

Отчёт по лабораторной работе №13

**Средства, применяемые при разработке программного обеспечения в
ОС типа UNIX/Linux**

Желдакова Виктория Алексеевна

Содержание

1	Цель работы	5
2	Теоретическое введение	6
3	Выполнение лабораторной работы	7
4	Выводы	13
5	Контрольные вопросы	14

Список иллюстраций

3.1	Содержимое файла calculate.c	7
3.2	Компиляция программы	8
3.3	Содержимое Makefile	8
3.4	Запуск отладчика и программы внутри него	9
3.5	Просмотр исходного кода	9
3.6	Просмотр содержимого не основного файла и установка точки останова	10
3.7	Вывод информации о точках останова, запуск программы, вывод стека функций, вывод значения переменной, удаление точки останова и проверка	11
3.8	Анализ кода calculate.c	12
3.9	Анализ кода main.c	12

Список таблиц

1 Цель работы

Приобрести простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

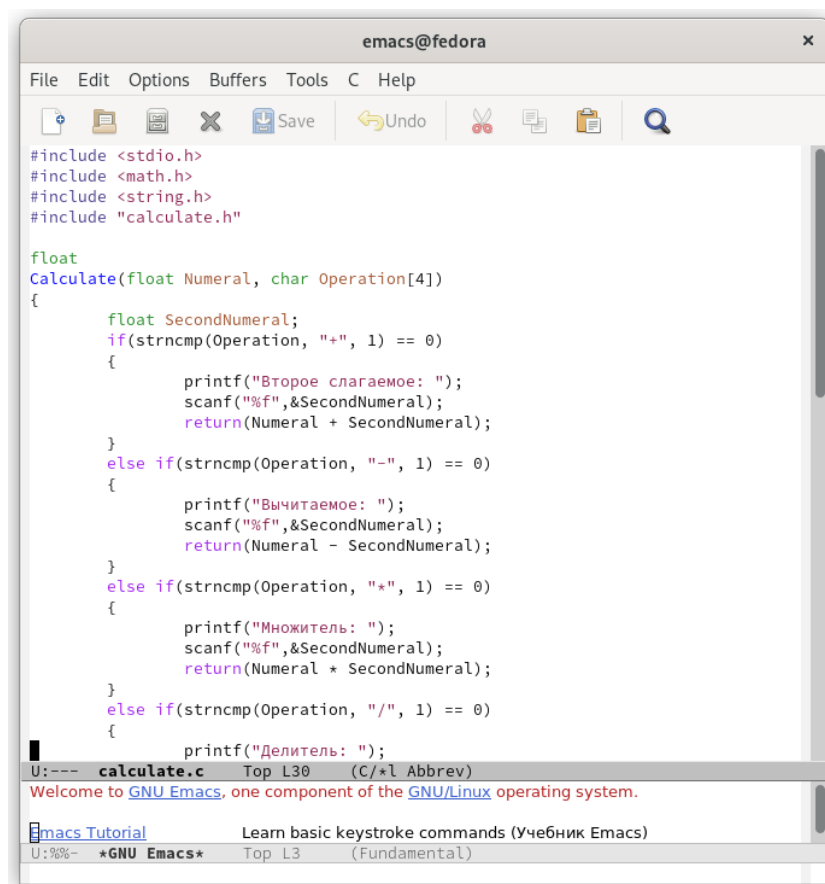
2 Теоретическое введение

GNU Debugger — переносимый отладчик проекта GNU, который работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования, включая Си, C++, Free Pascal, FreeBASIC, Ada, Фортран и Rust. GDB — свободное программное обеспечение, распространяемое по лицензии GPL.

Первоначально написан Ричардом Столлманом в 1988 году. За основу был взят отладчик DBX, поставлявшийся с дистрибутивом BSD. С 1990 до 1993 гг. проект поддерживался Джоном Джилмором, во время его работы в компании Cygnus Solutions. В настоящее время разработка координируется Управляющим комитетом GDB (GDB Steering Committee), назначенным Free Software Foundation.

