Отчёт по лабораторной работе №13

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

Дарья Сергеевна Кочина

Содержание

Список иллюстраций

1 Цель работы

Целью данной лабораторной работы является приобретение простейших навыков разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linux на примере создания на языке программирования C калькулятора с простейшими функциями.

2 Задание

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.

3 Теоретическое введение

Этапы разработки приложений

Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

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

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

– непосредственная разработка приложения:

– кодирование — по сути создание исходного текста программы (возможно в нескольких вариантах);

– анализ разработанного кода;

– сборка, компиляция и разработка исполняемого модуля;

– тестирование и отладка, сохранение произведённых изменений;

– документирование.

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.

4 Выполнение лабораторной работы

1. В домашнем каталоге создаю подкаталог /work/os/lab_progc помощью команды «mkdir -p/work/os/lab_prog». (рис. [4.1])

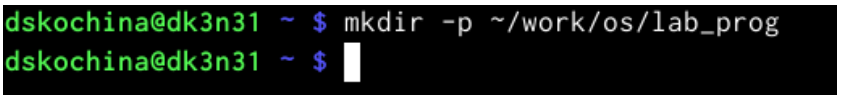

Рис. 4.1: Создание подкаталога

2. Создала в каталоге файлы: calculate.h, calculate.c, main.c, используя команды «cd ~/work/os/lab_prog» и «touch calculate.h calculate.c main.c». (рис. [4.2])

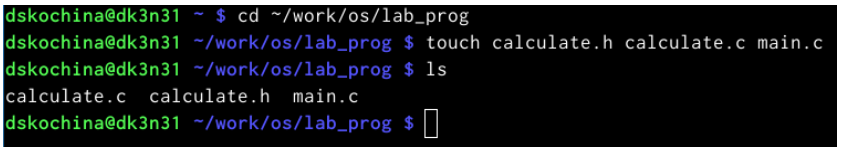

Рис. 4.2: Создание файлов

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, приступила к редактированию созданных файлов. Реализация функций калькулятора в файле calculate.c. (рис. [4.3], [4.4], [4.5])

```
///////////////////////////////////////////////////
// calculate.c

#include<studio.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);
```

Рис. 4.3: Программа в calculate.c

```
    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)
```

Рис. 4.4: Программа в calculate.c

```
  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);
    }
}
```

Рис. 4.5: Программа в calculate.c

Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора. (рис. [4.6])

```
////////////////////////////////////////////////////
// calculate.h

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

Рис. 4.6: Программа в calculate.h

Основной файл main.c, реализующий интерфейс пользователя к калькулятору. (рис. [4.7])

```
///////////////////////////////////////////////////////////////
// main.c

#include <studio.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;
}
```

Рис. 4.7: Программа в main.c

3. Выполнила 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». (рис. [4.8])

