**Тема**: Раздельная компиляция

1. **Формулировка задачи**

1) На языке C разработать функцию, реализующую определенную вариантом задания функциональность. Поместить определение функции в отдельный исходный файл, оформить заголовочный файл. Разработать тестовую программу на языке C.

2) Собрать программу «по шагам». Проанализировать выход препроцессора и компилятора. Проанализировать состав и содержимое секций, таблицы символов, таблицы перемещений и отладочную информацию, содержащуюся в объектных файлах исполняемом файле.

3) Выделить разработанную функцию в статическую библиотеку. Разработать make-файлы ля сборки библиотеки и использующей ее тестовой программы. Проанализировать ход сборки библиотеки и программы, созданные файлы зависимостей.

1. **Вариант задания**

В массиве чисел сделать перестановку – вначале все числа с нечетными индексами, потом – все с четными.

1. **Ход решения**
   1. **Текст программ, реализующих определенную вариантом задания функциональность**

#include <stdio.h>

int \*changeArray(int array[10]) {

static int myArray[10];

for (int i = 0; i < 10; i++) {

myArray[i] = array[i];

}

size\_t size = sizeof(myArray) / sizeof(myArray[0]);

int k = 0;

int temp;

for (int i = 1; i < size; i = i + 2) {

for (int j = i - 1; j >= k; j--) {

temp = myArray[j];

myArray[j] = myArray[j + 1];

myArray[j + 1] = temp;

}

k++;

}

for (int i = 0; i < size; i++) {

printf("%d ", myArray[i]);

}

return myArray;

}

int test(int expected[10], int real[10]) {

for (int i = 0; i < 10; i++) {

if (expected[i] != real[i]) {

return 0;

}

}

return 1;

}

int main()

{

int \*p;

int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

p = changeArray(array);

printf("\n0 = false, 1 = true");

int a[10] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};

printf("\n%d", test(a, p));

printf("\n%d", test(array, p));

}

Алгоритм работает путем сдвига каждого элемента с нечетным индексом перед всеми элементами с четными индексами. Таким образом все нечетные оказываются в начале, а четные индексы – в конце.   
 В файле «main.c» реализована функция changeArray() в которой и находится сам алгоритм преобразования массива чисел. Также реализована функция test(), которая сравнивает два числовых массива и выводит 1, если все сходится, или 0, если есть какое-то различие.

* 1. **Сборка программы**

Для сборки программы выполним следующую команду:

riscv64-unknown-elf-gcc --save-temps -march=rv32i -mabi=ilp32 -O1 -v main.c >log 2>&1

Программа *riscv64-unknown-elf-gcc* является драйвером компилятора gcc, в данном случае она запускается со следующими параметрами командной строки:

*--save-temps* – сохранять промежуточные файлы, создаваемые в процессе сборки;

*-march=rv32i -mabi=ilp32* – целевым является процессор с базовой архитектурой системы команд RV32I;

*-O1* – выполнять простые оптимизации генерируемого кода (мы используем эту опцию в примерах, потому что обычно генерируемый код получается более простым);

*-v* – печатать (в стандартный поток ошибок) выполняемые драйвером команды, а также дополнительную информацию.

В конце команды используется т.н. «перенаправление вывода»:

*>log* - вместо печати в консоли (обычно, это означает «на экране») вывод программы направляется в файл с именем “log” (если файл не существует, он создается; если файл существует, его содержимое будет утеряно);

*2>&1* – поток вывода ошибок (2 – стандартный «номер» этого потока) «связывается» с поток вывода («номер» 1), т.е. сообщения об ошибках (и информация, вывод которой вызван использованием флага “-v”, см.выше) также выводятся в файл “log”.

В результате исполнения приведенной команды в текущем каталоге будут созданы следующие файлы:

*a.out* – исполняемый файл, сгенерированный компоновщиком в результате сборки.

*log* – текстовый файл, содержащий сообщения компилятора, ассемблера и компоновщика, а также выполняемые команды и дополнительную информацию;

* 1. **Сборка программы по шагам**

Начнем сборку созданных программ на языке Cпо шагам. Первым шагом является препроцессирование файлf исходного текста “main.c” в файл “main.i”.



Драйвер компилятора gcc– riscv64-unknown-elf-gcc– запускается с параметрами командной строки “-march=rv32i -mabi=ilp32”, указывающих что целевым является процессор с базовой архитектурой системы команд RV32I;-O1 – указание выполнять простые оптимизации генерируемого кода; -E – указание остановить процесс сборки после препроцессирования.

