

# 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: Губина Ольга Вячеславовна

Группа: НПИбд-01-20

Преподаватель: Велиева Татьяна Рефатовна

МОСКВА

2021 г.

#### 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.

#### Задачи:

1. Научиться выполнять компиляцию по средствам командной строки;
2. Освоить отладчик GDB;
3. Научиться анализировать исходные коды.

#### Теоретическое введение:

##### *Этапы разработки приложений*

- 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, например, файла `main.c` используют команду:

```
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 -o hello main.c
```

С прочими опциями компилятора `gcc` можно ознакомиться в статье ["Опции компиляторов \[1\]"](#).

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.

##### *Тестирование и отладка*

Для использования отладчика 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
```

Затем можно использовать по мере необходимости различные команды gdb.

### **Анализ исходного текста программы**

Для анализа кода программы example.c следует выполнить следующую команду:

```
splint example.c
```

**В ходе работы** над понадобится установить утилиту splint на Centos 7, поэтому воспользуемся статьёй "[УСТАНОВКА ПАКЕТОВ В CENTOS 7](#)"<sup>[2]</sup>.

Все коды, которые использовались во время выполнения работы были взяты из *Лабораторной работы №14*<sup>[3]</sup>.

## **Выполнение работы:**

1, 2. В домашнем каталоге нам нужно создать подкаталог ~/work/os/lab\_prog.

Каталоги work и os уже были созданы во время выполнения предыдущих лабораторных работ. Поэтому просто переходим в каталог ~/work/os и создаем в нем подкаталог lab\_prog - `mkdir lab_prog` (*рисунок 1*). Перейдем в него командой `cd lab_prog`. Создадим в нём файлы: calculate.h, calculate.c, main.c с помощью текстового редактора emacs (*рисунок 1*). Это будет примитивнейший калькулятор.

```
[ovgubina@localhost ~]$ cd
[ovgubina@localhost ~]$ cd work
[ovgubina@localhost work]$ cd os
[ovgubina@localhost os]$ mkdir lab_prog
[ovgubina@localhost os]$ cd lab_prog
[ovgubina@localhost lab_prog]$
[ovgubina@localhost lab_prog]$ emacs calculate.h
[ovgubina@localhost lab_prog]$ emacs calculate.c
[ovgubina@localhost lab_prog]$ emacs main.c
[ovgubina@localhost lab_prog]$
```

рисунок 1: создание подкаталога ~/work/os/lab\_prog и файлов calculate.h, calculate.c, main.c

В файл calculate.h вводим код на языке программирования C, предоставленный в материалах к ЛР №14 (*рисунок 2*)

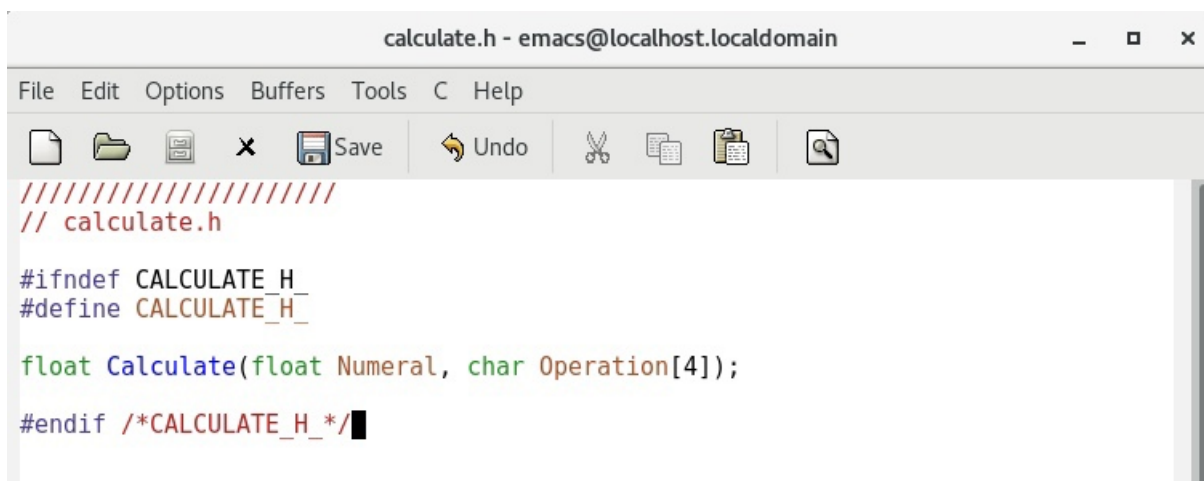


рисунок 2: файл calculate.h

То же делаем с файлами calculate.c (*рисунок 3*) и main.c (*рисунок 4*).

