

# **Лабораторная работа 13**

**Средства, применяемые при разработке программного обеспечения в  
ОС типа UNIX/Linux**

Генералов Даниил, НПИ-01-21, 1032212280

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>7</b>
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>8</b>
<b>5</b>	<b>Контрольные вопросы</b>	<b>13</b>
<b>6</b>	<b>Выводы</b>	<b>17</b>

## Список иллюстраций

4.1	Ввод существующего кода . . . . .	8
4.2	Создание Makefile . . . . .	9
4.3	Изменение Makefile . . . . .	10
4.4	Запуск программы в GDB . . . . .	10
4.5	Отладка программы в GDB . . . . .	11
4.6	Проверка main.c с помощью splint . . . . .	11
4.7	Проверка calculate.c с помощью splint . . . . .	12

## Список таблиц

# 1 Цель работы

Целью данной работы является:

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

## 2 Задание

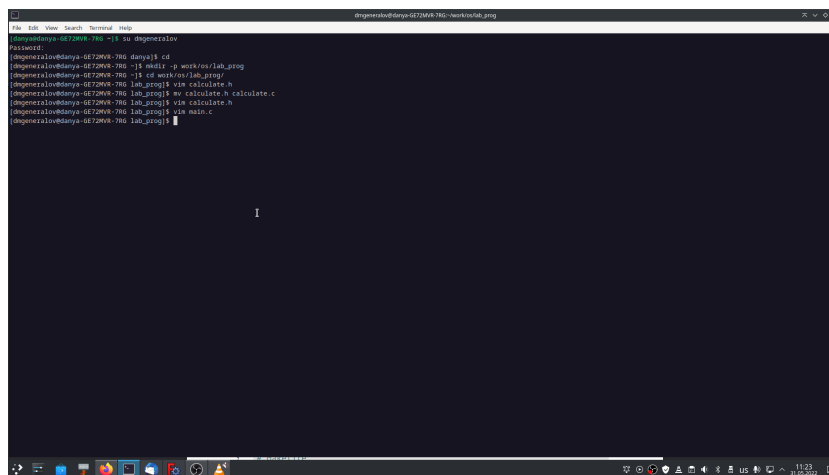
Требуется взять код на C, скомпилировать его, затем отладить его с помощью GDB и проанализировать исходный код с помощью `splint`.

## 3 Теоретическое введение

При разработке программ на Linux можно использовать широкий набор инструментов, которые помогают в этом. В этой лабораторной работе мы рассматриваем некоторые из них – помимо стандартных утилит вроде `make` и `gcc`, мы используем отладчик GDB и программу статического анализа исходного кода `splint`.

## 4 Выполнение лабораторной работы

Сначала нам требуется ввести существующий код, как показано на рисунке 4.1.

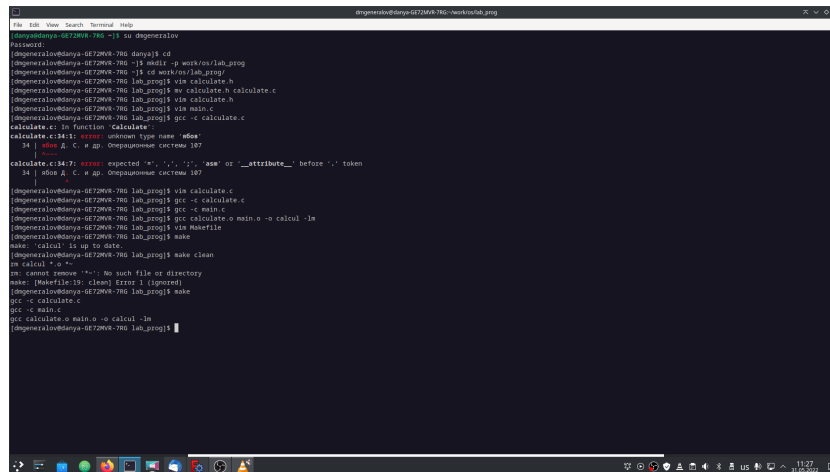


```
dogenerali@denya:~$ cd /tmp
dogenerali@denya:~/tmp$ mkdir lab_prog
dogenerali@denya:~/tmp$ cd lab_prog
dogenerali@denya:~/tmp/lab_prog$ nano calculate.h
dogenerali@denya:~/tmp/lab_prog$ mv calculate.h calculate.c
dogenerali@denya:~/tmp/lab_prog$ nano calculate.c
dogenerali@denya:~/tmp/lab_prog$ nano main.c
dogenerali@denya:~/tmp/lab_prog$
```