3 Выполнение лабораторной работы

В домашнем каталоге создали подкаталог `~/work/os/lab_prog` и в нём файлы `calculate.h`, `calculate.c`, `main.c` (рис. 3.1)



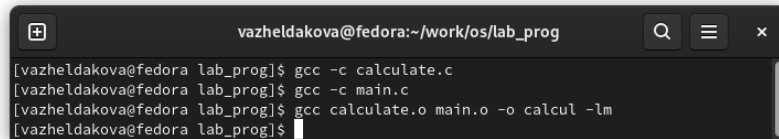
```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

float
Calculate(float Numeral, char Operation[4])
{
    float SecondNumeral;
    if(strcmp(Operation, "+", 1) == 0)
    {
        printf("Второе слагаемое: ");
        scanf("%f",&SecondNumeral);
        return(Numeral + SecondNumeral);
    }
    else if(strcmp(Operation, "-", 1) == 0)
    {
        printf("Вычитаемое: ");
        scanf("%f",&SecondNumeral);
        return(Numeral - SecondNumeral);
    }
    else if(strcmp(Operation, "*", 1) == 0)
    {
        printf("Множитель: ");
        scanf("%f",&SecondNumeral);
        return(Numeral * SecondNumeral);
    }
    else if(strcmp(Operation, "/", 1) == 0)
    {
        printf("Делитель: ");
        scanf("%f",&SecondNumeral);
        return(Numeral / SecondNumeral);
    }
}
```

U:--- calculate.c Top L30 (C/*l Abbrev)
Welcome to GNU Emacs, one component of the GNU/Linux operating system.
Emacs Tutorial Learn basic keystroke commands (Учебник Emacs)
U:%%~ *GNU Emacs* Top L3 (Fundamental)

Рис. 3.1: Содержимое файла `calculate.c`

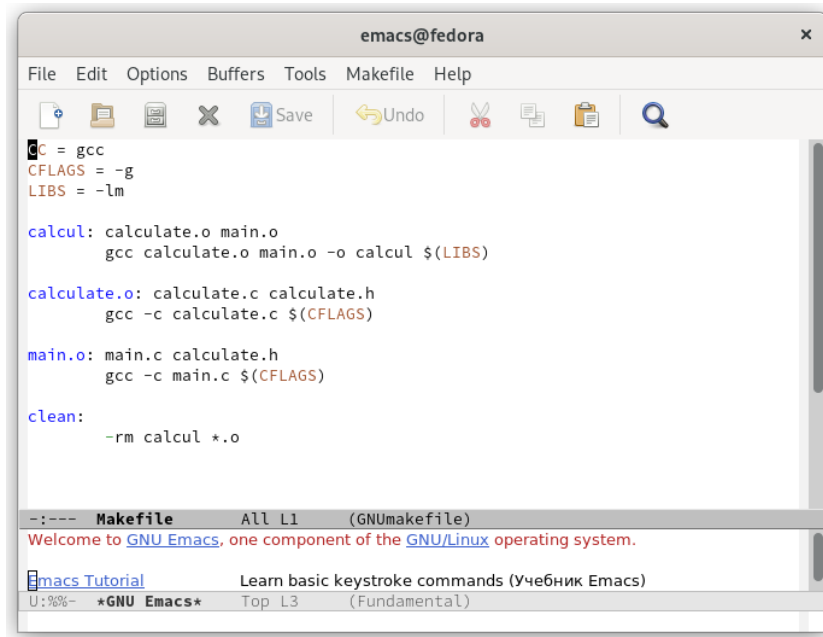
Выполнили компиляцию программы посредством `gcc` (рис. 3.2)

A terminal window titled 'vazheldakova@fedora:~/work/os/lab_prog' showing the compilation of a program. The commands entered are: 'gcc -c calculate.c', 'gcc -c main.c', and 'gcc calculate.o main.o -o calcul -lm'.

```
vazheldakova@fedora:~/work/os/lab_prog
[vazheldakova@fedora lab_prog]$ gcc -c calculate.c
[vazheldakova@fedora lab_prog]$ gcc -c main.c
[vazheldakova@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[vazheldakova@fedora lab_prog]$
```

Рис. 3.2: Компиляция программы

Создали Makefile (рис. 3.3)

