

РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра прикладной информатики и теории вероятностей

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 14

дисциплина: Операционные системы

Студент: Ким Реачна

Группа: НПИбд-02-20

Москва

2021г.

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.

Теоретическое введение:

Этапы разработки приложений:

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.

Компиляция исходного текста и построение исполняемого файла:

Для компиляции файла `main.c`, достаточно в командной строке ввести:

```
gcc -c main.c
```

Если требуется получить исполняемый файл с определённым именем (например, `hello`), то требуется воспользоваться опцией `-o` и в качестве параметра задать имя создаваемого файла:

```
gcc -c hello main.c
```

Тестирование и отладка:

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
```

Анализ исходного текста программы:

```
splint <file.c>
```

Выполнение работы:

1. В домашнем каталоге создайте подкаталог `~/work/os/lab_prog` (*Рисунок 1*)

Рисунок 1: создайте подкаталог `~/work/os/lab_prog`


```

#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)

```

2.3. Теперь создайте командный файл `main.c` (Рисунок 4-5)

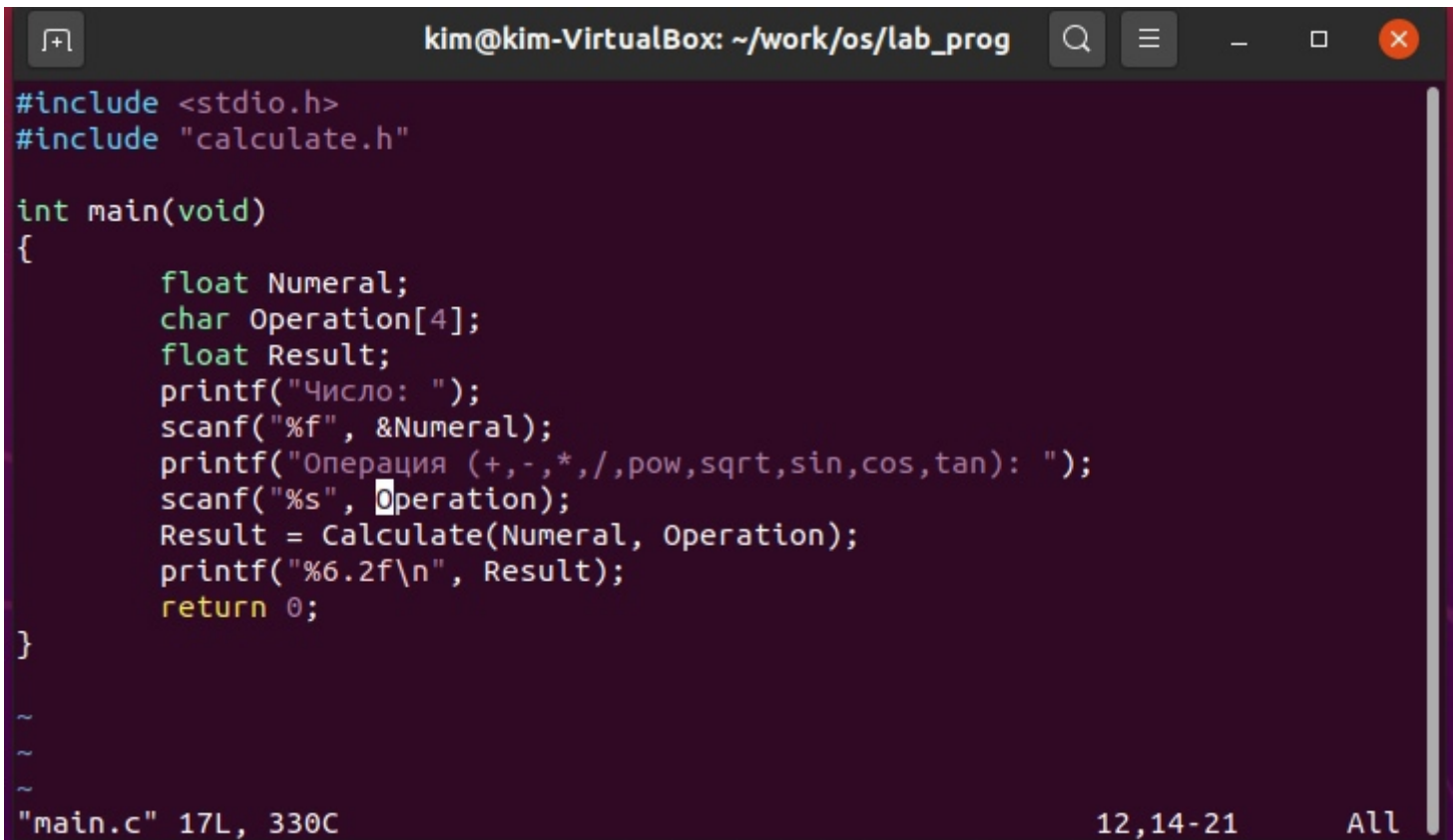
Рисунок 4: создайте командный файл `main.c`

