## Assignment\_28-08-2024

Author: ThanhTH10

Date: 29/08/2024

1. Explore make file. Write makefile for mathmatical functions in C sqrt, pow, factorial, square,cube...include header file

Step1: Mylib.h

double sqrt(double x);

double my\_pow(double base, int exp);

unsigned long long factorial(int x);

double square(double x);

double cube(double x);

Step2: Logic implementation

- cube.c

#include "mylib.h"

double cube(double x)

{

    return x \* x;

}

- factorial.c

#include "mylib.h"

unsigned long long factorial(int n)

{

    if (n < 0)

        return 0; // Factorial is not defined for negative numbers

    unsigned long long result = 1;

    for (int i = 1; i <= n; ++i)

    {

        result \*= i;

    }

    return result;

}

- my\_pow.c

#include "mylib.h"

double my\_pow(double base, int exp)

{

    if (exp < 0)

        return 1.0 / my\_pow(base, -exp); // Handle negative exponents

    double result = 1;

    while (exp > 0)

    {

        if (exp % 2 == 1)

            result \*= base;

        base \*= base;

        exp /= 2;

    }

    return result;

}

- sqrt.c

#include "Mylib.h"

double sqrt(double x)

{

    if (x < 0)

        return -1; // Return -1 for negative inputs as square root is not defined

    double tolerance = 1e-10;

    double guess = x;

    while ((guess \* guess - x) > tolerance || (x - guess \* guess) > tolerance)

    {

        guess = (guess + x / guess) / 2;

    }

    return guess;

}

- square.c

#include "mylib.h"

double square(double x)

{

    return x \* x;

}

Create main.cpp

#include <stdio.h>

#include "mylib.h"

int main(int argc, char const \*argv[])

{

    printf("sqrt of 10: %lf\n", sqrt(10));

    printf("2 pow 2: %lf\n", my\_pow(2, 2));

    printf("factorial of 10: %lld\n", factorial(10));

    printf("quare: %lf\n", square(12.0));

    printf("cube: %lf\n", cube(12.0));

    return 0;