An Emacs editor window titled 'emacs@fedora' showing the content of a Makefile. The Makefile defines variables for compiler, flags, and libraries, and includes rules for building 'calcul', 'calculate.o', 'main.o', and a 'clean' target.

```
emacs@fedora
File Edit Options Buffers Tools Makefile Help
[Icons] Save Undo [Icons]
C = gcc
CFLAGS = -g
LIBS = -lm

calcul: calculate.o main.o
    gcc calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
    gcc -c calculate.c $(CFLAGS)

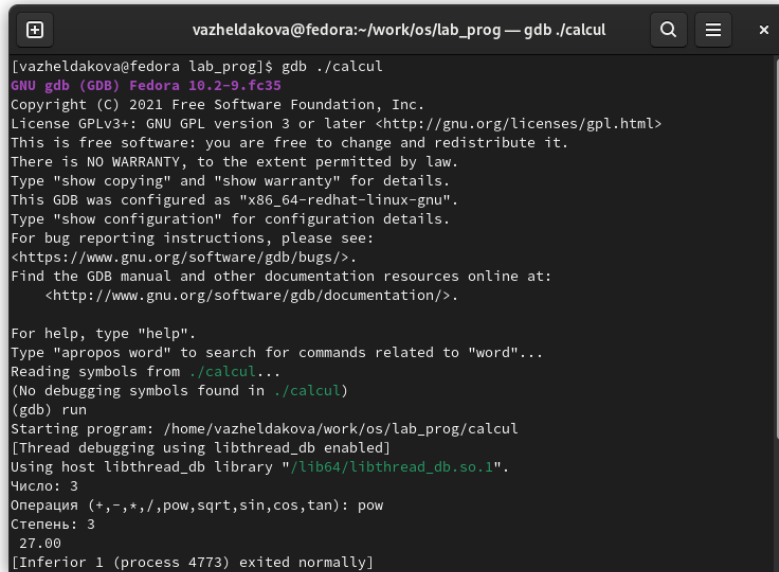
main.o: main.c calculate.h
    gcc -c main.c $(CFLAGS)

clean:
    -rm calcul *.o

--- Makefile All L1 (GNUmakefile)
Welcome to GNU Emacs, one component of the GNU/Linux operating system.
Emacs Tutorial Learn basic keystroke commands (Учебник Emacs)
U:%%- *GNU Emacs* Top L3 (Fundamental)
```

Рис. 3.3: Содержимое Makefile

Запустили GDB, загрузив в него программу для отладки и для запуска программы внутри отладчика использовали команду run (рис. 3.4)

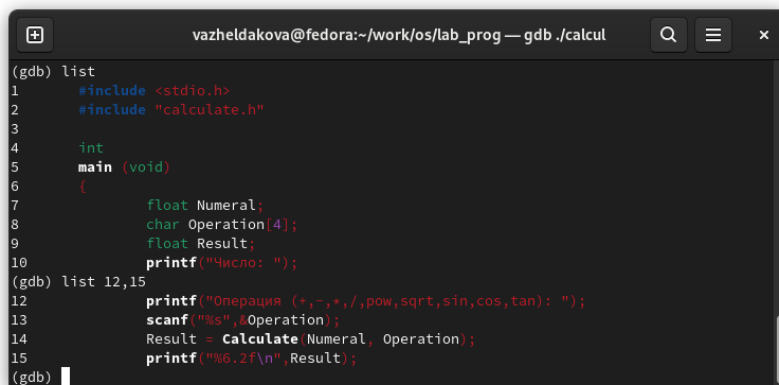


```
[vazheldakova@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 10.2-9.fc35
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./calcul...
(No debugging symbols found in ./calcul)
(gdb) run
Starting program: /home/vazheldakova/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 3
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): pow
Степень: 3
27.00
[Inferior 1 (process 4773) exited normally]
```

Рис. 3.4: Запуск отладчика и программы внутри него

Для постраничного вывода кода использовали команду `list`, затем просмотрели строки с 12 по 15 (рис. 3.5)



```
(gdb) list
1      #include <stdio.h>
2      #include "calculate.h"
3
4      int
5      main (void)
6      {
7          float Numeral;
8          char Operation[4];
9          float Result;
10         printf("Число: ");
(gdb) list 12,15
12         printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
13         scanf("%s",&Operation);
14         Result = Calculate(Numeral, Operation);
15         printf("%6.2f\n",Result);
(gdb)
```

Рис. 3.5: Просмотр исходного кода

Просмотрели содержимое файла `calculate.c`, используя команду `list` с параметрами и установили точку останова на 21 строке (рис. 3.6)

