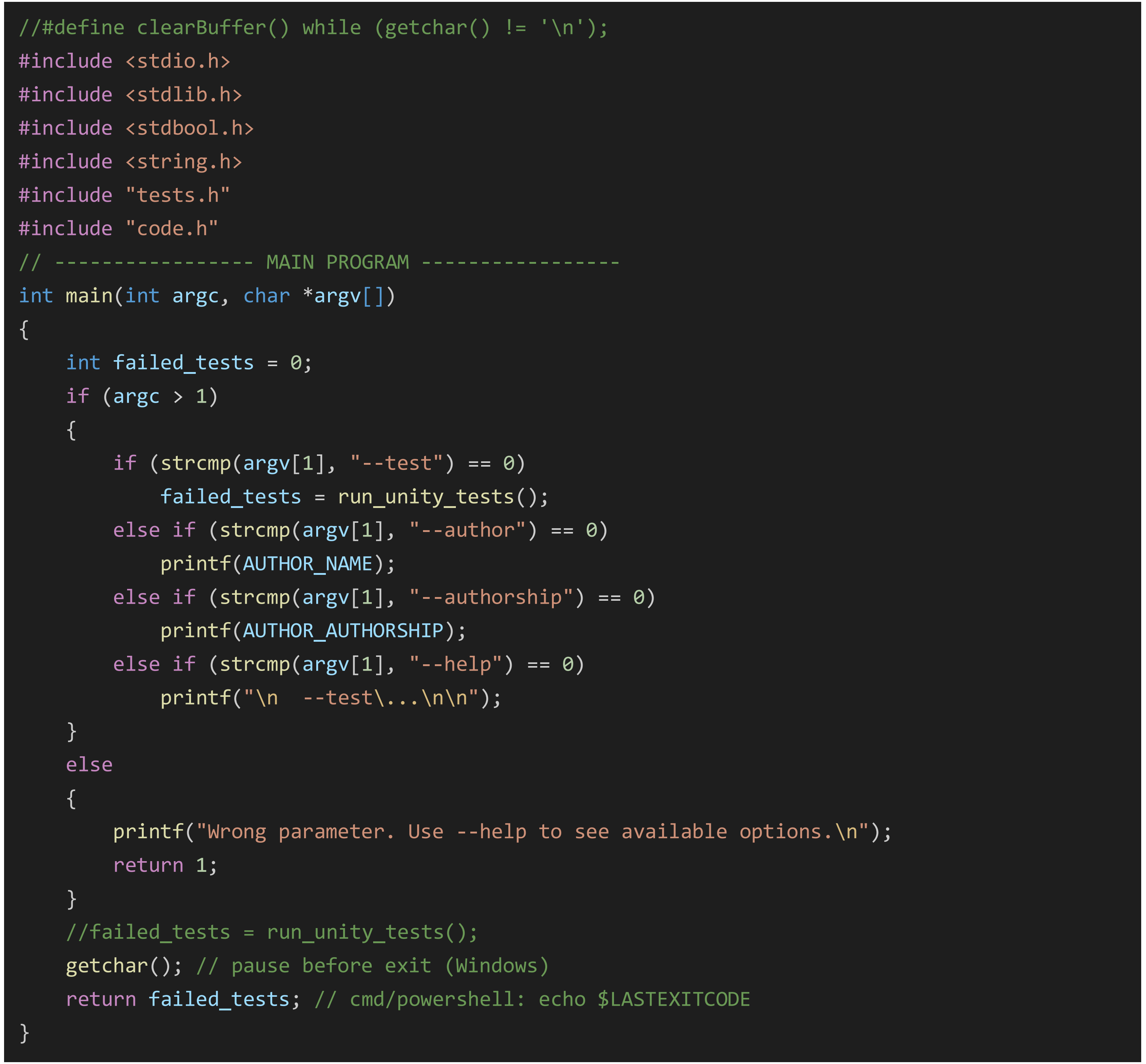
Review

Project Structure - Modular C Program with Unit Tests

- Separate logic, tests, and headers for clarity.
- Unity is a lightweight, header-based testing framework — ideal for C projects.
- This structure allows easy maintenance and clear organization.

