

Отчёт по лабораторной работе № 13

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

Паулу Антонию Жоау

Содержание

1	Цель работы	4
2	Задание	5
3	Выполнение лабораторной работы	6
3.1	Работа с программой калькулятор	6
4	Выводы	14
5	Ответы на контрольные вопросы	15

Список иллюстраций

3.1	lab_prog	6
3.2	calculate.h, calculate.c, main.c	6
3.3	calculate.c	7
3.4	calculate.h	8
3.5	main.c	8
3.6	gcc	9
3.7	Makefile	9
3.8	Makefile	9
3.9	gdb	10
3.10	run	10
3.11	list	10
3.12	list	10
3.13	list	11
3.14	breakpoint	11
3.15	info breakpoints	11
3.16	breakpoint	11
3.17	Numeral	12
3.18	Numeral	12
3.19	breakpoint	12
3.20	splint	12
3.21	splint	13

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

- Ознакомиться с теоретическим материалом.
- Выполнить упражнения.
- Ответить на контрольные вопросы.

3 Выполнение лабораторной работы

3.1 Работа с программой калькулятор

1. В домашнем каталоге создали подкаталог `~/work/os/lab_prog`. (рис. [3.1])

```
[azpaulu@fedora ~]$ mkdir ~/work/os/lab_prog  
[azpaulu@fedora ~]$ ls ~/work/os  
lab08 lab_prog  
[azpaulu@fedora ~]$
```

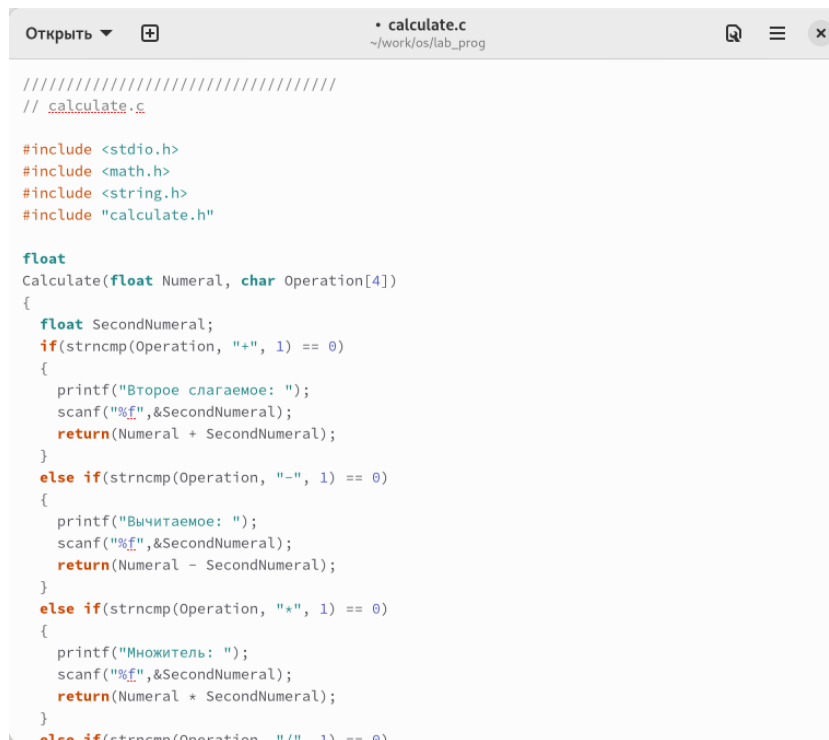
Рис. 3.1: lab_prog

2. Создали в нём файлы: `calculate.h`, `calculate.c`, `main.c`. (рис. [3.2])

```
[azpaulu@fedora ~]$ cd ~/work/os/lab_prog  
[azpaulu@fedora lab_prog]$ touch calculate.h calculate.c main.c  
[azpaulu@fedora lab_prog]$ ls  
calculate.c calculate.h main.c  
[azpaulu@fedora lab_prog]$
```

Рис. 3.2: calculate.h, calculate.c, main.c

Это примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он запрашивает первое число, операцию, второе число. После этого программа выводит результат и останавливается. Реализация функций калькулятора в файле `calculate.c`: (рис. [3.3])



```
////////////////////////////////////
// calculate.c

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

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

Рис. 3.3: calculate.c

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора: (рис. [3.4])