```

kim@kim-VirtualBox:~/work/os/lab_prog$ vi main.c
kim@kim-VirtualBox:~/work/os/lab_prog$ █

```

Рисунок 5: создайте командный файл `main.c`



```
kim@kim-VirtualBox: ~/work/os/lab_prog

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

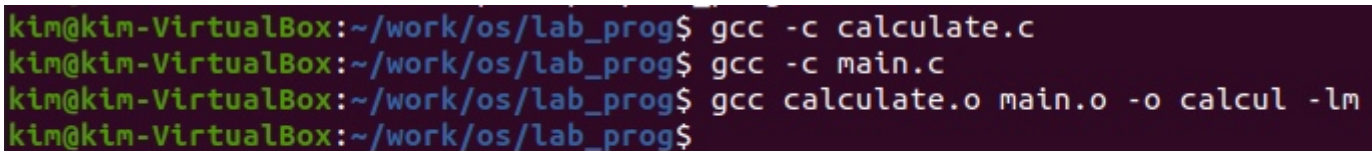
int main(void)
{
    float Numeral;
    char Operation[4];
    float Result;
    printf("Число: ");
    scanf("%f", &Numeral);
    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
    scanf("%s", Operation);
    Result = Calculate(Numeral, Operation);
    printf("%6.2f\n", Result);
    return 0;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

~
~
~
"main.c" 17L, 330C 12,14-21 All
```

3. Выполните компиляцию программы посредством `gcc` : (Рисунок 6)

```
gcc -c calculate.c
gcc -c main.c
gcc calculate.o main.o -o calcul -lm
```

Рисунок 6: `gcc`



```
kim@kim-VirtualBox:~/work/os/lab_prog$ gcc -c calculate.c
kim@kim-VirtualBox:~/work/os/lab_prog$ gcc -c main.c
kim@kim-VirtualBox:~/work/os/lab_prog$ gcc calculate.o main.o -o calcul -lm
kim@kim-VirtualBox:~/work/os/lab_prog$
```

4. При необходимости исправьте синтаксические ошибки: Как мы видим в (Рисунок 6) ошибок нет.

5. Создайте `Makefile` со следующим содержанием:

Рисунок 7: Создайте `Makefile`

```
kim@kim-VirtualBox: ~/work/os/lab_prog
#
# 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
~
~
~
"Makefile" 21L, 279C                                21,9    All
```

6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`):

Для этого сначала мы запустим некоторую команду (Рисунок 8), потому что при запуске команды `gdb` возникнут некоторые ошибки.

Рисунок 8: `gcc с -g`

```
kim@kim-VirtualBox:~/work/os/lab_prog$ gcc -c calculate.c -g
kim@kim-VirtualBox:~/work/os/lab_prog$ gcc -c main.c -g
kim@kim-VirtualBox:~/work/os/lab_prog$ gcc calculate.o main.o -o calcul -lm
kim@kim-VirtualBox:~/work/os/lab_prog$
```

- Запустите отладчик GDB, загрузив в него программу для отладки: `gdb ./calcul` (Рисунок 9)

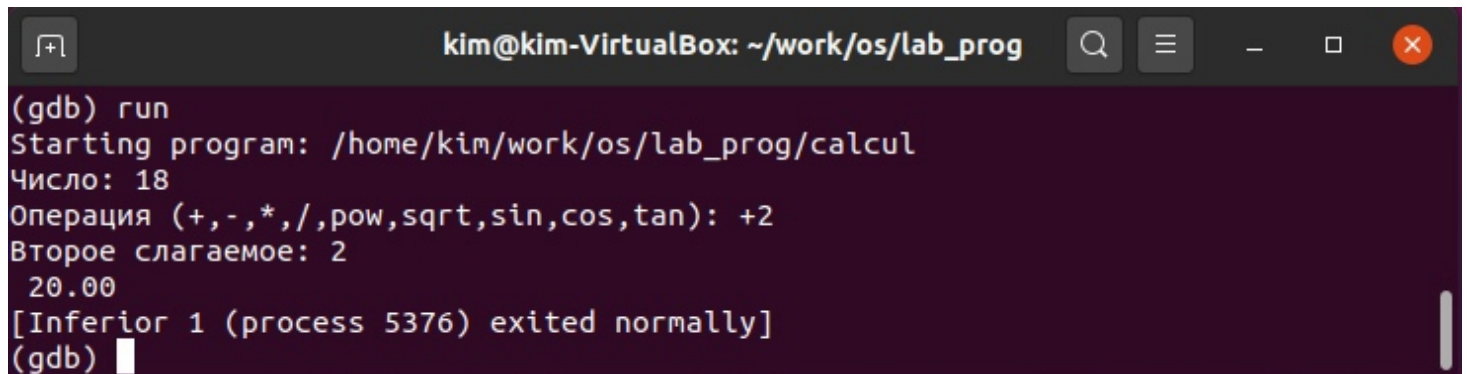
Рисунок 9: Запустите отладчик GDB