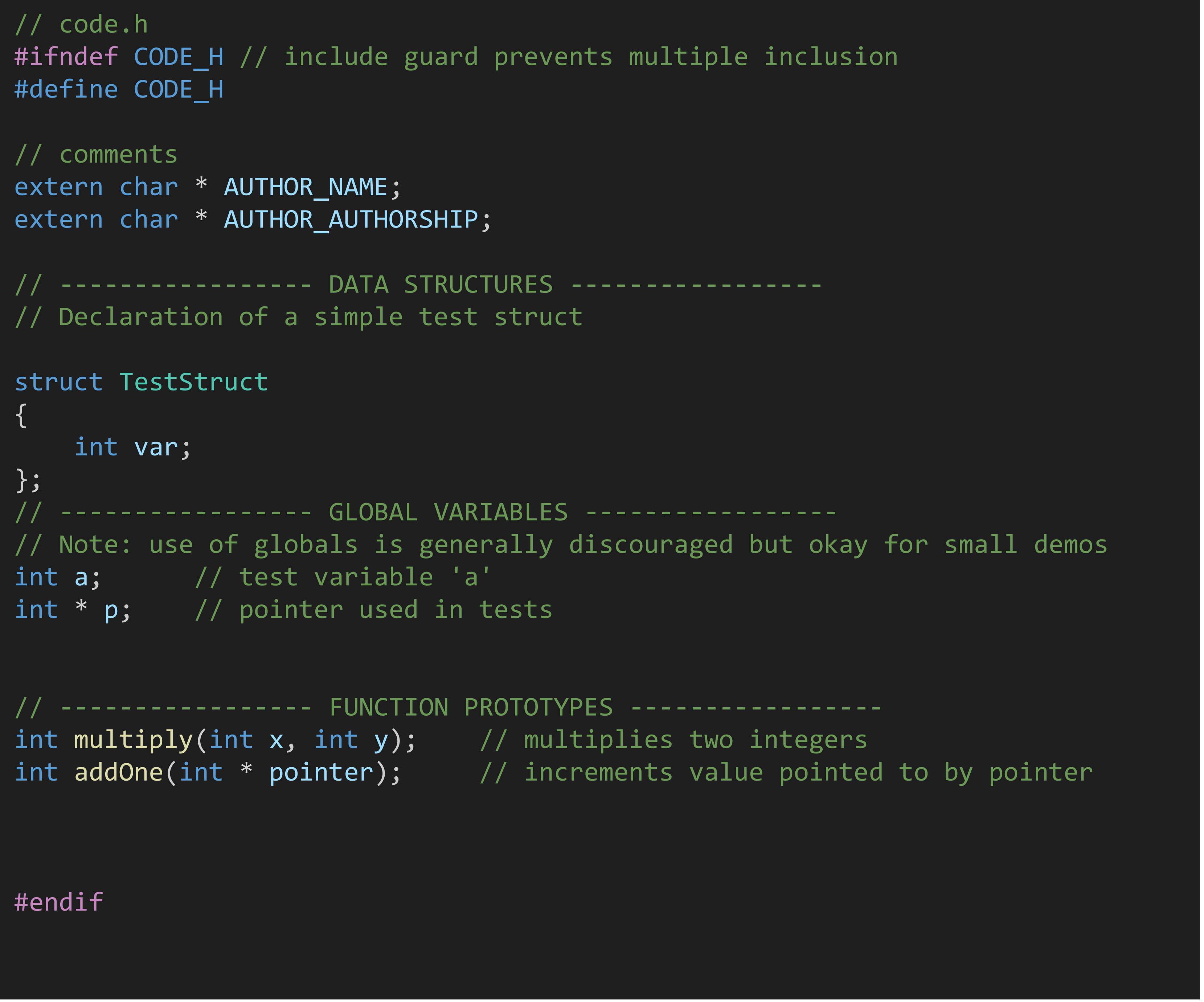
```
C project/
│
├── code.c               ← function implementations
├── code.h               ← function declarations and globals
│
├── main.c               ← main program entry point
│
├── tests.c              ← unit tests implementation
├── tests.h              ← declaration for test runner
│
├── unity.c / unity.h    ← Unity testing framework
│
└── Makefile             ← build automation
```

main.c



```c
//#define clearBuffer() while (getchar() != '\n');
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "tests.h"
#include "code.h"
// ----------------- MAIN PROGRAM -----------------
int main(int argc, char *argv[])
{
    int failed_tests = 0;
    if (argc > 1)
    {
        if (strcmp(argv[1], "--test") == 0)
            failed_tests = run_unity_tests();
        else if (strcmp(argv[1], "--author") == 0)
            printf(AUTHOR_NAME);
        else if (strcmp(argv[1], "--authorship") == 0)
            printf(AUTHOR_AUTHORSHIP);
        else if (strcmp(argv[1], "--help") == 0)
            printf("\n  --test\...\n\n");
    }
    else
    {
        printf("Wrong parameter. Use --help to see available options.\n");
        return 1;
    }
    //failed_tests = run_unity_tests();
    getchar(); // pause before exit (Windows)
    return failed_tests; // cmd/powershell: echo $LASTEXITCODE
}
```

