02.6.2022

Лабораторная работа 13

Манатов Рамазан Русланович

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.

Ход работы:

1) В домашнем каталоге создаю подкаталог ~/work/os/lab_prog с помощью команды «mkdir -p ~/work/os/lab_prog» (Рисунок 1).

```
[rrManatov@fedora ~]$ cd work
[rrManatov@fedora work]$ cd os
[rrManatov@fedora os]$ mkdir lab_prog
[rrManatov@fedora os]$ cd lab_prog
```

Рисунок 1

2) Создал в каталоге файлы: calculate.h, calculate.c, main.c, используя команды «cd ~/work/os/lab_prog» и «touch calculate.h calculate.c main.c» (Рисунок 2)

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

Рисунок 2

Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять sin, cos, tan. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится. Открыв редактор Emacs, приступил к редактированию созданных файлов. Реализация функций калькулятора в файле calculate.c (Рисунки 3, 4)

```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);
```

Рисунок 3

```
      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);
    }
}
```

Рисунок 4

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора (Рисунок 5)

```
//calculate.h

#ifndef CALCULATE_H_
#define CALCULATE_H_

float Calculate(float Numeral, char Operation[4]);

#endif /*CALCULATE_H_*/
```

Рисунок 5

Основной файл main.c, реализующий интерфейс пользователя к калькулятору (Рисунок 6)

```c
//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;
}
```

Рисунок 6

3) Выполнил компиляцию программы посредством gcc (версия компилятора: 8.3.0-19), используя команды «gcc -c calculate.c», «gcc -c main.c» и «gcc calculate.o main.o -o calcul -lm» (Рисунок 7)

```
[rrManatov@fedora lab_prog]$ gcc -c calculate.c
[rrManatov@fedora lab_prog]$ gcc -c main.c
[rrManatov@fedora lab_prog]$ gcc calculate.o main.o -o ca
lcul -lm
```

Рисунок 7

4) В ходе компиляции программы никаких ошибок выявлено не было.

5) Создал Makefile с необходимым содержанием (Рисунок 8)

```
#
# 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 8

Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цель main.o), а также их объединения в один исполняемый файл calcul (цель calcul). Цель clean нужна для автоматического удаления файлов. Переменная CC отвечает за утилиту для компиляции. Переменная CFLAGS отвечает за опции в данной утилите. Переменная LIBS отвечает за опции для объединения объектных файлов в один исполняемый файл.

6) Далее исправил Makefile (Рисунок 9).

```
#
# 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#
```

Рисунок 9

В переменную CFLAGS добавил опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Сделала так, что утилита компиляции выбирается с помощью переменной CC. После этого я удалил исполняемые и объектные файлы из каталога с помощью команды «make clear» (Рисунок 10). Выполнил компиляцию файлов, используя команды «make calculate.o», «make main.o», «male calcul» (Рисунок 11)

```
[rrManatov@fedora lab_prog]$ make clean
rm calcul *.o *~
```

Рисунок 10

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

Рисунок 11

Далее с помощью gdb выполнил отладку программы calcul. Запустил отладчик GDB, загрузив в него программу для отладки, используя команду: «gdb ./calcul» (Рисунок 12)

```
[rrManatov@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 10.2-9.fc35
Copyright (C) 2021 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.or
g/licenses/gpl.html>
This is free software: you are free to change and redistr
ibute 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 onl
ine at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "wo
rd"...
Reading symbols from ./calcul...
```

Рисунок 12

Для запуска программы внутри отладчика ввел команду «run» (Рисунок 13)

```
(gdb) run
Starting program: /home/rrManatov/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): +
Второе слагаемое 1
6.00
%6,2f/n[Inferior 1 (process 4256) exited normally]
(gdb)
```

Рисунок 13

Для постраничного (по 10 строк) просмотра исходного кода использовал команду «list» (Рисунок 14)

```
(gdb) list
16          Result = Calculate(Numeral, Operation);
17          printf("%6,2f/n",Result);
18          return 0;
19      }
```

Рисунок 14

Для просмотра строк с 12 по 15 основного файла использовал команду «list 12,15» (Рисунок 15)

```
(gdb) list 12,15
12          printf("Число: ");
13          scanf("%f",&Numeral);
14          printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan)
: ");
15          scanf("%s",&Operation);
```

Рисунок 15

Для просмотра определённых строк не основного файла использовал команду «list calculate.c:20,29» (Рисунок 16)

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

Рисунок 16

Установил точку останова в файле calculate.c на строке номер 21, используя команды «list calculate.c:20,27» и «break 21» (Рисунок 17)