Рис. 4.1: Ввод существующего кода

После этого, вручную скомпилировав и исправив ошибки, мы создаем и опробуем Makefile, чтобы компилировать программу автоматически. Это показано на рисунке 4.2.





```
degenerativdanya@72009-780: ~/workspace$ cd
degenerativdanya@72009-780: ~/workspace$ mkdir -p workspace/lab_prog
degenerativdanya@72009-780: ~/workspace/lab_prog$ cd
degenerativdanya@72009-780: ~/workspace/lab_prog$ vi calculate.h
degenerativdanya@72009-780: ~/workspace/lab_prog$ vi calculate.c
degenerativdanya@72009-780: ~/workspace/lab_prog$ vi calculate.h
degenerativdanya@72009-780: ~/workspace/lab_prog$ vi calculate.c
degenerativdanya@72009-780: ~/workspace/lab_prog$ vi calculate.c
calculate.c:14:7: error: expected '=', ',', ';', 'asm' or '__attribute__' before '.' token
  14 | #asm A C. # sp. Onepagummas cctnew 187
     |
calculate.c:14:7: error: expected '=', ',', ';', 'asm' or '__attribute__' before '.' token
  14 | #asm A C. # sp. Onepagummas cctnew 187
     |
degenerativdanya@72009-780: ~/workspace/lab_prog$ vi calculate.c
degenerativdanya@72009-780: ~/workspace/lab_prog$ gcc -c calculate.c
degenerativdanya@72009-780: ~/workspace/lab_prog$ gcc -c main.c
degenerativdanya@72009-780: ~/workspace/lab_prog$ gcc calculate.o main.o -o calcul -lm
degenerativdanya@72009-780: ~/workspace/lab_prog$ vi Makefile
make: 'calcul' is up to date.
degenerativdanya@72009-780: ~/workspace/lab_prog$ make clean
rm calcul.o *.o *.a
rm: cannot remove '*.o': No such file or directory
make: [Makefile:18: clean] Error 1 (ignored)
degenerativdanya@72009-780: ~/workspace/lab_prog$ make
gcc -c calculate.c
gcc -c main.c
gcc calculate.o main.o -o calcul -lm
degenerativdanya@72009-780: ~/workspace/lab_prog$
```

Рис. 4.2: Создание Makefile

После этого нужно сделать некоторые изменения в этом Makefile, и финальное состояние этого файла показано на рис. 4.3. В начале там задаются параметры вроде названия компилятора и флагов конфигурации к нему, вроде используемых библиотек и параметров добавления диагностической информации. После этого идут указания *целей*: файлов, которые можно собрать, тех файлов, которые должны быть собраны для сборки этого файла, и команд для сборки их. Например, чтобы собрать файл `calcul`, уже должны быть собраны файлы `calculate.o` и `main.o`, и нужно выполнить команду `gcc calculate.o main.o -o calcul -o calcul -lm -g`. Наконец, указывается цель `clean`, которая удаляет все собранные файлы.

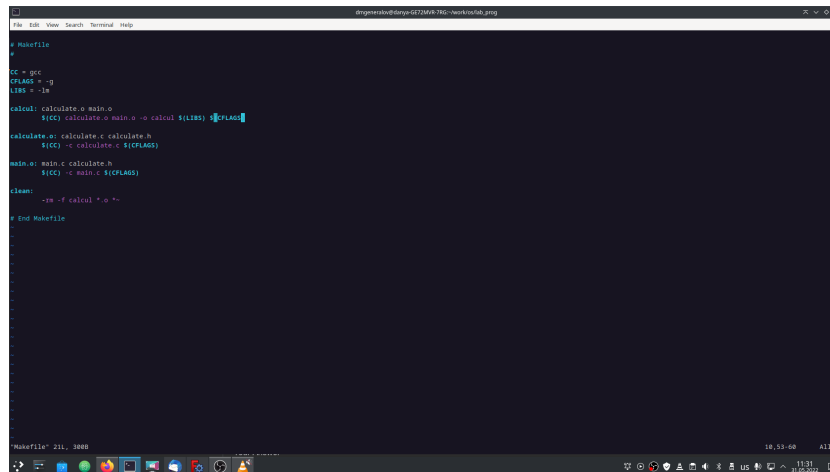


Рис. 4.3: Изменение Makefile

Теперь, когда программа собрана, можно запустить её в GDB. Это показано на рисунке 4.4. Команда `run` запускает программу для отладки, а команда `list` показывает исходный код программы вокруг места останова.