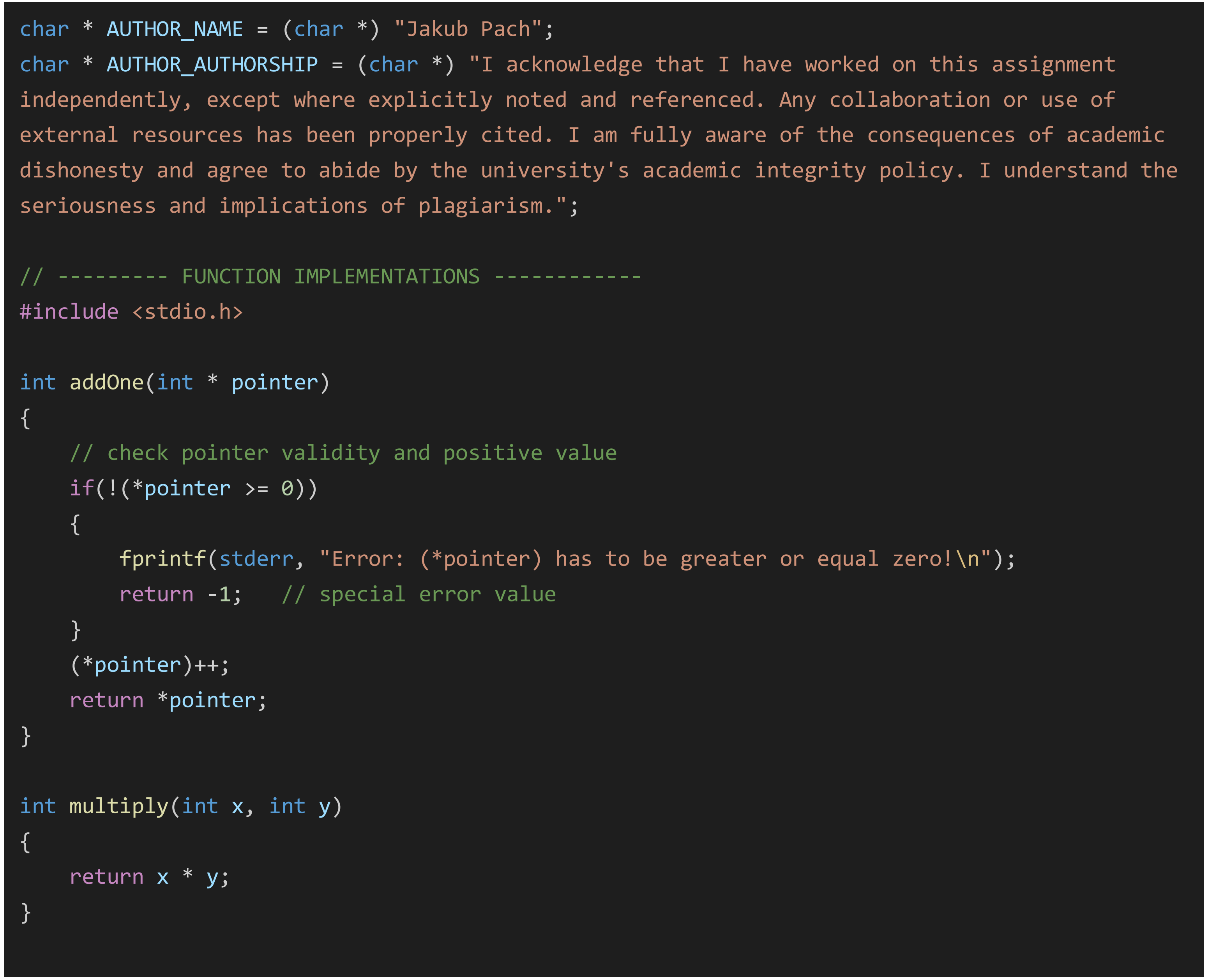
MONTANA
TECHNOLOGICAL UNIVERSITY



```c
// code.h
#ifndef CODE_H // include guard prevents multiple inclusion
#define CODE_H

// comments
extern char * AUTHOR_NAME;
extern char * AUTHOR_AUTHORSHIP;

// ----------------- DATA STRUCTURES -----------------
// Declaration of a simple test struct

struct TestStruct
{
    int var;
};
// ----------------- GLOBAL VARIABLES -----------------
// Note: use of globals is generally discouraged but okay for small demos
int a;      // test variable 'a'
int * p;    // pointer used in tests


// ----------------- FUNCTION PROTOTYPES -----------------
int multiply(int x, int y);    // multiplies two integers
int addOne(int * pointer);     // increments value pointed to by pointer



#endif
```

code.h

code.c



```c
char * AUTHOR_NAME = (char *) "Jakub Pach";
char * AUTHOR_AUTHORSHIP = (char *) "I acknowledge that I have worked on this assignment
independently, except where explicitly noted and referenced. Any collaboration or use of
external resources has been properly cited. I am fully aware of the consequences of academic
dishonesty and agree to abide by the university's academic integrity policy. I understand the
seriousness and implications of plagiarism.";

// --------- FUNCTION IMPLEMENTATIONS ------------
#include <stdio.h>

int addOne(int * pointer)
{
    // check pointer validity and positive value
    if(!(*pointer >= 0))
    {
        fprintf(stderr, "Error: (*pointer) has to be greater or equal zero!\n");
        return -1;   // special error value
    }
    (*pointer)++;
    return *pointer;
}

int multiply(int x, int y)
{
    return x * y;
}
```

