

Отчёт по лабораторной работе № 13

Королёв Иван Андреевич

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
4	Выполнение лабораторной работы	8
4.1	Подготовка файлов для создания примитивного калькулятора . .	8
4.2	Выполните компиляцию	10
4.3	Создайте Makefile	11
4.4	С помощью gdb выполните отладку программы calcul	12
4.5	С помощью утилиты splint попробуйте проанализировать коды файлов	15
5	Выводы	17
6	Ответы на контрольные вопросы	18

Список иллюстраций

4.1	Файлы	9
4.2	Файлы	10
4.3	Файлы	10
4.4	Выполните компиляцию	11
4.5	Создайте Makefile	11
4.6	gbd	12
4.7	run	12
4.8	list	13
4.9	list 12,15	13
4.10	list calculate.c:20,29	13
4.11	break 21	14
4.12	info breakpoints	14
4.13	run	15
4.14	splint	16
4.15	splint	16

Список таблиц

1 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.

2 Задание

В домашнем каталоге создайте подкаталог `~/work/os/lab_prog`. 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.

3 Теоретическое введение

Процесс разработки программного обеспечения обычно разделяется на следующие этапы:

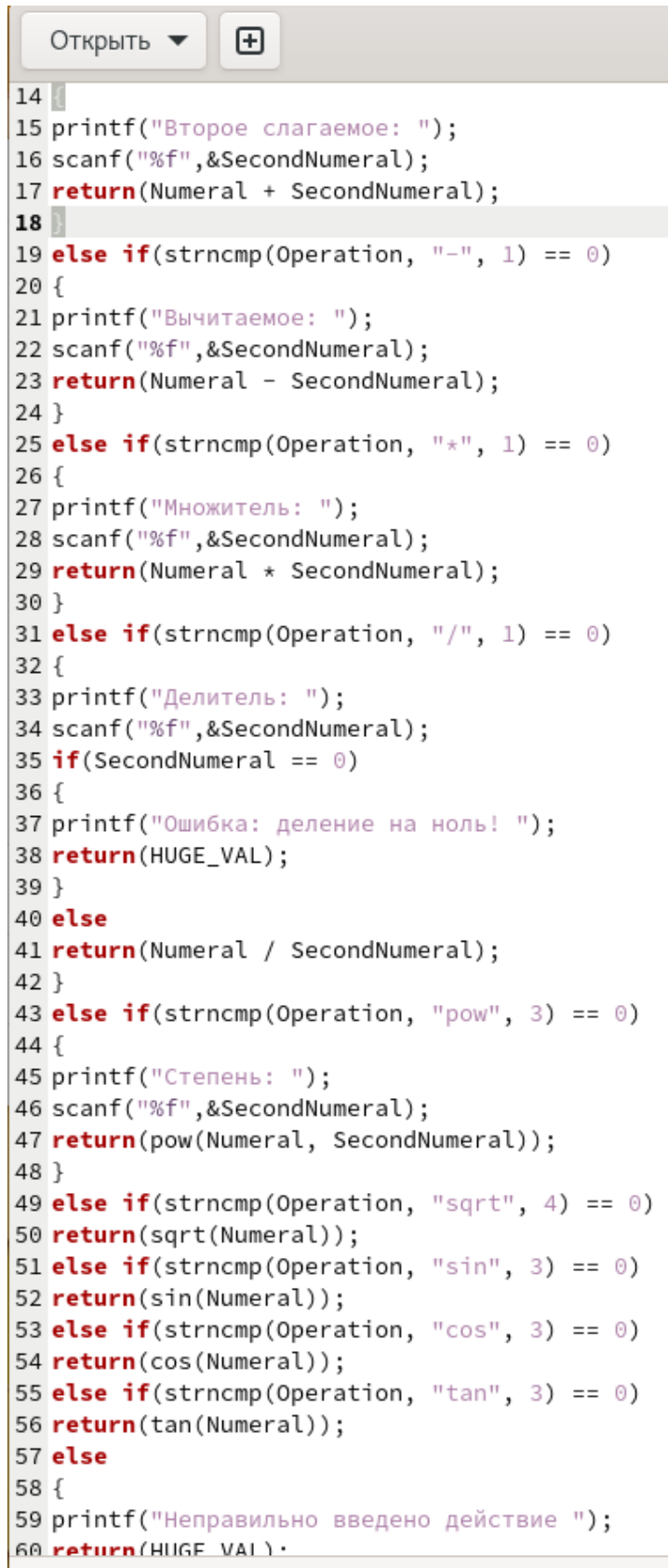
- планирование, включающее сбор и анализ требований к функционалу и другим характеристикам разрабатываемого приложения;
- проектирование, включающее в себя разработку базовых алгоритмов и спецификаций, определение языка программирования;
- непосредственная разработка приложения:
- кодирование — по сути 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