```
kim@kim-VirtualBox:~/work/os/lab_prog$ gdb ./calcul
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
Copyright (C) 2020 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:
<http://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...
(gdb) █
```

- Для запуска программы внутри отладчика введите команду run: run (Рисунок 10)

Рисунок 10: запуска программы внутри отладчика

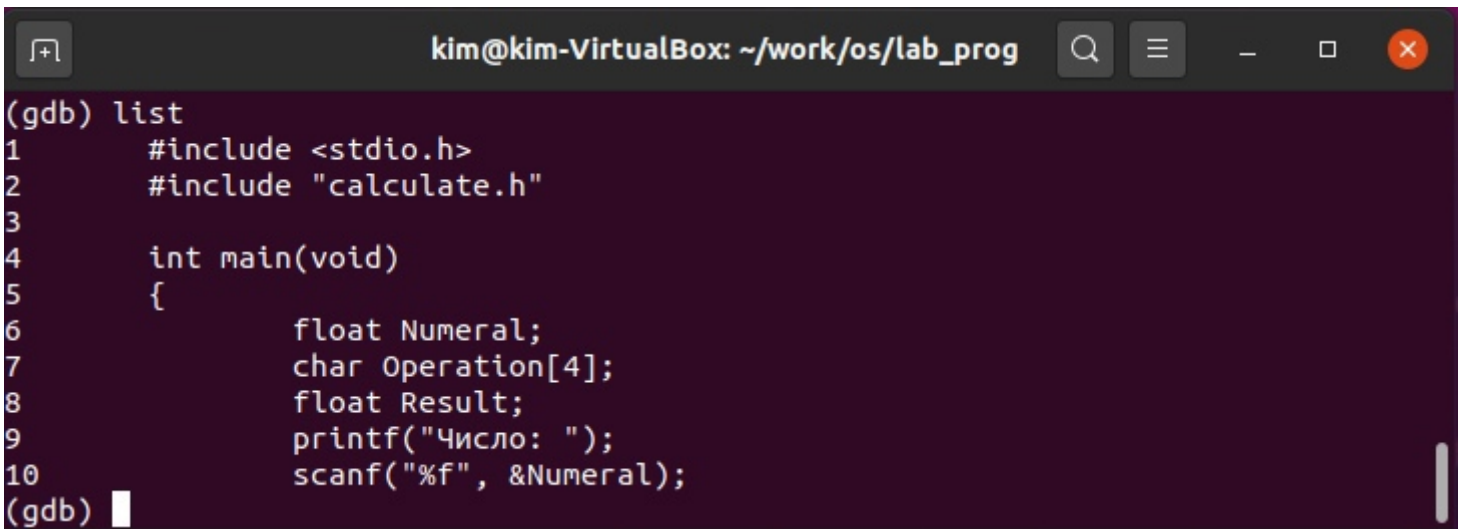


The screenshot shows a terminal window titled "kim@kim-VirtualBox: ~/work/os/lab_prog". The GDB prompt "(gdb)" is followed by the command "run". The output shows the program starting at "/home/kim/work/os/lab_prog/calcul", displaying "Число: 18", performing an operation "+2", and showing "Второе слагаемое: 2" and "20.00". The message "[Inferior 1 (process 5376) exited normally]" is displayed, and the prompt returns to "(gdb) █".

```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) run
Starting program: /home/kim/work/os/lab_prog/calcul
Число: 18
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): +2
Второе слагаемое: 2
20.00
[Inferior 1 (process 5376) exited normally]
(gdb) █
```

- Для постраничного (по 9 строк) просмотра исходного код используйте команду list: list (Рисунок 11)

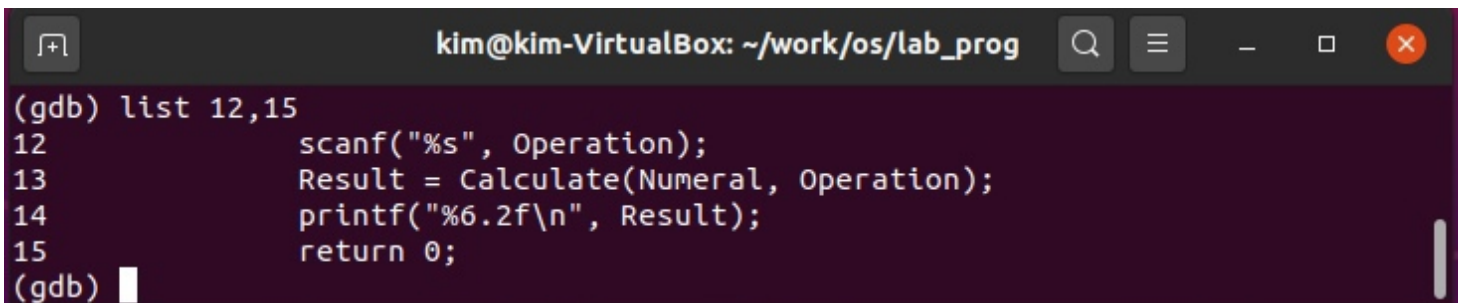
Рисунок 11: постраничного (по 9 строк)