```

////////////////////////////////////
// calculate.h

#ifndef CALCULATE_H_
#define CALCULATE_H_

float Calculate(float Numeral, char Operation[4]);

#endif /*CALCULATE_H_*/

```

Рис. 3.4: calculate.h

Основной файл main.c, реализующий интерфейс пользователя к калькулятору:(рис. [3.5])



```

////////////////////////////////////
// main.c

#include <stdio.h>
#include "calculate.h"
int
main (void)
{
    float Numeral;
    char Operation[4];
    float Result;
    printf("Число: ");
    scanf("%f",&Numeral);
    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
    scanf("%s",&Operation);
    Result = Calculate(Numeral, Operation);
    printf("%6.2f\n",Result);
    return 0;
}

```

Рис. 3.5: main.c

3. Выполнили компиляцию программы посредством gcc: (рис. [3.6])

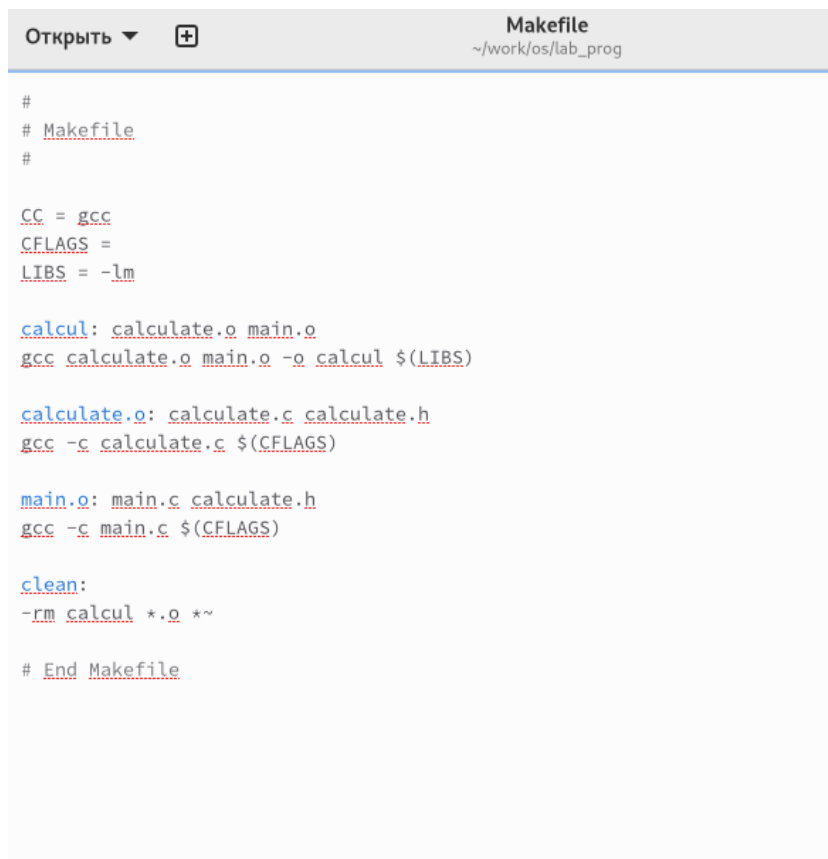

```
[azpaulu@fedora lab_prog]$ gcc -c calculate.c
[azpaulu@fedora lab_prog]$ gcc -c main.c
[azpaulu@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm
```

Рис. 3.6: gcc

4. Исправили синтаксические ошибки.
5. Создали Makefile. (рис. [3.7]), (рис. [3.8])

```
[azpaulu@fedora lab_prog]$ touch Makefile
```

Рис. 3.7: Makefile



```
#
# 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
```

Рис. 3.8: Makefile

6. С помощью gdb выполнили отладку программы calcul: – Запустили отладчик GDB, загрузив в него программу для отладки:(рис. [3.9])

```
[azpaulu@fedora lab_prog]$ gdb ./calcul
```

Рис. 3.9: gdb

– Для запуска программы внутри отладчика ввели команду run:(рис. [3.10])

```
(gdb) run
Starting program: /home/azpaulu/work/os/lab_prog/calcul
Downloading separate debug info for /home/azpaulu/work/os/lab_prog/system-suppl
ed DSO at 0x7ffff7fc4000...