}

Step 3: create Makefile

main: main.o sqrt.o pow.o factorial.o square.o cube.o

    gcc -o main main.o sqrt.o pow.o factorial.o square.o cube.o

main.o: main.c

gcc -c main.c -o main.o

sqrt.o: sqrt.c

    gcc -c sqrt.c -o sqrt.o

pow.o: pow.c

gcc -Wall -c pow.c -o pow.o

factorial.o: factorial.c

gcc -c factorial.c -o factorial.o

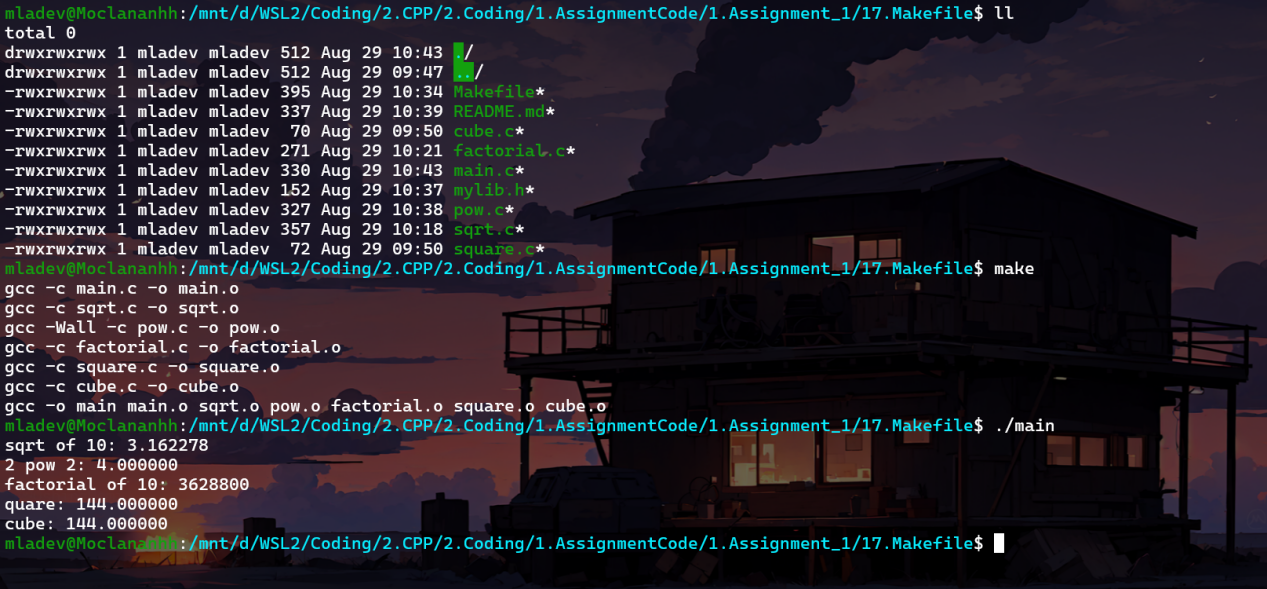
square.o: square.c

gcc -c square.c -o square.o

cube.o: cube.c

    gcc -c cube.c -o cube.o

Step 4: run program



2. Write makefile for C++ files, for add, subtract, mul,divide..main.cpp

g++ queue.cpp -fprofile-arcs -ftest-coverage

./a.out

gcov xxx.cpp

Step 1: Create header file

double add(double a, double b);

double subtract(double a, double b);

double multiply(double a, double b);

double divide(double a, double b);

Step 2: Logic implementation

- add.cpp

#include "mylib.h"

double add(double a, double b)

{

    return a + b;

}

- subtract.cpp

#include "mylib.h"

double subtract(double a, double b)

{

    return a - b;

}

- multiply.cpp

#include "mylib.h"

double multiply(double a, double b)

{

    return a \* b;

}

- divide.cpp

#include "mylib.h"

#include <iostream>

double divide(double a, double b)

{

    if (b == 0)

    {

        throw std::invalid\_argument("Division by zero");

    }

    return a / b;

}

Create main.cpp

#include <iostream>

#include "mylib.h"

int main()

{

    double a = 10.0, b = 5.0;

    std::cout << "Add: " << add(a, b) << std::endl;

    std::cout << "Subtract: " << subtract(a, b) << std::endl;

    std::cout << "Multiply: " << multiply(a, b) << std::endl;

    std::cout << "Divide: " << divide(a, b) << std::endl;

    return 0;

}

Step 3: Create Makefile

# Compiler

CXX = g++

# Compiler flags

CXXFLAGS = -Wall -std=c++17

# Target executable

TARGET = main

# Source files

SRCS = main.cpp \_add.cpp \_subtract.cpp \_divide.cpp \_multiply.cpp

# Test source files

TEST\_SRCS = \_add.cpp \_subtract.cpp \_divide.cpp \_multiply.cpp test.cpp test\_main.cpp

# Object files

OBJS = $(SRCS:.cpp=.o)

TEST\_OBJS = $(TEST\_SRCS:.cpp=.o)

# Default rule

all: $(TARGET)

# Linking rule

$(TARGET): $(OBJS)

    $(CXX) $(CXXFLAGS) -o $@ $^ -fprofile-arcs -ftest-coverage -L/usr/lib/x86\_64-linux-gnu/CppUTest -lCppUTest

# Test executable

test: $(TEST\_OBJS)

    $(CXX) $(CXXFLAGS) $(TEST\_OBJS) -fprofile-arcs -ftest-coverage -o test -L/usr/lib/x86\_64-linux-gnu/CppUTest -lCppUTest

# Compilation rule

%.o: %.cpp

    $(CXX) $(CXXFLAGS) -fprofile-arcs -ftest-coverage -c $< -o $@

# Clean rule

clean:

    rm -f $(TARGET) test $(OBJS) $(TEST\_OBJS) \*.gcda \*.gcno \*.gcov

# Run rule

run: $(TARGET)

    ./$(TARGET)

# Coverage rule

coverage: $(TEST\_OBJS)

