



МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ЛИПЕЦКИЙ
ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Институт компьютерных наук
Кафедра автоматизированных систем управления

Лабораторная работа №2
по операционным системам

Студент АС-21-1 _____ Станиславчук С. М.
(подпись, дата)

Руководитель
Доцент, к.п.н. _____ Кургасов В. В.
(подпись, дата)

Липецк 2024

Содержание

3. Цель работы	3
4. Задание кафедры	4
5. Исходные тексты созданных программ, содержимое созданных make- файлов, иллюстрация результатов работы	5
6. Вывод.....	12

3. Цель работы

Ознакомиться с техникой компиляции программ на языке программирования C (C++) в среде ОС семейства Unix, а также получить практические навыки использования утилиты GNU make для сборки проекта

4. Задание кафедры

Изучить особенности работы с утилитой make при создании проекта на языке C (C++) в ОС Unix, а также получить практические навыки использования утилиты GNU make при создании и сборке проекта

2. Самостоятельно сформулировать задачу, решение которой потребует создание не менее двух исходных файлов (с программным кодом). Предметные области в учебной студенческой группе не должны повторяться.

3. Используя любой текстовый редактор, создать программу на языке C (C++).

4. Для автоматизации сборки проекта утилитой make создать make-файл (см. п. «Пример создания более сложного make-файла»).

5. Выполнить программу (скомпилировать, при необходимости отладить).

6. Показать, что при изменении одного исходного файла и последующем вызове make будут исполнены только необходимые команды компиляции (неизмененные файлы перекомпилированы не будут) и изменены атрибуты и/или размер объектных файлов (файлы с расширением .o).

7. Создать make-файл с высоким уровнем автоматизированной обработки исходных файлов программы согласно следующим условиям:

- имя скомпилированной программы (выполняемый или бинарный файл), флаги компиляции и имена каталогов с исходными файлами и бинарными файлами (каталоги src, bin и т. п.) задаются с помощью переменных в makefile;
- зависимости исходных файлов на языке C (C++) и цели в make-файле должны формироваться динамически;
- наличие цели clean, удаляющей временные файлы;
- каталог проекта должен быть структурирован следующим образом:
 - src – каталог с исходными файлами;
 - bin – каталог с бинарными файлами (скомпилированными);
 - makefile

5. Исходные тексты созданных программ, содержимое созданных make-файлов, иллюстрация результатов работы

Мой комплекс программ будет реализовывать музыкальный плеер, в который можно добавлять песни (просто строки), а также проигрывать их (выводить в терминал). Думаю, в качестве образца комплексной компиляции этого будет вполне достаточно.

Я написал 2 скриптовых .cpp файла: MusicPlayer.cpp, MusicList.cpp и один хедер файл MusicList.h.

MusicPlayer.cpp – основной скрипт, имеющий main функцию, в которой мы создадим наш класс, добавим песни, и воспроизведем их.

MusicList.cpp – скрипт, реализующий класс MusicList.

MusicList.h – заголовочный файл, содержащий объявление класса MusicList.

MusicPlayer.cpp:

```
#include <iostream>
#include "MusicList.h"

int main() {
    MusicList list;
    list.addTrack("Track 1");
    list.addTrack("Track 2");
    list.addTrack("Track 3");

    list.play();

    return 0;
}
```

MusicList.cpp:

```
#include <iostream>

#include <vector>

#include "MusicList.h"

void MusicList::addTrack(const std::string& trackName) {
    tracks.push_back(trackName);
}

void MusicList::play() {
    std::cout << "Playing music tracks:" << std::endl;
    for (const auto& track : tracks) {
        std::cout << "- " << track << std::endl;
    }
}
```

MusicList.h:

```
#ifndef MUSICLIST_H
#define MUSICLIST_H

#include <string>
#include <vector>

class MusicList {
public:
    void addTrack(const std::string& trackName);
    void play();

private:
    std::vector<std::string> tracks;
};

#endif // MUSICLIST_H
```

