

Лабораторная работа №13

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

Легиньких Галина Андреевна НФИбд-02-21

Содержание

1	Цель работы	5
2	Теоретическое введение	6
3	Выполнение работа	7
4	Вывод	17
5	Контрольные вопросы	18

Список иллюстраций

3.1	Подкаталог	7
3.2	calculate.c	8
3.3	calculate.h	8
3.4	main.c	9
3.5	gcc	9
3.6	Makefile	10
3.7	gdb	11
3.8	run	11
3.9	list	12
3.10	list 12,15	12
3.11	list calculate.c	13
3.12	break	13
3.13	info	13
3.14	run	14
3.15	Информация	14
3.16	print	15
3.17	display	15
3.18	delete	15
3.19	splint	16

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

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 Выполнение работа

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



Рис. 3.1: Подкаталог

2. Создала в подкаталоге файлы: `calculate.h`, `calculate.c`, `main.c`.

У меня получился 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 выведет результат и останавливается.

Реализация функций калькулятора в файле `calculate.c`:(рис. 3.2)

```

#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)
    {
        printf("Делитель: ");
        scanf("%f",&SecondNumeral);
        if(SecondNumeral == 0)
        {
            printf("Ошибка: деление на ноль! ");
            return(HUGE_VAL);
        }
        else
            return(Numeral / SecondNumeral);
    }
    else if(strncmp(Operation, "pow", 3) == 0)
    {
        printf("Степень: ");
        scanf("%f",&SecondNumeral);
    }
}

```

Рис. 3.2: calculate.c

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора:(рис. 3.3)

```

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
~
~
~

```

Рис. 3.3: calculate.h

Основной файл main.c, реализующий интерфейс пользователя к калькулято-

ру:(рис. 3.4)

```
#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. 3.4: main.c

3. Выполнила компиляцию программы посредством gcc:(рис. 3.5)

```
galeginkikh@dk6n39 ~/work/os/lab_prog $ gcc -c calculate.c
galeginkikh@dk6n39 ~/work/os/lab_prog $ gcc -c main.c
main.c: В функции «main»:
main.c:13:10: предупреждение: формат «%s» ожидает аргумент типа «char *», но аргумент 2 имеет тип «char (*)[4]» [-Wformat=]
   13 |     scanf("%s",&Operation);
      |           ^~
      |           | |
      |           | | char (*)[4]
      |           | char *
galeginkikh@dk6n39 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm
galeginkikh@dk6n39 ~/work/os/lab_prog $ ls
calcul calculate.c calculate.o calculate.h calculate.h~ calculate.o main.c main.c~ main.o Makefile Makefile~
```

Рис. 3.5: gcc

4. Создала Makefile со следующим содержанием:(рис. 3.6)

```

#
# 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. 3.6: Makefile

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.

5. С помощью gdb выполнила отладку программы calcul (перед использованием gdb исправьте Makefile):

– Запустила отладчик GDB, загрузив в него программу для отладки:

```
gdb ./calcul
```

(рис. 3.7)

```

galeginjkkh@dk6n39 ~/work/os/lab_prog $ gdb ./calcul
GNU gdb (Gentoo 10.2 vanilla) 10.2
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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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)

```

Рис. 3.7: gdb

– Для запуска программы внутри отладчика введите команду run:

run

(рис. 3.8)

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/g/a/galeginjkkh/work/os/lab_prog/calcul
Число: 17
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 9
      8.00
[Inferior 1 (process 3934) exited normally]

```

Рис. 3.8: run

– Для постраничного (по 9 строк) просмотра исходного код использовала команду list:

list

(рис. 3.9)

```
(gdb) list
1      #include <stdio.h>
2      #include "calculate.h"
3
4      int main (void)
5      {
6          float Numeral;
7          char Operation[4];
8          float Result;
9          printf("Число: ");
10         scanf("%f",&Numeral);
```

Рис. 3.9: list

– Для просмотра строк с 12 по 15 основного файла использовала list с параметрами:

list 12,15

(рис. 3.10)

```
(gdb) list 12,15
12         scanf("%s",&Operation);
13         Result = Calculate(Numeral, Operation);
14         printf("%.2f\n",Result);
15         return 0;
```

Рис. 3.10: list 12,15

– Для просмотра определённых строк не основного файла использовала list с параметрами:

list calculate.c:20,29

(рис. 3.11)

```
(gdb) list calculate.c:20,29
20         return Numeral * SecondNumeral);
21     }
22     else if(strncmp(Operation, "/", 1) == 0)
23     {
24         printf("Делитель: ");
25         scanf("%f",&SecondNumeral);
26         if(SecondNumeral == 0)
27         {
28             printf("Ошибка: деление на ноль! ");
29             return HUGE_VAL);
(gdb)
```

Рис. 3.11: list calculate.c

– Установила точку останова в файле calculate.c на строке номер 21:

```
list calculate.c:20,27
```

```
break 21
```

(рис. 3.12)

```
(gdb) list calculate.c:20,27
20         return(Numeral * SecondNumeral);
21     }
22     else if(strncmp(Operation, "/", 1) == 0)
23     {
24         printf("Делитель: ");
25         scanf("%f",&SecondNumeral);
26         if(SecondNumeral == 0)
27     {
(gdb) break 21
Breakpoint 1 at 0x401295: file calculate.c, line 22.
(gdb)
```

Рис. 3.12: break

– Вывела информацию об имеющихся в проекте точка останова:

```
info breakpoints
```

(рис. 3.13)

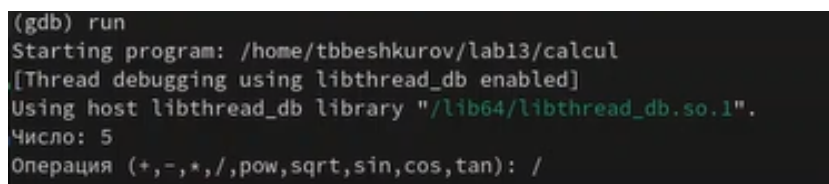
```
(gdb) info breakpoints
Num   Type             Disp Enb Address                  What
1     breakpoint       keep y  0x0000000000401295 in Calculate at calculate.c:22
```

Рис. 3.13: info

– Запустила программу внутри отладчика и убедилась, что программа остановится в момент прохождения точки останова:

```
run
5
-
backtrace
```

(рис. 3.14)



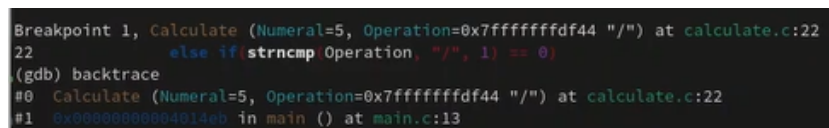
```
(gdb) run
Starting program: /home/tbbeshkurov/lab13/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): /
```

Рис. 3.14: run

– Отладчик выдал следующую информацию:

```
#0 Calculate (Numeral=5, Operation=0x7fffffffdf44 "/")
at calculate.c:22
#1 0x00000000004014eb in main () at main.c:17
```

(рис. 3.15)



```
Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdf44 "/") at calculate.c:22
22      else if (strcmp (Operation, "/") == 0)
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffdf44 "/") at calculate.c:22
#1 0x00000000004014eb in main () at main.c:13
```

Рис. 3.15: Информация

а команда backtrace показала весь стек вызываемых функций от начала программы до текущего места.

– Посмотрела, чему равно на этом этапе значение переменной Numeral, введя:

```
print Numeral
```

(рис. 3.16)

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

Рис. 3.16: print

– Сравнила с результатом вывода на экран после использования команды:

```
display Numeral
```

(рис. 3.17)

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

Рис. 3.17: display

– Убрала точки останова:

```
info breakpoints
```

```
delete 1
```

(рис. 3.18)

```
(gdb) delete 1  
(gdb) info breakpoints  
No breakpoints or watchpoints.
```

Рис. 3.18: delete

6. С помощью утилиты `splint` попробовала проанализировать коды файлов `calculate.c` и `main.c`. (рис. 3.19)

```

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:5:37: Function parameter Operation declared as manifest array (size
        constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:9:3: 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:14:3: Return value (type int) ignored: scanf("%f", &Sec...
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:26:6: 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:29:9: Return value type double does not match declared type float:
        (HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:37:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:38:9: Return value type double does not match declared type float:
        (pow(Numeral, SecondNumeral))
calculate.c:41:9: Return value type double does not match declared type float:

```

Рис. 3.19: splint

4 Вывод

Приобрела 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 утилика 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:
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 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 функ-

- ций; – 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.

- 1) Выполнили компиляцию программы
- 2) Увидели ошибки в программе
- 3) Открыли редактор и исправили программу
- 4) Загрузили программу в отладчик gdb
- 5) run — отладчик выполнил программу, мы ввели требуемые значения.
- 6) программа завершена, gdb не видит ошибок.

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

- 1 и 2.) Мы действительно забыли закрыть комментарии; 3.) отладчику не понравился формат %s для &Operation, т.к %s — символьный формат, а значит необходим только Operation.

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 slint?

1. Проверка корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений;

2. Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;
3. Общая оценка мобильности пользовательской программы.