```
(gdb) list calculate.c:20,29
20              printf("Вычитаемое: ");
21              scanf("%f",&SecondNumeral);
22              return(Numeral - SecondNumeral);
23            }
24          else if(strncmp(Operation, "*", 1) == 0)
25            {
26              printf("Множитель: ");
27              scanf("%f",&SecondNumeral);
28              return(Numeral * SecondNumeral);
29            }
(gdb) break 21
Breakpoint 1 at 0x40121e: file calculate.c, line 21.
```

Рисунок 17

Вывел информацию об имеющихся в проекте точках останова с помощью команды «info breakpoints» (Рисунок 18)

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040121e in Cal
culate at calculate.c:21
```

Рисунок 18

Запустил программу внутри отладчика и убедился, что программа остановилась в момент прохождения точки останова. Использовала команды «run», «5», «−» и «backtrace» (Рисунок 19)

```
(gdb) run
Starting program: /home/rrManatov/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): -

Breakpoint 1, Calculate (Numeral=5,
    Operation=0x7fffffffdf64 "-") at calculate.c:21
21              scanf("%f",&SecondNumeral);
(gdb) backtrace
#0  Calculate (Numeral=5, Operation=0x7fffffffdf64 "-")
    at calculate.c:21
#1  0x00000000004014eb in main () at main.c:16
```

Рисунок 19

Посмотрел, чему равно на этом этапе значение переменной Numeral, введя команду «print Numeral» (Рисунок 20)

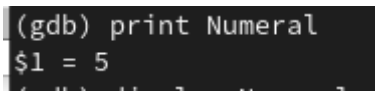

Рисунок 20

Сравнил с результатом вывода на экран после использования команды «display Numeral». Значения совпадают (Рисунок 21)

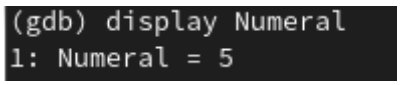

Рисунок 21

Убрал точки останова с помощью команд «info breakpoints» и «delete 1» (Рисунок 22)

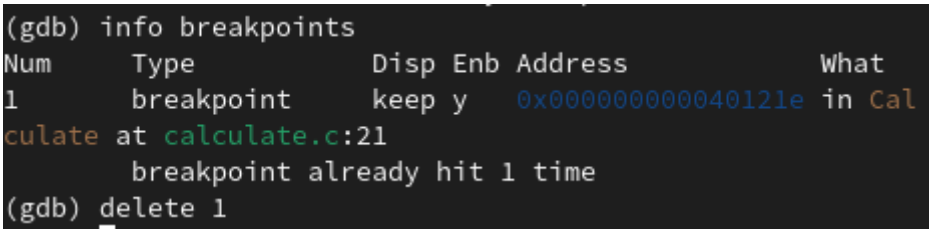

Рисунок 22

7) С помощью утилиты splint проанализировал коды файлов calculate.c и main.c. Предварительно я установил данную утилиту когда мне предложили это сделать(Рисунок 24)

Далее воспользовался командами «splint calculate.c» и «splint main.c» (Рисунки 24, 25). С помощью утилиты splint выяснилось, что в файлах calculate.c и main.c присутствует функция чтения scanf, возвращающая целое число (тип int), но эти числа не используются и нигде не сохранятся. Утилита вывела предупреждение о том, что в файле calculate.c происходит сравнение вещественного числа с нулем. Также возвращаемые значения (тип double) в функциях pow, sqrt, sin, cos и tan записываются в переменную типа float, что свидетельствует о потери данных. (рисунок 23 ,25)

