МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

## ***Факультет информационных технологий и робототехники***

Кафедра программного обеспечения информационных систем и технологий

**Отчет по лабораторной работе № 8**

по дисциплине:” Системное программирование ”

на тему: ”Автоматическая сборка многофайловых проектов***”***

Выполнил**:** студент группы 10701321 Кругликовский М.А.

Принял**:** ст.пр. Станкевич C.П.

Минск 2023

# **Лабораторная работа № 8**

**Цель работы**

Закрепить на практике принципы создания проектов с помощью скриптов SHELL, освоить средства примитивного графического интерфейса в Linux-скриптах.

**Задание**

**Задание 1**

Используя материалы (исходные файлы) упражнения №1 проведите компиляцию однофайлового проекта с прохождением всех стадий компиляции. Для ускорения работы примените скрипты bash. Исходные файлы программ обязательно должны содержать коментарии. Сборка проекта должна содержать файлы с результатами препроцессинга. Исследуйте фалы препроцессора, найдите в них код своей программы. Определите размеры исходных, препроцессорных, ассемблерных, объектных и исполняемых файлов. С помощью соответствующий консольных команд определите форматы этих файлов. Результаты подтвердите скриншотами.

**Задание 2**

Используя материалы (исходные файлы) упражнения №1 и №2 создайте один многофайловый проект, руководствуясь принципом Single Responsibility Principle. Каждое задание должно быть представлено отдельным исходным файлом. Для связывания файлов проекта обязательно используйте заголовочные файлы. Все файлы могут располагаться в одной директории. Все задания должны вызываться консольным меню. Для выхода из приложения предусмотрите отдельный параметр. Проведите автосборку проект с прохождением всех стадий компиляции с использованием утилиты make.

**Задание 3**

Используя материалы (исходные файлы) задания №2 создайте один многофайловый проект, руководствуясь принципом Single Responsibility Principle. Каждое задание должно быть представлено отдельным исходным файлом. Используйте архитектурный шаблон проектирования MVC. Здесь под шаблоном MVC понимается файлы или группа файлов в трех отдельных директориях (папках), а не просто три отдельных файла. Все задания должны вызываться консольным меню. Для выхода из приложения предусмотрите отдельный параметр. Для связывания файлов проекта обязательно используйте заголовочные файлы. Проведите рекурсивную автосборку проект с прохождением всех стадий компиляции с использованием утилиты make.

**Результаты выполнения**

**Задание 1**

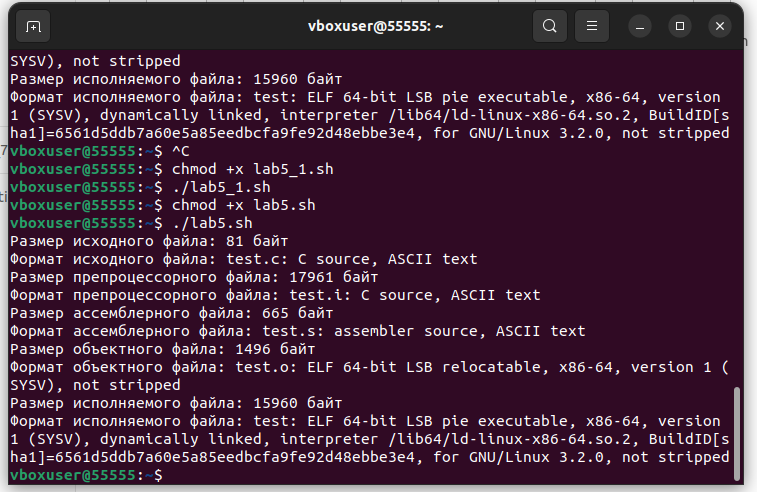
****

Рисунок 1. Скрипт bash, который выполнит компиляцию однофайлового проекта и соберет все промежуточные файлы, а также определит их размеры и форматы.

#!/bin/bash

# Указываем имя исходного файла

source\_file="test.c"

# Выполняем препроцессинг и сохраняем результат в файле .i

gcc -E "$source\_file" -o "${source\_file%.c}.i"

# Выполняем компиляцию препроцессированного файла и сохраняем результат в файле .s

gcc -S "${source\_file%.c}.i" -o "${source\_file%.c}.s"

# Выполняем компиляцию ассемблерного файла и сохраняем результат в файле .o

gcc -c "${source\_file%.c}.s" -o "${source\_file%.c}.o"

# Собираем исходный файл в исполняемый файл

gcc -o "${source\_file%.c}" "${source\_file%.c}.o"

# Определяем размеры и форматы файлов

source\_size=$(stat -c %s "$source\_file")

preprocessed\_size=$(stat -c %s "${source\_file%.c}.i")

assembly\_size=$(stat -c %s "${source\_file%.c}.s")

object\_size=$(stat -c %s "${source\_file%.c}.o")

executable\_size=$(stat -c %s "${source\_file%.c}")

source\_format=$(file "$source\_file")

preprocessed\_format=$(file "${source\_file%.c}.i")

assembly\_format=$(file "${source\_file%.c}.s")

object\_format=$(file "${source\_file%.c}.o")

executable\_format=$(file "${source\_file%.c}")

