

# **Лабораторная работа №14**

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

Пафренова Елизавета Евгеньевна

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Выполнение лабораторной работы</b>	<b>7</b>
<b>4</b>	<b>Выводы</b>	<b>12</b>
<b>5</b>	<b>Контрольные вопросы</b>	<b>13</b>

## Список иллюстраций

3.1	Создание каталогов и файлов . . . . .	7
3.2	Компиляция программы . . . . .	7
3.3	Makefile . . . . .	8
3.4	Запуск отладчика . . . . .	8
3.5	Запуск программы . . . . .	9
3.6	Просмотр исходного кода . . . . .	9
3.7	Точки останова . . . . .	10
3.8	Значение переменной . . . . .	10
3.9	Удаление точки останова . . . . .	10
3.10	Анализ кода файла calculate.c . . . . .	11
3.11	Анализ кода файла main.c . . . . .	11

## **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 Задание

1. В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`.
2. Создайте в нём файлы: `calculate.h`, `calculate.c`, `main.c`. Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится.
3. Выполните компиляцию программы посредством `gcc`
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile` со следующим содержанием.
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`)
7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

### 3 Выполнение лабораторной работы

Первым шагом я создала каталог `~/work/os/lab_prog` с помощью ***mkdir -p*** и файлы `calculate.h`, `calculate.c`, `main.c` с помощью ***touch***. (рис. 3.1)

```
[eeparfenova@fedora ~]$ mkdir -p ~/work/os/lab_prog
[eeparfenova@fedora ~]$ cd ~/work/os/lab_prog
[eeparfenova@fedora lab_prog]$ touch calculate.h
[eeparfenova@fedora lab_prog]$ touch calculate.c
[eeparfenova@fedora lab_prog]$ touch main.c
[eeparfenova@fedora lab_prog]$ ls
calculate.c calculate.h main.c
```

Рис. 3.1: Создание каталогов и файлов

Далее я записала программы в эти файлы. Листинги я взяла из файла Лабораторной работы. Затем выполнила компиляцию программы посредством gcc, используя команды:

***gcc -c calculate.c***

***gcc -c -g main.c***

***gcc calculate.o main.o -o calcul -lm*** (рис. 3.2)

```
[eeparfenova@fedora lab_prog]$ gcc -c calculate.c
[eeparfenova@fedora lab_prog]$ gcc -c -g main.c
[eeparfenova@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
[eeparfenova@fedora lab_prog]$ ls
calcul calculate.c calculate.h calculate.h~ calculate.o main.c main.o
```

Рис. 3.2: Компиляция программы

Далее я создала Makefile с помощью ***touch*** и записала туда код из файла Лабораторной работы, внося некоторые изменения в 6 и 19 строках. (рис. 3.3)

```

#
# Makefile
#

CC = 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

# End Makefile

```

Рис. 3.3: Makefile

После я запустила отладчик GDB командой ***gdb ./calcul***, загрузив в него программу для отладки.(рис. 3.4)

```

[eeeparfenova@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 11.2-2.fc35
Copyright (C) 2022 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...

```

Рис. 3.4: Запуск отладчика

Запустила программу внутри отладчика, используя команду ***run***.(рис. 3.5)



```
(gdb) run
Starting program: /home/eeparfenova/work/os/lab_prog/calcul

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /home/eeparfenova/work/os/lab_prog/system-suppl
ied DSO at 0x7ffff7fc5000...
Downloading separate debug info for /lib64/libm.so.6...
Downloading separate debug info for /lib64/libc.so.6...
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): pow
Степень: 2
25.00
[Inferior 1 (process 11015) exited normally]
```

Рис. 3.5: Запуск программы

Постранично просмотрела код, используя **list**, а после посмотрела исходный код с 12 строки по 15 строку командой **list 12,15** (рис. 3.6)

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

Рис. 3.6: Просмотр исходного кода

Затем установила точку останова в файле calculate.c на строке номер 14, сразу после ввода числа. Сделала это, используя команду **break 14**. Посмотрела информацию об имеющихся в проекте точка останова командой **info breakpoints**. Запустила программу и проверила, когда она остановится. Она остановилась правильно, перед вводом операции. (рис. 3.7)

```
(gdb) break 14
Breakpoint 1 at 0x4014b3: file main.c, line 14.
(gdb) info breakpoints
Num      Type             Disp Enb Address              What
1        breakpoint      keep y   0x00000000004014b3 in main at main.c:14
(gdb) run
Starting program: /home/eparfenova/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5

Breakpoint 1, main () at main.c:14
14      printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
```

Рис. 3.7: Точки останова

Посмотрела, чему равно на этом этапе значение переменной *Numeral*, введя ***print Numeral***. Оно равно пяти, так как я ввела именно это число. Затем сравнила с результатом вывода на экран, который сделала командой ***display Numeral***. Значения совпали. (рис. 3.8)

```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
```

Рис. 3.8: Значение переменной

После просто убрали точки останова, вызвав вначале информацию о них через ***info breakpoints***, а после удалив, используя ***delete 1*** (точка останова номер 1) (рис. 3.9)

```
(gdb) info breakpoints
Num      Type             Disp Enb Address              What
1        breakpoint      keep y   0x00000000004014b3 in main at main.c:14
breakpoint already hit 1 time
(gdb) delete 1
```

Рис. 3.9: Удаление точки останова

Последним шагом с помощью утилиты *splint* проанализировала коды файлов *calculate.c* и *main.c*. (рис. 3.10) (рис. 3.11)

```
[eeparfenova@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:7:37: 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:10:31: Function parameter Operation declared as manifest array
        (size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:3: 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:22:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:6: 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:38:10: Return value type double does not match declared type float:
        (HUGE_VAL)
```

Рис. 3.10: Анализ кода файла calculate.c

```
[eeparfenova@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2021

calculate.h:7:37: 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:13:2: 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:15:13: 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:15:10: Corresponding format code
main.c:15:2: Return value (type int) ignored: scanf("%s", &Ope...

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

Рис. 3.11: Анализ кода файла main.c

## 4 Выводы

Мы 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 более полную информацию о программах: 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:

- создание исходного кода программы;
- представляется в виде файла;
- сохранение различных вариантов исходного текста;
- анализ исходного текста; (необходимо отслеживать изменения исходного кода, а также при работе более двух программистов над проектом программы нужно, чтобы они не делали изменений кода в одно время).
- компиляция исходного текста и построение исполняемого модуля;
- тестирование и отладка;
- проверка кода на наличие ошибок
- сохранение всех изменений, выполняемых при тестировании и отладке.

3. Что такое суффиксы и префиксы? Основное их назначение. Приведите примеры их 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. Основное назначение компилятора с языка Си 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 структуру make-файла. Дайте характеристику основным элементам этого файла.

makefile для программы abcd.c мог бы иметь вид:

```
Makefile
```

```
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 ~
```

```
#End Makefile
```

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 ис-

полняемый 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 (если они есть)

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.