

Отчёт по лабораторной работе №13

Дисциплина: Операционные системы

Кристина Алексеевна Антипина

Содержание

Цель работы	1
Задание.....	1
Контрольные вопросы.....	11
Вывод.....	14

Цель работы

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

Задание

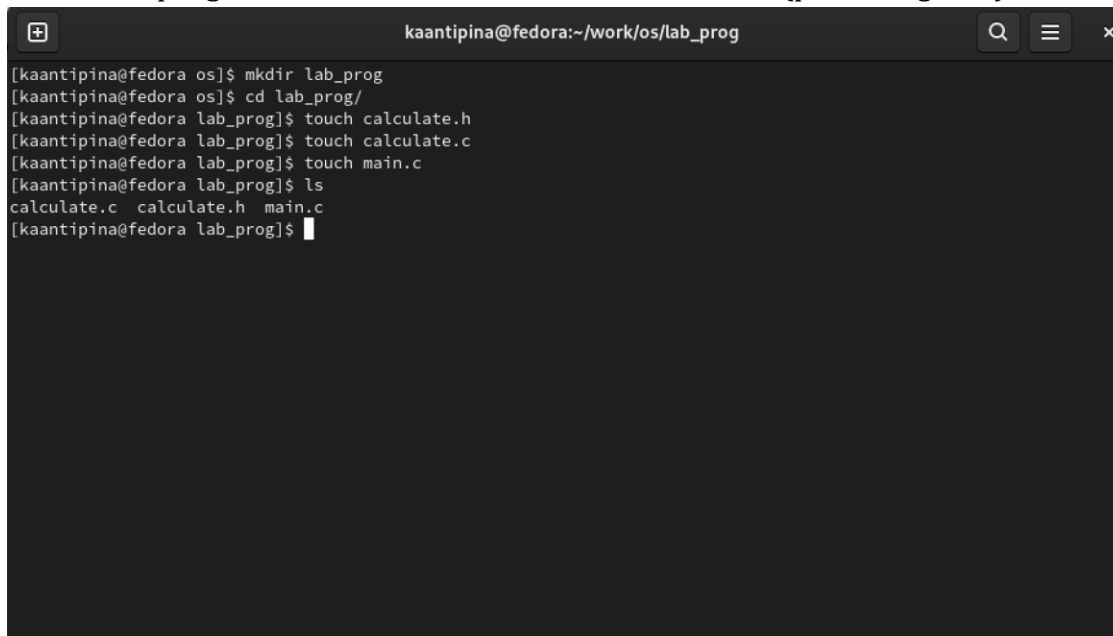
1. В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`.
2. Создайте в нём файлы: `calculate.h`, `calculate.c`, `main.c`. Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результаты и остановится. Реализация функций калькулятора в файле `calculate.h`: Интерфейсный файл `calculate.h`, описывающий формат вызова функции калькулятора:
3. Выполните компиляцию программы посредством `gcc`:
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile`
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`): – Запустите отладчик `GDB`, загрузив в него программу для отладки: `gdb ./calcul` – Для запуска программы внутри отладчика введите команду `run` – Для постраничного (по 9 строк) просмотра исходного кода используйте команду `list` – Для просмотра строк с 12 по 15 основного файла

используйте list с параметрами: list 12,15 – Для просмотра определённых строк не основного файла используйте list с параметрами: list calculate.c:20,29 – Установите точку останова в файле calculate.c на строке номер 21: list calculate.c:20,27 break 21

- Выведите информацию об имеющихся в проекте точках останова: info breakpoints – Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова: run 5
 - backtrace – Отладчик выдаст следующую информацию: 1 #0 Calculate (Numeral=5, Operation=0x7fffffff280 "-") 2 at calculate.c:21 3 #1 0x000000000400b2b in main () at main.c:17 а команда backtrace покажет весь стек вызываемых функций от начала программы до текущего места. – Посмотрите, чему равно на этом этапе значение переменной Numeral, введя: print Numeral На экран должно быть выведено число 5. – Сравните с результатом вывода на экран после использования команды: display Numeral – Уберите точки останова: info breakpoints delete 1
7. С помощью утилиты splint попробуйте проанализировать коды файлов calculate.c и main.c.

