

Лабораторная работа No 13.

**Программирование в командном процессоре ОС UNIX. Расширенное
программирование**

Боровиков Даниил Александрович

Содержание

1	Цель работы	5
2	Выполнение лабораторной работы	6
3	Выводы	12
4	Контрольные вопросы	13

Список иллюстраций

2.1	Создание нового подкаталога и файлов в нем	6
2.2	Реализация функций калькулятора в файле calculate.h:	7
2.3	Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора:	8
2.4	Основной файл main.c, реализующий интерфейс пользователя к калькулятору	8
2.5	Компиляция программы посредством gcc:	9
2.6	Создайте Makefile со следующим содержанием:	9
2.7	Отладка программы calcul	10
2.8	Анализ кода файла calculate.c	11
2.9	Анализ кода файла main.c.	11

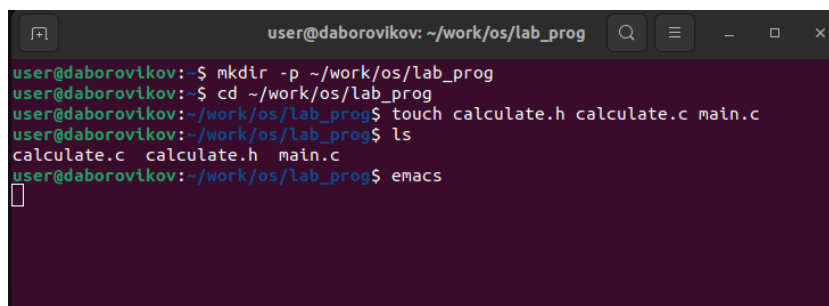
Список таблиц

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 Выполнение лабораторной работы

В домашнем каталоге создадим подкаталог `~/work/os/lab_prog`. Создадим 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.(рис. fig. 2.1).

A terminal window with a dark background and light text. The window title is 'user@daborovikov: ~/work/os/lab_prog'. The terminal shows the following commands and output:

```
user@daborovikov:~$ mkdir -p ~/work/os/lab_prog
user@daborovikov:~$ cd ~/work/os/lab_prog
user@daborovikov:~/work/os/lab_prog$ touch calculate.h calculate.c main.c
user@daborovikov:~/work/os/lab_prog$ ls
calculate.c calculate.h main.c
user@daborovikov:~/work/os/lab_prog$ emacs
```

Рис. 2.1: Создание нового подкаталога и файлов в нем

Реализация функций калькулятора в файле `calculate.h`:(рис. fig. 2.2).

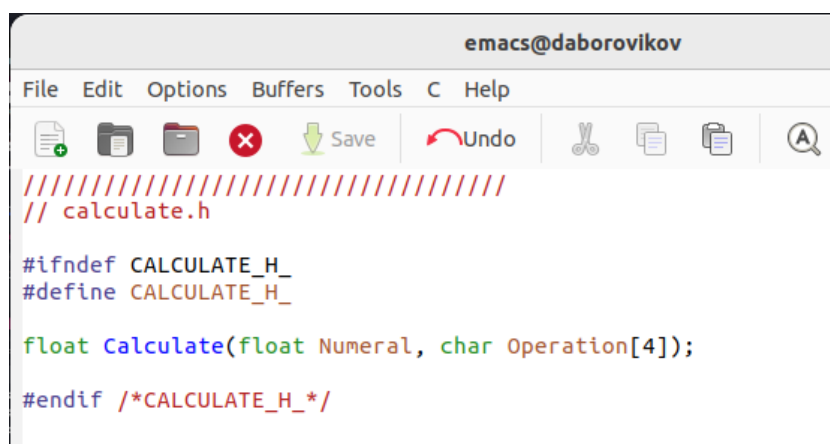
The screenshot shows an Emacs editor window titled 'emacs@daborovikov'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for saving, undo, and other editing functions. The main text area displays the following C code:

```
////////////////////////////////////  
// 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)  
    {  
        printf("Делитель: ");  
        scanf("%f",&SecondNumeral);  
    }  
}
```

The status bar at the bottom shows 'U:--- calculate.c Top L34 (C/*l Abbrev)' and 'menu-bar file open-file'.