```
dskochina@dk3n31 ~/work/os/lab_prog $ gcc -c calculate.c
dskochina@dk3n31 ~/work/os/lab_prog $ gcc -c main.c
dskochina@dk3n31 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm
dskochina@dk3n31 ~/work/os/lab_prog $
```

Рис. 4.8: Компиляция программы

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

5. Создала Makefile с необходимым содержанием. (рис. [4.9])

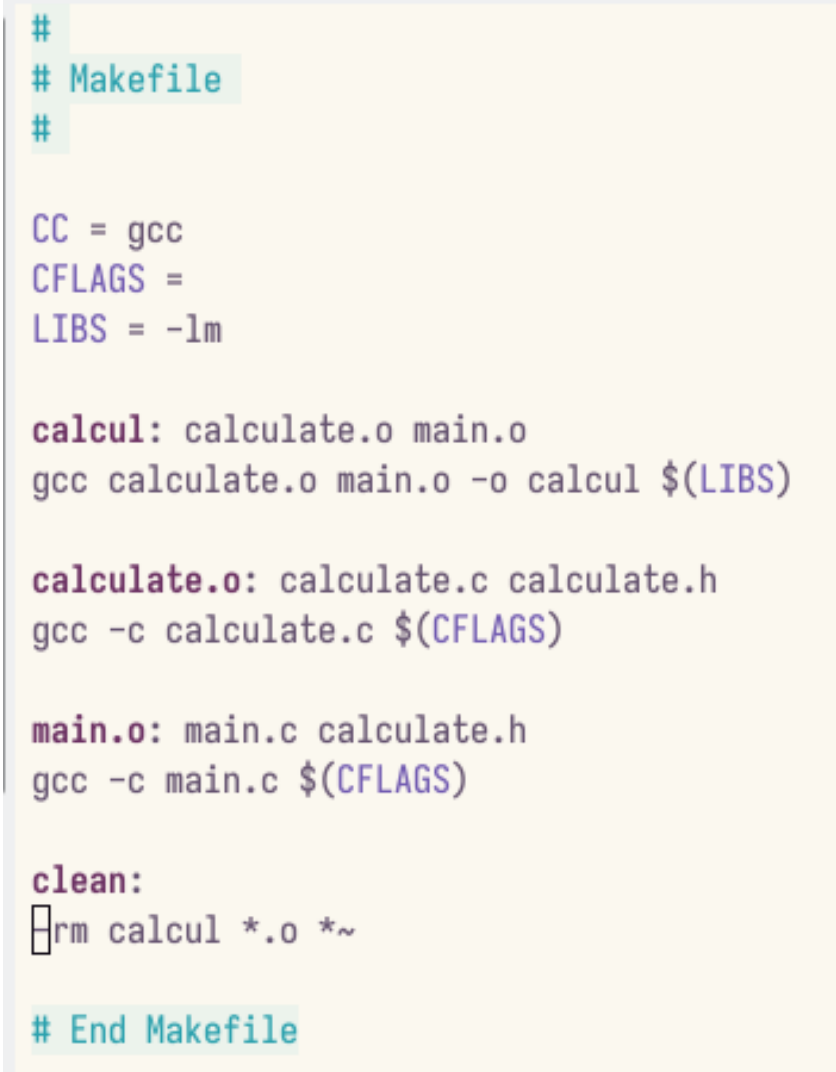

Рис. 4.9: Программа в Makefile

Данный файл необходим для автоматической компиляции файлов calculate.c (цель calculate.o), main.c (цельmain.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 исправила Makefile. (рис. [4.10])

```
#
# 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. 4.10: Изменённая программа в Makefile

В переменную CFLAGS добавила 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 удалила исполняемые и объектные файлы из каталога с помощью команды «make clear». Выполнила компиляцию файлов, используя команды «make calculate.o», «make main.o», «make calcul». (рис. [4.11])

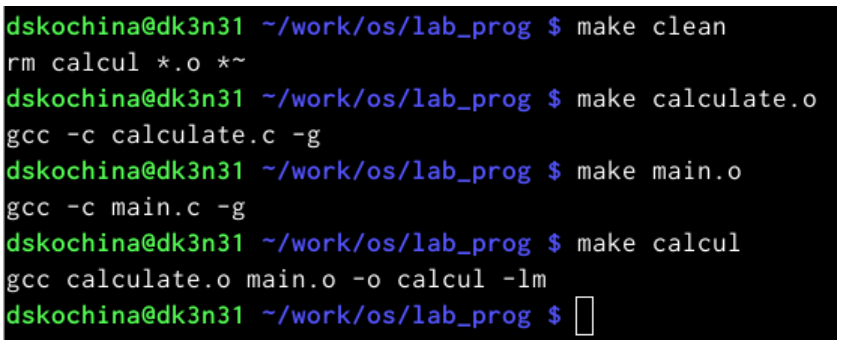

Рис. 4.11: Удаление и компиляция файлов

Далее с помощью gdb выполнила отладку программы calcul. Запустила отладчик GDB, загрузив в него программу для отладки, используя команду: «gdb./calcul». (рис. [4.12])