Выполнение лабораторной работы

1. В домашнем каталоге создаю подкаталог ~/work/os/lab_prog с помощью команды «mkdir lab_prog».
2. Создаю в каталоге файлы: calculate.h, calculate.c, main.c, используя команды «cd lab_prog» и «touch calculate.h calculate.c main.c» (рис. -@fig:001).

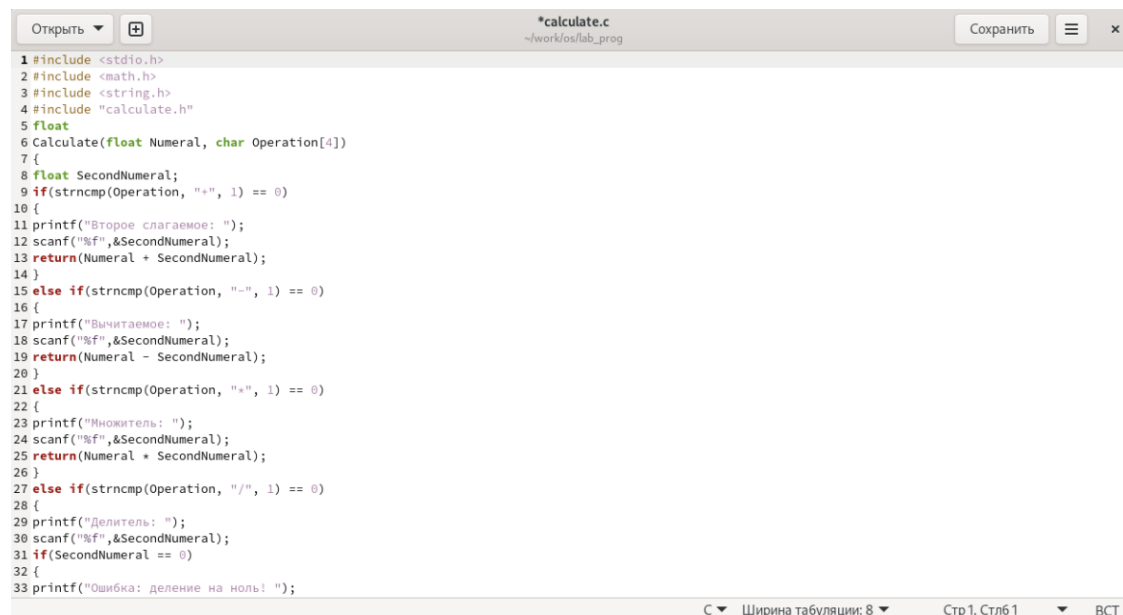


```
kaantipina@fedora:~/work/os/lab_prog
[kaantipina@fedora os]$ mkdir lab_prog
[kaantipina@fedora os]$ cd lab_prog/
[kaantipina@fedora lab_prog]$ touch calculate.h
[kaantipina@fedora lab_prog]$ touch calculate.c
[kaantipina@fedora lab_prog]$ touch main.c
[kaantipina@fedora lab_prog]$ ls
calculate.c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

Создаю каталог и файлы в нём

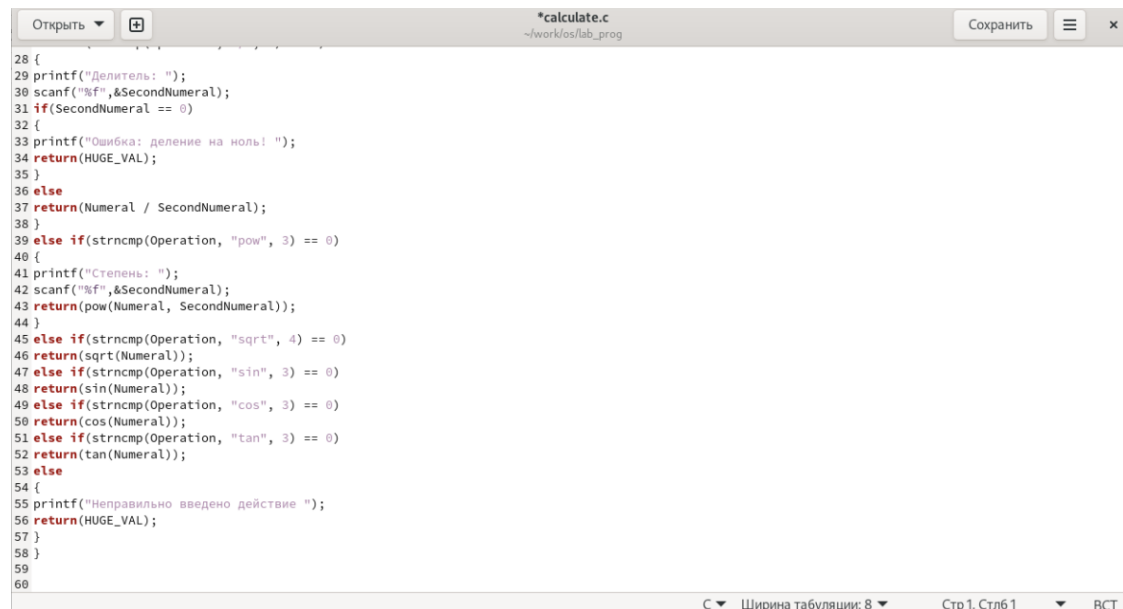
Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень,

вычислять \sin , \cos , \tan . При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится. Открыв редактор Емасс, приступаю к редактированию созданных файлов. Реализация функций калькулятора в файле `calculate.c` (рис. -@fig:002) (рис. -@fig:003).