Теперь создадим Makefile, который будет компилировать оба исходных файла:

Makefile:

```
# Это Makefile для компиляции программы MusicPlayer из исходных файлов MusicPlayer.cpp и
MusicList.cpp

# Переменная CC указывает компилятор C++
CC = g++

# Переменная CFLAGS содержит флаги компиляции
CFLAGS = -Wall -g

# Переменная TARGET задает имя целевого файла
TARGET = MusicPlayer

# Переменная SRCS содержит список исходных файлов программы
SRCS = MusicPlayer.cpp MusicList.cpp

# Переменная OBJS содержит список объектных файлов, которые будут созданы из исходных файлов
OBJS = $(SRCS:.cpp=.o)

# Цель "all" зависит от цели $(TARGET), что означает, что для выполнения цели "all" необходимо
сначала создать целевой файл $(TARGET)

all: $(TARGET)

# Правило для создания целевого файла $(TARGET)
$(TARGET): $(OBJS)

    # Команда для компиляции всех объектных файлов вместе для создания целевого файла
    $(CC) $(CFLAGS) -o $@ $^

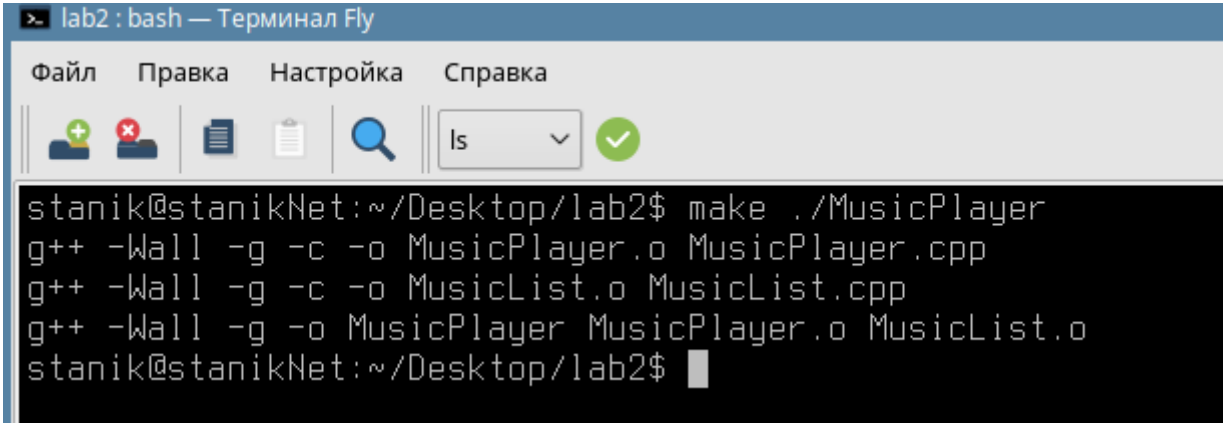
# Правило для компиляции каждого исходного файла .cpp в объектный файл .o
%.o: %.cpp

    # Команда для компиляции каждого исходного файла в объектный файл
    $(CC) $(CFLAGS) -c -o $@ $<

# Цель "clean" указывает, как удалить все созданные объектные файлы и целевой файл
clean:

    # Команда для удаления всех объектных файлов и целевого файла
```