В начале файла main.i содержится порядка 1200 строк с инструкциями по линковке stdio.h к проекту, а затем следует код на C, который мало отличаются от исходных версий программ:

…

# 797 "c:\\users\\user\\documents\\lab4files\\riscv64-unknown-elf-gcc-8.3.0-2020.04.1-x86\_64-w64-mingw32\\riscv64-unknown-elf\\include\\stdio.h" 3

# 10 "main.c" 2

# 11 "main.c"

int \*changeArray(int array[10]) {

static int myArray[10];

for (int i = 0; i < 10; i++) {

myArray[i] = array[i];

}

size\_t size = sizeof(myArray) / sizeof(myArray[0]);

int k = 0;

int temp;

for (int i = 1; i < size; i = i + 2) {

for (int j = i - 1; j >= k; j--) {

temp = myArray[j];

myArray[j] = myArray[j + 1];

myArray[j + 1] = temp;

}

k++;

}

for (int i = 0; i < size; i++) {

printf("%d ", myArray[i]);

}

return myArray;

}

int test(int expected[10], int real[10]) {

for (int i = 0; i < 10; i++) {

if (expected[i] != real[i]) {

return 0;

}

}

return 1;

}

int main()

{

int \*p;

int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

p = changeArray(array);

printf("\n0 = false, 1 = true");

int a[10] = {1, 3, 5, 7, 9, 0, 2, 4, 6, 8};

printf("\n%d", test(a, p));

printf("\n%d", test(array, p));

}

Появившиеся нестандартные директивы, начинающиеся с символа “#”, используются для передачи информации об исходном тексте из препроцессора в компилятор.

Следующим шагом является компиляция файла “main.i” в код на языке ассемблера “main.s”:



Драйвер компилятораriscv64-unknown-elf-gcc запускается с параметрами командной строки “-march=rv32i -mabi=ilp32”, указывающих что целевым является процессор с базовой архитектурой системы команд RV32I;-O1 – указание выполнять простые оптимизации генерируемого кода; -S – указание остановить процесс сборки после компиляции (без запуска ассемблера).

.file "main.c"

.option nopic

.attribute arch, "rv32i2p0"

.attribute unaligned\_access, 0

.attribute stack\_align, 16

.text

.align 2

.globl changeArray

.type changeArray, @function

changeArray:

addi sp,sp,-16

sw ra,12(sp)

sw s0,8(sp)

sw s1,4(sp)

sw s2,0(sp)

lui a5,%hi(.LANCHOR0)

addi s0,a5,%lo(.LANCHOR0)

addi s1,s0,40

addi a5,a5,%lo(.LANCHOR0)

.L2:

lw a4,0(a0)

sw a4,0(a5)

addi a0,a0,4

addi a5,a5,4

bne a5,s1,.L2

lui a0,%hi(.LANCHOR0+4)

addi a0,a0,%lo(.LANCHOR0+4)

mv a2,s0

li a1,0

li a6,0

li a7,10

.L5:

bgt a6,a1,.L3

mv a5,a0

.L4:

lw a4,-4(a5)

lw a3,0(a5)

sw a3,-4(a5)

sw a4,0(a5)

addi a5,a5,-4

bne a5,a2,.L4

.L3:

addi a6,a6,1

addi a1,a1,2

addi a0,a0,8

addi a2,a2,4

bne a1,a7,.L5

lui s2,%hi(.LC2)

.L6:

lw a1,0(s0)

addi a0,s2,%lo(.LC2)

call printf

addi s0,s0,4

bne s0,s1,.L6

lui a0,%hi(.LANCHOR0)

addi a0,a0,%lo(.LANCHOR0)

lw ra,12(sp)

lw s0,8(sp)

lw s1,4(sp)

lw s2,0(sp)

addi sp,sp,16

jr ra

.size changeArray, .-changeArray

.align 2

.globl test

.type test, @function

test:

lw a4,0(a0)

lw a5,0(a1)

bne a4,a5,.L15

addi a5,a0,4

addi a1,a1,4

addi a0,a0,40

.L14:

lw a3,0(a5)

lw a4,0(a1)

bne a3,a4,.L16

addi a5,a5,4

addi a1,a1,4

bne a5,a0,.L14

li a0,1

ret

.L15:

li a0,0

ret

.L16:

li a0,0

ret

.size test, .-test

.align 2

.globl main

.type main, @function

main:

addi sp,sp,-96

sw ra,92(sp)

sw s0,88(sp)

sw s1,84(sp)

lui s0,%hi(.LANCHOR1)

addi s0,s0,%lo(.LANCHOR1)