```
1 #include <stdio.h>
2 #include <math.h>
3 #include <string.h>
4 #include "calculate.h"
5 float
6 Calculate(float Numeral, char Operation[4])
7 {
8     float SecondNumeral;
9     if(strncmp(Operation, "+", 1) == 0)
10 {
11     printf("Второе слагаемое: ");
12     scanf("%f",&SecondNumeral);
13     return(Numeral + SecondNumeral);
14 }
15 else if(strncmp(Operation, "-", 1) == 0)
16 {
17     printf("Вычитаемое: ");
18     scanf("%f",&SecondNumeral);
19     return(Numeral - SecondNumeral);
20 }
21 else if(strncmp(Operation, "*", 1) == 0)
22 {
23     printf("Мультипликатор: ");
24     scanf("%f",&SecondNumeral);
25     return(Numeral * SecondNumeral);
26 }
27 else if(strncmp(Operation, "/", 1) == 0)
28 {
29     printf("Делитель: ");
30     scanf("%f",&SecondNumeral);
31     if(SecondNumeral == 0)
32 {
33     printf("Ошибка: деление на ноль! ");
34     return(HUGE_VAL);
35 }
36 else
37     return(Numeral / SecondNumeral);
38 }
39 else if(strncmp(Operation, "pow", 3) == 0)
40 {
41     printf("Степень: ");
42     scanf("%f",&SecondNumeral);
43     return(pow(Numeral, SecondNumeral));
44 }
45 else if(strncmp(Operation, "sqrt", 4) == 0)
46     return(sqrt(Numeral));
47 else if(strncmp(Operation, "sin", 3) == 0)
48     return(sin(Numeral));
49 else if(strncmp(Operation, "cos", 3) == 0)
50     return(cos(Numeral));
51 else if(strncmp(Operation, "tan", 3) == 0)
52     return(tan(Numeral));
53 else
54 {
55     printf("Неправильно введено действие ");
56     return(HUGE_VAL);
57 }
58 }
59
60
```

Реализация функций калькулятора в файле `calculate.c`