tests.h

```c
// tests.h
#ifndef TESTS_H
#define TESTS_H



// prototype for running all Unity tests
int run_unity_tests(void);



#endif // TESTS_H
```

tests.c

```c
//#include <assert.h>    // commented out, not needed
#include "code.h"
#include "tests.h"
#include "unity.h"
int result; // global variable for test results
// ----------------- UNITY SETUP / TEARDOWN -----------------
void setUp()
{
    a = 4;
    p = (int*) malloc(sizeof(int));
    *p = 5;
    result = 0;
}
void tearDown()
{
    free(p);
    result = 0;
}
// ----------------- TEST FUNCTIONS -----------------
void test_multiply_basic()
{
    result = multiply(a, *p);
    TEST_ASSERT_EQUAL(20, result);
}
void test_multiply_with_zero()
{
    *p = 0;
    result = multiply(a, *p);
    TEST_ASSERT_EQUAL(0, result);
}
void test_multiply_negative()
{
    *p = -3;
    result = multiply(a, *p);
    TEST_ASSERT_EQUAL(-12, result);
}
```

```c
void test_addOne_basic()
{
    result = addOne(p);
    TEST_ASSERT_EQUAL(6, result);
}
void test_addOne_negative()
{
    *p = -3;
    result = addOne(p);
    TEST_ASSERT_EQUAL(-1, result); // tests proper error handling
}
// ----------------- RUN ALL TESTS -----------------
int run_unity_tests(void)
{
    UNITY_BEGIN();
    RUN_TEST(test_multiply_basic);
    RUN_TEST(test_multiply_with_zero);
    RUN_TEST(test_multiply_negative);
    RUN_TEST(test_addOne_basic);
    RUN_TEST(test_addOne_negative);
    return UNITY_END();
}
```

MONTANA
TECHNOLOGICAL UNIVERSITY

code.c and code.h – Functional Module

The Logic Module

```c
int multiply(int x, int y);     // multiplies two integers
int addOne(int * pointer);      // increments value pointed to by pointer
```

```c
int addOne(int * pointer)
{
    // check pointer validity and positive value
    if(!(*pointer >= 0))
    {
        fprintf(stderr, "Error: (*pointer) has to be greater or equal zero!\n");
        return -1;   // special error value
    }
    (*pointer)++;
    return *pointer;
}
int multiply(int x, int y)
{
    return x * y;
}
```

- code.h declares the interface.
- code.c contains the implementation.

- addOne() checks input validity → demonstrates defensive programming.Simple, testable logic → perfect for unit testing.

main.c – Program Entry Point

The Main Program and Command Interface

- The program supports several command-line arguments.
- --test runs all unit tests.
- The same executable can run in:
  - normal mode (program logic)
  - test mode (verification)
- Demonstrates separation of logic and testing.

```c
int main(int argc, char *argv[])
{
    int failed_tests = 0;
    if (argc > 1)
    {
        if (strcmp(argv[1], "--test") == 0)
            failed_tests = run_unity_tests();
        else if (strcmp(argv[1], "--author") == 0)
            printf(AUTHOR_NAME);
        else if (strcmp(argv[1], "--authorship") == 0)
            printf(AUTHOR_AUTHORSHIP);
        else if (strcmp(argv[1], "--help") == 0)
            printf("\n  --test\...\n\n");
    }
    else
    {
        printf("Wrong parameter. Use --help to see available options.\n");
        return 1;
    }
    //failed_tests = run_unity_tests();
    getchar(); // pause before exit (Windows)
    return failed_tests; // cmd/powershell: echo $LASTEXITCODE
}
```

Unit Testing with Unity - Writing Tests with Assertions

- Each test_* function verifies one behavior.
- TEST_ASSERT_EQUAL(expected, actual) checks correctness.
- Assertion = a condition that must be true for the test to pass.
- Failures are automatically reported by Unity.

```c
void test_multiply_basic()
{
    result = multiply(a, *p);
    TEST_ASSERT_EQUAL(20, result);
}
void test_multiply_with_zero()
{
    *p = 0;
    result = multiply(a, *p);
    TEST_ASSERT_EQUAL(0, result);
}
void test_multiply_negative()
{
    *p = -3;
    result = multiply(a, *p);
    TEST_ASSERT_EQUAL(-12, result);
}
```

Unit Testing with Unity - Setup and Teardown Functions

Preparing and Cleaning the Test Environment

- setUp() runs before each test → initializes variables.
- tearDown() runs after each test → cleans up memory.
- Ensures tests run independently and do not affect each other.
- Mimics a controlled test environment.

```c
void setUp()
{
    a = 4;
    p = (int*)
malloc(sizeof(int));
    *p = 5;
    result = 0;
}
void tearDown()
{
    free(p);
    result = 0;
}
```