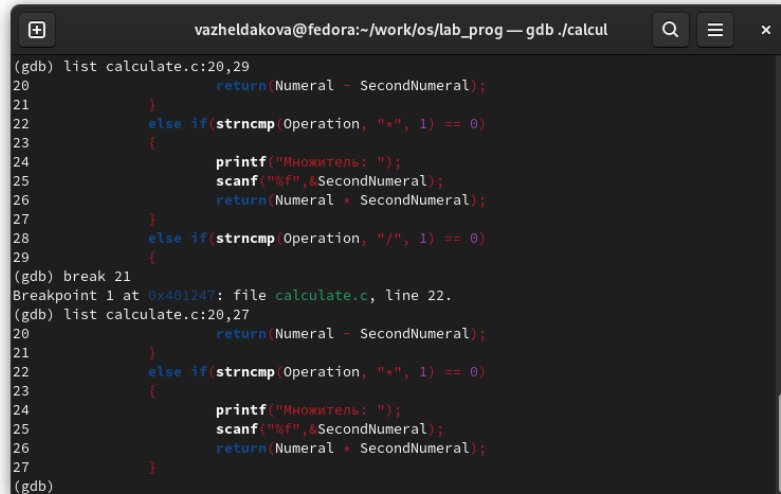
The image shows a GDB terminal window with a dark background. The title bar at the top reads 'vazheldakova@fedora:~/work/os/lab_prog — gdb ./calcul'. The terminal content shows the following sequence of commands and output:
(gdb) list calculate.c:20,29
20 return(Numeral - SecondNumeral);
21 }
22 else if strcmp(Operation, "+", 1) == 0)
23 {
24 printf("Множитель: ");
25 scanf("%f",&SecondNumeral);
26 return(Numeral * SecondNumeral);
27 }
28 else if strcmp(Operation, "/", 1) == 0)
29 {
(gdb) break 21
Breakpoint 1 at 0x401247: file calculate.c, line 22.
(gdb) list calculate.c:20,27
20 return(Numeral - SecondNumeral);
21 }
22 else if strcmp(Operation, "+", 1) == 0)
23 {
24 printf("Множитель: ");
25 scanf("%f",&SecondNumeral);
26 return(Numeral * SecondNumeral);
27 }
(gdb)

Рис. 3.6: Просмотр содержимого не основного файла и установка точки останова

Вывели информацию о всех точках останова, запустили программу и убедились, что программа останавливается в момент прохождения точки останова, использовали команду Backtrace для вывода всего стека вызываемых функций, вывели значение переменной Numeral с помощью команд print и display, удалили точку останова и вывели информацию о имеющихся точках останова (рис. 3.7)

```
vazheldakova@fedora:~/work/os/lab_prog — gdb ./calcul
(gdb) info breakpoints
Num   Type       Disp Enb Address            What
1      breakpoint keep y   0x0000000000401247 in Calculate
                                at calculate.c:22

(gdb) run
Starting program: /home/vazheldakova/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
5Число:5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): /

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdf14 "/")
at calculate.c:22
22      else if(strncmp(Operation, "+", 1) == 0)
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffdf14 "/") at calculate.c:22
#1 0x00000000004014eb in main () at main.c:14
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)
```

Рис. 3.7: Вывод информации о точках останова, запуск программы, вывод стека функций, вывод значения переменной, удаление точки останова и проверка

С помощью утилиты splint проанализировали коды файлов calculate.c и main.c (рис. 3.8 и рис. 3.9)

```
vazheldakova@fedora:~/work/os/lab_prog
[vazheldakova@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:4:37: Function parameter Operation declared as manifest array (size
constant is meaningless)
A formal parameter is declared as an array with size. The size of the array
is ignored in this context, since the array formal parameter is treated as a
pointer. (Use -fixedformalarray to inhibit warning)
calculate.c:7:31: Function parameter Operation declared as manifest array (size
constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:13:3: Return value (type int) ignored: scanf("%f", &Sec...
Result returned by function call is not used. If this is intended, can cast
result to (void) to eliminate message. (Use -retvalint to inhibit warning)
calculate.c:19:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:25:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:31:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:32:6: Dangerous equality comparison involving float types:
SecondNumeral == 0
Two real (float, double, or long double) values are compared directly using
== or != primitive. This may produce unexpected results since floating point
representations are inexact. Instead, compare the difference to FLT_EPSILON
or DBL_EPSILON. (Use -realcompare to inhibit warning)
calculate.c:35:10: Return value type double does not match declared type float:
(HUGE_VAL)
To allow all numeric types to match, use +relaxtypes.
calculate.c:43:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:44:10: Return value type double does not match declared type float:
(pow(Numeral, SecondNumeral))
calculate.c:47:9: Return value type double does not match declared type float:
(sqrt(Numeral))
calculate.c:49:9: Return value type double does not match declared type float:
(sin(Numeral))
calculate.c:51:10: Return value type double does not match declared type float:
(cos(Numeral))
calculate.c:53:9: Return value type double does not match declared type float:
(tan(Numeral))
calculate.c:57:9: Return value type double does not match declared type float:
(HUGE_VAL)

Finished checking --- 15 code warnings
[vazheldakova@fedora lab_prog]$
```