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: 2
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 1
1.00
[Inferior 1 (process 191487) exited normally]
(gdb)
```

Рис. 3.10: run

– Для постраничного (по 9 строк) просмотра исходного код использовали ко-
манду list (рис. [3.11])

```
(gdb) list
Downloading source file /usr/src/debug/glibc-2.35-4.fc36.x86_64/elf/sofini.c...
1      /* Terminate the frame unwind info section with a 4byte 0 as a sentinel;
2         this would be the 'length' field in a real FDE.  */
3
4      typedef unsigned int ui32 __attribute__ ((mode (SI)));
5      static const ui32 __FRAME_END__[1]
6          __attribute__ ((used, section ("eh_frame")))
7          = { 0 };
(gdb) █
```

Рис. 3.11: list

– Для просмотра строк с 1 по 4 основного файла использовали list с параметра-
ми:(рис. [3.12])

```
(gdb) list 1, 4
1      /* Terminate the frame unwind info section with a 4byte 0 as a sentinel;
2         this would be the 'length' field in a real FDE.  */
3
4      typedef unsigned int ui32 __attribute__ ((mode (SI)));
(gdb) █
```

Рис. 3.12: list

– Для просмотра определённых строк не основного файла использовали list с параметрами: (рис. [3.13])

```
(gdb) list calculate.c:20,29
```

Рис. 3.13: list

– Установили точку останова в файле calculate.c на строке номер 21: (рис. [3.14])

```
(gdb) break 21
No line 21 in the current file.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (21) pending.
```

Рис. 3.14: breakpoint

– Вывели информацию об имеющихся в проекте точка останова: (рис. [3.15])

```
Breakpoint 1 (21) pending.
(gdb) i b
Num      Type      Disp Enb Address  What
1        breakpoint keep  y   <PENDING> 21
```

Рис. 3.15: info breakpoints

– Запустили программу внутри отладчика и убедились, что программа остановится в момент прохождения точки останова: (рис. [3.16])

```
(gdb) run
Starting program: /home/azpaulu/work/os/lab_prog/calcul
Downloading separate debug info for /home/azpaulu/work/os/lab_prog/system-...
ed DSO at 0x7ffff7fc4000...

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".
```

Рис. 3.16: breakpoint

– Посмотрели, чему равно на этом этапе значение переменной Numeral, введя: (рис. [3.17])

```
(gdb) print Numeral
```

Рис. 3.17: Numeral

– Сравнили с результатом вывода на экран после использования команды:
(рис. [3.18])

```
(gdb) display Numeral
```

Рис. 3.18: Numeral

– Убрали точки останова: (рис. [3.19])

```
(gdb) i b
Num      Type      Disp Enb Address  What
1        breakpoint keep  y   <PENDING> 21
(gdb) delete 1
(gdb) i b
No breakpoints or watchpoints.
```

Рис. 3.19: breakpoint

7. С помощью утилиты splint попробовали проанализировать коды файлов
calculate.c и main.c (рис. [3.20]), (рис. [3.21])

```
* Ожидание в очереди...
* Ожидание аутентификации...
* Ожидание в очереди...
* Загрузка пакетов...
* Запрос данных...
* Проверка изменений...
* Установка пакетов...
Splint 3.1.2 --- 22 Jan 2022

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:16:5: 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:22:5: Return value (type int) ignored: scanf("%f", &Sec...
```

Рис. 3.20: splint

```

[azpaulu@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 22 Jan 2022

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:13:3: 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:15:14: 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:15:11: Corresponding format code
main.c:15:3: Return value (type int) ignored: scanf("%s", &Ope...