```
kim@kim-VirtualBox: ~/work/os/lab_prog
(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);
(gdb)
```

- Для просмотра строк с 12 по 15 основного файла используйте list с параметрами:
list 12,15 (Рисунок 12)

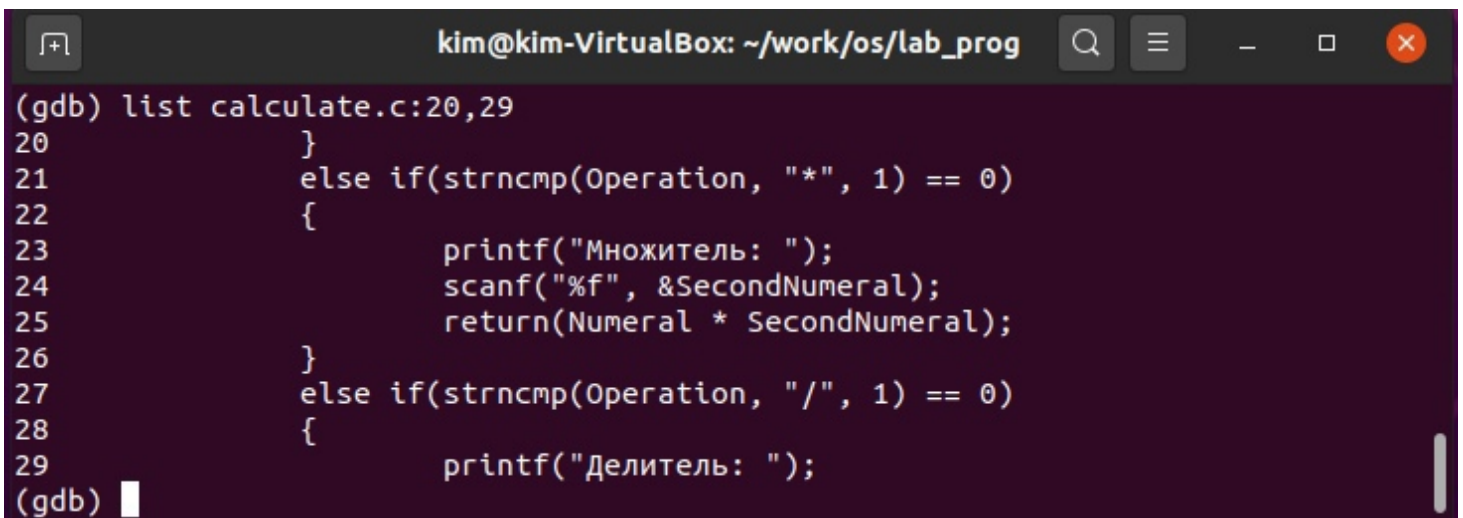
Рисунок 12: просмотра строк с 12 по 15



```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) list 12,15
12         scanf("%s", Operation);
13         Result = Calculate(Numeral, Operation);
14         printf("%6.2f\n", Result);
15         return 0;
(gdb)
```

- Для просмотра определённых строк не основного файла используйте list с параметрами:
list calculate.c:20,29 (Рисунок 13)

Рисунок 13: просмотра определённых строк

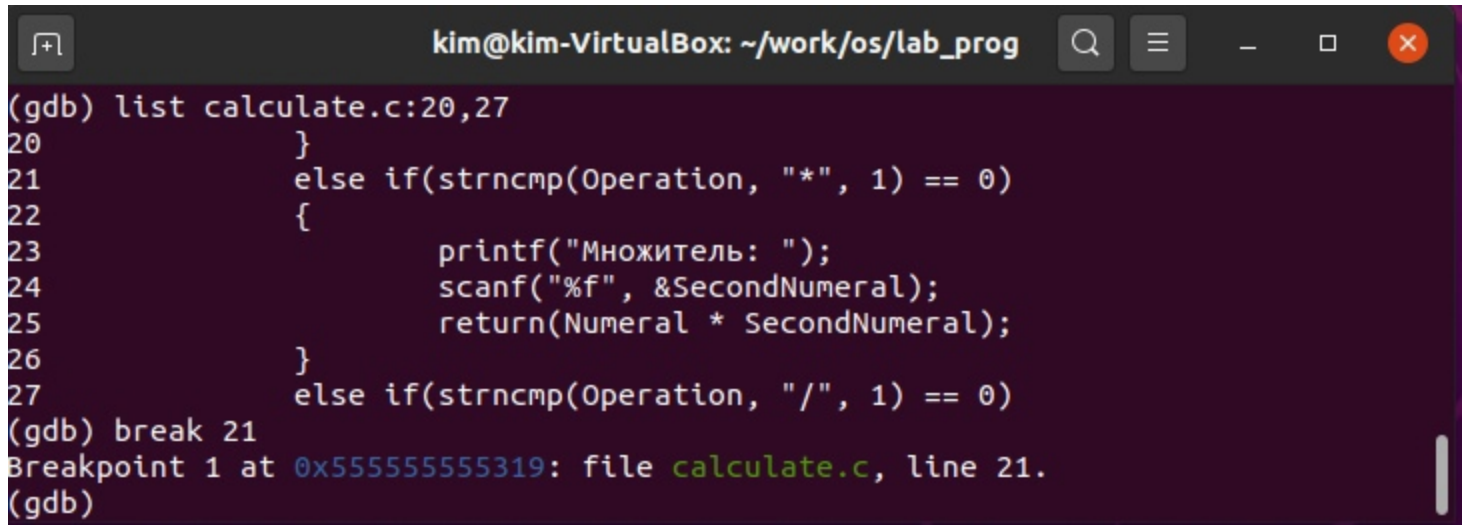