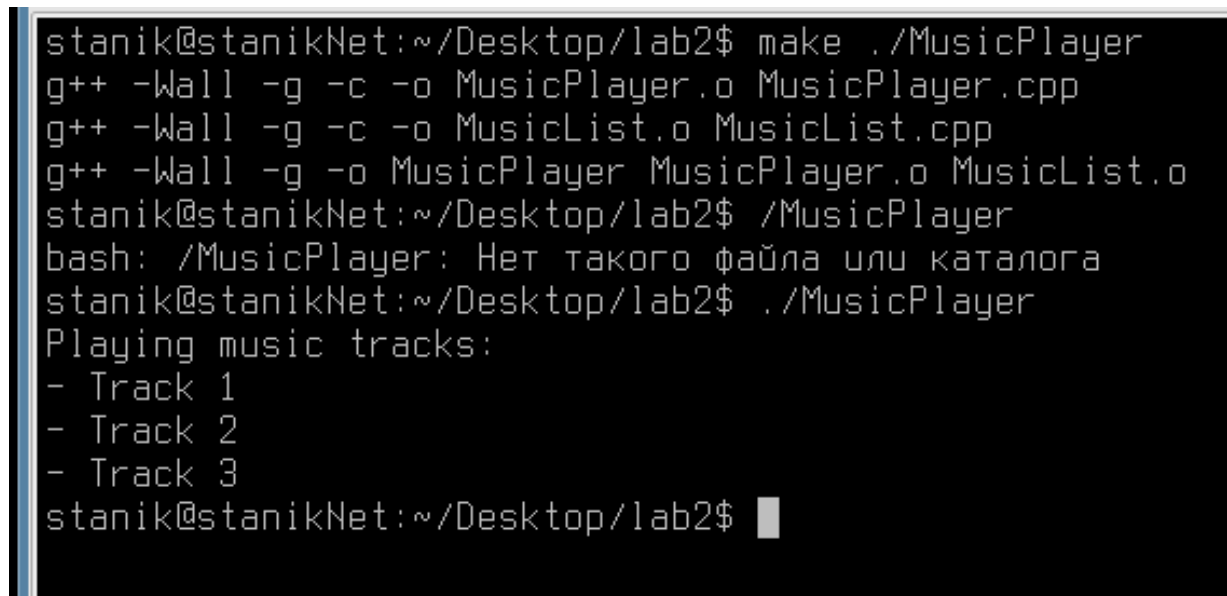
```
rm -f $(OBJS) $(TARGET)
```



```
stanik@stanikNet:~/Desktop/lab2$ make ./MusicPlayer
g++ -Wall -g -c -o MusicPlayer.o MusicPlayer.cpp
g++ -Wall -g -c -o MusicList.o MusicList.cpp
g++ -Wall -g -o MusicPlayer MusicPlayer.o MusicList.o
stanik@stanikNet:~/Desktop/lab2$
```

Рисунок 1. Пример выполнения команды make

Теперь сделаем сборку проекта, то есть выполним комплекс программ



```
stanik@stanikNet:~/Desktop/lab2$ make ./MusicPlayer
g++ -Wall -g -c -o MusicPlayer.o MusicPlayer.cpp
g++ -Wall -g -c -o MusicList.o MusicList.cpp
g++ -Wall -g -o MusicPlayer MusicPlayer.o MusicList.o
stanik@stanikNet:~/Desktop/lab2$ ./MusicPlayer
bash: ./MusicPlayer: Нет такого файла или каталога
stanik@stanikNet:~/Desktop/lab2$ ./MusicPlayer
Playing music tracks:
- Track 1
- Track 2
- Track 3
stanik@stanikNet:~/Desktop/lab2$
```

Рисунок 2. Пример выполнения

Теперь внесем изменение в один из исходных файлов. Для примера, добавим новый метод в класс MusicList(MusicList.cpp), а также объявим метод stop() (в MusicList.h):

```
#include <iostream>

#include <vector>

#include "MusicList.h"

void MusicList::addTrack(const std::string& trackName) {

    tracks.push_back(trackName);

}

void MusicList::play() {

    std::cout << "Playing music tracks:" << std::endl;
```



```

        for (const auto& track : tracks) {

            std::cout << "- " << track << std::endl;

        }

    }

}

void MusicList::stop() {

    std::cout << "Stopping music playback" << std::endl;

}

```

До повторного вызова make:

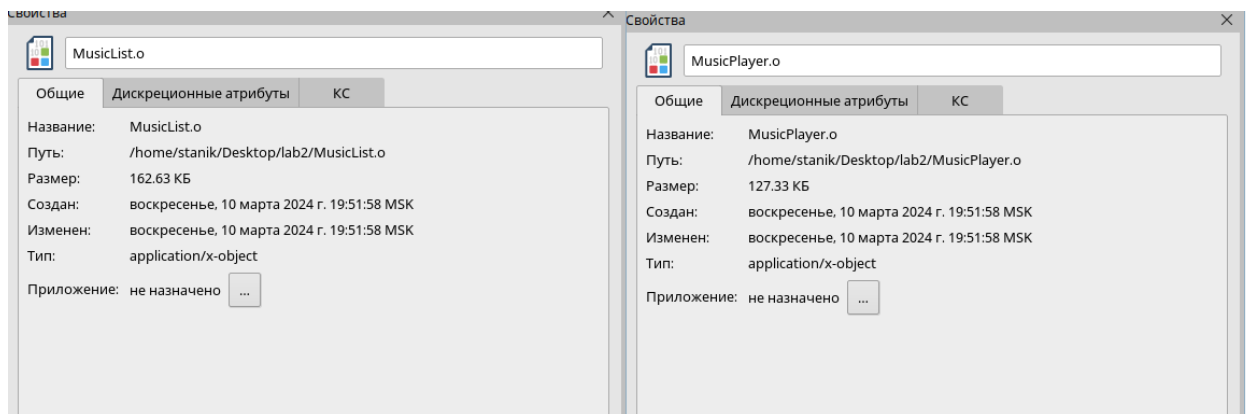


Рисунок 3. Свойства объектных файлов

Теперь вызовем make снова:

```

stanik@stanikNet:~/Desktop/lab2$ make
g++ -Wall -g -c -o MusicList.o MusicList.cpp
g++ -Wall -g -o MusicPlayer MusicPlayer.o MusicList.o
stanik@stanikNet:~/Desktop/lab2$ █

```

Рисунок 4. Повторный вызов команды make

Посмотрим на новые свойства объектных файлов:

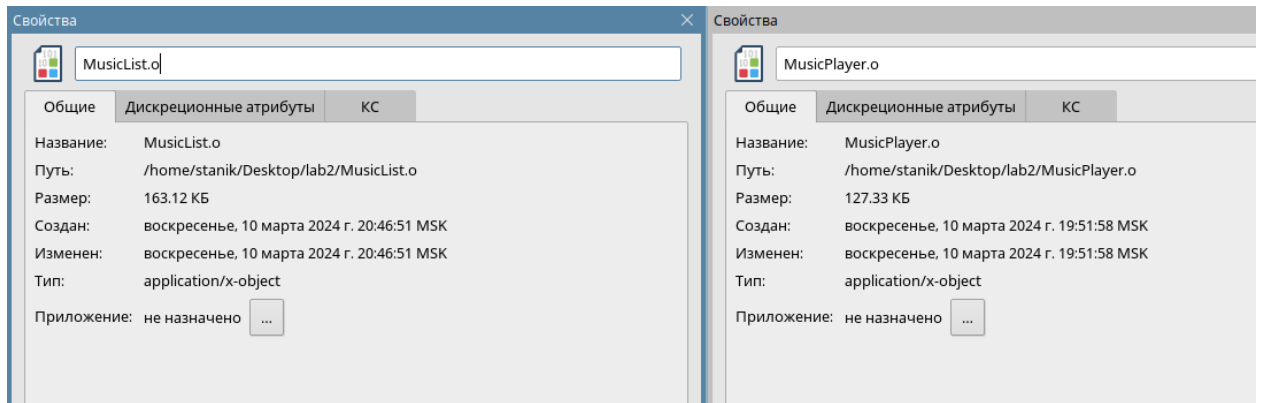


Рисунок 5. Новые свойства объектных файлов

MusicPlayer.o не изменился, так как мы его не редактировали, следовательно, он не стал перекомпилироваться, а вот MusicList.o стал весить чуть больше, из-за нового добавленного метода stop()

Подготовим почву для нового makefile скрипта

Директория проекта должна быть структурирована следующим образом:

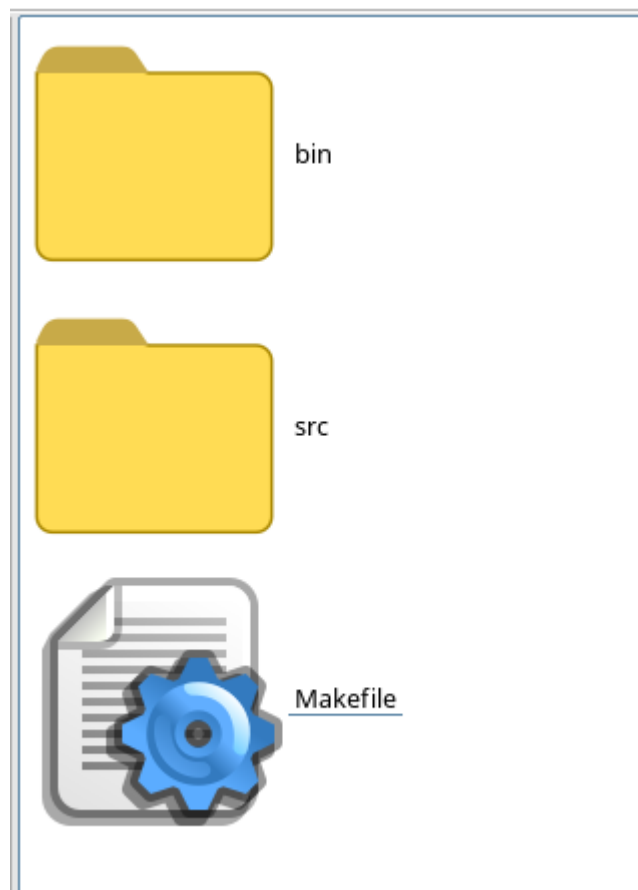


Рисунок 6. Директория проекта

В src будут .cpp и .h файлы, а в bin сгенерированные объектные файлы и сама программа

```
stanik@stanikNet:~/Desktop/lab2$ make
g++ -Wall -g -c -o bin/MusicList.o src/MusicList.cpp
g++ -Wall -g -c -o bin/MusicPlayer.o src/MusicPlayer.cpp
g++ -Wall -g -o bin/program bin/MusicList.o bin/MusicPlayer.o
```

Рисунок 7. Выполнение команды make для нового makefile скрипта

```
stanik@stanikNet:~/Desktop/lab2$ ./bin/program
Playing music tracks:
- Track 1
- Track 2
- Track 3
stanik@stanikNet:~/Desktop/lab2$
```

Рисунок 8. Запуск сгенерированного файла program из папки bin

```

1 # Переменные
2 CC = g++
3 CFLAGS = -Wall -g
4 SRC_DIR = src
5 BIN_DIR = bin
6 TARGET = $(BIN_DIR)/program
7 SRCS := $(wildcard $(SRC_DIR)/*.cpp)
8 OBJS := $(patsubst $(SRC_DIR)/%.cpp, $(BIN_DIR)/%.o, $(SRCS))
9
10 # Правило для цели по умолчанию
11 all: $(TARGET)
12
13 # Правило для компиляции объектных файлов
14 $(BIN_DIR)/%.o: $(SRC_DIR)/%.cpp | $(BIN_DIR)
15     $(CC) $(CFLAGS) -c -o $@ $<
16
17 # Правило для сборки исполняемого файла
18 $(TARGET): $(OBJS)
19     $(CC) $(CFLAGS) -o $@ $^
20
21 # Создание директории bin, если она отсутствует
22 $(BIN_DIR):
23     mkdir -p $(BIN_DIR)
24
25 # Цель для очистки временных файлов
26 clean:
27     rm -rf $(BIN_DIR)

```

Рисунок 9. Содержимое Makefile

6. Вывод

В ходе выполненной работы научился проводить сборку проекта при помощи утилиты make; скомпилировал проект, содержащий несколько скриптовых файлов.