Unit Testing with Unity - Running and Reporting Tests

- Each test is executed via RUN_TEST().
- Unity reports all results in a readable format.
- Helps trace logical errors quickly and accurately.

```
Result:
tests.c:53:test_multiply_basic:PASS
tests.c:54:test_multiply_with_zero:PASS
tests.c:55:test_multiply_negative:PASS
tests.c:56:test_addOne_basic:PASS
Error: (*pointer) has to be greater or equal zero!
tests.c:57:test_addOne_negative:PASS


-----------------------
5 Tests 0 Failures 0 Ignored
OK
```

```c
void test_addOne_basic()
{
    result = addOne(p);
    TEST_ASSERT_EQUAL(6, result);
}
void test_addOne_negative()
{
    *p = -3;
    result = addOne(p);
    TEST_ASSERT_EQUAL(-1, result); // tests proper error handling
}
// ----------------- RUN ALL TESTS -----------------
int run_unity_tests(void)
{
    UNITY_BEGIN();
    RUN_TEST(test_multiply_basic);
    RUN_TEST(test_multiply_with_zero);
    RUN_TEST(test_multiply_negative);
    RUN_TEST(test_addOne_basic);
    RUN_TEST(test_addOne_negative);
    return UNITY_END();
}
```

Summary and Best Practices

- Why do all these test functions have no parameters and return void? Because their **function signature** must match what Unity expects — the function's address is passed to the macro: `RUN_TEST(test_multiply_basic);` inside the `run_unity_tests()` function.

- If all tests run successfully, `UNITY_END()` returns the number of **failed tests**. This allows us to conveniently pass that information to the operating system using: `return UNITY_END();` in main.c, since main.c contains:

```
failed_tests = run_unity_tests();

return failed_tests;
```

Unions

An example - unions

```c
#include <stdio.h>
struct Example1
{
    char c;
    int i;
    short s;
};
union Example2
{
    short s;
    char c;
    int i;
};
int main(int argc, char *argv[])
{
    printf( "Size of a struct Example is = %d\n", sizeof(struct Example1) );
    printf( "Size of a struct Example is = %d\n", sizeof(union Example2) );
    struct Example1 example1;
    printf( "Struct Example1:\n" );
    printf( "Address of variable c = %d\n", &example1.c );
    printf( "Address of variable i = %d\n", &example1.i );
    printf( "Address of variable s = %d\n", &example1.s );
    union Example2 example2;
    printf( "Union Example2:\n" );
    printf( "Address of variable s = %d\n", &example2.s );
    printf( "Address of variable c = %d\n", &example2.c );
    printf( "Address of variable i = %d\n", &example2.i );
    return 0;
}
```

Result:

```
Size of a struct Example is = 12
Size of a struct Example is = 4
Struct Example1:
Address of variable c = 6487828
Address of variable i = 6487832
Address of variable s = 6487836
Union Example2:
Address of variable s = 6487824
Address of variable c = 6487824
Address of variable i = 6487824
```

Unions

- Unions in C are special data types that allow different data types to be stored in the same memory location.

Syntax:

```
unions symbolic_name1
{
    <statement1>
}<symbolic_name2, ...>;
```

Everything that is in angle brackets <> is optional.

An example - unions

```c
#include <stdio.h>
union Example
{
    short s;
    char c;
    int i;
};

int main(int argc, char *argv[])
{
    union Example example;
    printf( "Union Example:\n" );

    example.s = 5;
    printf( "The value of [example.s] = %d\n", example.s );
    example.c = 'A';
    printf( "The value of [example.c] = %d\n", example.c );
    example.i = 63123;
    printf( "The value of [example.i] = %d\n", example.i );
    /*but...*/
    printf( "The value of [example.s] = %d\n", example.s );

    return 0;
}
```

Result:

```
Union Example:
The value of [example.s] = 5
The value of [example.c] = 65
The value of [example.i] = 63123
The value of [example.s] = -2413
```

Unions in C are a mechanism that allows different data types to be stored in the same memory location, which can be useful in specific situations but requires caution due to potential pitfalls associated with their use.

A summary on unions:

Unions in C are special data types that allow different data types to be stored in the same memory location. Unlike structures, where each member has its own dedicated space, all members of a union share the same location. This means that at any given time, only one member of the union can hold a defined value, and changing the value of one member automatically overwrites the values of the others. This makes unions ideal for situations where we need to store different types of data in the same place, but only one of these types is active at any moment. Unions have various applications, from representing variable data types to creating more memory-efficient data structures. However, it’s important to note that improper use of unions can lead to programming errors, such as accessing uninitialized data or violating memory alignment rules

Enums

Enums

- An enum (enumeration) is a user-defined data type in C that consists of a set of named integer constants. These constants are often used to represent a fixed set of values.
- Enum syntax is the same as struct.