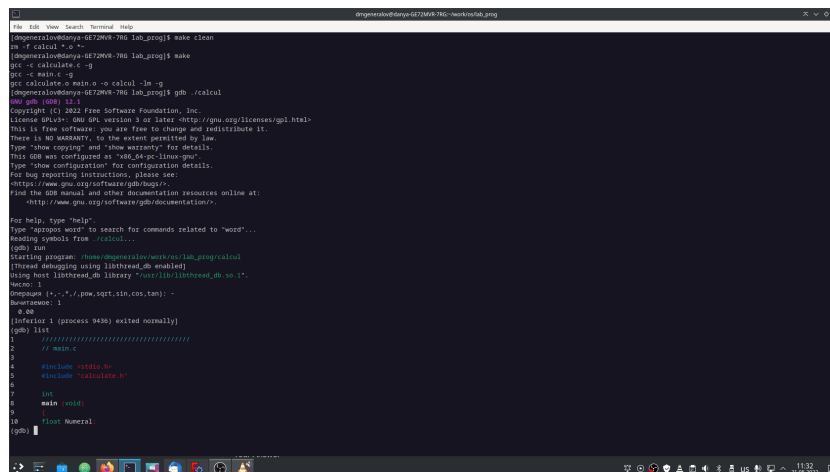
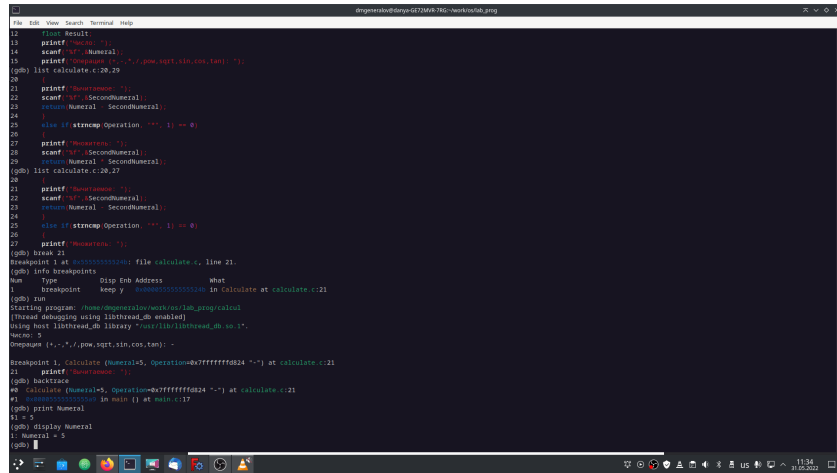


Рис. 4.4: Запуск программы в GDB

Команда `list` принимает аргумент – это может быть диапазон строк, которые следует вывести, или название файла и диапазон строк в нем. После команды `list` можно набрать команду `break`, которая создаст точку останова на указанной строке в последнем отображенном файле. Когда точка останова создана и программа запущена, она будет остановлена перед выполнением этой строки. После этого можно использовать команды `backtrace`, чтобы посмотреть на стек

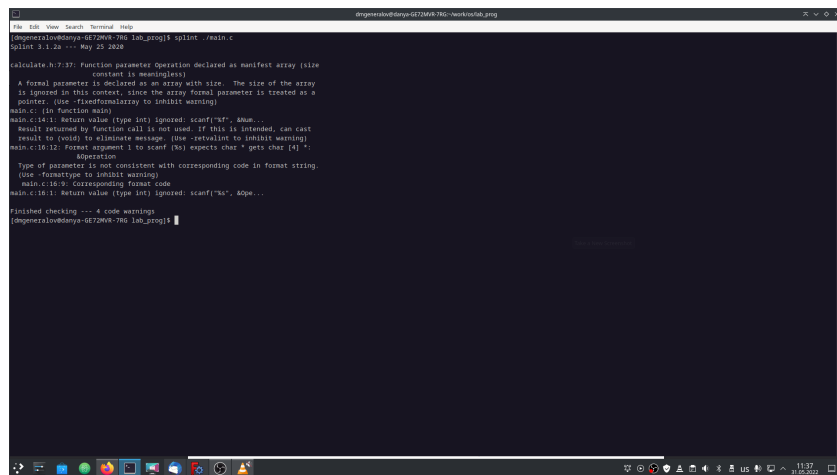
вызовов. Здесь же можно отобразить информацию о переменных с помощью команд `print` и `display`.



```
12 printf("Result: ");
13 printf("%d\n", i);
14 scanf("%d", &Numeral);
15 printf("Input: %d\n", *pow_sqrt_sin_cos_tan);
(gdb) list calculate.c:20:29
20
21 printf("NewNumeral: ");
22 scanf("%d", &SecondNumeral);
23 return Numeral * SecondNumeral;
24 }
25 else if (strcmp(Operation, "*") != 0)
26 {
27 printf("NewNumeral: ");
28 scanf("%d", &SecondNumeral);
29 return Numeral / SecondNumeral;
(gdb) list calculate.c:20:27
20
21 printf("NewNumeral: ");
22 scanf("%d", &SecondNumeral);
23 return Numeral * SecondNumeral;
24 }
25 else if (strcmp(Operation, "/") != 0)
26 {
27 printf("NewNumeral: ");
28 scanf("%d", &SecondNumeral);
29 return Numeral / SecondNumeral;
(gdb) break 21
Breakpoint 1 at 0x401000: File calculate.c, line 21.
(gdb) info breakpoints
Num Type Disp Enb Address What
-----
1 breakpoint keep y 0x0000000000000021 In calculate at calculate.c:21
(gdb) run
Starting program: /home/depgeralov@depgeralov:~/lib_prog/calculat
(thread debugging using libthread_db enabled)
Using host libthread_db library "/usr/lib/libthread_db.so.1".
NewNumeral: 5
Breakpoint 1, calculate (Numeral=5, Operation=0x7fffffffdb24 "") at calculate.c:21
21 printf("NewNumeral: ");
(gdb) backtrace
#0 calculate (Numeral=5, Operation=0x7fffffffdb24 "") at calculate.c:21
#1 main (argc=2, argv=0x7fffffffdb24) at main.c:17
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
Numeral = 5
(gdb)
```