```
calculate.c - emacs@localhost.localdomain
File Edit Options Buffers Tools C Help
Save Undo
////////////////////////////////////
// calculate.c

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

float
Calculate(float Numeral, char Operation[4])
{
    float SecondNumeral;
    if(strcmp(Operation, "+", 1) == 0)
    {
        printf("Второе слагаемое: ");
        scanf("%f", &SecondNumeral);
        return(Numeral + SecondNumeral);
    }
    else if(strcmp(Operation, "-", 1) == 0)
    {
        printf("Вычитаемое: ");
        scanf("%f", &SecondNumeral);
        return(Numeral - SecondNumeral);
    }
    else if(strcmp(Operation, "*", 1) == 0)
    {
        printf("Множитель: ");
        scanf("%f", &SecondNumeral);
        return(Numeral * SecondNumeral);
    }
}
U:** calculate.c Top L28 (C/l Abbrev)
```

рисунок 3: файл calculate.c

```
main.c - emacs@localhost.localdomain
File Edit Options Buffers Tools C Help
Save Undo
////////////////////////////////////
// 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;
}
U:** main.c Top L28 (C/l Abbrev)
```

рисунок 4: файл main.c

3, 4. Теперь выполним компиляцию программы посредством gcc, ввода следующие команды (рисунок 5):

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

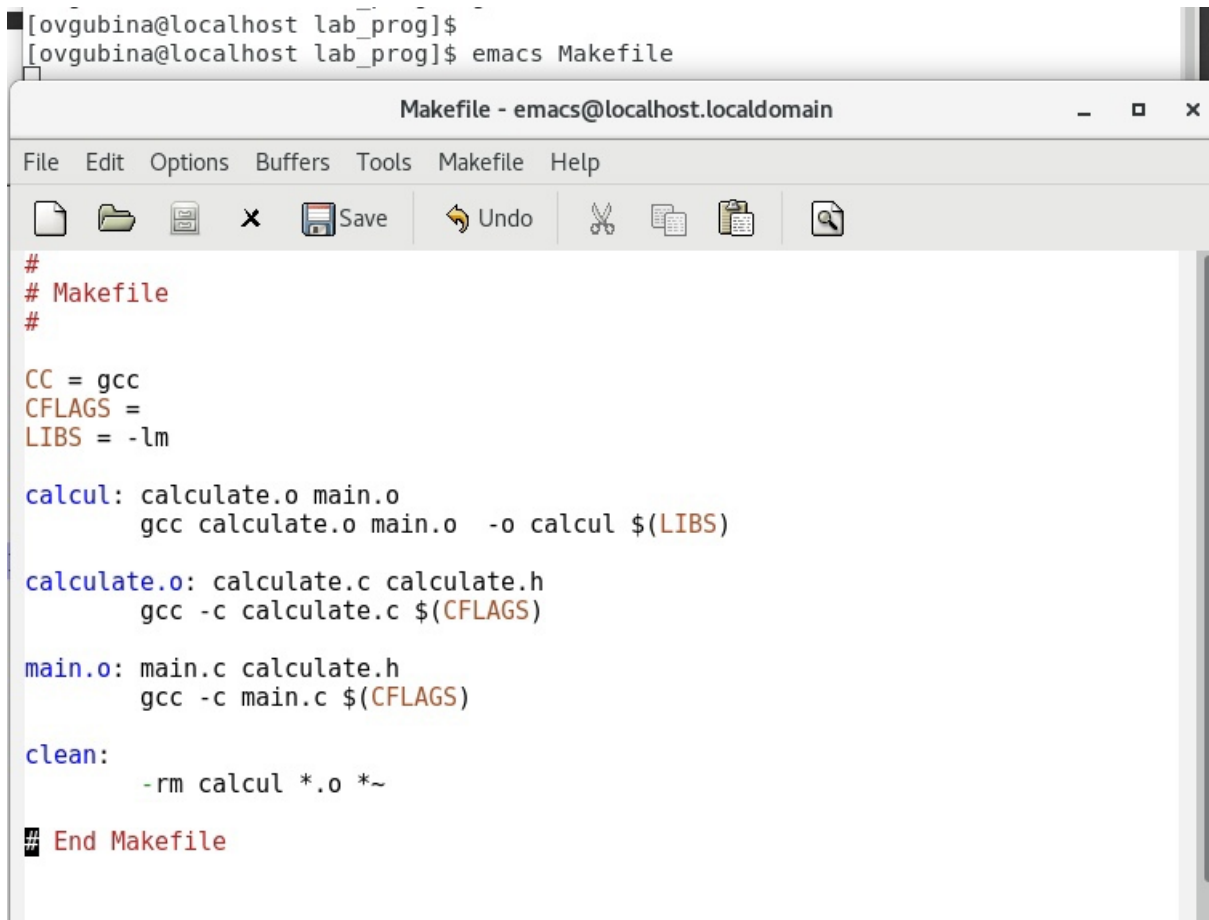
```
[ovgubina@localhost lab_prog]$ gcc -c calculate.c
[ovgubina@localhost lab_prog]$ gcc -c main.c
[ovgubina@localhost lab_prog]$ gcc calculate.o main.o -o calcul -lm
[ovgubina@localhost lab_prog]$
```

рисунок 5: компиляция программы посредством gcc

Видим, что система не выдает нам сообщений об ошибках, следовательно код написан правильно, и нам нечего исправлять.

5. Создадим Makefile, который будет расположен в каталоге lab\_prog, поскольку makefile должен находиться в том же месте, где и проект, связанный с ним. Создаем файл с помощью редактора emacs и вводим в него предложенный код файла из лабораторной работы (рисунок 6).

```
[ovgubina@localhost lab_prog]$
[ovgubina@localhost lab_prog]$ emacs Makefile
```



```
#
# Makefile
#
CC = gcc
CFLAGS =
LIBS = -lm

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

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

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

clean:
    -rm calcul *.o *~

## End Makefile
```

рисунок 6: создание Makefile

6. С помощью gdb выполним отладку программы calcul. Для использования GDB нам необходимо сначала скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле (рисунок 7). Для этого следует воспользоваться опцией -g компилятора gcc, тогда синтаксис компиляции будет следующим:

```
gcc -c [имя файла] -g
```

```
[ovgubina@localhost lab_prog]$ gcc -c calculate.c -g
[ovgubina@localhost lab_prog]$ gcc -c main.c -g
[ovgubina@localhost lab_prog]$ gcc calculate.o main.o -o calcul -lm
[ovgubina@localhost lab_prog]$
```

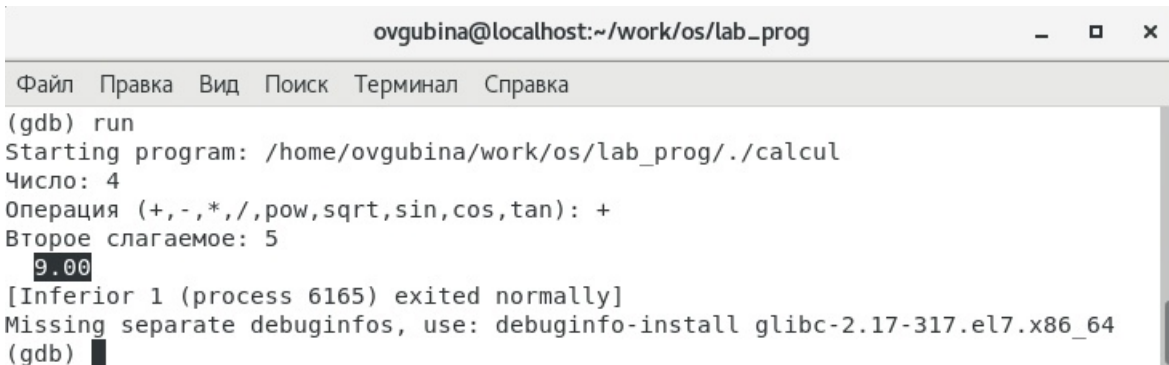
рисунок 7: компиляция перед запуском GDB

- Запустим отладчик GDB, загрузив в него программу для отладки: `gdb ./calcul` (рисунок 8).

```
[ovgubina@localhost lab_prog]$  
[ovgubina@localhost lab_prog]$ gdb ./calcul  
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-120.el7  
Copyright (C) 2013 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-redhat-linux-gnu".  
For bug reporting instructions, please see:  
<http://www.gnu.org/software/gdb/bugs/>...  
Reading symbols from /home/ovgubina/work/os/lab_prog/calcul...done.  
(gdb) █
```

рисунок 8: запуск отладчика GDB

- Теперь нам нужно запустить программу внутри отладчика. Для этого внутри отладчика введем команду `run` (рисунок 9).

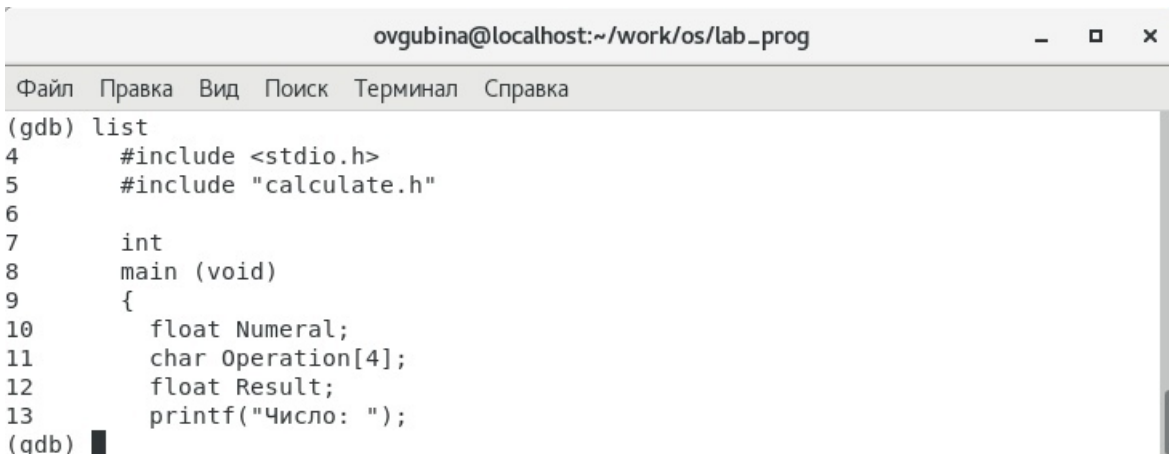