Рис. 3.8: Анализ кода calculate.c

```
vazheldakova@fedora:~/work/os/lab_prog
[vazheldakova@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:4:37: Function parameter Operation declared as manifest array (size
constant is meaningless)
A formal parameter is declared as an array with size. The size of the array
is ignored in this context, since the array formal parameter is treated as a
pointer. (Use -fixedformalarray to inhibit warning)
main.c: (in function main)
main.c:11:2: Return value (type int) ignored: scanf("%f", &Num...
Result returned by function call is not used. If this is intended, can cast
result to (void) to eliminate message. (Use -retvalint to inhibit warning)
main.c:13:13: Format argument 1 to scanf (%s) expects char * gets char [4] *:
&Operation
Type of parameter is not consistent with corresponding code in format string.
(Use -formattype to inhibit warning)
main.c:13:10: Corresponding format code
main.c:13:2: Return value (type int) ignored: scanf("%s", &Ope...

Finished checking --- 4 code warnings
[vazheldakova@fedora lab_prog]$
```

Рис. 3.9: Анализ кода main.c

4 Выводы

Приобрели простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

5 Контрольные вопросы

1. Как получить информацию о возможностях программ gcc, make, gdb и др.?

Дополнительную информацию о этих программах можно получить с помощью функций `info` и `man`.

2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.

Unix поддерживает следующие основные этапы разработки приложений:

- планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
- непосредственная разработка приложения;
- кодирование — по сути создание исходного текста программы (возможно в нескольких вариантах);
- анализ разработанного кода;
- сборка, компиляция и разработка исполняемого модуля;
- тестирование и отладка, сохранение произведённых изменений;
- документирование.

3. Что такое суффикс в контексте языка программирования? Приведите примеры использования.

Использование суффикса “.с” для имени файла с программой на языке Си отражает удобное и полезное соглашение, принятое в ОС UNIX. Для любого имени входного файла суффикс определяет какая компиляция требуется. Суффиксы и префиксы указывают тип объекта. Одно из полезных свойств компилятора Си — его способность по суффиксам определять типы файлов. По суффиксу .с компилятор распознает, что файл abcd.c должен компилироваться, а по суффиксу .о, что файл abcd.o является объектным модулем и для получения исполняемой программы необходимо выполнить редактирование связей. Простейший пример командной строки для компиляции программы abcd.c и построения исполняемого модуля abcd имеет вид: `gcc -o abcd abcd.c`.

Некоторые проекты предпочитают показывать префиксы в начале текста изменений для старых (old) и новых (new) файлов. Опция – `prefix` может быть использована для установки такого префикса. Плюс к этому команда `bzr diff -p1` выводит префиксы в форме которая подходит для команды `patch -p1`.

4. Каково основное назначение компилятора языка С в UNIX?

Основное назначение компилятора с языка Си заключается в компиляции всей программы в целом и получении исполняемого модуля.

5. Для чего предназначена утилита make?

При разработке большой программы, состоящей из нескольких исходных файлов заголовков, приходится постоянно следить за файлами, которые требуют перекомпиляции после внесения изменений. Программа `make` освобождает пользователя от такой рутинной работы и служит для документирования взаимосвязей между файлами. Описание взаимосвязей и соответствующих действий хранится в так называемом `make`-файле, который по умолчанию имеет имя `makefile` или `Makefile`.

