

Лабораторная работа №14 по предмету  
Операционные системы

Группа НПМбв-01-19

Бондаренко Артем Федорович

# Содержание

Цель работы	5
Задание	6
Выполнение лабораторной работы	13
Выводы	21
Ответы на контрольные вопросы	22

## Список иллюстраций

1	Процесс создания директорий в терминале . . . . .	13
2	Процесс создания необходимых файлов в терминале . . . . .	13
3	Фрагмент одного из заполненных файлов . . . . .	14
4	Результат компиляции . . . . .	14
5	Makefile с содержанием из лабораторной работы . . . . .	15
6	Исправленный Makefile . . . . .	16
7	Запуск отладчика и введённая команда run . . . . .	16
8	Результат выполнения команд в терминале . . . . .	17
9	Команда list с параметрами calculate.c:20,29 . . . . .	17
10	Установка точки останова . . . . .	18
11	Информацию о точках останова . . . . .	18
12	Информация о точках останова . . . . .	18
13	Информация по backtrace . . . . .	18
14	Информация после ввода print Numeral и display Numeral . . . . .	19
15	Удаление точек останова . . . . .	19
16	Анализ кода файла calculate.c утилитой splint . . . . .	19
17	Анализ кода файла main.c утилитой splint . . . . .	20

## Список таблиц

## 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.

# 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 результат и остановится.

Реализация функций калькулятора в файле `calculate.h`:

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

```

```

scanf("%f",&SecondNumeral);
return(pow(Numeral, SecondNumeral));
}
else if(strncmp(Operation, "sqrt", 4) == 0)
return(sqrt(Numeral));
else if(strncmp(Operation, "sin", 3) == 0)
return(sin(Numeral));
else if(strncmp(Operation, "cos", 3) == 0)
return(cos(Numeral));
else if(strncmp(Operation, "tan", 3) == 0)
return(tan(Numeral));
else
{
printf("Неправильно введено действие ");
return(HUGE_VAL);
}
}

```

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

```

////////////////////////////////////
// calculate.h
#ifndef CALCULATE_H_
#define CALCULATE_H_
float Calculate(float Numeral, char Operation[4]);
#endif /*CALCULATE_H_*/

```

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

```

////////////////////////////////////

```



```
// 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;
}
```

3. Выполните компиляцию программы посредством gcc:

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

4. При необходимости исправьте синтаксические ошибки.

5. Создайте Makefile со следующим содержанием:

```
#
# 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
```

Поясните в отчёте его содержание.

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:

```
run
```

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

```
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 точка останова:

```
info breakpoints
```

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

```
run
```

```
5
```

```
-
```

```
backtrace
```

– Отладчик выдаст следующую информацию:

```
#0 Calculate (Numeral=5, Operation=0x7ffffffd280 "-")
```

```
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. – 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

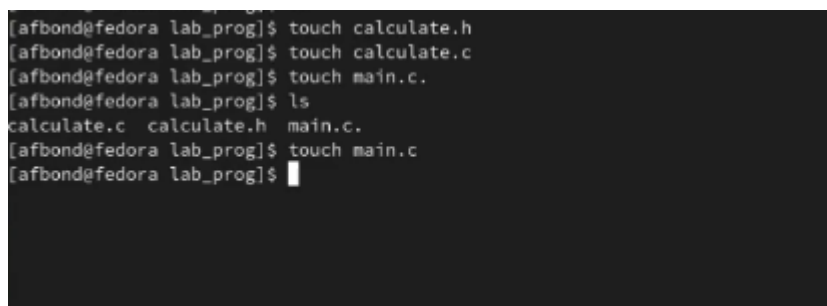
В домашнем каталоге создал подкаталог `~/work/os/lab_prog`. (Ссылка: Рис. 1)

A terminal window titled 'afbond@fedora:~/work/os/lab\_prog' with search and menu icons in the top right. The terminal shows the following commands and output:

```
[afbond@fedora ~]$ mkdir -p ~/work/os/lab_prog
[afbond@fedora ~]$ cd ~/work/os/lab_prog
[afbond@fedora lab_prog]$
```

Рис. 1: Процесс создания директорий в терминале

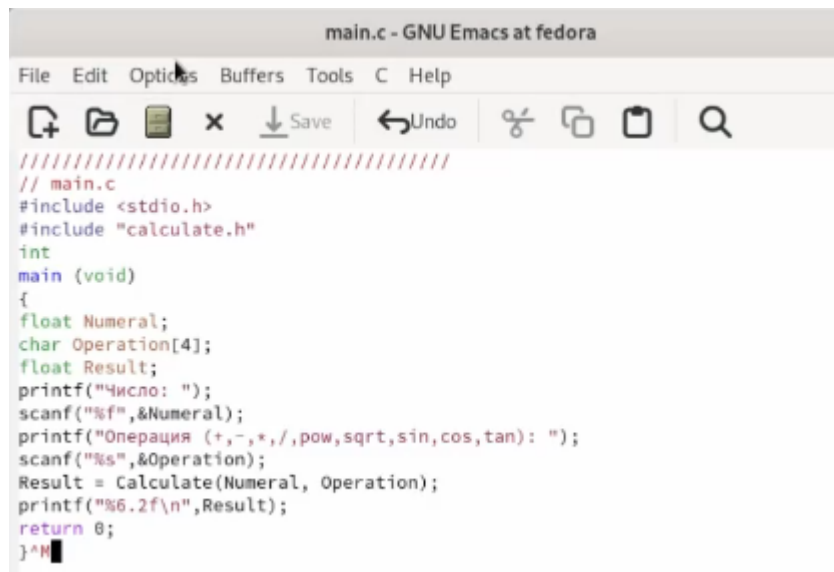
Создал в нём файлы: `calculate.h`, `calculate.c`, `main.c`. (Ссылка: Рис. 2)

A terminal window showing the creation of files in the 'lab\_prog' directory. The commands and output are:

```
[afbond@fedora lab_prog]$ touch calculate.h
[afbond@fedora lab_prog]$ touch calculate.c
[afbond@fedora lab_prog]$ touch main.c
[afbond@fedora lab_prog]$ ls
calculate.c calculate.h main.c
[afbond@fedora lab_prog]$ touch main.c
[afbond@fedora lab_prog]$
```

Рис. 2: Процесс создания необходимых файлов в терминале

Каждый из созданных файлов заполнил кодом из лабораторной работы. (Ссылка: Рис. 3)

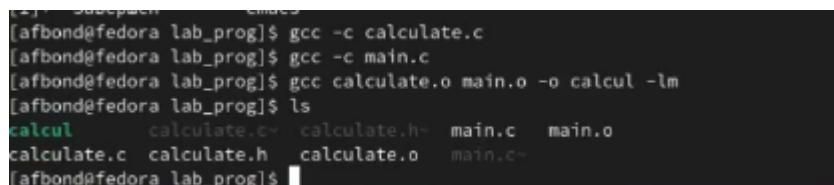


```
main.c - GNU Emacs at fedora
File Edit Options Buffers Tools C Help
[Icons: Open, Save, Undo, Redo, Find, etc.]
////////////////////////////////////
// main.c
#include <stdio.h>
#include "calculate.h"
int
main (void)
{
    float Numeral;
    char Operation[4];
    float Result;
    printf("Число: ");
    scanf("%f",&Numeral);
    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
    scanf("%s",&Operation);
    Result = Calculate(Numeral, Operation);
    printf("%.2f\n",Result);
    return 0;
}
```

Рис. 3: Фрагмент одного из заполненных файлов

Получился 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.

Выполнил компиляцию программы посредством gcc. (Ссылка: Рис. 4)



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

Рис. 4: Результат компиляции

Создал Makefile со следующим содержанием: (Ссылка: Рис. 5)

```

#
# 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

```

Рис. 5: Makefile с содержанием из лабораторной работы

Пояснение по Makefile, он используется для сборки программы calcul.

CC = gcc: это задание переменной CC, которая используется для определения компилятора.

CFLAGS =: это задание переменной CFLAGS, которая используется для добавления дополнительных флагов компилятора.

LIBS = -lm: это задание переменной LIBS, которая используется для добавления библиотек в программу.

calcul: calculate.o main.o: это задание цели calcul, которая зависит от файлов calculate.o и main.o.

gcc calculate.o main.o -o calcul \$(LIBS): это команда сборки для создания исполняемого файла calcul.

calculate.o: calculate.c calculate.h: это задание для сборки файла calculate.o из файлов calculate.c и calculate.h.

gcc -c calculate.c \$(CFLAGS): это команда компиляции для создания файла calculate.o.

main.o: main.c calculate.h: это задание для сборки файла main.o из файлов main.c и calculate.h.

gcc -c main.c \$(CFLAGS): это команда компиляции для создания файла main.o.

clean: -rm calcul .o ~: это задание для очистки (удаления) всех созданных файлов и объектов.

С помощью gdb выполняю отладку программы calcul (перед использованием gdb исправил Makefile(Ссылка: Рис. 6)):