lw t3,0(s0)

lw t1,4(s0)

lw a7,8(s0)

lw a6,12(s0)

lw a0,16(s0)

lw a1,20(s0)

lw a2,24(s0)

lw a3,28(s0)

lw a4,32(s0)

lw a5,36(s0)

sw t3,40(sp)

sw t1,44(sp)

sw a7,48(sp)

sw a6,52(sp)

sw a0,56(sp)

sw a1,60(sp)

sw a2,64(sp)

sw a3,68(sp)

sw a4,72(sp)

sw a5,76(sp)

addi a0,sp,40

call changeArray

mv s1,a0

lui a0,%hi(.LC3)

addi a0,a0,%lo(.LC3)

call printf

lw t3,40(s0)

lw t1,44(s0)

lw a7,48(s0)

lw a6,52(s0)

lw a0,56(s0)

lw a1,60(s0)

lw a2,64(s0)

lw a3,68(s0)

lw a4,72(s0)

lw a5,76(s0)

sw t3,0(sp)

sw t1,4(sp)

sw a7,8(sp)

sw a6,12(sp)

sw a0,16(sp)

sw a1,20(sp)

sw a2,24(sp)

sw a3,28(sp)

sw a4,32(sp)

sw a5,36(sp)

mv a1,s1

mv a0,sp

call test

mv a1,a0

lui s0,%hi(.LC4)

addi a0,s0,%lo(.LC4)

call printf

mv a1,s1

addi a0,sp,40

call test

mv a1,a0

addi a0,s0,%lo(.LC4)

call printf

li a0,0

lw ra,92(sp)

lw s0,88(sp)

lw s1,84(sp)

addi sp,sp,96

jr ra

.size main, .-main

.section .rodata

.align 2

.set .LANCHOR1,. + 0

.LC0:

.word 0

.word 1

.word 2

.word 3

.word 4

.word 5

.word 6

.word 7

.word 8

.word 9

.LC1:

.word 1

.word 3

.word 5

.word 7

.word 9

.word 0

.word 2

.word 4

.word 6

.word 8

.bss

.align 2

.set .LANCHOR0,. + 0

.type myArray.2550, @object

.size myArray.2550, 40

myArray.2550:

.zero 40

.section .rodata.str1.4,"aMS",@progbits,1

.align 2

.LC2:

.string "%d "

.LC3:

.string "\n0 = false, 1 = true"

.zero 3

.LC4:

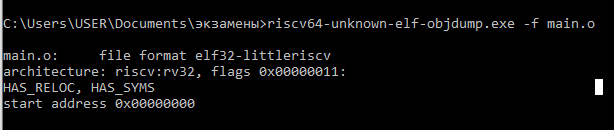
.string "\n%d"

Следующим шагом является ассемблирование файла “main.s”в объектный файл “main.o”:



Драйвер компилятораriscv64-unknown-elf-gcc запускается с параметрами командной строки “-march=rv32i -mabi=ilp32”, указывающих что целевым является процессор с базовой архитектурой системы команд RV32I; -c – указание остановить процесс сборки после ассемблирования.

Объектный файл не является текстовым и не может быть напрямую выведен на экран в читаемом формате, для изучения его содержимого используем утилиту objdump:



Файл имеет формат ELF, является объектным файлом 32-разрядной архитектуры RISC-V, содержит символы (флаг HAS\_SYMS), содержит таблицу перемещений (флаг HAS\_RELOC).

Компоновка

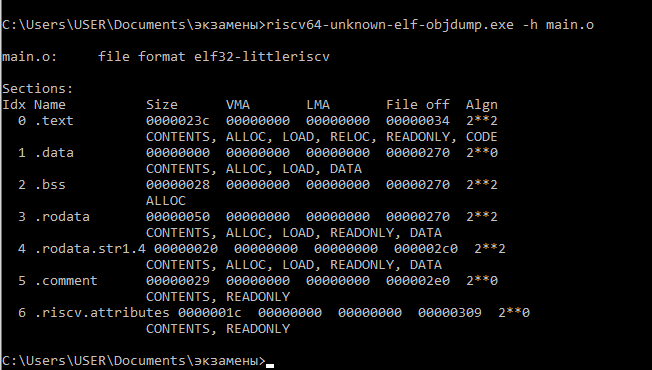
Компоновка программы выполняется по следующей команде:



Результатом является исполняемый файл *“a.out”*

**Объектный файл**

Как известно, содержательная часть объектного файла разбита на «разделы», называемые обычно секциями. Следующая команда обеспечивает отображение заголовков секций файла ”gray.o”:



В файле “main*.o”* имеются следующие секции:

*.text* – секция кода, в которой содержатся коды инструкций (название секции обусловлено историческими причинами);

.*data* – секция инициализированных данных;

*.bss* – секция неинициализированных статических переменных (название секции также обусловлено историческими причинами);

*.rodata* – аналог .data для неизменяемых данных

*.comment* – секция данных о версиях размером 12 байт

.riscv.attributes – информация про RISC-V

Изучим таблицу символов файла:

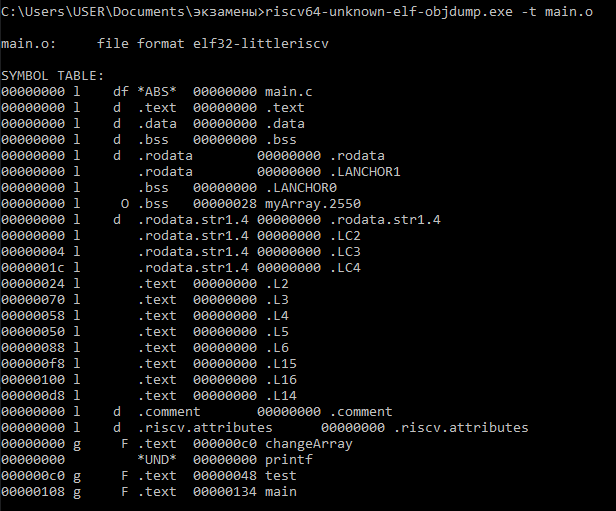


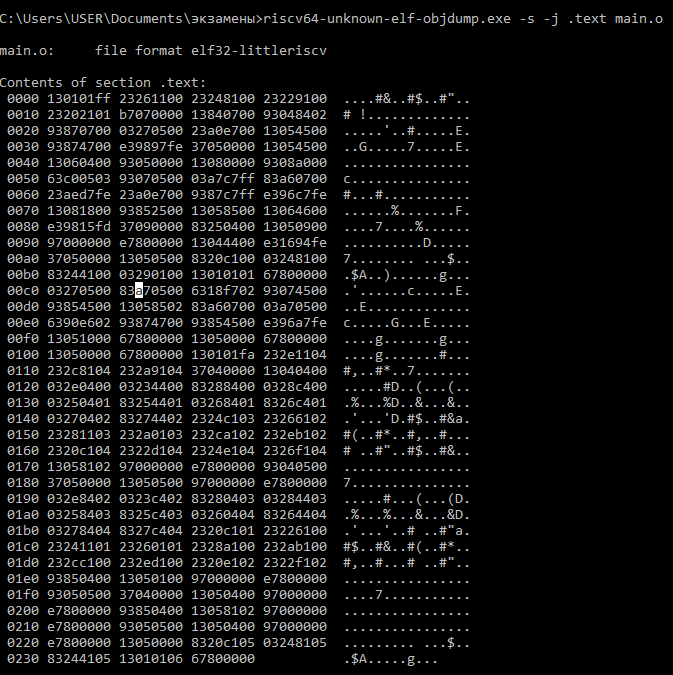
Таблица содержит 3 глобальные (флаг g) функции (флаг F) – main, changeArray, test, а также один неопределённый символ (UND).

UND означает, что символ printf использовался в ассемблерном коде, из которого был получен данный объектный фал, но не был определён;

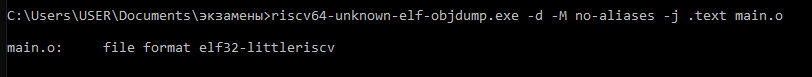
Ассемблер сделал вывод о том, что символ должен быть определён где-то ещё, и отразил это в таблице символов.

**Секция .text**

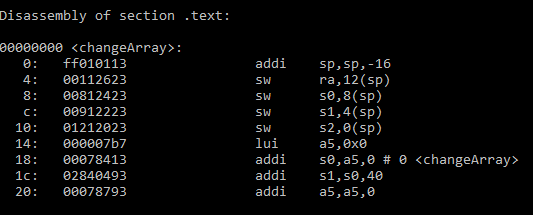
Изучим содержимое секции “.text”:

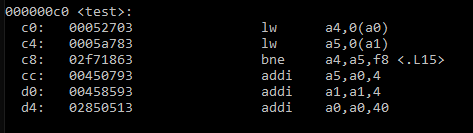


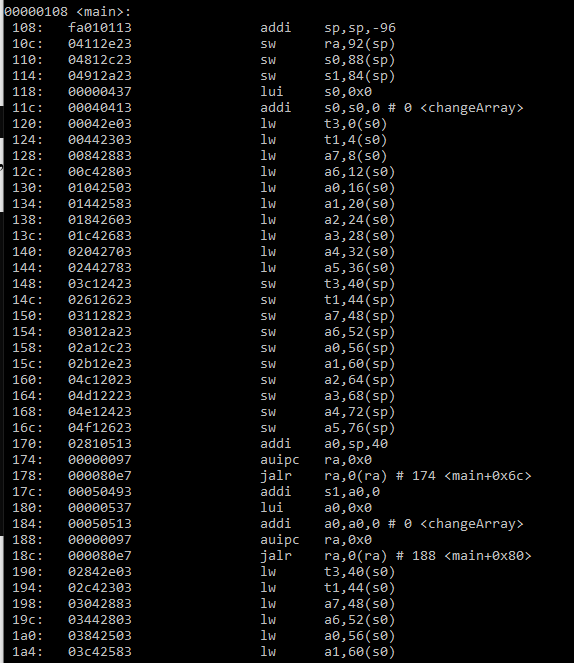
Разумеется, процедура декодирования кодов инструкций является «механической» (иначе как бы ее реализовывало техническое устройство – процессор), следовательно, разумно поручить ее выполнение ЭВМ:

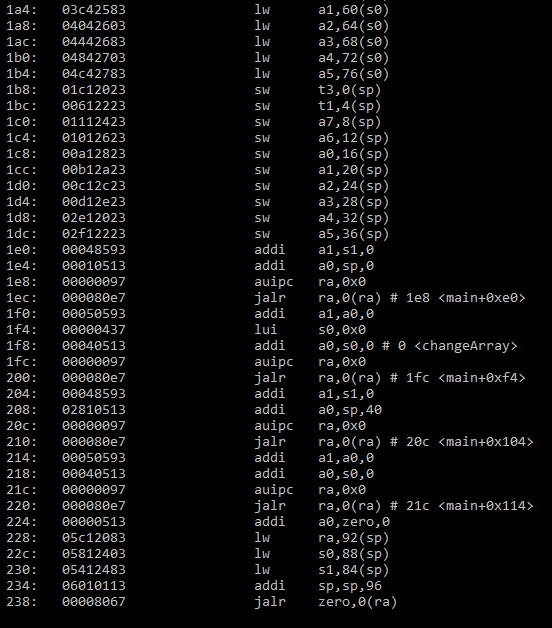


Опция *“-d”* инициирует процесс дизассемблирования, опция *“-M no-aliases”* требует использовать в выводе только инструкции системы команд (но не псевдоинструкции ассемблера). Секция кода теперь содержит намного большее количество строк, поэтому рассмотрим только самые важные участки:

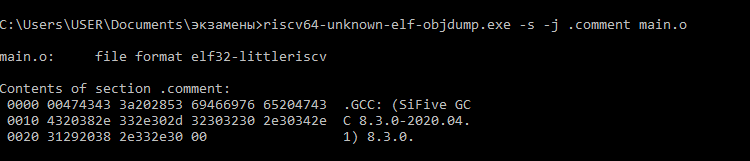








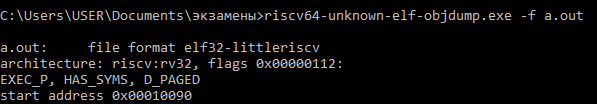
**Секция .comment**

****

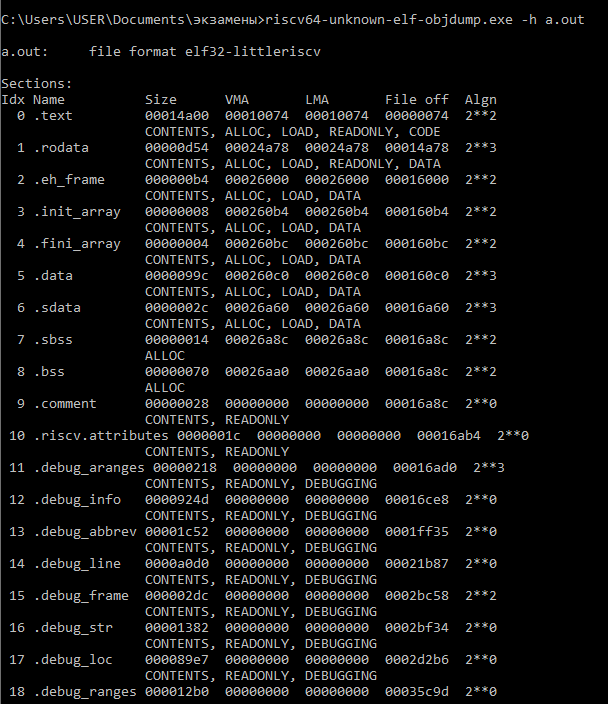
В секции .comment, записаны инициалы компилятора, которым проводилась процедура.