6. Приведите пример структуры Makefile. Дайте характеристику основным элементам этого файла.

```

CC = gcc
CFLAGS =
LIBS = -lm

calcul: calculate.o main.o
    gcc calculate.o main.o -o calcul $(LIBS)

calculate.o: calculate.c calculate.h
    gcc -c calculate.c $(CFLAGS)

main.o: main.c calculate.h
    gcc -c main.c $(CFLAGS)

clean:
    -rm calcul *.o *~

```

В общем случае make-файл содержит последовательность записей (строк), определяющих зависимости между файлами. Первая строка записи представляет собой список целевых (зависимых) файлов, разделенных пробелами, за которыми следует двоеточие и список файлов, от которых зависят целевые. Текст, следующий за точкой с запятой, и все последующие строки, начинающиеся с литеры табуляции, являются командами ОС UNIX, которые необходимо выполнить для обновления целевого файла. Таким образом, спецификация взаимосвязей имеет формат: target1 [target2...]: [:] [dependment1...] [(tab)commands] [#commentary] [(tab)commands] [#commentary], где # — специфицирует начало комментария, так как содержимое строки, начиная с # и до конца строки, не будет обрабатываться командой make; : — последовательность команд ОС UNIX должна содержаться в одной строке make-файла (файла описаний), есть возможность переноса команд (), но она считается как одна строка; :: — последовательность команд ОС UNIX может содержаться

в нескольких последовательных строках файла описаний. Приведённый выше make-файл для программы `abcd.c` включает два способа компиляции и построения исполняемого модуля. Первый способ предусматривает обычную компиляцию с построением исполняемого модуля с именем `abcd`. Второй способ позволяет включать в исполняемый модуль `testabcd` возможность выполнить процесс отладки на уровне исходного текста.

7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

Пошаговая отладка программ заключается в том, что выполняется один оператор программы и, затем контролируются те переменные, на которые должен был воздействовать данный оператор. Если в программе имеются уже отлаженные подпрограммы, то подпрограмму можно рассматривать, как один оператор программы и воспользоваться вторым способом отладки программ. Если в программе существует достаточно большой участок программы, уже отлаженный ранее, то его можно выполнить, не контролируя переменные, на которые он воздействует. Использование точек останова позволяет пропускать уже отлаженную часть программы. Точка останова устанавливается в местах, где необходимо проверить содержимое переменных или просто проконтролировать, передаётся ли управление данному оператору. Практически во всех отладчиках поддерживается это свойство (а также выполнение программы до курсора и выход из подпрограммы). Затем отладка программы продолжается в пошаговом режиме с контролем локальных и глобальных переменных, а также внутренних регистров микроконтроллера и напряжений на выводах этой микросхемы.

8. Назовите и дайте основную характеристику основным командам отладчика `gdb`.

- `backtrace` – выводит весь путь к текущей точке останова, то есть названия всех функций, начиная от `main()`; иными словами, выводит весь стек функций;

- `break` – устанавливает точку останова; параметром может быть номер строки или название функции;
 - `clear` – удаляет все точки останова на текущем уровне стека (то есть в текущей функции);
 - `continue` – продолжает выполнение программы от текущей точки до конца;
 - `delete` – удаляет точку останова или контрольное выражение;
 - `display` – добавляет выражение в список выражений, значения которых отображаются каждый раз при остановке программы;
 - `finish` – выполняет программу до выхода из текущей функции; отображает возвращаемое значение, если такое имеется;
 - `info breakpoints` – выводит список всех имеющихся точек останова;
 - `info watchpoints` – выводит список всех имеющихся контрольных выражений;
 - `splist` – выводит исходный код; в качестве параметра передаются название файла исходного кода, затем, через двоеточие, номер начальной и конечной строки;
 - `next` – пошаговое выполнение программы, но, в отличие от команды `step`, не выполняет пошагово вызываемые функции;
 - `print` – выводит значение какого-либо выражения (выражение передаётся в качестве параметра);
 - `run` – запускает программу на выполнение;
 - `set` – устанавливает новое значение переменной
 - `step` – пошаговое выполнение программы;
 - `watch` – устанавливает контрольное выражение, программа остановится, как только значение контрольного выражения изменится;
9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.
- Выполнили компиляцию программы
 - Увидели ошибки в программе

- Открыли редактор и исправили программу
- Загрузили программу в отладчик gdb
- run — отладчик выполнил программу, мы ввели требуемые значения.
- программа завершена, gdb не видит ошибок.

10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.

Ошибок не было.

11. Назовите основные средства, повышающие понимание исходного кода программы.

Если вы работаете с исходным кодом, который не вами разрабатывался, то назначение различных конструкций может быть не совсем понятным. Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся: - cscope - исследование функций, содержащихся в программе; - splint — критическая проверка программ, написанных на языке Си.

12. Каковы основные задачи, решаемые программой splint?

- Проверка корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений;
- Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;
- Общая оценка мобильности пользовательской программы.