# Выводим информацию о размерах и форматах файлов

echo "Размер исходного файла: $source\_size байт"

echo "Формат исходного файла: $source\_format"

echo "Размер препроцессорного файла: $preprocessed\_size байт"

echo "Формат препроцессорного файла: $preprocessed\_format"

echo "Размер ассемблерного файла: $assembly\_size байт"

echo "Формат ассемблерного файла: $assembly\_format"

echo "Размер объектного файла: $object\_size байт"

echo "Формат объектного файла: $object\_format"

echo "Размер исполняемого файла: $executable\_size байт"

echo "Формат исполняемого файла: $executable\_format

**Задание 2**

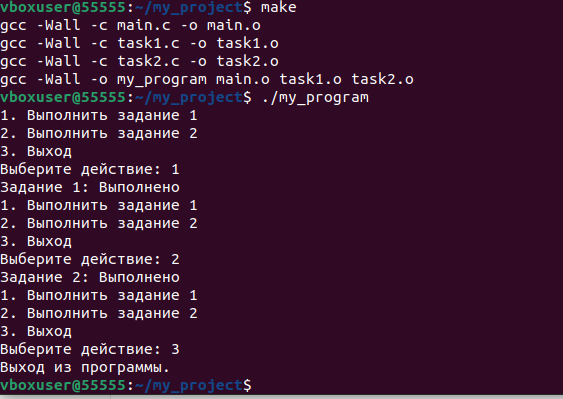


Рисунок 2. создания многофайлового проекта, удовлетворяющего принципу Single Responsibility Principle и автоматизации сборки с использованием make

# Компилятор и флаги компиляции

CC = gcc

CFLAGS = -Wall

# Список исходных файлов

SRCS = main.c task1.c task2.c

# Преобразование .c файлов в .o файлы

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

# Заголовочные файлы

HEADERS = task1.h task2.h

# Имя исполняемого файла

TARGET = my\_program

# Правило сборки для исполняемого файла

$(TARGET): $(OBJS)

$(CC) $(CFLAGS) -o $(TARGET) $(OBJS)

# Правило сборки для .o файлов из .c файлов

%.o: %.c $(HEADERS)

$(CC) $(CFLAGS) -c $< -o $@

# Правило для очистки собранных файлов

clean:

rm -f $(OBJS) $(TARGET)

**Задание 3**

Makefile

# Compiler

CXX = g++

# Compiler flags

CXXFLAGS = -std=c++11 -Wall

# Directories

MODEL\_DIR = Model

VIEW\_DIR = View

CONTROLLER\_DIR = Controller

# Source files

MODEL\_SOURCES = $(wildcard $(MODEL\_DIR)/\*.cpp)

VIEW\_SOURCES = $(wildcard $(VIEW\_DIR)/\*.cpp)

CONTROLLER\_SOURCES = $(wildcard $(CONTROLLER\_DIR)/\*.cpp)

MAIN\_SOURCE = main.cpp

# Object files

MODEL\_OBJECTS = $(MODEL\_SOURCES:.cpp=.o)

VIEW\_OBJECTS = $(VIEW\_SOURCES:.cpp=.o)

CONTROLLER\_OBJECTS = $(CONTROLLER\_SOURCES:.cpp=.o)

MAIN\_OBJECT = $(MAIN\_SOURCE:.cpp=.o)

# Executable name

EXECUTABLE = app

all: $(EXECUTABLE)

$(EXECUTABLE): $(MODEL\_OBJECTS) $(VIEW\_OBJECTS) $(CONTROLLER\_OBJECTS) $(MAIN\_OBJECT)

$(CXX) $(CXXFLAGS) $(MODEL\_OBJECTS) $(VIEW\_OBJECTS) $(CONTROLLER\_OBJECTS) $(MAIN\_OBJECT) -o $(EXECUTABLE)

# Rule for creating object files

$(MODEL\_DIR)/%.o: $(MODEL\_DIR)/%.cpp

$(CXX) $(CXXFLAGS) -c $< -o $@

$(VIEW\_DIR)/%.o: $(VIEW\_DIR)/%.cpp

$(CXX) $(CXXFLAGS) -c $< -o $@

$(CONTROLLER\_DIR)/%.o: $(CONTROLLER\_DIR)/%.cpp

$(CXX) $(CXXFLAGS) -c $< -o $@

%.o: %.cpp

$(CXX) $(CXXFLAGS) -c $< -o $@

clean:

rm -f $(MODEL\_DIR)/\*.o $(VIEW\_DIR)/\*.o $(CONTROLLER\_DIR)/\*.o \*.o $(EXECUTABLE)