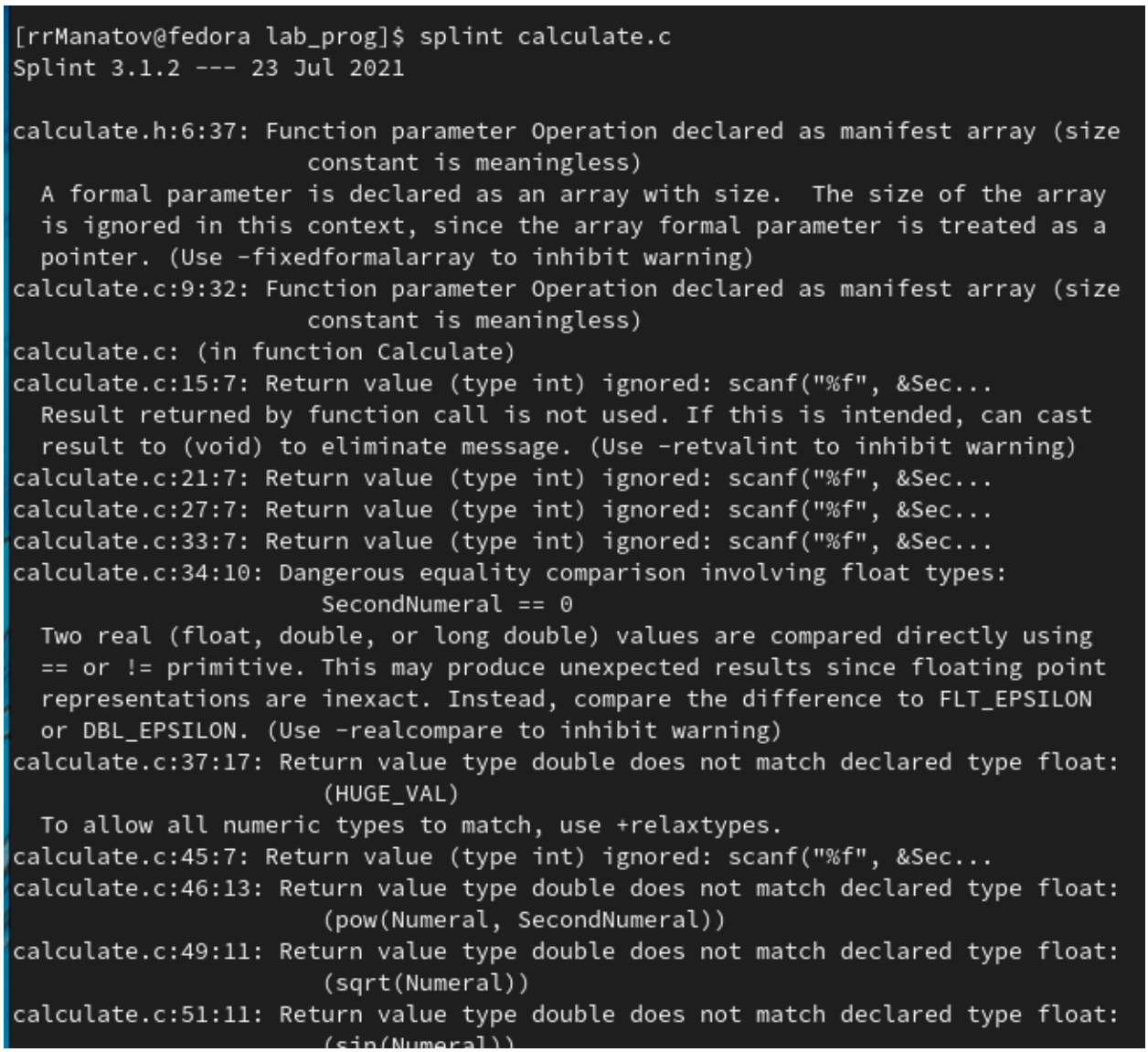
[rrManatov@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2021
calculate.h:6:37: Function parameter Operation declared as manifest array (size
constant is meaningless)
A formal parameter is declared as an array with size. The size of the array
is ignored in this context, since the array formal parameter is treated as a
pointer. (Use -fixedformalarray to inhibit warning)
calculate.c:9:32: Function parameter Operation declared as manifest array (size
constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:15:7: 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:21:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:27:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:33:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:10: 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:37:17: Return value type double does not match declared type float:
(HUGE_VAL)
To allow all numeric types to match, use +relaxtypes.
calculate.c:45:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:46:13: Return value type double does not match declared type float:
(pow(Numeral, SecondNumeral))
calculate.c:49:11: Return value type double does not match declared type float:
(sqrt(Numeral))
calculate.c:51:11: Return value type double does not match declared type float:
(sin(Numeral))