**Вывод компоновщика – исполняемый файл**

Сформированный компоновщиком файл *“a.out”*, разумеется, также является «бинарным», и для изучения его содержимого будем пользоваться утилитой *objdump*:

****

Изучим секции файла:

****

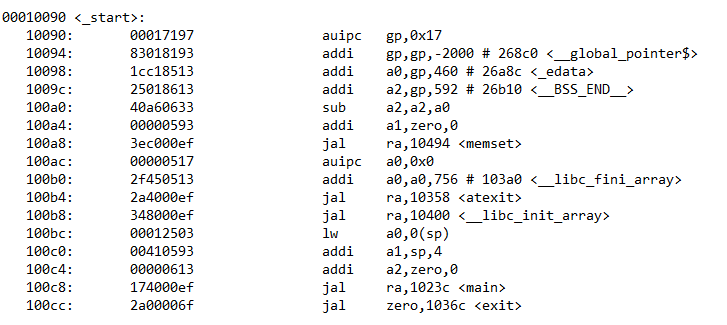
Состав секций *“a.out”* значительно расширен по сравнению с *“main.o”*. Также увеличились размеры секций *“.text”*, *“.data”*, *“.bss”* и *“.comment”*. Дополнительные секции появились из других объектных файлов, переданных на вход компоновщика.

**Инструкции программы**

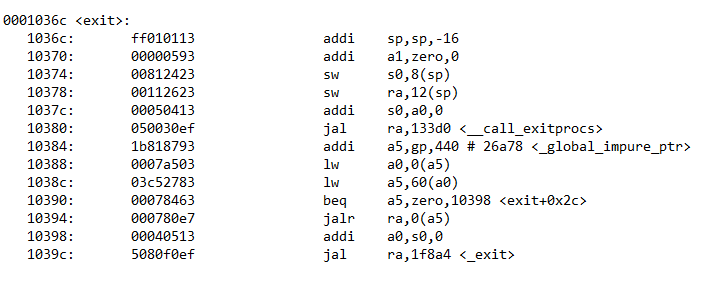
Изучим содержимое секции “.text”:



В результате выполнения получили файл *“a.ds”*. Изучим его.



“\_start” – “точка входа” в нашу программу. Код, начинающийся с метки “\_start” обеспечивает инициализацию памяти, регистров процессора и среды времени выполнения, после чего передаёт управление определённой нами функции main.

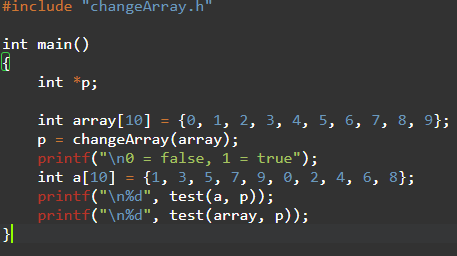


Можно видеть, что в конце *“exit”* управление передается на символ *“\_exit”*

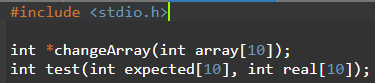
**Раздельная компиляция**

Разобьём исходную программу на 3 файла.