```
#
# Makefile
#

CC = gcc
CFLAGS = -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. 6: Исправленный Makefile

Запустил отладчик GDB, загрузив в него программу для отладки. (Ссылка: Рис. 7)

Для запуска программы внутри отладчика ввел команду run.(Ссылка: Рис. 7)

```
[afbond@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 12.1-4.fc37
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.ht
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".
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...
(gdb) run
Starting program: /home/afbond/work/os/lab_prog/calcul

This GDB supports auto-downloading debuginfo from the following URLs:
https://debuginfod.fedoraproject.org/
Enable debuginfod for this session? (y or [n])
```

Рис. 7: Запуск отладчика и введенная команда run





```
(gdb) list calculate.c:20,27
20     scanf("%f",&SecondNumeral);
21     return(Numeral - SecondNumeral);
22 }
23 else if(strncmp(Operation, "+", 1) == 0)
24 {
25     printf("Множитель: ");
26     scanf("%f",&SecondNumeral);
27     return(Numeral + SecondNumeral);
(gdb) break 21
Breakpoint 1 at 0x0000000000401234: file calculate.c, line 21.
(gdb)
```

Рис. 10: Установка точки останова

Вывел информацию об имеющихся в проекте точка останова. (Ссылка: Рис. 11)

```
(gdb) info break
Num   Type           Disp Enb Address            What
1     breakpoint     keep y   0x0000000000401234 in calculate
                                at calculate.c:21
(gdb)
```

Рис. 11: Информацию о точках останова

Запустил программу внутри отладчика и убедился, что программа остановилась в момент прохождения точки останова. (Ссылка: Рис. 12)

```
(gdb) run
Starting program: /home/afbond/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: 2

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdf14 "-") at calculate.c:21
21     return(Numeral - SecondNumeral);
(gdb)
```

Рис. 12: Информацию о точках останова

Ввёл backtrace и отладчик выдал следующую информацию: (Ссылка: Рис. 13)

```
(gdb) backtrace
#0 Calculate (Numeral=5, Operation=0x7fffffffdf14 "-") at calculate.c:21
#1 0x00000000004014eb in main () at main.c:15
(gdb)
```

Рис. 13: Информация по backtrace

Посмотрел, чему равно на этом этапе значение переменной Numeral, введя: `print Numeral`. Затем ввёл `display Numeral` и сравнил результаты. И там и там значение 5, но с небольшими отличиями. (Ссылка: Рис. 14)

```
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) i
```

Рис. 14: Информация после ввода `print Numeral` и `display Numeral`

Убрал точки останова (Ссылка: Рис. 15)

```
(gdb) info break
Num Type      Disp Enb Address      What
1   breakpoint keep y  0x00000000401234 in Calculate
                        at calculate.c:21
      breakpoint already hit 1 time
(gdb) delete 1
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

Рис. 15: Удаление точек останова

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

```
[afbond@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:5:37: 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:8:31: Function parameter Operation declared as manifest array (size
constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:14:1: 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:20:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:26:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:32:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:33:4: Dangerous equality comparison involving float types:
```

Рис. 16: Анализ кода файла `calculate.c` утилитой `splint`

```
[afbond@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 2022

calculate.h:5:37: 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:12:1: 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:14:12: 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:14:9: Corresponding format code
main.c:14:1: Return value (type int) ignored: scanf("%s", &Ope...

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

Рис. 17: Анализ кода файла main.c утилитой splint

## Выводы

Таким образом, были приобретены простейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями.

# Ответы на контрольные вопросы

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

Для получения информации о возможностях программ `gcc`, `make`, `gdb` и других утилит, установленных в системе, можно использовать команды `man` и `info` в терминале Linux.

Команда `man` позволяет просмотреть мануал (руководство пользователя) по конкретной утилите.

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.

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

Суффикс в контексте языка программирования - это часть имени файла, которая указывает на его тип или назначение. Например, в операционной системе Unix/Linux суффикс “.c” обычно используется для исходных файлов программ на языке C, а суффикс “.o” - для объектных файлов, получаемых в результате компиляции исходных файлов.

Еще одним примером может быть суффикс “.py” для файлов программ на языке Python, суффикс “.java” для файлов программ на языке Java, и т.д.