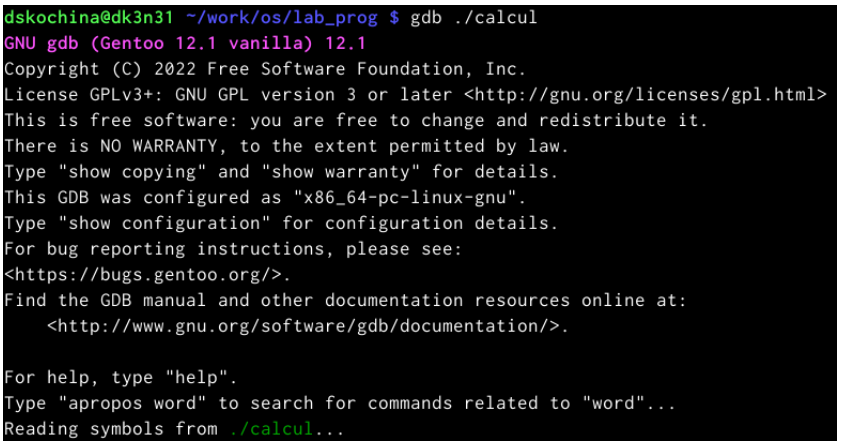

Рис. 4.12: Работа с gdb

Для запуска программы внутри отладчика ввела команду «run». (рис. [4.13])

Рис. 4.13: Работа с gdb - run

Для постраничного просмотра исходного кода использовала команду «list». (рис. [4.14])

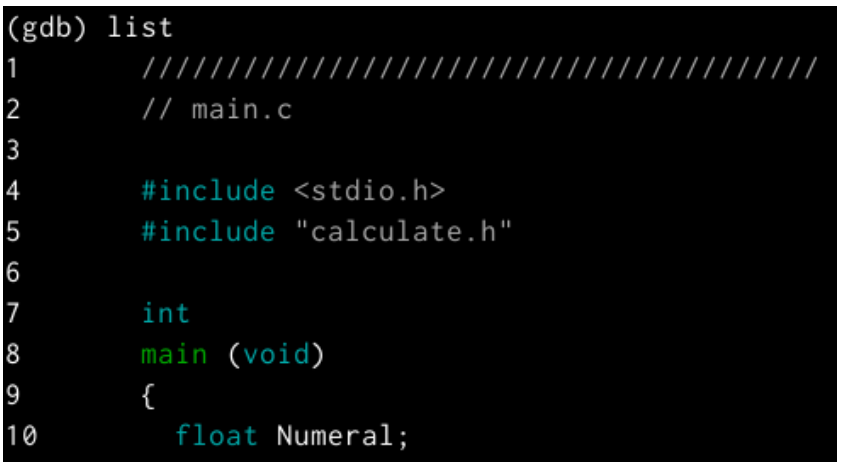

Рис. 4.14: Работа с gdb - list

Для просмотра строк с 12 по 15 основного файла использовала команду «list 12,15». (рис. [4.15])

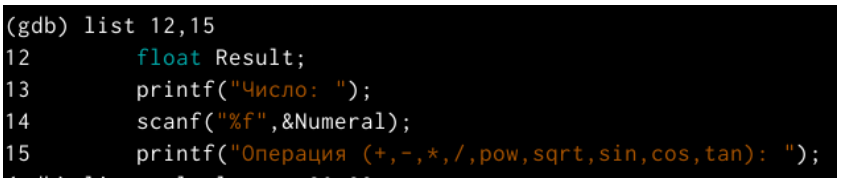

Рис. 4.15: Работа с gdb - list 12,15

Для просмотра определённых строк не основного файла использовала команду «list calculate.c:20,29». (рис. [4.16])

```
(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);
```

Рис. 4.16: Работа с gdb - list calculate.c:20,29

Установила точку останова в файле calculate.c на строке номер 21, используя команды «list calculate.c:20,27» и «break 21». (рис. [4.17])

```
(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) info breakpoints
No breakpoints or watchpoints.
(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 0x555555555247: file calculate.c, line 21.
```

Рис. 4.17: Работа с gdb - list calculate.c:20,27

Вывела информацию об имеющихся в проекте точках останова с помощью

команды «info breakpoints». (рис. [4.18])

Рис. 4.18: Работа с gdb - info breakpoints

Запустила программу внутри отладчика и убедилась, что программа остановилась в момент прохождения точки останова. Использовала команды «run», «5», «*» и «backtrace». (рис. [4.19])