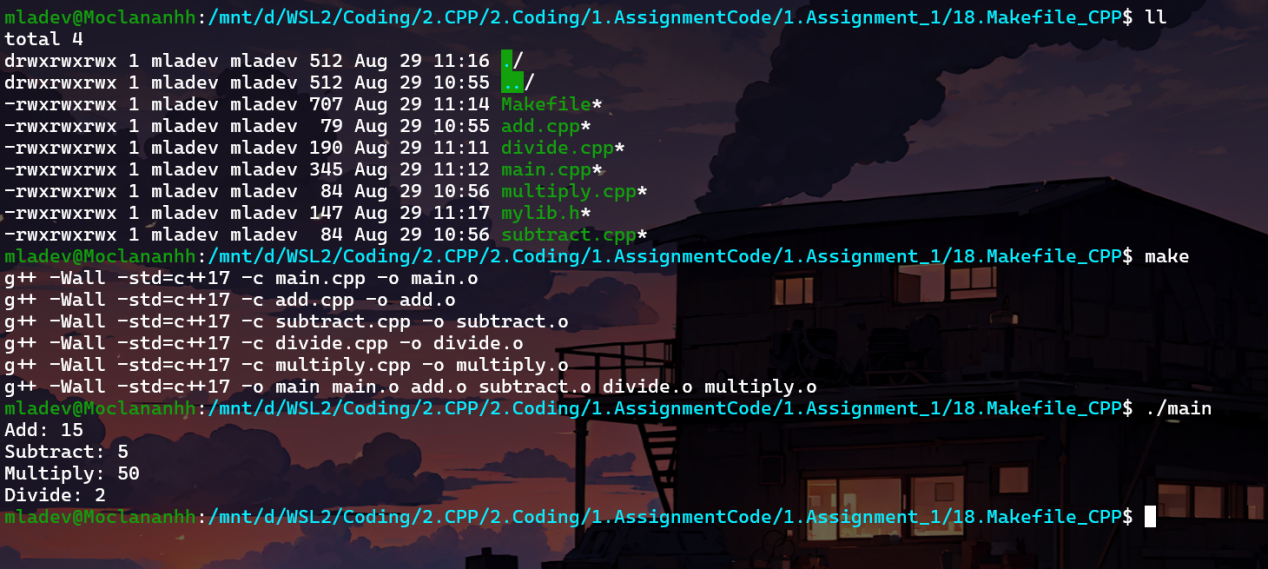
    $(CXX) $(CXXFLAGS) -fprofile-arcs -ftest-coverage -o $(TARGET) $(TEST\_OBJS) -L/usr/lib/x86\_64-linux-gnu/CppUTest -lCppUTest

    ./$(TARGET)

    gcov \_\*.cpp

.PHONY: all clean run coverage test

Step 4: run program



Create test coverage

Step 1: Create test\_main.cpp

#include <CppUTest/CommandLineTestRunner.h>

int main(int ac, char \*\*av)

{

    return CommandLineTestRunner::RunAllTests(ac, av);

}

Step 2: Create test.cpp

#include <CppUTest/Utest.h>

#include <CppUTest/UtestMacros.h>

#include "CppUTest/TestHarness.h"

#include "mylib.h"

#include <stdexcept>

// Test Group

TEST\_GROUP(DivideGroup){};

// Test case for normal division

TEST(DivideGroup, HandlesPositiveNumbers)

{

    DOUBLES\_EQUAL(5.0, divide(10.0, 2.0), 0.0001);

    DOUBLES\_EQUAL(3.0, divide(9.0, 3.0), 0.0001);

}

// Test case for division by zero

TEST(DivideGroup, HandlesDivisionByZero)

{

    CHECK\_THROWS(std::invalid\_argument, divide(10.0, 0.0));

}

// Test case for negative numbers

TEST(DivideGroup, HandlesNegativeNumbers)

{

    DOUBLES\_EQUAL(-5.0, divide(-10.0, 2.0), 0.0001);

    DOUBLES\_EQUAL(-5.0, divide(10.0, -2.0), 0.0001);

    DOUBLES\_EQUAL(5.0, divide(-10.0, -2.0), 0.0001);

}

// Test case for division resulting in a fraction

TEST(DivideGroup, HandlesFractionResult)

{

    DOUBLES\_EQUAL(2.5, divide(10.0, 4.0), 0.0001);