Суффиксы используются для облегчения процесса компиляции, линковки и сборки программного кода, так как позволяют автоматически определять тип файлов и выполнять соответствующие действия. Кроме того, суффиксы также используются для определения типа файлов в операционных системах и приложениях, что позволяет им обрабатывать файлы соответствующим образом.

4. Каково основное назначение компилятора языка C в UNIX?

Основное назначение компилятора языка C в UNIX - это преобразование исходного кода программы, написанной на языке C, в машинный код, который может быть исполнен процессором компьютера. Компилятор языка C является одним из основных инструментов разработки программного обеспечения в UNIX-подобных операционных системах. Он позволяет разработчикам писать эффективный и портируемый код на языке C, который может быть выполнен на различных аппаратных платформах и операционных системах. Кроме того, компилятор языка C может использоваться для оптимизации кода, анализа его качества и обнаружения ошибок во время компиляции.

5. Для чего предназначена утилита make?

Утилита make предназначена для автоматизации процесса компиляции программ. Она позволяет автоматически определять, какие файлы программы были изменены,

и перекомпилировать только их, а также устанавливать программу в систему и запускать тесты. В результате использования утилиты make значительно сокращается время, необходимое для сборки программы, и уменьшается вероятность ошибок, возникающих при ручной компиляции. Основным принципом работы make является использование файла Makefile, в котором задаются правила сборки программы и зависимости между файлами.

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

Общий синтаксис Makefile имеет вид:

```
target1 [target2...]:[.] [dependment1...]  
[(tab)commands] [#commentary]  
[(tab)commands] [#commentary]
```

7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

Основным свойством, присущим всем программам отладки, является возможность остановки выполнения программы на определенном месте, наблюдения за ее состоянием и изменения этого состояния в процессе выполнения.

Чтобы использовать это свойство, необходимо подготовить исходный код программы и запустить ее в среде отладки, установив точки останова в нужных местах. При достижении точки останова выполнение программы приостанавливается, и пользователь может проанализировать текущее состояние программы и выполнить необходимые действия для ее дальнейшего исполнения.

8. Назовите и дайте основную характеристику основным командам 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

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

Исправил `Makefile`, после запустил отладку `gdb ./calcul` проверил на работоспособность программу, установил точку останова и удалил её, далее попробовал проанализировать код с помощью утилиты `splint`.

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

Компилятор не выдал никаких ошибок при первом компилировании созданных файлов через `gcc`. Сам исполняемый файл работал корректно.

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

Основные средства, которые могут помочь повысить понимание исходного кода программы, включают:

Комментарии: они могут предоставить дополнительную информацию о том, что делает код и как он это делает.

Документация: она может содержать инструкции по использованию программы, описание функций, классов и методов, которые содержатся в программе, и другую полезную информацию.

Отладчики: они позволяют выполнять код по шагам, отслеживать значения переменных и тестировать различные сценарии выполнения программы.

Редакторы кода: они обычно имеют функции подсветки синтаксиса, автодополнения и форматирования кода, которые упрощают чтение и понимание исходного кода.

Программы анализа кода: такие программы могут автоматически проверять исходный код на наличие ошибок, обнаруживать недостатки и предлагать рекомендации по улучшению кода.

Книги и руководства: они могут содержать общие принципы программирования, описывать конкретные алгоритмы и структуры данных, а также предоставлять примеры кода для изучения и анализа.

12. Каковы основные задачи, решаемые программой splint?

Splint (Secure Programming Lint) - это инструмент статического анализа кода на языке C, который помогает обнаруживать ошибки в коде, связанные с безопасностью и потенциальными уязвимостями.

Основные задачи, решаемые программой Splint, включают:

Поиск ошибок в коде: Splint может обнаружить ошибки в коде, такие как инициализированные переменные, неправильное использование указателей, переполнение буфера и другие.

Анализ потока данных: Splint может проанализировать поток данных в программе и найти возможные ошибки в обработке данных, такие как некорректное приведение типов, ошибки при работе с файлами и сетью и другие.

Проверка безопасности: Splint может проверять код на наличие уязвимостей безопасности, связанных с ошибками в обработке пользовательского ввода, уязвимостями буфера и другими проблемами безопасности.

Проверка соответствия стандартам: Splint может проверять код на соответствие стандартам языка C, таким как ANSI C и POSIX.

Улучшение качества кода: Splint помогает улучшить качество кода, предоставляя рекомендации по улучшению стиля программирования и снижению уровня сложности кода.