```
28 {
29     printf("Делитель: ");
30     scanf("%f",&SecondNumeral);
31     if(SecondNumeral == 0)
32 {
33     printf("Ошибка: деление на ноль! ");
34     return(HUGE_VAL);
35 }
36 else
37     return(Numeral / SecondNumeral);
38 }
39 else if(strncmp(Operation, "pow", 3) == 0)
40 {
41     printf("Степень: ");
42     scanf("%f",&SecondNumeral);
43     return(pow(Numeral, SecondNumeral));
44 }
45 else if(strncmp(Operation, "sqrt", 4) == 0)
46     return(sqrt(Numeral));
47 else if(strncmp(Operation, "sin", 3) == 0)
48     return(sin(Numeral));
49 else if(strncmp(Operation, "cos", 3) == 0)
50     return(cos(Numeral));
51 else if(strncmp(Operation, "tan", 3) == 0)
52     return(tan(Numeral));
53 else
54 {
55     printf("Неправильно введено действие ");
56     return(HUGE_VAL);
57 }
58 }
59
60
```

Реализация функций калькулятора в файле `calculate.c`

Интерфейсный файл `calculate.h`, описывающий формат вызова функции калькулятора (рис. -@fig:004).



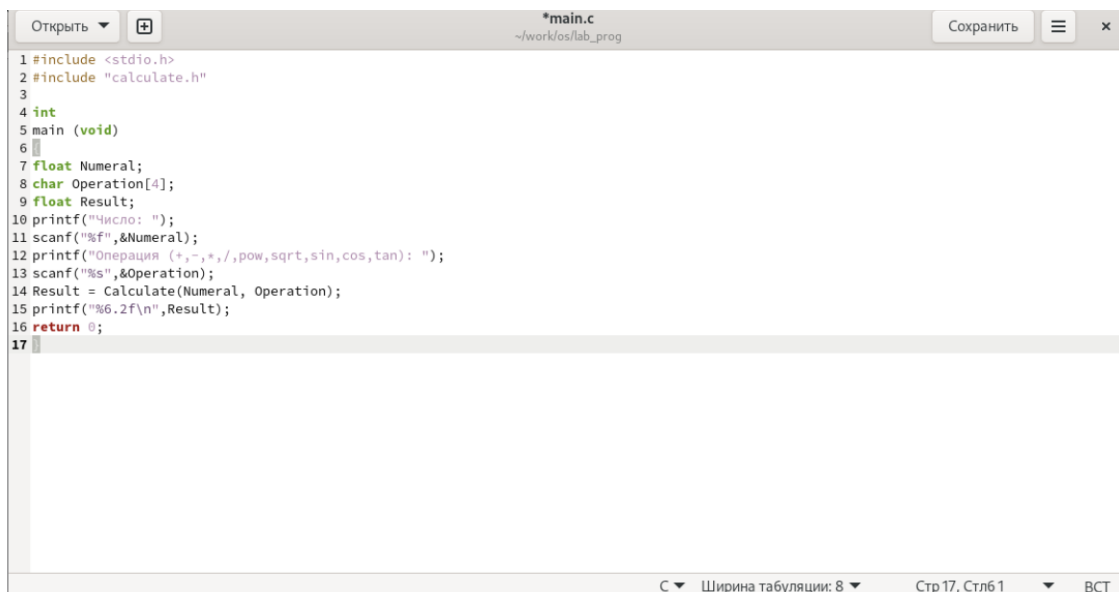
The screenshot shows a code editor window titled `*calculate.h` with the path `~/work/os/lab_prog`. The editor contains the following C header code:

```
1 #ifndef CALCULATE_H_
2 #define CALCULATE_H_
3
4 float Calculate(float Numeral, char Operation[4]);
5
6 #endif /*CALCULATE_H_*/
```

The status bar at the bottom indicates the language is C/Object-C Header, the tab width is 8, and the current position is line 4, column 1.

Интерфейсный файл *calculate.h*

Основной файл *main.c*, реализующий интерфейс пользователя к калькулятору (рис. - @fig:005).



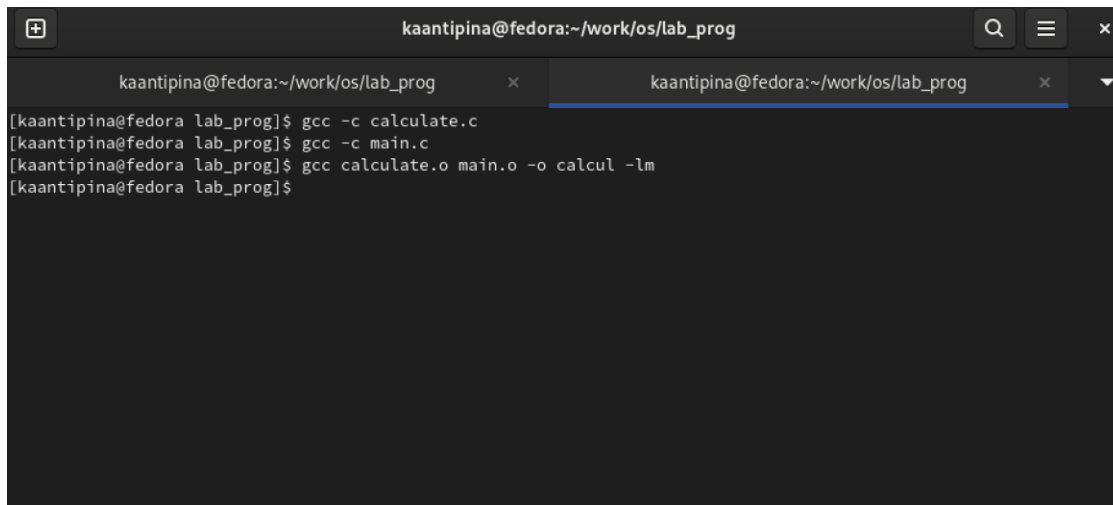
The screenshot shows a code editor window titled `*main.c` with the path `~/work/os/lab_prog`. The editor contains the following C code:

```
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: ");
11    scanf("%f",&Numeral);
12    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
13    scanf("%s",&Operation);
14    Result = Calculate(Numeral, Operation);
15    printf("%6.2f\n",Result);
16    return 0;
17 }
```

The status bar at the bottom indicates the language is C, the tab width is 8, and the current position is line 17, column 1.

Основной файл *main.c*

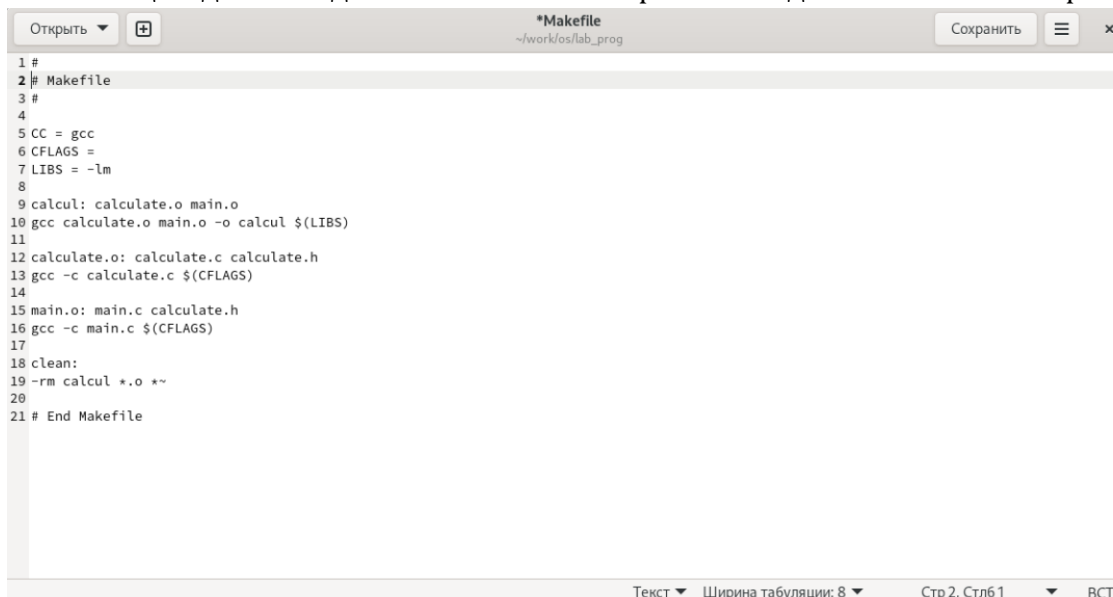
3. Выполню компиляцию программы посредством gcc, используя команды «gcc -c calculate.c», «gcc -c main.c» и «gcc calculate.o main.o -o calcul -lm» (рис. - @fig:006).

A terminal window titled 'kaantipina@fedora:~/work/os/lab_prog' with two tabs. The first tab is active and shows the following commands and their outputs:

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

Выполняю компиляцию программы посредством gcc

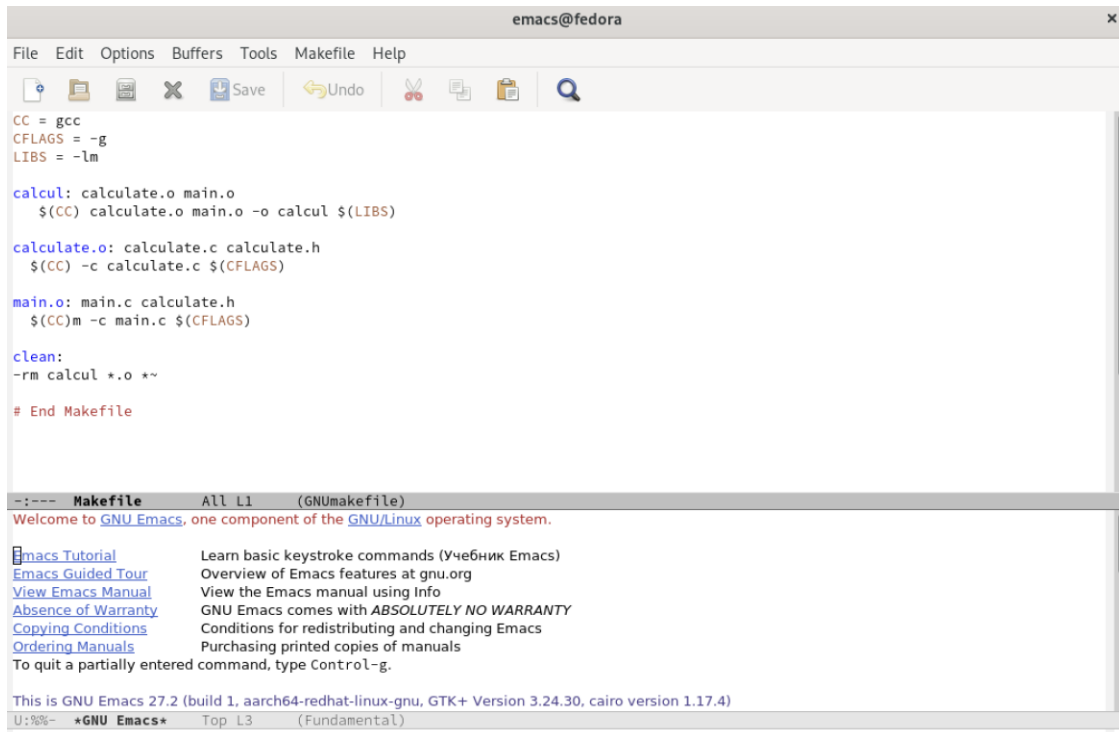
4. В ходе компиляции программы никаких ошибок выявлено не было.
5. Создаю Makefile с необходимым содержанием (рис. -@fig:007). Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цель main.o), а также их объединения в один исполняемый файл calcul (цель calcul). Цель clean нужна для автоматического удаления файлов. Переменная CC отвечает за утилиту для компиляции. Переменная CFLAGS отвечает за опции в данной утилите. Переменная LIBS отвечает за опции для объединения объектных файлов в один исполняемый файл.

A text editor window titled '*Makefile' with the path '~/.work/os/lab_prog'. It contains the following text:

```
1 #
2 # Makefile
3 #
4
5 CC = gcc
6 CFLAGS =
7 LIBS = -lm
8
9 calcul: calculate.o main.o
10 gcc calculate.o main.o -o calcul $(LIBS)
11
12 calculate.o: calculate.c calculate.h
13 gcc -c calculate.c $(CFLAGS)
14
15 main.o: main.c calculate.h
16 gcc -c main.c $(CFLAGS)
17
18 clean:
19 -rm calcul *.o *~
20
21 # End Makefile
```

Создаю Makefile с необходимым содержанием

6. Далее исправляю Makefile (рис. -@fig:008). В переменную CFLAGS добавляю опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Сделаю так, что утилита компиляции выбирается с помощью переменной CC.



Далее исправлю Makefile

После этого я удалю исполняемые и объектные файлы из каталога с помощью команды «make clean». Выполню компиляцию файлов, используя команды «make calculate.o», «make main.o», «male calcul» (рис. -@fig:009).

```
[kaantipina@fedora lab_prog]$ make clean
rm calcul *.o *~
[kaantipina@fedora lab_prog]$ make calculate.o
gcc -c calculate.c -g
[kaantipina@fedora lab_prog]$ make main.o
gcc -c main.c -g
[kaantipina@fedora lab_prog]$ male calcul
bash: male: command not found...
[kaantipina@fedora lab_prog]$ make calcul
gcc calculate.o main.o -o calcul -lm
```

Используя команды make

Далее с помощью gdb выполню отладку программы calcul. Запускаю отладчик GDB, загрузив в него программу для отладки, используя команду: «gdb ./calcul» (рис. -@fig:010).

```
[kaantipina@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 "aarch64-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)
```

Запускаю отладчик GDB

Для запуска программы внутри отладчика ввожу команду «run» (рис. -@fig:011).

```
(gdb) run
Starting program: /home/kaantipina/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 4
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -
Вычитаемое: 1
          3.00
[Inferior 1 (process 125530) exited normally]
```

Запуск программы внутри отладчика

Для постраничного (по 10 строк) просмотра исходного кода использую команду «list» (рис. -@fig:012).

```
(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) █
```

Использую команду «list»

Для просмотра строк с 12 по 15 основного файла использую команду «list 12,15» (рис. -@fig:013).

```
(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);
```

Просмотр строк с 12 по 15

Для просмотра определённых строк не основного файла использую команду «list calculate.c:20,29» (рис. -@fig:014).

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

Просмотр определённых строк не основного файла

Устанавливаю точку останова в файле calculate.c на строке номер 21, используя команды «list calculate.c:20,27» и «break 21» (рис. -@fig:015).

```
(gdb) list calculate.c:20,27
20         return(Numeral - SecondNumeral);
21     }
22     else if(strncmp(Operation, "*", 1) == 0)
23     {
24         printf("Множитель: ");
25         scanf("%f",&SecondNumeral);
26         return(Numeral * SecondNumeral);
27     }
(gdb) break 21
Breakpoint 1 at 0x400938: file calculate.c, line 22.
(gdb)
```

Устанавливаю точку останова в файле calculate.c

Вывожу информацию об имеющихся в проекте точках останова с помощью команды «info breakpoints» (рис. -@fig:016).