Syntax:

```
enums symbolic_name1
{
    <statement1>
}<symbolic_name2, ...>;
```

Everything that is in angle brackets <> is optional.

1 example

```c
#include <stdio.h>
enum color
{
    red,
    green,
    blue,
};


int main(int argc, char *argv[])
{
    printf("The size of [enum color] is %d\n", sizeof(enum color)); /*default as sizeof(int)*/

    enum color my_color = red;
    printf("The value of my_color is %d\n", my_color);

    int x = green;
    printf("The value of x is %d\n", x);

    my_color = 99;  /* be careful */
    printf("The value of my_color is %d\n", my_color);

    if(my_color == red || my_color == green || my_color == blue )
        printf("Proper value\n");
    else
        printf("Improper value\n");
    return 0;
}
```

These constants are often used to represent a fixed set of values, such as days of the week, colors, or error codes. Enums provide a way to make code more readable and maintainable by using meaningful names instead of raw integer values.

Result:

```
The size of [enum color] is 4
The value of my_color is 0
The value of x is 1
The value of my_color is 99
Improper value
```

2 example

```c
#include <stdio.h>
enum color
{
    red = 51,
    green,
    blue = 91,
    orange,
};

int main(int argc, char *argv[])
{
    int array[red];

    printf("The value of red is %d\n", red);
    printf("The value of green is %d\n", green);
    printf("The value of blue is %d\n", blue);
    printf("The value of blue is %d\n", orange);
    return 0;
}
```

If we don't specify a value for an enumeration field, the default value is 0. The subsequent field will have a value one greater than the previous one. However, if we define a different value for a field, that field will have an individually defined value and the next field will be one more than the last one.

When using elements from a defined enum, we don't need to use the enum's name itself. They are treated by the compiler as integer constants, which means they can be used to define array sizes just like the `#define` preprocessor directive.

Result:

```
The value of red is 51
The value of green is 52
The value of blue is 91
The value of orange is 92
```

Enums

- An enum (enumeration) is a user-defined data type in C that consists of a set of named integer constants. These constants are often used to represent a fixed set of values.
- Enum syntax is the same as struct.

Syntax:

```
enums symbolic_name1
{
    <statement1>
}<symbolic_name2, ...>;
```

Everything that is in angle brackets <> is optional.

3 example

```c
#include <stdio.h>
enum color
{
    red = 51,
    green,
    blue = 91,
    orange,
}my_global_color, * my_pointer_color;

enum color my_global = orange;
int main(int argc, char *argv[])
{
    enum color my_local_color = red;
    printf("The value of my_local_color is %d\n", my_local_color);
    my_pointer_color = &my_local_color;
    *my_pointer_color = blue;
    printf("The value of my_local_color is %d\n", my_local_color);
    switch (my_global)
    {
        case red:
                printf("red\n");
                break;
        case green:
                printf("green\n");
                break;
        case blue:
                printf("blue\n");
                break;
        case orange:
                printf("orange\n");
                break;
        default:
                printf("none\n");
                break;
    }
    return 0;
}
```

Given that enums can also create global variables within their definition (including pointers), it's important to keep this in mind. Enums are often used with switch statements, allowing us to convert numerical values into strings.

Result:

```
The value of my_local_color is 51
The value of my_local_color is 91
orange
```

MONTANA TECHNOLOGICAL UNIVERSITY

Bit-fields

An example

```c
#include <stdio.h>
struct Data
{
    unsigned char even_number : 1;
    /* instead of using the char data type,
       if you use the int data type, it will occupy 4 bytes */
    unsigned char greater_than_10 : 1;
    unsigned char is_the_power_of_two : 1;
};
int main(int argc, char *argv[])
{
    printf("The size of Data is %d\n", sizeof(struct Data));

    struct Data my_data;
    int value;
    printf("Give me a number\n");
    _ = scanf("%d", &value);

    my_data.even_number = !(value % 2);
    my_data.greater_than_10 = ( value > 10 );
    my_data.is_the_power_of_two = value && !(value & (value - 1));

    printf("%d %d %d \n", my_data.even_number, my_data.greater_than_10, my_data.is_the_power_of_two );
    return 0;
}
```

A bit-field structure occupies as much space as the largest defined field type, instead of assigning a value '=' to fields, the number of BITS for this flag ':' is specified

Result1:

```
The size of Data is 1
Give me a number
8
1 0 1
```

Result2:

```
The size of Data is 1
Give me a number
12
1 1 0
```

Result3:

```
The size of Data is 1
Give me a number
11
0 1 0
```

File Input / Output

Introduction to File Access in C

- Why File Access Matters:
  - Storing and retrieving data beyond program execution.
- Types of File Modes:
  - Text Mode: Reads and writes data as readable characters, char-by-char, line-by-line, etc.
  - Binary Mode: Reads and writes data as raw bytes, allowing efficient handling of complex data structures.