**// main.cpp**

#include "Model/Person.h"

#include "View/PersonView.h"

#include "Controller/PersonController.h"

int main() {

// Create model, view, and controller

Person person;

PersonView personView;

PersonController personController(&person, &personView);

// Set person details

personController.setPersonDetails("John Doe", 30);

// Update and print details

personController.updateView();

// Update person details and print again

personController.setPersonDetails("Jane Smith", 35);

personController.updateView();

return 0;

}

**// Controller/PersonController.cpp**

#include "PersonController.h"

PersonController::PersonController(Person\* model, PersonView\* view) : m\_model(model), m\_view(view) {}

void PersonController::setPersonDetails(const std::string& name, int age) {

m\_model->setName(name);

m\_model->setAge(age);

}

void PersonController::updateView() const {

m\_view->printPersonDetails(m\_model->getName(), m\_model->getAge());

}

**// Controller/PersonController.h**

#ifndef PERSONCONTROLLER\_H

#define PERSONCONTROLLER\_H

#include "Person.h"

#include "PersonView.h"

class PersonController {

public:

PersonController(Person\* model, PersonView\* view);

void setPersonDetails(const std::string& name, int age);

void updateView() const;

private:

Person\* m\_model;

PersonView\* m\_view;

};

#endif

**// View/PersonView.cpp**

#include "../Model/Person.h"

void PersonView::printPersonDetails(const std::string& name, int age) const {

std::cout << "Person Details:\n";

std::cout << "Name: " << name << "\n";

std::cout << "Age: " << age << "\n";

}

**// View/PersonView.h**

#ifndef PERSONVIEW\_H

#define PERSONVIEW\_H

#include <iostream>

#include "Person.h"

class PersonView {

public:

void printPersonDetails(const std::string& name, int age) const;

};

#endif

**// Model/Person.cpp**

#include "Person.h"

void Person::setName(const std::string& name) {

m\_name = name;

}

void Person::setAge(int age) {

m\_age = age;

}

std::string Person::getName() const {

return m\_name;

}

int Person::getAge() const {

return m\_age;

}

**// Model/Person.h**

#ifndef PERSON\_H

#define PERSON\_H

#include <string>

class Person {

public:

void setName(const std::string& name);

void setAge(int age);

std::string getName() const;

int getAge() const;

private:

std::string m\_name;

int m\_age;

};

#endif

- ProjectDirectory

- Makefile

- main.cpp

- Model

- Person.h

- Person.cpp

- View

- PersonView.h

- PersonView.cpp

- Controller

- PersonController.h

- PersonController.cpp

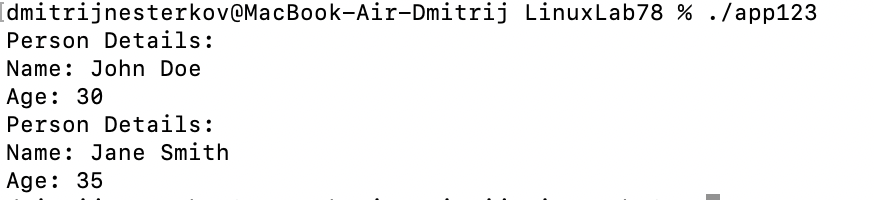


Рисунок 3. Скриншот результата

**Контрольные вопросы**

**В чем различие компиляции и интерпритации программы?**

Компиляция и интерпретация - это два основных способа выполнения программного кода, и они отличаются по способу обработки и исполнения программ.

Компиляция:

Процесс: При компиляции исходный код программы (например, на языке C, C++) преобразуется в машинный код (или в промежуточный код), который выполняется компьютером.

Время выполнения: Компиляция происходит до запуска программы, и в результате компиляции создается исполняемый файл (например, .exe для Windows или .out для Unix/Linux).

Пример: Программы на C, C++, Rust, и большинство языков программирования высокого уровня компилируются.

Преимущества: Компилированные программы, как правило, выполняются быстрее, так как они уже переведены в машинный код, и нет необходимости в интерпретации на этапе выполнения. Также компиляция позволяет обнаруживать множество ошибок на этапе компиляции, что может уменьшить количество ошибок во время выполнения.

Интерпретация:

Процесс: При интерпретации исходный код программы читается и выполняется построчно или в более мелких единицах виртуальной машины или интерпретатора.

Время выполнения: Интерпретация происходит во время выполнения программы. Отсутствует создание отдельного исполняемого файла.

Пример: Программы на Python, JavaScript, Ruby, PHP и многих других языках программирования интерпретируются.

Преимущества: Интерпретация позволяет более гибкое взаимодействие с программой, так как она может адаптироваться к различным условиям во время выполнения. Также она упрощает кросс-платформенную разработку, так как интерпретатор может быть запущен на разных платформах без необходимости перекомпиляции.