Рисунок 23

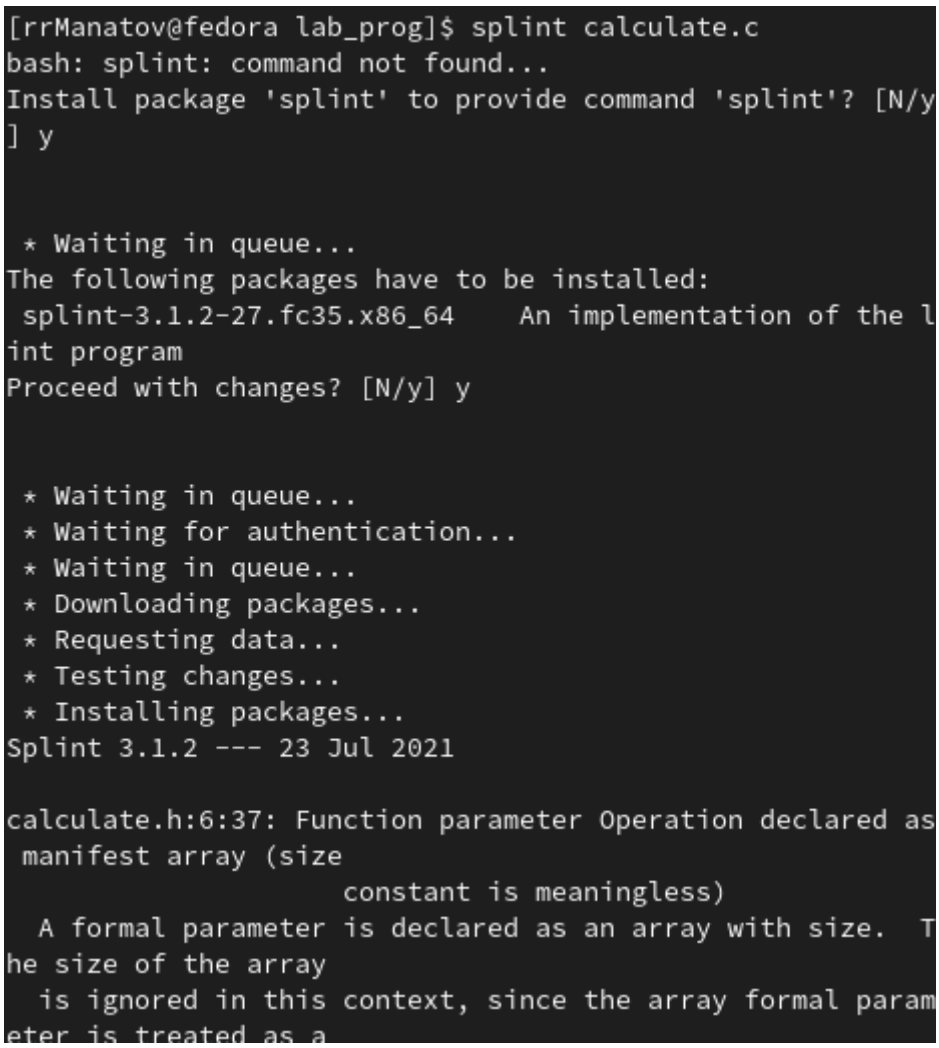

Рисунок 24 установка splint

Вывод:В ходе выполнения данной лабораторной работы я приобрел простейшие навыки разработки,анализа,тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования С калькулятора с простейшими функциями