Finished checking --- 4 code warnings

```

Рис. 3.21: splint

4 Выводы

В ходе выполнения были приобретены 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 Ответы на контрольные вопросы

1. Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой `man` или опцией `-help (-h)` для каждой команды.
2. Процесс разработки программного обеспечения обычно разделяется на следующие этапы: • 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. Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: `vi`, `vim`, `mceditor`, `emacs`, `geany` и др. После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.
3. Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расширением (суффиксом) `.c` воспринимаются gcc как программы на языке C, файлы с расширением `.cc` или `.C` – как файлы на языке C++, а файлы с расширением `.o` считаются

объектными. Например, в команде «gcc -c main.c»: gcc по расширению (суффиксу) .c распознает тип файла для компиляции и формирует объектный модуль – файл с расширением .o. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o и в качестве параметра задать имя создаваемого файла: «gcc -o hello main.c».

4. Основное назначение компилятора языка Си в UNIX заключается в компиляции всей программы и получении исполняемого файла/модуля.
5. Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. Она позволяет автоматизировать процесс преобразования файлов программы из одной формы в другую, отслеживает взаимосвязи между файлами.
6. Для работы с утилитой make необходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefile или Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса. В самом простом случае Makefile имеет следующий синтаксис: ... : ...
<команда 1> ... Сначала задаётся список целей, разделённых пробелами, за которым идёт двоеточие и список зависимостей. Затем в следующих строках указываются команды. Строки с командами обязательно должны начинаться с табуляции. В качестве цели в Makefile может выступать имя файла или название какого-то действия. Зависимость задаёт исходные параметры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды – собственно действия, которые необходимо выполнить для достижения цели. Общий синтаксис Makefile имеет вид: target1 [target2...]:[:] [dependment1...] [(tab)commands] [#commentary] [(tab)commands] [#commentary] Здесь знак # определяет начало комментария (содержимое от знака # и до конца строки не будет обрабатываться. Одинарное двоеточие указывает на то, что последовательность

команд должна содержаться в одной строке. Для переноса можно в длинной строке команд можно использовать обратный слэш (). Двойное двоеточие указывает на то, что последовательность команд может содержаться в нескольких последовательных строках. Пример более сложного синтаксиса Makefile: `Makefile for abcd.c # CC = gcc CFLAGS = #Compile abcd.c normaly abcd: abcd.c (CFLAGS) abcd.c clean: -rm abcd .o ~ #End Makefile for abcd.c` В этом примере в начале файла заданы три переменные: CC и CFLAGS. Затем указаны цели, их зависимости и соответствующие команды. В командах происходит обращение к значениям переменных. Цель с именем clean производит очистку каталога от файлов, полученных в результате компиляции. Для её описания использованы регулярные выражения.

7. Во время работы над кодом программы программист неизбежно сталкивается с появлением ошибок в ней. Использование отладчика для поиска и устранения ошибок в программе существенно облегчает жизнь программиста. В комплект программ GNU для ОС типа UNIX входит отладчик GDB (GNU Debugger). Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле. Для этого следует воспользоваться опцией -g компилятора gcc: `gcc -c file.c -g` После этого для начала работы с gdb необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: `gdb file.o`

8. Основные команды отладчика gdb:

`backtrace` – вывод на экран пути к текущей точке останова (по сути вывод названий всех функций) `break` – установить точку останова (в качестве параметра может быть указан номер строки или название функции) `clear` – удалить все точки останова в функции `continue` – продолжить выполнение программы `delete` – удалить точку останова `display` – добавить выражение в список выражений,

значения которых отображаются при достижении точки останова программы `finish` – выполнить программу до момента выхода из функции `info breakpoints` – вывести на экран список используемых точек останова `info watchpoints` – вывести на экран список используемых контрольных выражений `list` – вывести на экран исходный код (в качестве параметра может быть указано название файла и через двоеточие номера начальной и конечной строк) `next` – выполнить программу пошагово, но без выполнения вызываемых в программе функций `print` – вывести значение указываемого в качестве параметра выражения `run` – запуск программы на выполнение `set` – установить новое значение переменной `step` – пошаговое выполнение программы `watch` – установить контрольное выражение, при изменении значения которого программа будет остановлена Для выхода из `gdb` можно воспользоваться командой `quit` (или её сокращённым вариантом `q`) или комбинацией клавиш `Ctrl-d`. Более подробную информацию по работе с `gdb` можно получить с помощью команд `gdb -h` и `man gdb`. Схема отладки программы показана в 6 пункте лабораторной работы.

9. При первом запуске компилятор не выдал никаких ошибок, но в коде программы `main.c` допущена ошибка, которую компилятор мог пропустить (возможно, из-за версии 8.3.0-19): в строке `scanf("%s", &Operation);` нужно убрать знак `&`, потому что имя массива символов уже является указателем на первый элемент этого массива.
10. Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся:
 - `cscope` – исследование функций, содержащихся в программе,
 - `lint` – критическая проверка программ, написанных на языке Си.
12. Утилита `splint` анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, обнаруживает синтаксические и семантические ошибки.

В отличие от компилятора C анализатор splint генерирует комментарии с описанием разбора кода программы и осуществляет общий контроль, обнаруживая такие ошибки, как одинаковые объекты, определённые в разных файлах, или объекты, чьи значения не используются в работе программы, переменные с некорректно заданными значениями и типами и многое другое