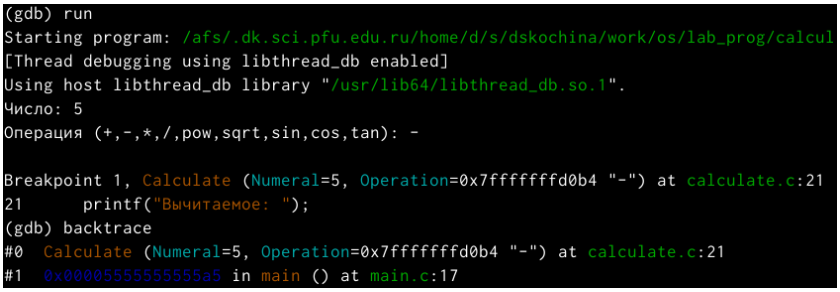

Рис. 4.19: Работа с gdb - run

Посмотрела, чему равно на этом этапе значение переменной Numeral, введя команду «print Numeral». Сравнила с результатом вывода на экран после использования команды «display Numeral». Значения совпадают. (рис. [4.20])

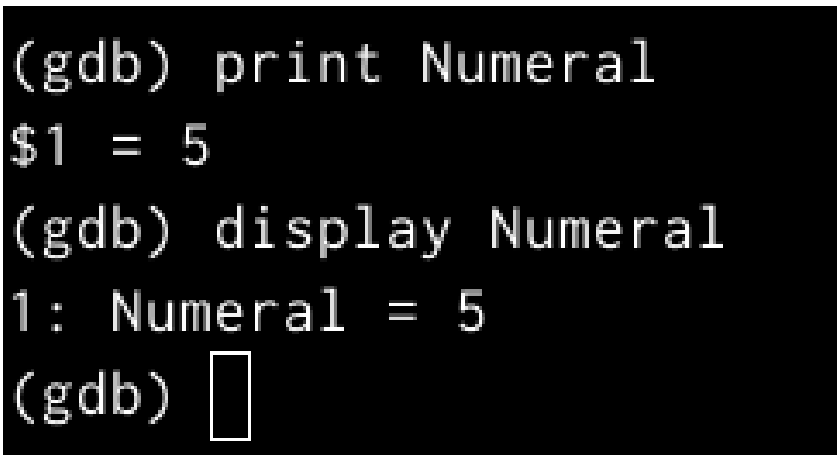

Рис. 4.20: Print Numeral и display Numeral

Убрала точки останова с помощью команд «info breakpoints» и «delete 1». (рис. [4.21])

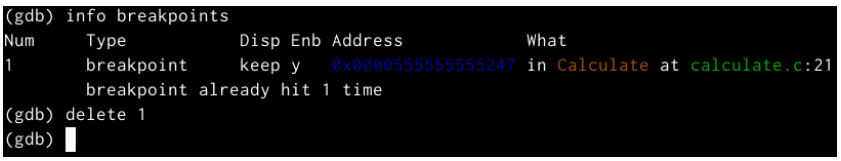

Рис. 4.21: Работа с gdb - info breakpoints

7. Далее воспользовалась командами «splint calculate.c» и «splint main.c». 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. (рис. [4.22], [4.23], [4.24])

```
dskochina@dk3n31 ~/work/os/lab_prog $ splint calculate.c
Splint 3.1.2 --- 07 Dec 2021

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:16:1: 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:22:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:1: 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:38:7: Return value type double does not match declared type float:
                    (HUGE_VAL)
  To allow all numeric types to match, use +relaxtypes.
calculate.c:46:1: 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))
```

Рис. 4.22: Результат команды splint calculate.c

```
calculate.c:38:7: Return value type double does not match declared type float:
                    (HUGE_VAL)
  To allow all numeric types to match, use +relaxtypes.
calculate.c:46:1: 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:
                    (HUGE_VAL)

Finished checking --- 15 code warnings
```