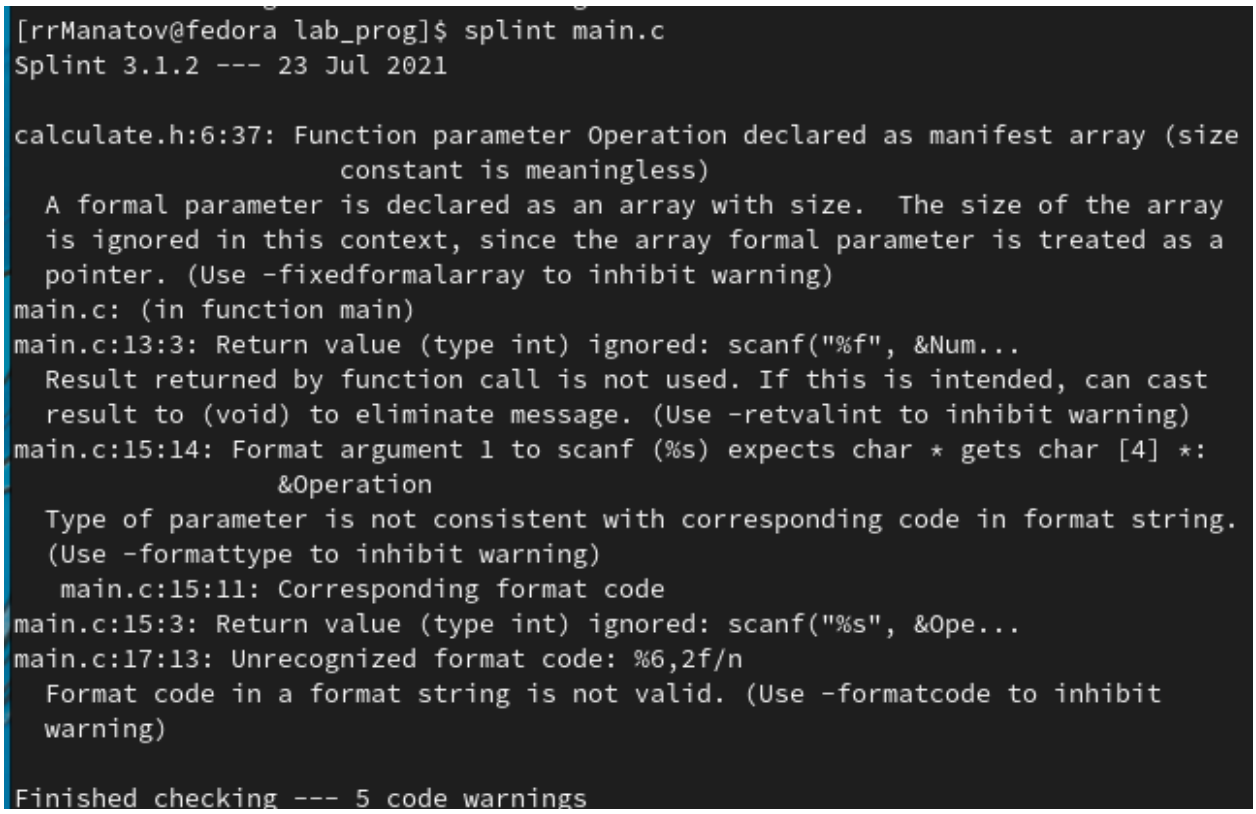

Рисунок 25

Далее воспользовался командами «splint calculate.c» и «splint main.c» (Рисунки 24, 25). С помощью утилиты splint выяснилось, что в файлах calculate.c и main.c присутствует функция чтения scanf, возвращающая целое число (тип int), но эти числа не используются и нигде не сохранятся. Утилита вывела предупреждение о том, что в файле calculate.c происходит сравнение вещественного числа с нулем. Также возвращаемые значения (тип double) в функциях pow, sqrt, sin, cos и tan записываются в переменную типа float, что св

идетельствует о потери данных.

3. Контрольные вопросы:

1) Чтобы получить информацию о возможностях программ gcc, make, gdb и др. нужно воспользоваться командой man или опцией -help (-h) для каждой команды.

2) Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

• планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;

• проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;

• непосредственная разработка приложения: о кодирование − по сути создание исходного текста программы (возможно в нескольких вариантах); – анализ разработанного кода; о сборка, компиляция и разработка исполняемого модуля; о тестирование и отладка, сохранение произведённых изменений;

• документирование. Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geany и др.

После завершения написания исходного кода программы (возможно состоящей из нескольких файлов), необходимо её скомпилировать и получить исполняемый модуль.

3) Для имени входного файла суффикс определяет какая компиляция требуется. Суффиксы указывают на тип объекта. Файлы с расширением (суффиксом) .c воспринимаются gcc как программы на языке C, файлы с расширением .cc или .C − как файлы на языке C++, а файлы с расширением .o считаются объектными. Например, в команде «gcc -c main.c»: gcc по расширению (суффиксу) .c распознает тип файла для компиляции и формирует объектный модуль − файл с расширением .o. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -o и в качестве параметра задать имя создаваемого файла: «gcc -o hello main.c».

4) Основное назначение компилятора языка Си в UNIX заключается в компиляции всей программы и получении исполняемого файла/модуля.

5) Для сборки разрабатываемого приложения и собственно компиляции полезно воспользоваться утилитой make. Она позволяет автоматизировать процесс преобразования файлов программы из одной формы в другую, отслеживает взаимосвязи между файлами.