4.1 Подготовка файлов для создания примитивного калькулятора

В домашнем каталоге создайте подкаталог `~/work/os/lab_prog1`. Создайте в нём файлы: `calculate.h`, `calculate.c`, `main.c`. Реализация функций калькулятора в файле `calculate.h` (рис. 4.1). Интерфейсный файл `calculate.h`, описывающий формат вызова функции калькулятора (рис. 4.2). Основной файл `main.c`, реализующий интерфейс пользователя к калькулятору (рис. 4.3).



```
14 {
15 printf("Второе слагаемое: ");
16 scanf("%f",&SecondNumeral);
17 return(Numeral + SecondNumeral);
18 }
19 else if(strncmp(Operation, "-", 1) == 0)
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);
30 }
31 else if(strncmp(Operation, "/", 1) == 0)
32 {
33 printf("Делитель: ");
34 scanf("%f",&SecondNumeral);
35 if(SecondNumeral == 0)
36 {
37 printf("Ошибка: деление на ноль! ");
38 return(HUGE_VAL);
39 }
40 else
41 return(Numeral / SecondNumeral);
42 }
43 else if(strncmp(Operation, "pow", 3) == 0)
44 {
45 printf("Степень: ");
46 scanf("%f",&SecondNumeral);
47 return(pow(Numeral, SecondNumeral));
48 }
49 else if(strncmp(Operation, "sqrt", 4) == 0)
50 return(sqrt(Numeral));
51 else if(strncmp(Operation, "sin", 3) == 0)
52 return(sin(Numeral));
53 else if(strncmp(Operation, "cos", 3) == 0)
54 return(cos(Numeral));
55 else if(strncmp(Operation, "tan", 3) == 0)
56 return(tan(Numeral));
57 else
58 {
59 printf("Неправильно введено действие ");
60 return(HUGE_VAL);
61 }
```

Рис. 4.1: Файлы

```

1 //////////////////////////////////////////////////
2 // calculate.h
3
4 #ifndef CALCULATE_H_
5 #define CALCULATE_H_
6
7 float Calculate(float Numeral, char Operation[4]);
8
9 #endif /*CALCULATE_H_*/
10

```

Рис. 4.2: Файлы

```

1 //////////////////////////////////////////////////
2 // main.c
3
4 #include <stdio.h>
5 #include "calculate.h"
6 int
7 main (void)
8 {
9     float Numeral;
10    char Operation[4];
11    float Result;
12    printf("Число: ");
13    scanf("%f",&Numeral);
14    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
15    scanf("%s",&Operation);
16    Result = Calculate(Numeral, Operation);
17    printf("%6.2f\n",Result);
18    return 0;
19 }

```

Рис. 4.3: Файлы

4.2 Выполните компиляцию

Выполните компиляцию программы посредством gcc (рис. 4.4).

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

Рис. 4.4: Выполните компиляцию

4.3 Создайте Makefile

Создайте Makefile со следующим содержанием (рис. 4.5).

```
1 #
2 # Makefile
3 #
4
5 CC = gcc
6 CFLAGS = -g
7 LIBS = -lm
8
9 calcul: calculate.o main.o
10 gcc calculate.o main.o -o calcul $(LIBS)
11
12 calculate.o: calculate.c calculate.h
13 gcc -c calculate.c $(CFLAGS)
14
15 main.o: main.c calculate.h
16 gcc -c main.c $(CFLAGS)
17
18 clean:
19 -rm calcul *.o *~
20
21 # End Makefile
22
```

Рис. 4.5: Создайте Makefile

4.4 С помощью gdb выполните отладку программы calcul

1. Запустите отладчик GDB, загрузив в него программу для отладки (рис. 4.6).

```
[iakorolev@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora Linux 13.1-2.fc37
Copyright (C) 2023 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".
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...

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

Рис. 4.6: gdb

2. Для запуска программы внутри отладчика введите команду run (рис. 4.7).

```
(gdb) run
Starting program: /home/iakorolev/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 7
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): +
Второе слагаемое: 7
14.00
[Inferior 1 (process 79283) exited normally]
(gdb)
```

Рис. 4.7: run

3. Для постраничного (по 9 строк) просмотра исходного код используйте команду list (рис. 4.8).

```
[Inferior 1 (process 80553) exited normally]
(gdb) list
1  ///////////////////////////////////////////////////
2  // main.c
3
4  #include <stdio.h>
5  #include "calculate.h"
6  int main (void)
7  {
8      float Numeral;
9      char Operation[4];
10     float Result;
(gdb) list
11     printf("Число: ");
12     scanf("%f",&Numeral);
13     printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
14     scanf("%s",Operation);
15     Result = Calculate(Numeral, Operation);
16     printf("%6.2f\n",Result);
17     return 0;
18 }
(gdb)
```

Рис. 4.8: list

4. Для просмотра строк с 12 по 15 основного файла используйте list с параметрами (рис. 4.9).

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

Рис. 4.9: list 12,15

5. Для просмотра определённых строк не основного файла используйте list с параметрами (рис. 4.10).

```
19     Result = Calculate(Numeral, Operation);
(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)
```

Рис. 4.10: list calculate.c:20,29

6. Установите точку остановки в файле `calculate.c` на строке номер 21 (рис. 4.11).

```
(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
```

Рис. 4.11: `break 21`

7. Выведите информацию об имеющихся в проекте точка остановки (рис. 4.12).