```
ovgubina@localhost:~/work/os/lab_prog  
Файл Правка Вид Поиск Терминал Справка  
(gdb) run  
Starting program: /home/ovgubina/work/os/lab_prog/./calcul  
Число: 4  
Операция (+, -, *, /, pow, sqrt, sin, cos, tan): +  
Второе слагаемое: 5  
9.00  
[Inferior 1 (process 6165) exited normally]  
Missing separate debuginfos, use: debuginfo-install glibc-2.17-317.el7.x86_64  
(gdb) █
```

рисунок 9: запуск программы в отладчике

Видим, что программа была успешно запущена. На ввод нам предлагается ввести какое-либо число (вводим 4), операцию, которая будет производится с ним (в нашем случае это сложение), далее нам предлагается ввести второе слагаемое (5). Результат выделен черным - 9. Программа работает исправно.

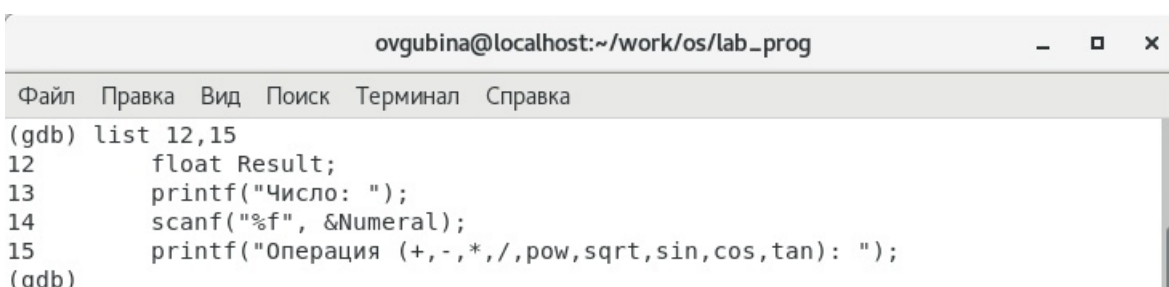
- Для постраничного (по 9 строк) просмотра исходного код используем команду `list` (рисунок 10). Видим, что действительно вывелось 9 строк (4-13).



```
ovgubina@localhost:~/work/os/lab_prog  
Файл Правка Вид Поиск Терминал Справка  
(gdb) list  
4      #include <stdio.h>  
5      #include "calculate.h"  
6  
7      int  
8      main (void)  
9      {  
10         float Numeral;  
11         char Operation[4];  
12         float Result;  
13         printf("Число: ");  
(gdb) █
```

рисунок 10: постраничный вывод list

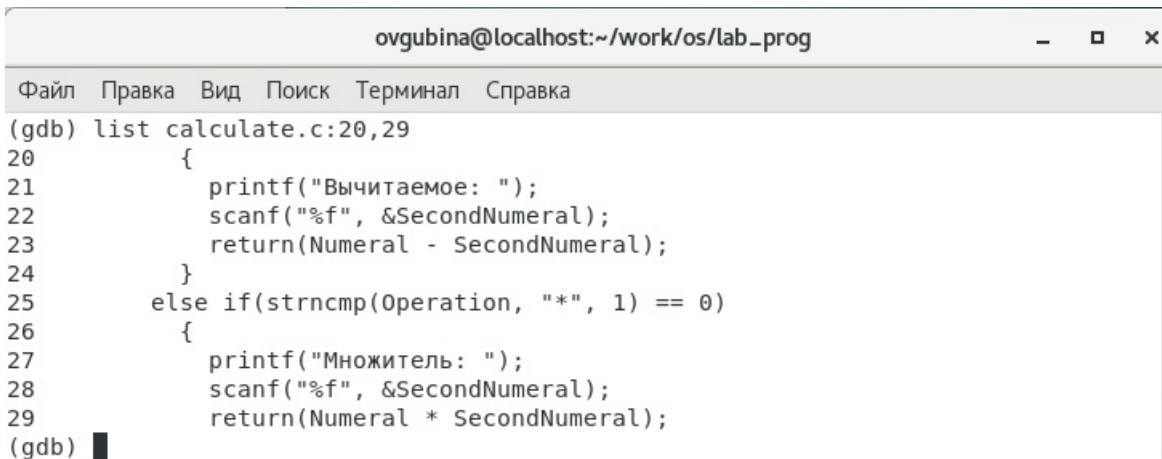
- Для просмотра строк с 12 по 15 основного файла используем list с параметрами - `list 12,15` (рисунок 11).