```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) list calculate.c:20,29
20         }
21         else if(strncmp(Operation, "*", 1) == 0)
22         {
23             printf("Множитель: ");
24             scanf("%f", &SecondNumeral);
25             return(Numeral * SecondNumeral);
26         }
27     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

- Установите точку останова в файле calculate.c на строке номер 21: (Рисунок 14)


```
list calculate.c:20,27
break 21
```

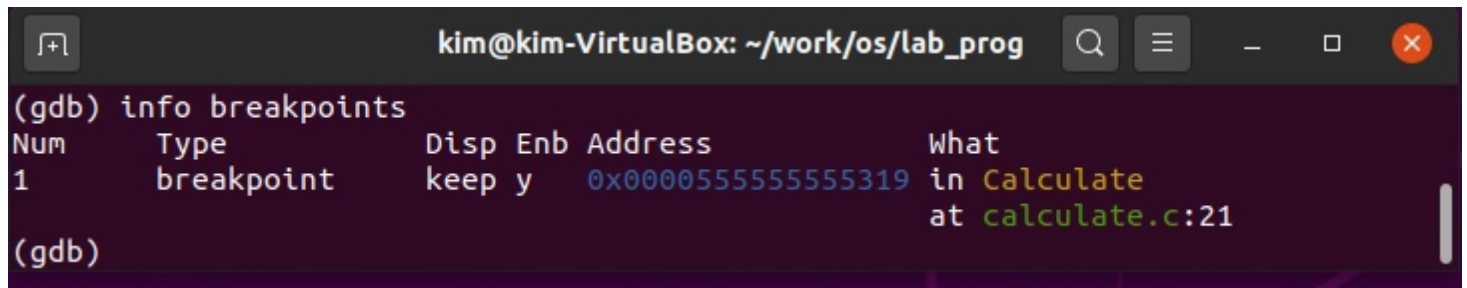
Рисунок 14: Установите точку останова



```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) list calculate.c:20,27
20         }
21         else if(strncmp(Operation, "*", 1) == 0)
22         {
23             printf("Множитель: ");
24             scanf("%f", &SecondNumeral);
25             return(Numeral * SecondNumeral);
26         }
27         else if(strncmp(Operation, "/", 1) == 0)
(gdb) break 21
Breakpoint 1 at 0x555555555319: file calculate.c, line 21.
(gdb)
```

- Выведите информацию об имеющихся в проекте точка останова: info breakpoints (Рисунок 15)

Рисунок 15: Выведите информацию об имеющихся



```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) info breakpoints
Num      Type      Disp Enb Address          What
1        breakpoint keep  y   0x0000555555555319 in Calculate
                                     at calculate.c:21
(gdb)
```

- Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова:

```
run
5
*
backtrace
```

Рисунок 16: Запустите программу

```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) run
Starting program: /home/kim/work/os/lab_prog/calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdded4 "*")
    at calculate.c:21
21      else if(strncmp(Operation, "*", 1) == 0)
(gdb)
```

Как мы видим , здесь я использовал символ*, потому что мои точки останова здесь в строке 21- символ * , поэтому нам нужно поставить * , иначе он не будет показывать сообщения, как в (Рисунок 16)

Рисунок 17: gdb backtrace

```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffdded4 "*") at calculate.c:21
#1 0x00005555555555bd in main () at main.c:13
(gdb)
```