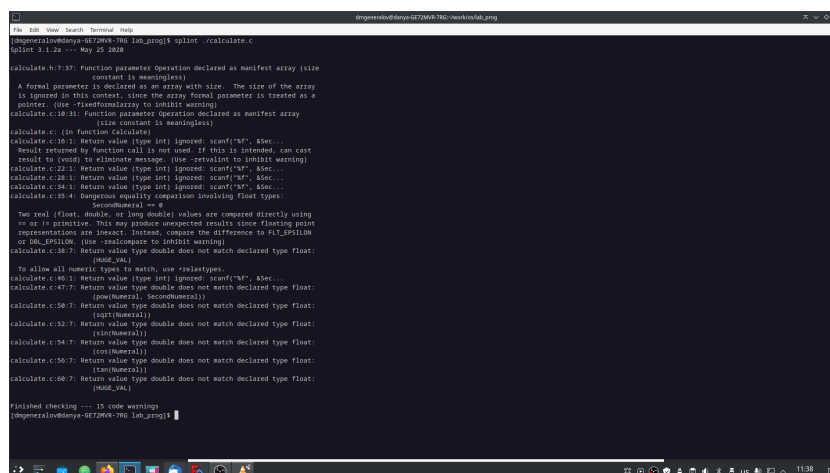
Рис. 4.5: Отладка программы в GDB

Также можно использовать программу `splint`, чтобы проверить код на возможные ошибки. Результаты вызова этой программы на двух файлах показаны на рисунках 4.6 и 4.7.



```
calculate.h:7:27: 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 -fvarformalarray to inhibit warning)
main.c:10: function main
main.c:14:1: Return value (type int) ignored: scanf("%d", &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:16:12: Format argument 1 to scanf (%s) expects char * gets char [4] ".
Deprecation
Type of parameter is not consistent with corresponding code in format string.
(Use -formattype to inhibit warning)
main.c:16:9: Corresponding format code
main.c:16:1: Return value (type int) ignored: scanf("%s", &ope...
Finished checking --- 4 code warnings
depgeralov@depgeralov:~/lib_prog$
```

Рис. 4.6: Проверка main.c с помощью splint



```
File Edit View Search Terminal Help
@ngeneralov@bony: 0172049-760 lab_prog$ splint -Icalculate.c
splint 3.1.2a --- May 23 2020

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 -fignoreformalarray 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:10:1: 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 -castvoid to inhibit warning)
calculate.c:22:1: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:23:1: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:34:1: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:35:4: 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:36:7: Return value type double does not match declared type float:
    (Mod_Val)
To allow all numeric types to match, use %platypes.
calculate.c:40:1: Return value (type int) ignored: scanf("%f", &sec...
calculate.c:47:7: Return value type double does not match declared type float:
    (pow(Numeral, SecondNumeral))
calculate.c:50:7: Return value type double does not match declared type float:
    (sqrt(Numeral))
calculate.c:52:7: Return value type double does not match declared type float:
    (sin(Numeral))
calculate.c:54:7: Return value type double does not match declared type float:
    (cos(Numeral))
calculate.c:56:7: Return value type double does not match declared type float:
    (tan(Numeral))
calculate.c:60:7: Return value type double does not match declared type float:
    (Mod_Val)
Finished checking --- 15 code warnings
@ngeneralov@bony: 0172049-760 lab_prog$
```