Рис. 4.23: Результат команды splint calculate.c

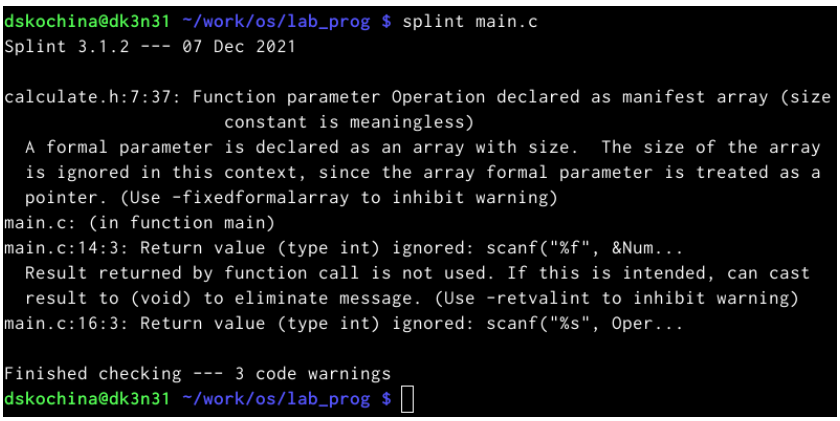

Рис. 4.24: Результат команды splint main.c

Ответы на контрольные вопросы:

1. Чтобы получить информацию о возможностях программ gcc, make, gdbи др.нужно воспользоваться командой manили 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: окодирование –по сути создание исходного текста программы (возможно в нескольких вариантах); –анализ разработанного кода; осборка, компиляция и разработка исполняемого модуля; отестирование и отладка, сохранение произведённых изменений;
- документирование. Для создания исходного текста программы разработчик может воспользоваться любым удобным для него редактором текста: vi, vim, mceditor, emacs, geanyи 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) .своспринимаются gcскак программы на языке С, файлы с расширением .ссили .С–как файлы на языке С++, а файлы срасширением .осчитаются объектными.Например, в команде «gcc-cmain.c»:gcспо расширению (суффиксу) .сраспознает тип файла для компиляции и формирует объектный модуль –файл с расширением .о. Если требуется получить исполняемый файл с определённым именем (например, hello), то требуется воспользоваться опцией -оиви качестве параметра задать имя создаваемого файла: «gcc-ohellomaiВ ходе выполнения данной лабораторной работы я приобрелапростейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linuxна примере создания на языке программирования С калькулятора с простейшими функциями.n.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 makeнеобходимо в корне рабочего каталога с Вашим проектом создать файл с названием makefileили Makefile, в котором будут описаны правила обработки файлов Вашего программного комплекса. В самом простом случае Makefile имеет следующий синтаксис: … : …<команда 1>…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 Makefileможет выступать имя файла или название какого-то действия. Зависимость задаёт исходные пара-

метры (условия) для достижения указанной цели. Зависимость также может быть названием какого-то действия. Команды – собственно действия, которые необходимо выполнить для достижения цели.Общий синтаксис Makefileимеет 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 = gccCFLAGS =# Compile abcd.c normalyabcd: abcd.c$(CC) -o abcd $(CFLAGS) abcd.cclean:-rm abcd .*o* ~# EndMakefileforabcd.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 cleanпроизводит 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 GNUдля ОС типа UNIXвходит отладчик GDB(GNUDebugger). 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-cfile.c-g. После этого для начала работы с gdbнеобходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный файл: gdbfile.o

8. Основные команды отладчика gdb:

backtrace – вывод на экран пути к текущей точке останова (по сутивывод – названий всех функций); break – установить точку останова (в качестве параметра можетбыть 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 (вВ ходе выполнения данной лабораторной работы я приобрелапростейшие навыки разработки, анализа, тестирования и отладки приложений в ОС типа UNIX/Linuxна примере создания на языке программирования С калькулятора с простейшими функциями. качестве параметра может быть указано название файла и через двоеточие номера начальнойи 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 gdbможно 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-hи mangdb.

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 массивасимволов уже является указателемна первый элементэтого 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 Санализатор splintгенерирует 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 работt программы, переменные с некорректно заданными значениямии типами и многое другое.

5 Выводы

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