Opening and closing files

- Function: `fopen()`
- Syntax:

```
FILE *fopen(const char *filename, const char *mode);
```

- Error Handling:
  - Always check if `fopen()` returned `NULL`, indicating failure (e.g., file not found).
- Function: `fclose()`
- Syntax:

```
int fclose(*FILE);
```

- Error Handling:
  - Always check if `fclose()` returned `NULL`, indicating failure (e.g., file not found).

- In the context of file access, the `fopen()` and `fclose()` functions serve a role analogous to curly braces `{, }` in defining a code block. Opening a file with `fopen()` is akin to entering a new scope of operations on that file, similar to how an opening curly brace signals the beginning of a new block of statements. Conversely, `fclose()` marks the end of this scope, closing the file and thus concluding the block, much like a closing curly brace.
- While the compiler ensures that code blocks are properly closed, it is the programmer's responsibility to ensure files are closed correctly. An unclosed file is like an unclosed curly brace - it can lead to unexpected errors and hinder further operations.

Opening files

- Function: `fopen()`
- Syntax:

file path/file name

mode

```
FILE *fopen(const char *filename, const char *mode);
```

- Error Handling:
  - Always check if fopen() returned `NULL`, indicating failure (e.g., file not found).

`const char *filename`

- The `const char *filename` argument in the `fopen()` function can specify three types of file locations:

- File Name Only:
  - When only the file name is provided, `fopen()` attempts to open the file in the same directory as the currently executing program.
  - Example: `fopen("data.txt", "rt")` will try to open the file `"data.txt"` in the directory where the program is located.

- Relative Path:
  - A relative path specifies the location of the file relative to the current working directory.
  - Example: `fopen("data/results.txt", "wt")`* will try to open the file `"results.txt"` in a subdirectory named `"data"` within the current working directory.

- Absolute Path:
  - An absolute path provides the complete path to the file, starting from the root directory of the file system.
  - Example: `fopen("/home/user/documents/project/data.txt", "at")` will open the file `"data.txt"` in the specified directory, regardless of the current working directory.

*Platform-specific path separators: The specific character used to separate directories in a path (e.g., / in Unix-like systems, \ \ in Windows) depends on the operating system.

File Access Modes in C

- Access Modes:

Mode	Description
r	Opens an existing file for reading only.
w	Opens a file for writing. Creates a new file or clears the content of an existing one.
a	Opens a file for appending data. Creates the file if it does not already exist.
x	Creates the file if it does not already exist. Fails if the file already exists.
+	Combined with r, w, a, or x, allows both reading and writing.

- File Types:

Type Modifier	Description
t	Text mode (default). Treats file as a sequence of characters.
b	Binary mode. Treats file as a sequence of bytes with no translation.

MONTANA
TECHNOLOGICAL UNIVERSITY

Binary File I/O Functions

- For reading blocks of binary data.
  - Syntax:

```
int fread(void *ptr, int size, int count, FILE *stream);
```

- For writing blocks of binary data.
  - Syntax:

```
int fwrite(const void *ptr, int size, int count, FILE *stream);
```

`ptr:`	Pointer to the data you want to write(read).
`size:`	Size, in bytes, of each element to be written(read).
`count:`	Number of elements to write(read), which is the third argument.
`stream:`	File pointer to the open file where data should be written(read).

An example – Binary file

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    FILE *file;
    file =  fopen("binaryFile.bin", "wb");
    if(file)
    {
        int x = 5;
        _ =  fwrite(&x, sizeof(x), 1, file);

        char text[10] = "Some text";
        _ =  fwrite(text, sizeof(*text), 10, file);

        float fnumber = 3.14f;
        _ =  fwrite(&fnumber, sizeof(fnumber), 1, file);
        fclose(file);
    }
    else
        printf("Error");
    return 0;
}
```

binaryFile.bin:

```
Some text ĂőH@
```

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    FILE *file;
    file =  fopen("binaryFile.bin", "rb");
    if(file)
    {
        int x;
        _ =  fread(&x, sizeof(x), 1, file);

        char text[10];
        _ =  fread(text, sizeof(*text), 10, file);

        float fnumber;
        _ =  fread(&fnumber, sizeof(fnumber), 1, file);

        fclose(file);

        printf("%d %s %f", x, text, fnumber);
    }
    else
        printf("Error");
    return 0;
}
```

Result:

```
15 Some text 3.140000
```

`fseek()` function

- The function `fseek()` moves the file pointer to a specified location, allowing you to read from or write to a specific part of the file.
- Syntax:

```
int fseek(FILE *stream, long offset, int origin);
```

`stream:`	A pointer to the FILE object that identifies the file.
`offset:`	The number of bytes to move the file pointer.
`origin:`	The starting position for the offset; it can be one of the following constants:

`SEEK_SET`:	Start from the beginning of the file (0).
`SEEK_CUR`:	Start from the current position of the file pointer (1).
`SEEK_END`:	Start from the end of the file(2).