Рис. 4.7: Проверка calculate.c с помощью splint

Среди обнаруженных проблем можно отметить следующие:

- Результат вызова `scanf` игнорируется, что может скрывать ошибки при чтении с клавиатуры.
- Функция `calculate.c:Calculate` принимает параметр `char[4]`, но на самом деле он используется как `char*`, из-за чего параметр длины не имеет значения.
- Осуществляется сравнение на равенство с помощью оператора `==` над значениями типа `float`, что может приводить к ошибкам из-за погрешности вычислений дробных чисел.
- Некоторые из используемых функций возвращают значение типа `double`, а не `float`, и используется скрытое преобразование между ними.

## 5 Контрольные вопросы

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

Это можно сделать, как и для любой другой программы, с помощью `man` или через флаг `--help`.

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

- Написание кода в текстовом редакторе – используется `vim` или `emacs`.
- Создание структуры компиляции программы – используется `make` или `cmake`.
- Статическая проверка кода на ошибки – используется `lint` или `splint`.
- Отладка программы – используется `gdb` или `lldb`.
- Сохранение изменений – используется `git` или `hg`.

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

В C/C++ можно добавить к численному литералу дополнительные символы, чтобы обозначить тип данных этого литерала. Например, по умолчанию запись `1.0` имеет тип `double`, а запись `1.0f` имеет тип `float`. Аналогично, по умолчанию запись `123` имеет тип `int`, запись `123l` имеет тип `long int`, а запись `123u` имеет тип `unsigned int`.

4. Каково основное назначение компилятора языка C в UNIX?

Как следует из названия, он нужен для компиляции кода на языке C. Это нужно для того, чтобы программы были портативными между системами с разными архитектурами и процессорами.

5. Для чего предназначена утилита make?

Она нужна для того, чтобы иметь одну команду для выполнения всех необходимых действий по сборке программы. Используя собственный анализатор, make определяет, какие действия по сборке программы уже выполнены, а какие ещё предстоит выполнить, и делает только необходимые действия параллельно.

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

```
all: program # указание основной цели

program: a.o b.o c.o # указание зависимостей
        gcc -o program a.o b.o c.o # указание команды компиляции

a.o: # указание отдельных компонентов программы
        gcc -c a.c # указание команды компиляции

b.o:
        gcc -c b.c

c.o:
        gcc -c c.c

clean: # указание команды для очистки всех собранных файлов
        rm -f *.o program
```

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

Все программы отладки позволяют просматривать исходный код программы, останавливать исполнение на определенных местах и рассматривать состояние остановленной программы, если эта программа была скомпилирована с отладочной информацией (например, флагом `-g` в GCC).

8. Назовите и дайте основную характеристику основным командам отладчика `gdb`.

- `list` – показывает исходный код программы
- `break` – позволяет поставить точку останова на определенном месте программы
- `run` – запускает программу
- `step` – позволяет произвести один шаг в программе
- `next` – позволяет произвести один шаг в программе, не заходя внутрь стека
- `finish` – делает шаги до конца текущего фрейма стека
- `backtrace` – показывает стек вызовов
- `display` – позволяет просматривать значения переменных в программе

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

```
(gdb) run
(gdb) list
(gdb) list 12,15
(gdb) list calculate.c:20,29
(gdb) list calculate.c:20,27
(gdb) break 21
(gdb) info breakpoints
(gdb) run
5
-
```

```
(gdb) backtrace
(gdb) print Numeral
(gdb) display Numeral
(gdb) info breakpoints
(gdb) delete 1
(gdb) quit
```

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

Он вывел сообщения, указывающие на строки с ошибками и сообщающие, что именно неожиданно об этом синтаксисе в этом месте кода.

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

Среди таких средств можно считать функции редактора текста – подсветку синтаксиса, автоматическую табуляцию и свертку блоков кода – сообщения компилятора, если такие присутствуют, а также утилиты вроде `lint/splint`, которые анализируют код на частые ошибки.

12. Каковы основные задачи, решаемые программой `splint`?

Она обращает внимание программиста на возможные ошибки в коде, которые могут привести к неправильной работе программы в неожиданных ситуациях.



## 6 Выводы

В этой лабораторной работе мы рассмотрели (во второй раз) средства разработки программ в Linux. Хотя для большинства этих средств есть графические интерфейсы, знать про их использование с командной строки может быть полезным.