```
ovgubina@localhost:~/work/os/lab_prog  
Файл Правка Вид Поиск Терминал Справка  
(gdb) list 12,15  
12         float Result;  
13         printf("Число: ");  
14         scanf("%f", &Numeral);  
15         printf("Операция (+, -, *, /, pow, sqrt, sin, cos, tan): ");  
(gdb) █
```

рисунок 11: просмотр определенных строк

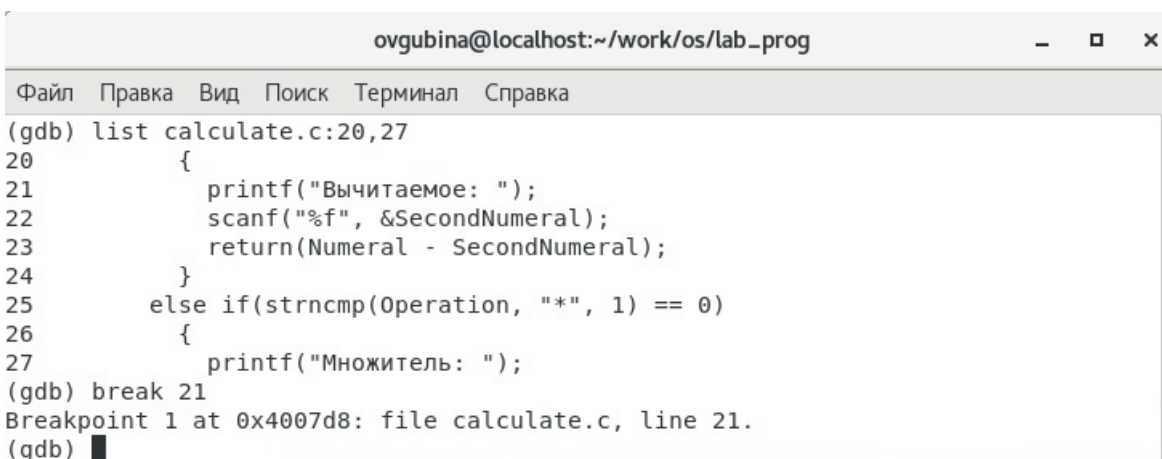
- Для просмотра определённых строк не основного файла используем `list` с параметрами: `list calculate.c:20,29` (рисунок 12).



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

рисунок 12: просмотр определённых строк не основного файла

- Установим точку останова в файле `calculate.c` на строке номер 21: `break 21` (рисунок 13).

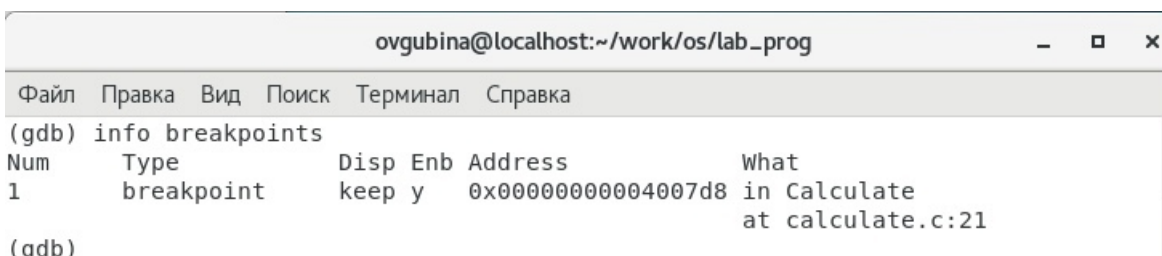


```
ovgubina@localhost:~/work/os/lab_prog
Файл Правка Вид Поиск Терминал Справка
(gdb) list calculate.c:20,27
20      {
21          printf("Вычитаемое: ");
22          scanf("%f", &SecondNumeral);
23          return(Numeral - SecondNumeral);
24      }
25      else if(strncmp(Operation, "*", 1) == 0)
26      {
27          printf("Множитель: ");
(gdb) break 21
Breakpoint 1 at 0x4007d8: file calculate.c, line 21.
(gdb)
```

рисунок 13: установка точки останова

Видим, что точка была успешно установлена.

- Выведем информацию об имеющихся в проекте точка останова. Для этого введем команду `info breakpoints` (рисунок 14).