Рис. 2.2: Реализация функций калькулятора в файле calculate.h:

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора:(рис. fig. 2.3).

The image shows the Emacs editor window titled 'emacs@daborovikov'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations and editing. The code in the buffer is for 'calculate.h', starting with a red comment line '////////////////////////////////////', followed by '#ifndef CALCULATE_H_', '#define CALCULATE_H_', a function declaration 'float Calculate(float Numeral, char Operation[4]);', and ending with '#endif /*CALCULATE_H_*/'.

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

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

Рис. 2.3: Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора:

Основной файл main.c, реализующий интерфейс пользователя к калькулятору(рис. fig. 2.4).

The image shows the Emacs editor window titled 'emacs@daborovikov'. The menu bar and toolbar are the same as in the previous screenshot. The code in the buffer is for 'main.c', starting with a red comment line '////////////////////////////////////', followed by '#include <stdio.h>', '#include "calculate.h"', the 'main' function signature 'int main (void)', and the function body which declares 'float Numeral;', 'char Operation[4];', 'float Result;', and contains logic for input, calculation using 'Calculate', and output using 'printf' and 'scanf'.

```
////////////////////////////////////
// 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. 2.4: Основной файл main.c, реализующий интерфейс пользователя к калькулятору

Выполните компиляцию программы посредством gcc:(рис. fig. 2.5).


```
user@daborovikov: ~/work/os/lab_prog
user@daborovikov:~$ gcc -c calculate.c
cc1: fatal error: calculate.c: Нет такого файла или каталога
compilation terminated.
user@daborovikov:~$ cd ~/work/os/lab_prog
user@daborovikov:~/work/os/lab_prog$ gcc -c calculate.c
user@daborovikov:~/work/os/lab_prog$ gcc -c main.c
main.c: In function 'main':
main.c:16:11: warning: format '%s' expects argument of type 'char *', but argu-
nt 2 has type 'char (*)[4]' [-Wformat=]
   16 |     scanf("%s", &Operation);
      |           ^~
      |           |
      |           | char (*)[4]
      |           | char *
user@daborovikov:~/work/os/lab_prog$ gcc calculate.o main.o -o calcul -lm
user@daborovikov:~/work/os/lab_prog$ ls
calcul      calculate.c~  calculate.h~  main.c      main.o
calculate.c  calculate.h  calculate.o  main.c~
user@daborovikov:~/work/os/lab_prog$
```

Рис. 2.5: Компиляция программы посредством gcc:

Создайте Makefile со следующим содержанием:(рис. fig. 2.6).

```
emacs@daborovikov
File Edit Options Buffers Tools Makefile Help
[Icons: New, Open, Save, Undo, Cut, Copy, Paste, Find]

#
# Makefile
#

CC = gcc
GFLAGS = -g
LIBS = -lm

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

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

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

clean:
-rm calcul *.o *~

# End Makefile
```

Рис. 2.6: Создайте Makefile со следующим содержанием:

С помощью gdb выполните отладку программы calcul (перед использованием gdb исправьте Makefile)(рис. fig. 2.7).

```

user@daborovikov:~/work/os/lab_prog$ emacs Makefile
user@daborovikov:~/work/os/lab_prog$ emacs
user@daborovikov:~/work/os/lab_prog$ make clean
rm calcul *.o *-
user@daborovikov:~/work/os/lab_prog$ make calculate.o
gcc -c calculate.c
user@daborovikov:~/work/os/lab_prog$ make main.o
gcc -c main.c
main.c: In function 'main':
main.c:16:11: warning: format '%s' expects argument of type 'char **', but argument 2 has type 'char (*)[4]' [-Wformat=]
   16 |     scanf("%s", &operation);
      |           ^~
      |           |
      |           | char (*)[4]
      |           char *
user@daborovikov:~/work/os/lab_prog$ make calcul
gcc calculate.o main.o -o calcul -lm
user@daborovikov:~/work/os/lab_prog$ gdb ./calcul
GNU gdb (Ubuntu 12.0.90-0ubuntu1) 12.0.90
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-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/user/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Число: 5
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): *
Множитель: 6
30.00
[Inferior 1 (process 30033) exited normally]
(gdb)

```

Рис. 2.7: Отладка программы calcul

С помощью утилиты splint попробуйте проанализировать коды файлов calculate.c и main.c. (рис. fig. 2.8).