6) Для работы с утилитой make необходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefile или Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса. В самом простом случае Makefile имеет следующий синтаксис: ... : ... ... Сначала задаётся список целей, разделённых пробелами, за которым идёт двоеточие и список зависимостей. Затем в следующих строках указываются команды. Строки с командами обязательно должны начинаться с табуляции. В качестве цели в Makefile может выступать имя файла или название какого-то действия. Зависимость задаёт исходные параметры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды − собственно действия, которые необходимо выполнить для достижения цели. Общий синтаксис Makefile имеет вид: target1 [target2...]:[:] [dependment1...] [(tab)commands] [#commentary] [(tab)commands] [#commentary] Здесь знак # определяет начало комментария (содержимое от знака # и до конца строки не будет обрабатываться. Одинарное двоеточие указывает на то, что последовательность команд должна содержаться в одной строке. Для переноса можно в длинной строке команд можно использовать обратный слэш (\). Двойное двоеточие указывает на то, что последовательность команд может содержаться в нескольких последовательных строках. Пример более сложного синтаксиса Makefile: # # Makefile for abcd.c # CC = gcc CFLAGS = # Compile abcd.c normaly abcd: abcd.c $(CC) -o abcd $(CFLAGS) abcd.c clean: -rm abcd *.o *~ # End Makefile for abcd.c В этом примере в начале файла заданы три переменные: CC и CFLAGS. Затем указаны цели, их зависимости и соответствующие команды. В командах происходит обращение к значениям переменных. Цель с именем clean производит очистку каталога от файлов, полученных в результате компиляции. Для её описания использованы регулярные выражения.

7) Во время работы над кодом программы программист неизбежно сталкивается с появлением ошибок в ней. Использование отладчика для поиска и устранения ошибок в программе существенно облегчает жизнь программиста. В комплект программ GNU для ОС типа UNIX входит отладчик GDB (GNU Debugger). Для использования GDB необходимо скомпилировать анализируемый код программы таким образом, чтобы отладочная информация содержалась в результирующем бинарном файле. Для этого следует воспользоваться опцией -g компилятора gcc: gcc -c file.c -g После этого для начала работы с gdb необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: gdb file.o

8) Основные команды отладчика gdb:

• backtrace − вывод на экран пути к текущей точке останова (по сути вывод − названий всех функций)

• break − установить точку останова (в качестве параметра может быть указан номер строки или название функции)

• clear − удалить все точки останова в функции • continue − продолжить выполнение программы • delete − удалить точку останова

• display − добавить выражение в список выражений, значения которых отображаются при достижении точки останова программы

• finish − выполнить программу до момента выхода из функции

• info breakpoints − вывести на экран список используемых точек останова

• info watchpoints − вывести на экран список используемых контрольных выражений

• list − вывести на экран исходный код (в качестве параметра может быть указано название файла и через двоеточие номера начальной и конечной строк)

• next − выполнить программу пошагово, но без выполнения вызываемых в программе функций • print − вывести значение указываемого в качестве параметра выражения

• run − запуск программы на выполнение

• set − установить новое значение переменной

• step − пошаговое выполнение программы

• watch − установить контрольное выражение, при изменении значения которого программа будет остановлена Для выхода из gdb можно воспользоваться командой quit (или её сокращённым вариантом q) или комбинацией клавиш Ctrl-d. Более подробную информацию по работе с gdb можно получить с помощью команд gdb -h и man gdb. 9) Схема отладки программы показана в 6 пункте лабораторной работы. 10) При первом запуске компилятор не выдал никаких ошибок, но в коде программы main.c допущена ошибка, которую компилятор мог пропустить (возможно, из-за версии 8.3.0-19): в строке scanf(“%s”, &Operation); нужно убрать знак &, потому что имя массива символов уже является указателем на первый элемент этого массива. 11) Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся:

• cscope − исследование функций, содержащихся в программе,

• lint − критическая проверка программ, написанных на языке Си.

12) Утилита splint анализирует программный код, проверяет корректность задания аргументов использованных в программе функций и типов возвращаемых значений, обнаруживает синтаксические и семантические ошибки. В отличие от компилятора С анализатор splint генерирует комментарии с описанием разбора кода программы и осуществляет общий контроль, обнаруживая такие ошибки, как одинаковые объекты, определённые в разных файлах, или объекты, чьи значения не используются в работе программы, переменные с некорректно заданными значениями и типами и многое другое.