```
ovgubina@localhost:~/work/os/lab_prog
Файл Правка Вид Поиск Терминал Справка
(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1      breakpoint       keep y   0x00000000004007d8 in Calculate
                                at calculate.c:21
(gdb)
```

рисунок 14: информация об имеющихся в проекте точка останова

Можем наблюдать информацию о точке, которую мы только что установили: ее номер, тип, адрес, место установки.

- Запустим программу внутри отладчика с помощью `run` и убедимся, что программа остановится в момент прохождения точки останова (рисунок 15). На ввод подаем число 5, в качестве операции - вычитание. Видим, что программа останавливается на строке 21, куда была установлена точка останова, и дальше не идет.



```
ovgubina@localhost:~/work/os/lab_prog
Файл Правка Вид Поиск Терминал Справка
(gdb) run
Starting program: /home/ovgubina/work/os/lab_prog/./calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdf30 "-")
  at calculate.c:21
21      printf("Вычитаемое: ");
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffdf30 "-") at calculate.c:21
#1 0x00000000400a90 in main () at main.c:17
(gdb)
```

рисунок 15: запуск программы с установленной точкой останова

При вводе команды `backtrace` отладчик выдает следующую информацию:

```
#0 Calculate (Numeral=5, Operation=0x7fffffffdf30 "-")
  at calculate.c:21
#1 0x00000000400b2b in main () at main.c:17
```

Она показывает весь стек вызываемых функций от начала программы до текущего места.

- Посмотрим, чему равно на этом этапе значение переменной `Numeral`, введя `print Numeral` и сравним ее вывод с результатом вывода `display Numeral` (рисунок 16). Видим, что в обоих случаях выводится число 5, что и ожидалось.

```
ovgubina@localhost:~/work/os/lab_prog
Файл Правка Вид Поиск Терминал Справка
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb)
```

рисунок 16: сравнение вывода `print Numeral` и `display Numeral`

- Убираем точки останова. Сначала посмотрим информацию о текущих точках `info breakpoints`, чтобы узнать номер точки, которую мы собираемся удалить. Далее удаляем ее `delete 1`, где 1 - номер точки. Снова просматриваем `info breakpoints`, чтобы убедиться в удалении точки (рисунок 17).

```
ovgubina@localhost:~/work/os/lab_prog
Файл Правка Вид Поиск Терминал Справка
(gdb) info breakpoints
Num  Type      Disp Enb Address          What
1    breakpoint keep y  0x000000004007d8 in Calculate
                                at calculate.c:21

    breakpoint already hit 1 time
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)
```

рисунок 17: удаление точки останова

Видим по последнему выводу, что точка была успешно удалена.

- Теперь с помощью утилиты `splint` нам нужно проанализировать коды файлов `calculate.c` и `main.c`. Для этого сначала установим данную утилиту, перейдя в режим суперпользователя `su` (рисунки 18-19).

```

[ovgubina@localhost lab_prog]$ su
Пароль:
[root@localhost lab_prog]# yum install epel-release
Загружены модули: fastestmirror, langpacks
Loading mirror speeds from cached hostfile
 * base: mirror.axelname.ru
 * extras: mirror.docker.ru
 * updates: mirror.docker.ru
base | 3.6 kB | 00:00
extras | 2.9 kB | 00:00
updates | 2.9 kB | 00:00
Разрешение зависимостей
--> Проверка сценария
---> Пакет epel-release.noarch 0:7-11 помечен для установки
--> Проверка зависимостей окончена

Зависимости определены

```

```

=====
Package                Архитектура  Версия      Репозиторий  Размер
=====
Установка:

```

рисунок 18: установка утилиты splint