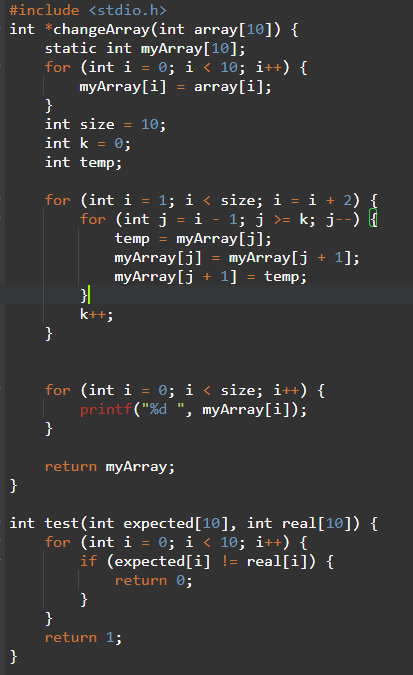
Main.c



changeArray.h

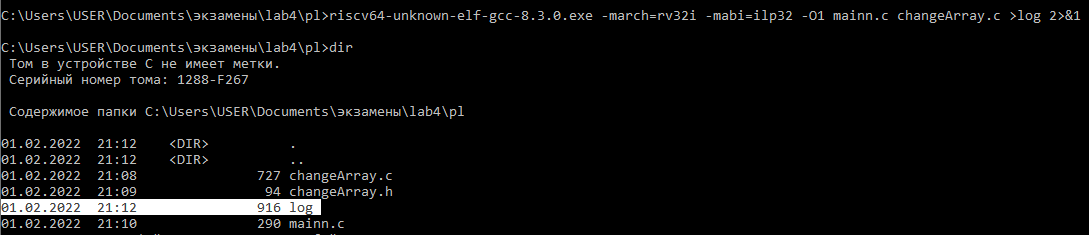


changeArray.c



**Сборка программы**

Сборка программы осуществляется следующей командой:



Ранее препроцессирование, компиляция и ассемблирование выполнялось нами по шагам, но на практике это требуется редко, обычно необходимо выполнить все стадии обработки исходного файла, получив в результате объектный файл.



Параметры:

* *“-c”* - приводит к останову процесса сборки после ассемблирования, т.е. после формирования объектного файла

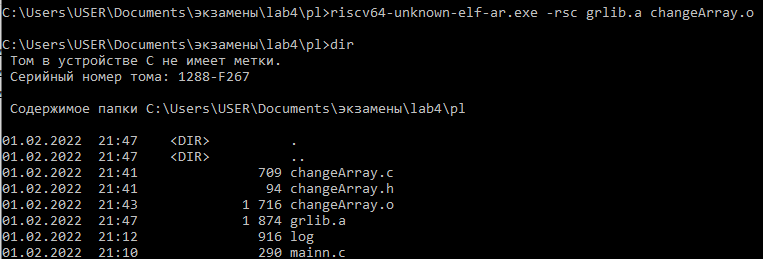
После создания объектного файла его нужно скомпоновать с нашей программой



**Создание и использование статической библиотеки**

Статическая библиотека (static library) является, по сути, архивом (набором, коллекцией) объектных файлов, среди которых компоновщик выбирает «полезные» для данной программы: объектный файл считается «полезным», если в нем определяется еще не разрешенный компоновщиком символ.

Поместим changeArray.o в такой архив:

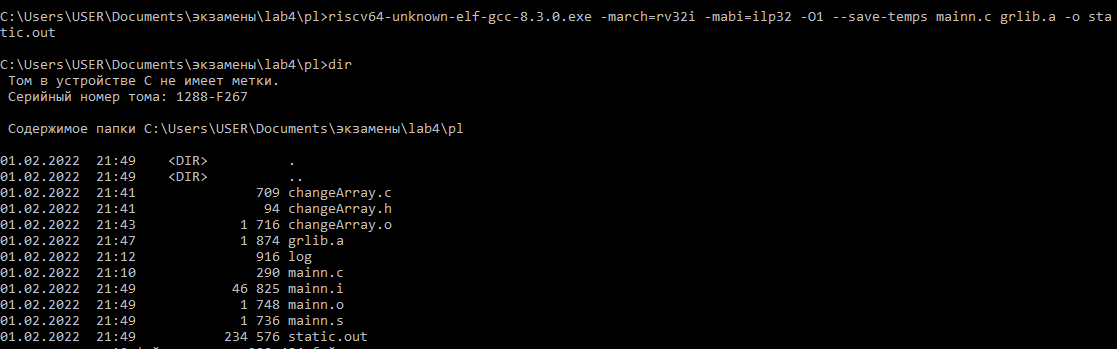


Параметры:

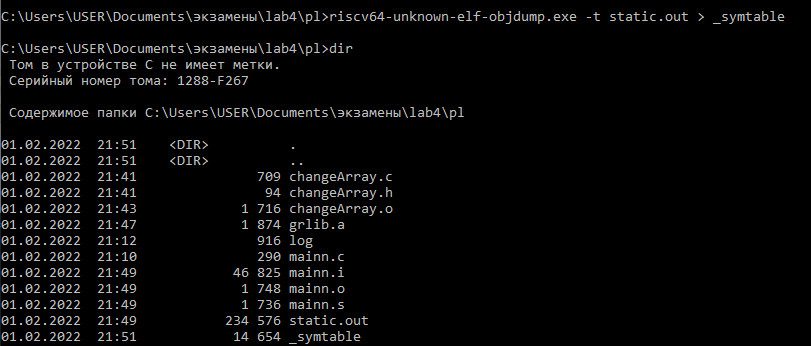
* -r – заменить старые файлы с такими названиями (*changeArray.o*), если они уже есть в архиве
* -s – записать «index» в архив. Index – это список всех символов, объявленных во включенных в архив объектных файлах, и его присутствие ускоряет линковку
* -с – создать архив, если его еще не было

Результирующим файлом является “grlib.a” (“.a” – от “archive”).