`ftell()` and `rewind()` functions

- The function `ftell()` returns the current file position as a long integer. If an error occurs, it returns -1.
- Syntax:

```
long ftell(FILE *stream);
```

- Unlike `fseek()` and `ftell()`, `rewind()` is simpler to use and is intended to reset the file pointer to the beginning of the file.
- Syntax:

```
void rewind(FILE *stream);
```

1 example – fseek()

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    FILE *file;
    file =  fopen("binaryFile.bin", "rb");
    if(file)
    {
          /* Move to the 5th byte from the beginning */
          fseek(file, 5, SEEK_SET);
          printf("Position after fseek: %ld\n", ftell(file));

          /* Move forward by 10 bytes from the current position */
          fseek(file, 10, SEEK_CUR);
          printf("Position after another fseek: %ld\n", ftell(file));

          /* Move to the end of the file */
          fseek(file, 0, SEEK_END);
          printf("Position at the end of file: %ld\n", ftell(file));
          fclose(file);
    }
    else
        printf("Error");
    return 0;
}
```

Result:

```
Position after fseek: 5
Position after another fseek: 15
Position at the end of file: 18
```

2 example – fseek(), ftell()

```c
#include <stdio.h>
int main(int argc, char *argv[])
{
    FILE *file;
    file =  fopen("binaryFile.bin", "rb");
    if(file)
    {
        int x; int position;
        position = ftell(file);
        printf("The position in file = %d\n", position);
        _ =  fread(&x, sizeof(x), 1, file);
        position = ftell(file);
        printf("The position in file = %d\n", position);
        char text[10];
        _ =  fread(text, sizeof(*text), 10, file);
        position = ftell(file);
        printf("The position in file = %d\n", position);
        float fnumber;
        _ =  fread(&fnumber, sizeof(fnumber), 1, file);
        position = ftell(file);
        printf("The position in file = %d\n", position);
        printf("%d %s %f\n", x, text, fnumber);
        /* Move to the 0th byte from the beginning */
        fseek(file, 0, SEEK_SET);
        printf("Position after fseek(): %ld\n", ftell(file));
        x = 0;
        _ =  fread(&x, sizeof(x), 1, file);
        printf("%d\n", x);
        fclose(file);
    }
    else
        printf("Error");
    return 0;
}
```

Result:

```
The position in file = 0
The position in file = 4
The position in file = 14
The position in file = 18
5 Some text 3.140000
Position after fseek(): 0
5
```

Text File I/O Functions

- For reading text from file:
  - `fgetc()`: Reads a single character from a file.
  - `fgets()`: Reads a line from a file.
  - `fscanf()`: Reads formatted input from a file (similar to `scanf()`).
- For writing text for file:
  - `fputc()`: Writes a single character to a file.
  - `fputs()`: Writes a string to a file.
  - `fprintf()`: Prints formatted output to a file (similar to `printf()`).

Reading Text from a file

- **`int fgetc(FILE *stream)`**
  - **Description**: Reads a single character from the specified file stream.
  - **Return Type**: Returns the character read as an unsigned char cast to an int, or `EOF` on end of file or error.
  - **Example**: `int ch = fgetc(file_pointer);`
- **`char *fgets(char *str, int n, FILE *stream)`**
  - **Description**: Reads a line from the file and stores it in the character array str.
  - **Return Type**: Returns str on success, or `NULL` on error or when end of file occurs.
  - **Example**: `char *result = fgets(buffer, sizeof(buffer), file_pointer);`
- **`int fscanf(FILE *stream, const char *format, ...)`**
  - **Description**: Reads formatted input from a file based on the format string, similar to scanf.
  - **Return Type**: Returns the number of items successfully read, or `EOF` if an error or end of file occurs before any items are matched.
  - **Example**: `int read_count = fscanf(file_pointer, "%d %f", &int_var, &float_var);`

Writing Text to a file

- `int fputc(int char, FILE *stream)`
  - **Description**: Writes a single character to the specified file stream.
  - **Return Type**: Returns the written character as an unsigned char cast to an int, or `EOF` on error.
  - **Example**: `int result = fputc('A', file_pointer);`
- `int fputs(const char *str, FILE *stream)`
  - **Description**: Writes a string to the file.
  - **Return Type**: Returns a non-negative number on success, or `EOF` on error.
  - **Example**: `int result = fputs("Hello, World!\n", file_pointer);`
- `int fprintf(FILE *stream, const char *format, ...)`
  - **Description**: Writes formatted output to the specified file stream, similar to `printf()`.
  - **Return Type**: Returns the number of characters written, or a negative value if an error occurs.
  - **Example**: `int chars_written = fprintf(file_pointer, "Integer: %d, Float: %f\n", int_var, float_var);`
-

Thank you

Jakub Leszek Pach
jpach@mtech.edu