- Отладчик выдаст следующую информацию:

```
#0 Calculate (Numeral=5, Operation=0x7fffffffdd280 "-")
    at calculate.c:21
#1 0x0000000000400b2b 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 и Сравните с результатом вывода на экран после использования команды: display Numeral (Рисунок 18)

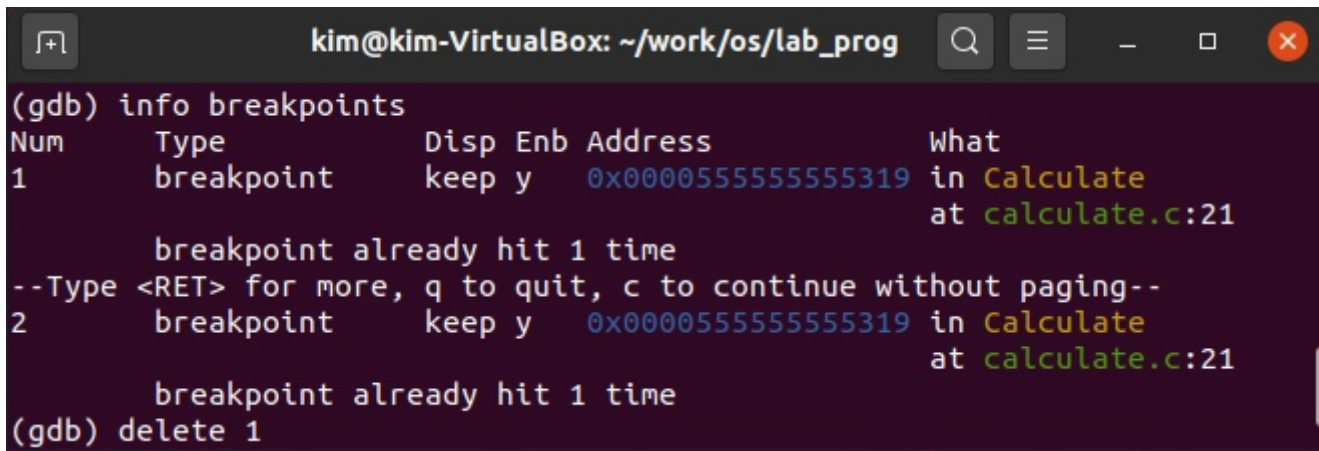
Рисунок 18: Посмотрите Numeral

```
kim@kim-VirtualBox: ~/work/os/lab_prog
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
```

- Уберите точки останова: (Рисунок 19)

```
info breakpoints
delete 1
```

Рисунок 19: Уберите точки останова

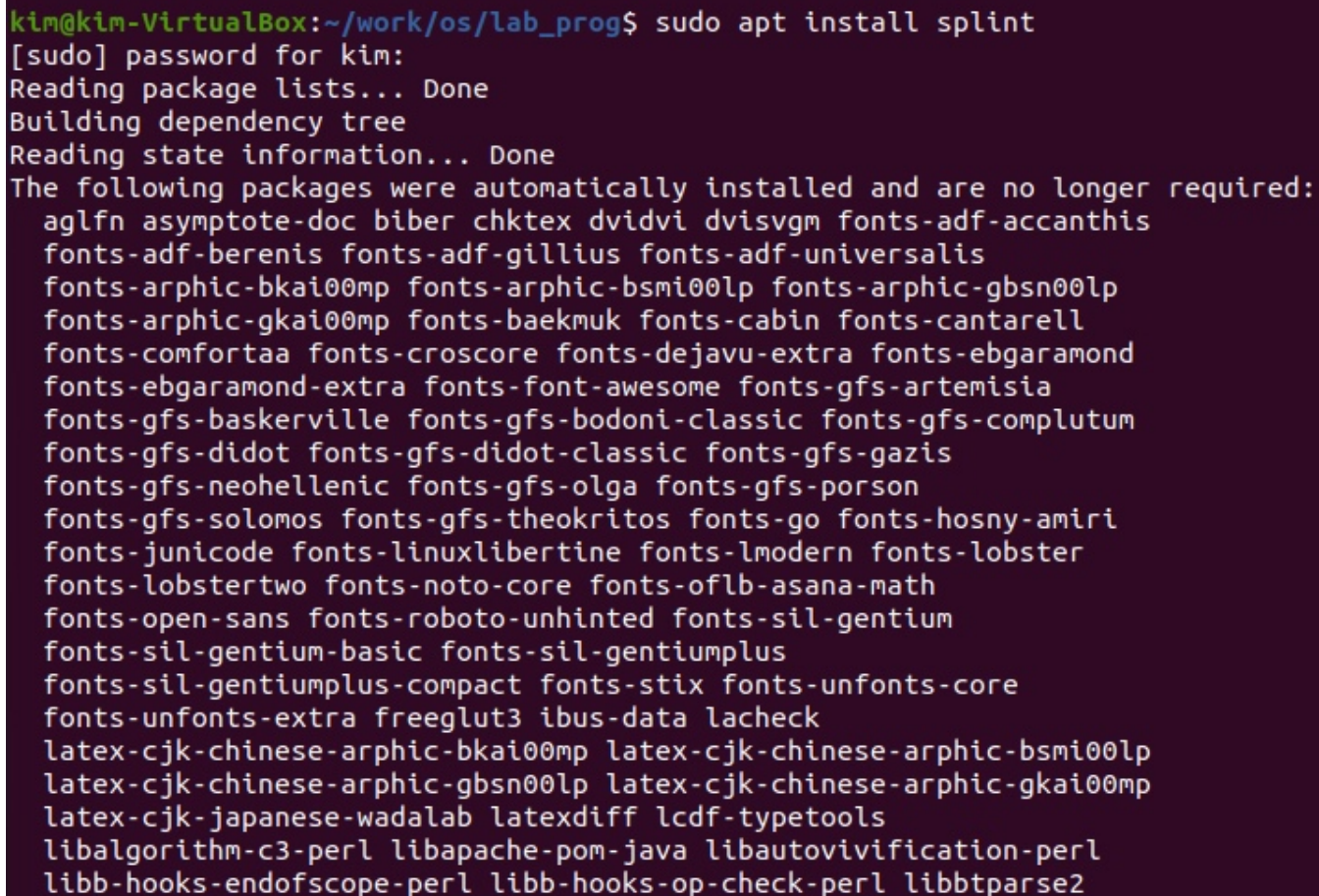