```
(gdb) info breakpoints
Num    Type           Disp Enb Address                  What
1      breakpoint    keep y   0x0000000000400938 in Calculate at calculate.c:22
2      breakpoint    keep y   0x0000000000400938 in Calculate at calculate.c:22
3      breakpoint    keep y   0x0000000000400928 in Calculate at calculate.c:20
breakpoint already hit 1 time
```

Вывожу информацию об имеющихся в проекте точках останова

Запускаю программу внутри отладчика, программа остановилась в момент прохождения точки останова. Использую команды «run», «5», «-» и «backtrace» (рис. - @fig:017).

```
(gdb) run
Starting program: /home/kaantipina/work/os/lab_prog/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): -
Вычитаемое: backtrace

Breakpoint 3, Calculate (Numeral=5, Operation=0xffffffffef00 "-") at calculate.c:20
20      return(Numeral - SecondNumeral);
(gdb)
```

Запускаю программу внутри отладчика до точки останова

Посмотрю, чему равно на этом этапе значение переменной Numeral, введя команду «print Numeral». Сравню с результатом вывода на экран после использования команды «display Numeral». Значения совпадают. Убираю точку останова с помощью команд «info breakpoints» и «delete 3» (рис. -@fig:018).

```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) info breakpoints
Num    Type           Disp Enb Address                  What
1      breakpoint    keep y   0x0000000000400938 in Calculate at calculate.c:22
2      breakpoint    keep y   0x0000000000400938 in Calculate at calculate.c:22
3      breakpoint    keep y   0x0000000000400928 in Calculate at calculate.c:20
breakpoint already hit 1 time
(gdb) delete 1
(gdb) splint calculate.c
Undefined command: "splint". Try "help".
(gdb) splint calculate.c
Undefined command: "splint". Try "help".
(gdb) splint
Undefined command: "splint". Try "help".
(gdb) delete 1
No breakpoint number 1.
(gdb) delete 2
(gdb) delete 3
(gdb)
```

Смотрю, чему равно Numeral, display Numeral и убираю точку останова

7. С помощью утилиты splint анализирую коды файлов calculate.c и main.c. Воспользуюсь командами «splint calculate.c» и «splint main.c» (рис. -@fig:019)

(рис. -@fig:020).

С помощью утилиты splint выяснилось, что в файлах calculate.c и main.c присутствует функция чтения scanf, возвращающая целое число (тип int), но эти числа не используются и нигде не сохраняются. Утилита вывела предупреждение о том, что в файле calculate.c происходит сравнение вещественного числа с нулем. Также возвращаемые значения (тип double) в функциях pow, sqrt, sin, cos и tan записываются в переменную типа float, что свидетельствует о потере данных.

```
[kaantipina@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:13:7: 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:19:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:25:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:31:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:32:10: 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: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:43:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:44:13: Return value type double does not match declared type float:
    (pow(Numeral, SecondNumeral))
calculate.c:47:11: Return value type double does not match declared type float:
    (sqrt(Numeral))
calculate.c:49:11: Return value type double does not match declared type float:
    (sin(Numeral))
calculate.c:51:11: Return value type double does not match declared type float:
```

Проанализирую код файла calculate.c

```
[kaantipina@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:11:1: 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:13:12: 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:13:9: Corresponding format code
main.c:13:1: Return value (type int) ignored: scanf("%s", &Ope...

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

Проанализирую код файла main.c

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 использовать обратный слэш (). Двойное двоеточие указывает на то, что последовательность команд может содержаться в нескольких последовательных строках.

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`.

9. Схема отладки программы показана в 6 пункте лабораторной работы.
10. При первом запуске компилятор не выдал никаких ошибок, но в коде программы `main.c` допущена ошибка, которую компилятор мог пропустить (возможно, из-за версии): в строке `scanf("%s", &Operation);` нужно убрать знак `&`, потому что имя массива символов уже является указателем на первый элемент этого массива.
11. 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 работепрограммы, переменные с некорректно заданными значениями и типами и многое другое.

Вывод

В ходе выполнения лабораторной работы № 13 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.