```

user@daborovikov:~/work/os/lab_prog$ splint calculate.c
Splint 3.1.2 --- 21 Feb 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:16:7: 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:22:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:10: 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)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:46:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:13: Return value type double does not match declared type float:
        (pow(Numeral, SecondNumeral))
calculate.c:50:11: Return value type double does not match declared type float:
        (sqrt(Numeral))
calculate.c:52:11: Return value type double does not match declared type float:
        (sin(Numeral))
calculate.c:54:11: Return value type double does not match declared type float:
        (cos(Numeral))
calculate.c:56:11: Return value type double does not match declared type float:
        (tan(Numeral))
calculate.c:60:13: Return value type double does not match declared type float:
        (HUGE_VAL)

Finished checking --- 15 code warnings
user@daborovikov:~/work/os/lab_prog$

```

Рис. 2.8: Анализ кода файла calculate.c

(рис. fig. 2.9).

```

user@daborovikov:~/work/os/lab_prog$ splint main.c
Splint 3.1.2 --- 21 Feb 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:14:3: 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:16:14: 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:16:11: Corresponding format code
main.c:16:3: Return value (type int) ignored: scanf("%s", &Op...

Finished checking --- 4 code warnings
user@daborovikov:~/work/os/lab_prog$

```

Рис. 2.9: Анализ кода файла main.c.

3 Выводы

В ходе лабораторной работы 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

4 Контрольные вопросы

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

Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой тап или опцией `-help(-h)` для каждой команды.

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

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`, `mc editor`, `emacs`, `geany` и др. После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

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

Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расширением (суффиксом) .с воспринимаются gcc как программы на языке C, файлы с расширением .с++ как файлы на языке C++, а файлы с расширением .о считаются объектными. Например, в команде «gcc -main.c»: gcc по расширению (суффиксу) .с распознает тип файла для компиляции и формирует объектный модуль файл с расширением .о. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o в качестве параметра задать имя создаваемого файла: «gcc -o hello main.c». В ходе выполнения данной лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями.

4. Каково основное назначение компилятора языка C в UNIX?

Основное назначение компилятора языка C в UNIX заключается в компиляции всей программы и получении исполняемого файла/модуля.

5. Для чего предназначена утилита make?

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

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 = gccCFLAGS =# Compile abcd.c normaly
abcd: abcd.c$(CC) -o abcd
$(CFLAGS) abcd.cc
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. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

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. Назовите и дайте основную характеристику основным командам отладчика gdb.

Основные команды отладчика gdb: 1. backtrace вывод на экран пути к текущей точке останова (по сути вывод названий всех функций); 2. break установить точку останова (в качестве параметра может быть указан номер строки или название функции); 3. clear удалить все точки останова в функции; 4. continue продолжить выполнение программы; 5. delete удалить точку останова; 6. display добавить выражение в список выражений, значения которых отображаются при достижении точки останова программы; 7. finish выполнить программу до момента выхода из функции; 8. info breakpoints вывести на экран список используемых точек останова; 9. info watchpoints вывести на экран список используемых контрольных выражений; 10. list вывести на экран исходный код (в ходе выполнения данной лабораторной работы я приобрела простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями. в качестве параметра может быть указано название файла и через двоеточие номера начальной и конечной строк); 11. next выполнить программу пошагово, но без выполнения вызываемых в программе функций; 12. print вывести значение указываемого в качестве параметра выражения; 13. run запуск программы

на выполнение; 14. `set` установить новое значение переменной; 15. `step` пошаговое выполнение программы; 16. `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`.

9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.

Схема отладки программы показана в 6 пункте лабораторной работы.

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

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.

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

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. Каковы основные задачи, решаемые программой `splint`?

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 Санализатор 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.