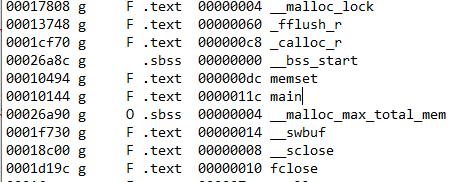
Используем статическую библиотеку для сборки программ:

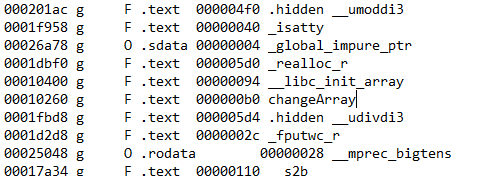


Изучим таблицы символов полученных исполняемых файлов:



Содержимое \_symtable





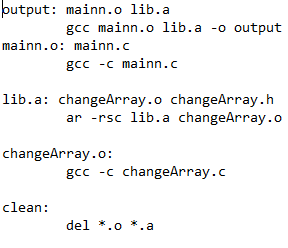
Видим, что все нужные символы вошли в исполняемый файл.

Несмотря на то, что в нашем случае компоновщик не используется, преимущества использования библиотеки очевидны: при компоновке были использованы необходимые объектные файлы и только они, причем задача выбора необходимых для сборки объектных файлов была возложена на компоновщик (а не нас).

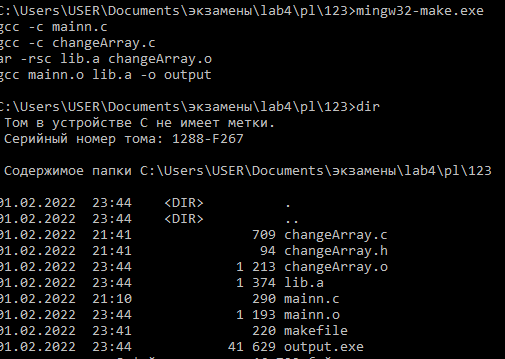
**Makefile**

Makefile - это набор инструкций для программы make, которая позволяет собирать проекты, состоящие из большого числа “\*.c” и “\*.h” файлов. Обычно эта программа используется в связке с системами сборки, например cmake, позволяя вести проекты модульно (т.е. проект с включенными подпроектами).

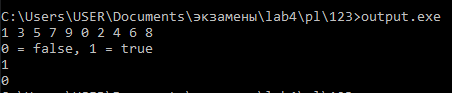
Текст makefile:



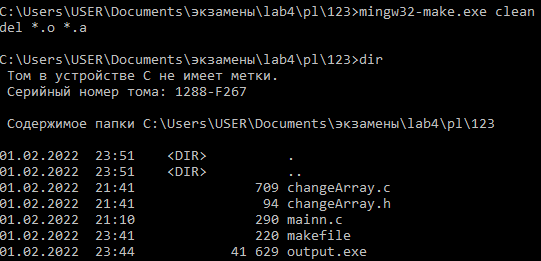
Сборка с помощью Makefile:



Демонстрация работы программы:



Очистка



Что происходит в Makefile:

1. Создаём объектный файл *mainn.o* из исходного *mainn.c*
2. Создаём объектный файл *changeArray.o* из исходного *changeArray.c*
3. Архивируем объектный файл changeArray.o (создаём статическую библиотеку *lib.a*)
4. Компонуем статическую библиотеку *lib.a* с объектным файлом *mainn.o* и получаем исполняемый файл *output*

ВЫВОД

В данной лабораторной работе мы познакомились с процессом сборки проекта на языке C.

Он состоит из:

1. **Препроцессирования**: исходного *.c* файл препроцессируем в *.i* файл
2. **Компиляции**: полученный *.i* файл компилируется в файл ассемблера *.s*
3. **Ассемблирования**: файл *.s* асссемблируется в объектный файл *.o*
4. **Компоновки**: объектный файл *.o* компонуется в исполняемый файл

Также мы ознакомились в *makefile’*ами*,* которые упрощают процесс сборки.

Утилита Make позволяет собирать проекты, состоящие из большого количества файлов, вместо использования PS/SH скриптов, и прописывания файлов вручную.