}

// Test Group

TEST\_GROUP(AddGroup){};

// Test case for adding two positive numbers

TEST(AddGroup, HandlesPositiveNumbers)

{

    DOUBLES\_EQUAL(10.0, add(3.0, 7.0), 0.0001);

    DOUBLES\_EQUAL(5.0, add(0.0, 5.0), 0.0001);

}

// Test case for adding two negative numbers

TEST(AddGroup, HandlesNegativeNumbers)

{

    DOUBLES\_EQUAL(-10.0, add(-3.0, -7.0), 0.0001);

    DOUBLES\_EQUAL(-10.0, add(-5.0, -5.0), 0.0001);

}

// Test case for adding a positive and a negative number

TEST(AddGroup, HandlesMixedSignNumbers)

{

    DOUBLES\_EQUAL(4.0, add(-3.0, 7.0), 0.0001);

    DOUBLES\_EQUAL(-4.0, add(3.0, -7.0), 0.0001);

}

// Test case for adding zero

TEST(AddGroup, HandlesZero)

{

    DOUBLES\_EQUAL(0.0, add(0.0, 0.0), 0.0001);

    DOUBLES\_EQUAL(10.0, add(10.0, 0.0), 0.0001);

    DOUBLES\_EQUAL(-10.0, add(0.0, -10.0), 0.0001);

}

// Test Group

TEST\_GROUP(SubtractGroup){};

// Test case for subtracting two positive numbers

TEST(SubtractGroup, HandlesPositiveNumbers)

{

    DOUBLES\_EQUAL(3.0, subtract(10.0, 7.0), 0.0001);

    DOUBLES\_EQUAL(0.0, subtract(5.0, 5.0), 0.0001);

}

// Test case for subtracting two negative numbers

TEST(SubtractGroup, HandlesNegativeNumbers)

{

    DOUBLES\_EQUAL(-2.0, subtract(-7.0, -5.0), 0.0001);

    DOUBLES\_EQUAL(-5.0, subtract(-10.0, -5.0), 0.0001);

}

// Test case for subtracting a positive number from a negative number

TEST(SubtractGroup, HandlesMixedSignNumbers)

{

    DOUBLES\_EQUAL(-10.0, subtract(-3.0, 7.0), 0.0001);

    DOUBLES\_EQUAL(10.0, subtract(3.0, -7.0), 0.0001);

}

// Test case for subtracting zero

TEST(SubtractGroup, HandlesZero)

{

    DOUBLES\_EQUAL(10.0, subtract(10.0, 0.0), 0.0001);

    DOUBLES\_EQUAL(-10.0, subtract(-10.0, 0.0), 0.0001);

    DOUBLES\_EQUAL(0.0, subtract(0.0, 0.0), 0.0001);

}

// Test Group

TEST\_GROUP(MultiplyGroup){};

// Test case for multiplying two positive numbers

TEST(MultiplyGroup, HandlesPositiveNumbers)

{

    DOUBLES\_EQUAL(20.0, multiply(4.0, 5.0), 0.0001);

    DOUBLES\_EQUAL(0.0, multiply(0.0, 5.0), 0.0001);

}

// Test case for multiplying two negative numbers

TEST(MultiplyGroup, HandlesNegativeNumbers)

{

    DOUBLES\_EQUAL(20.0, multiply(-4.0, -5.0), 0.0001);

    DOUBLES\_EQUAL(25.0, multiply(-5.0, -5.0), 0.0001);

}

// Test case for multiplying a positive number by a negative number

TEST(MultiplyGroup, HandlesMixedSignNumbers)

{

    DOUBLES\_EQUAL(-20.0, multiply(-4.0, 5.0), 0.0001);

    DOUBLES\_EQUAL(-15.0, multiply(3.0, -5.0), 0.0001);

}

// Test case for multiplying by zero

TEST(MultiplyGroup, HandlesZero)

{

    DOUBLES\_EQUAL(0.0, multiply(10.0, 0.0), 0.0001);

    DOUBLES\_EQUAL(0.0, multiply(0.0, 10.0), 0.0001);

    DOUBLES\_EQUAL(0.0, multiply(0.0, 0.0), 0.0001);

}

Step 3: Run test