```
(gdb) info breakpoints
Num      Type      Disp Enb Address              What
1        breakpoint keep y   0x000055555555319 in calculate
                                at calculate.c:21
        breakpoint already hit 1 time
--Type <RET> for more, q to quit, c to continue without paging--
2        breakpoint keep y   0x000055555555319 in calculate
                                at calculate.c:21
        breakpoint already hit 1 time
(gdb) delete 1
```

7. С помощью утилиты `splint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

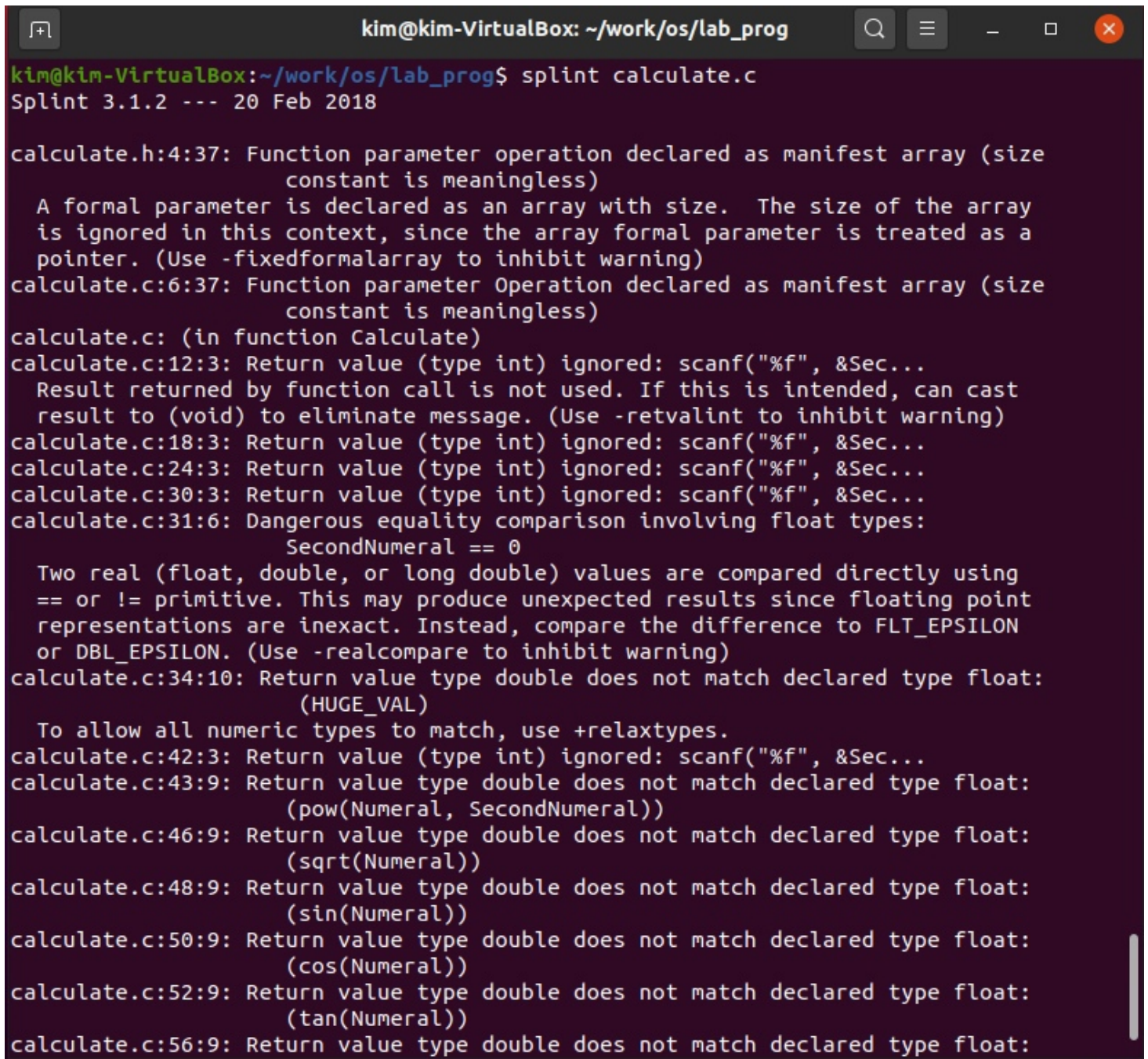
Сначала мы установим `splint` с помощью `sudo apt install splint` (Рисунок 20)

Рисунок 20: установим `splint`