```

ovgubina@localhost:/home/ovgubina/work/os/lab_prog
Файл  Правка  Вид  Поиск  Терминал  Справка
Установлено:
  epel-release.noarch 0:7-11

Выполнено!
[root@localhost lab_prog]# yum install -y splint
Загружены модули: fastestmirror, langpacks
Заблокировано /var/run/yum.pid: другая копия запущена с pid 6556.
Another app is currently holding the yum lock; waiting for it to exit...
  Другое приложение: PackageKit
    Память   : 32 М RSS (932 MB VSZ)
    Запущено : Sat Jun  5 12:49:34 2021 — 00:20 назад
    Статус   : Ожидание, pid: 6556
Another app is currently holding the yum lock; waiting for it to exit...
  Другое приложение: PackageKit
    Память   : 36 М RSS (928 MB VSZ)
    Запущено : Sat Jun  5 12:49:34 2021 — 00:22 назад
    Статус   : Запуск, pid: 6556
Another app is currently holding the yum lock; waiting for it to exit...
  Другое приложение: PackageKit
    Память   : 76 М RSS (974 MB VSZ)
    Запущено : Sat Jun  5 12:49:34 2021 — 00:24 назад
    Статус   : Запуск, pid: 6556
Another app is currently holding the yum lock; waiting for it to exit...
  Другое приложение: PackageKit

```

рисунок 19: установка утилиты splint

Выходим из режима суперпользователя и анализируем коды файлов calculate.c (рисунок 20) и main.c (рисунок 21) через `splint [имя файла]`.



```

[root@localhost lab_prog]# su ovgubina
[ovgubina@localhost lab_prog]$ splint calculate.c
Splint 3.1.2 --- 11 Oct 2015

```

```

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
[ovgubina@localhost lab_prog]$ █

```

рисунок 20: анализ кода calculate.c

```

[ovgubina@localhost lab_prog]$ splint main.c
Splint 3.1.2 --- 11 Oct 2015

```

```

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:16:15: 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", &Ope...

```

```

Finished checking --- 4 code warnings
[ovgubina@localhost lab_prog]$ █

```

рисунок 21: анализ кода main.c

Видим, что утилита `splint` анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, а также обнаруживает синтаксические и семантические ошибки.

## Вывод:

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.

## Библиография:

- [1]: [Опции компиляторов](#)
- [2]: [УСТАНОВКА ПАКЕТОВ В CENTOS 7](#)
- [3]: [Лабораторная работа №14](#)

## Контрольные вопросы:

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

Можно воспользоваться справкой `man`.

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

Этапы разработки приложений UNIX:

- создание исходного кода программы;
- 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;
- сохранение всех изменений, выполняемых при тестировании и отладке.

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

Суффикс определяет тип компиляции для файла. Суффиксы и префиксы указывают на тип объекта. Например: по суффиксу `.c` компилятор распознает, что файл должен компилироваться, а по суффиксу `.o`, что файл 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`.

### 4. Каково основное назначение компилятора языка C в UNIX?

Сборка команды перед работой и выявление ошибок синтаксиса и семантики.

### 5. Для чего предназначена утилита `make`?

`make` — утилита предназначенная для автоматизации преобразования файлов из одной формы в другую. Правила преобразования задаются в скрипте с именем `Makefile`, который должен находиться в корне рабочей директории проекта.

**\*\*6.** Приведите пример структуры `Makefile`. Дайте характеристику основным элементам этого файла.

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; `:` — 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. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

Все программы отладки позволяют отслеживать состояние программы на любом из этапов ее исполнения. Для того чтобы эту возможность использовать необходимо изучить документацию по использованию определенного отладчика. Понять общие принципы отладки

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

1. Запуск отладчика
2. Просмотр кода постранично
3. Установка точки останова
4. Тест программы путем ввода параметров
5. Снятие точек

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

Не сталкивалась с ошибками во время выполнения.

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

- `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, а также обнаруживает синтаксические и семантические ошибки.