```
kim@kim-VirtualBox:~/work/os/lab_prog$ sudo apt install splint
[sudo] password for kim:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following packages were automatically installed and are no longer required:
  aglfn asymptote-doc biber chktex dvidvi dvisvgm fonts-adf-accanthis
  fonts-adf-berenis fonts-adf-gillius fonts-adf-universalis
  fonts-arphic-bkai00mp fonts-arphic-bsmi00lp fonts-arphic-gbsn00lp
  fonts-arphic-gkai00mp fonts-baekmuk fonts-cabin fonts-cantarell
  fonts-comfortaa fonts-croscore fonts-dejavu-extra fonts-ebgaramond
  fonts-ebgaramond-extra fonts-font-awesome fonts-gfs-artemisla
  fonts-gfs-baskerville fonts-gfs-bodoni-classic fonts-gfs-complutum
  fonts-gfs-didot fonts-gfs-didot-classic fonts-gfs-gazis
  fonts-gfs-neohellenic fonts-gfs-olga fonts-gfs-porson
  fonts-gfs-solomos fonts-gfs-theokritos fonts-go fonts-hosny-amiri
  fonts-junicode fonts-linuxlibertine fonts-lmodern fonts-lobster
  fonts-lobstertwo fonts-noto-core fonts-oflb-asana-math
  fonts-open-sans fonts-roboto-unhinted fonts-sil-gentium
  fonts-sil-gentium-basic fonts-sil-gentiumplus
  fonts-sil-gentiumplus-compact fonts-stix fonts-unfonts-core
  fonts-unfonts-extra freeglut3 ibus-data lacheck
  latex-cjk-chinese-arphic-bkai00mp latex-cjk-chinese-arphic-bsmi00lp
  latex-cjk-chinese-arphic-gbsn00lp latex-cjk-chinese-arphic-gkai00mp
  latex-cjk-japanese-wadalab latexdiff lcdf-typetools
  libalgorithm-c3-perl libapache-pom-java libautovivification-perl
  libb-hooks-endofscope-perl libb-hooks-op-check-perl libbtparse2
```


Рисунок 21: проанализировать файлов calculate.c



```
kim@kim-VirtualBox: ~/work/os/lab_prog
kim@kim-VirtualBox:~/work/os/lab_prog$ splint calculate.c
Splint 3.1.2 --- 20 Feb 2018

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:6:37: Function parameter Operation declared as manifest array (size
                    constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:12:3: 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:18:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:24:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:30:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:31:6: 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:34:10: Return value type double does not match declared type float:
                    (HUGE_VAL)
    To allow all numeric types to match, use +relaxtypes.
calculate.c:42:3: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:43:9: Return value type double does not match declared type float:
                    (pow(Numeral, SecondNumeral))
calculate.c:46:9: Return value type double does not match declared type float:
                    (sqrt(Numeral))
calculate.c:48:9: Return value type double does not match declared type float:
                    (sin(Numeral))
calculate.c:50:9: Return value type double does not match declared type float:
                    (cos(Numeral))
calculate.c:52:9: Return value type double does not match declared type float:
                    (tan(Numeral))
calculate.c:56:9: Return value type double does not match declared type float:
```

Рисунок 22: проанализировать файлов main.c

```
kim@kim-VirtualBox: ~/work/os/lab_prog
kim@kim-VirtualBox:~/work/os/lab_prog$ splint main.c
Splint 3.1.2 --- 20 Feb 2018

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:10:2: 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:12:2: Return value (type int) ignored: scanf("%s", Oper...

Finished checking --- 3 code warnings
kim@kim-VirtualBox:~/work/os/lab_prog$
```

Контрольные вопросы:

1. С помощью функций `info` и `man`.
2.
 - создание исходного кода программы, которая 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. Использование суффикса `".c"` 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 `.c` компилятор распознает, что файл `abcd.c` должен компилироваться, а по суффиксу `.o`, 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. Компиляция всей программы в целом и получении исполняемого модуля.
5. 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.
6. 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.

7. 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.
 - `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;
- `list` — 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.
 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. Не возникло

11.
 - `cscope` - исследование функций, содержащихся в программе;
 - `splint` — критическая проверка программ, написанных на языке Си.
12.
 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.

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.

Библиография:

[1]:[Описание к Лабораторная №14](#)

[2]:[Командой splint](#)