```
(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1     breakpoint       keep y   0x000000000040120f in calculate
                                     at calculate.c:21
(gdb)
```

Рис. 4.12: `info breakpoints`

8. Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки остановки. Видим, что программа остановилась в момент прохождения точки остановки. Посмотрим, чему равно на этом этапе значение переменной `Numeral`. Далее, сравним с результатом вывода на экран после использования команды. И в конце, убедившись, что всё правильно, удалим точку остановки (рис. 4.13).

```

Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdee4 "-")
at calculate.c:21
21      printf("Вычитаемое: ");
(gdb) print Numeral
$1 = 5
(gdb) display Numeral
1: Numeral = 5
(gdb) delete 1
(gdb) info breakpoints
No breakpoints or watchpoints.
(gdb)

```

Рис. 4.13: run

4.5 С помощью утилиты splint попробуйте проанализировать коды файлов

1. С помощью утилиты splint анализирую кода файла calculate.c (рис. 4.14).

```

[iakorolev@fedora lab_prog1]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 2022

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:16:9: 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:22:9: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:5: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:5: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:8: 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:38:15: Return value type double does not match declared type float:
(HUGE_VAL)
To allow all numeric types to match, use +relaxtypes.
calculate.c:46:5: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:47:11: Return value type double does not match declared type float:
(pow(Numeral, SecondNumeral))
calculate.c:50:15: Return value type double does not match declared type float:
(sqrt(Numeral))
calculate.c:52:15: Return value type double does not match declared type float:
(sin(Numeral))
calculate.c:54:15: Return value type double does not match declared type float:
(cos(Numeral))
calculate.c:56:15: Return value type double does not match declared type float:
(tan(Numeral))
calculate.c:60:15: Return value type double does not match declared type float:
(HUGE_VAL)

Finished checking --- 15 code warnings
[iakorolev@fedora lab_prog1]$

```

Рис. 4.14: splint

2. С помощью утилиты splint анализирую кода файла main.c (рис. 4.15).

```

[iakorolev@fedora lab_prog1]$ splint main.c
Splint 3.1.2 --- 23 Jul 2022

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:12:5: 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:14:5: Return value (type int) ignored: scanf("%s", Oper...

Finished checking --- 3 code warnings
[iakorolev@fedora lab_prog1]$

```

Рис. 4.15: splint

5 Выводы

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.

6 Ответы на контрольные вопросы

1. С помощью `man`
2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX? Unix поддерживает следующие основные этапы разработки приложений:
 - создание исходного кода программы;
 - представляется в виде файла;
 - 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;
 - проверка кода на наличие ошибок
 - сохранение всех изменений, выполняемых при тестировании и отладке.
3. Что такое суффиксы и префиксы? Основное их назначение. Приведите примеры их использования.
 - Использование суффикса “.c” для имени файла с программой на языке Си отражает удобное и полезное соглашение, принятое в ОС UNIX. Для любого имени входного файла суффикс определяет какая компиляция требуется. Суффиксы и префиксы указывают тип объекта. Одно из полезных свойств

компилятора Си — его способность по суффиксам определять типы файлов. По суффиксу `.c` компилятор распознает, что файл `abcd.c` должен компилироваться, а по суффиксу `.o`, что файл `abcd.o` 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`. Некоторые проекты предпочитают показывать префиксы в начале текста изменений для старых (`old`) и новых (`new`) файлов. Опция `-prefix` может быть использована для установки такого префикса. Плюс к этому команда `bzr diff -p1` выводит префиксы в форме которая подходит для команды `patch -p1`.

4. Основное назначение компилятора с языка Си в UNIX?

- Основное назначение компилятора с языка Си заключается в компиляции всей программы в целом и получении исполняемого модуля.

5. Для чего предназначена утилита `make`.

- При разработке большой программы, состоящей из нескольких исходных файлов заголовков, приходится постоянно следить за файлами, которые требуют перекомпиляции после внесения изменений. Программа `make` освобождает пользователя от такой рутинной работы и служит для документирования взаимосвязей между файлами. Описание взаимосвязей и соответствующих действий хранится в так называемом `make`-файле, который по умолчанию имеет имя `makefile` или `Makefile`.

6. Приведите структуру `make`-файла. Дайте характеристику основным элементам этого файла.

- `makefile` для программы `abcd.c` мог бы иметь вид:

```

#
#
Makefile
#
CC = gcc
CFLAGS =
LIBS = -lm
calcul: calculate.o main.o gcc calculate.o main.o -o calcul $(LIBS) calculate.o:
c calculate.c $(CFLAGS) main.o: main.c calculate.h gcc -c main.c $(CFLAGS) clean:
rm calcul *.o *~
#End Makefile

```

- В общем случае make-файл содержит последовательность записей (строк), определяющих зависимости между файлами. Первая строка записи представляет собой список целевых (зависимых) файлов, разделенных пробелами, за которыми следует двоеточие и список файлов, от которых зависят целевые. 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, так как содержимое строки, начиная с `#` и до конца строки, не будет обрабатываться командой `make`; `:` — 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. Приведённый выше make-файл для программы `abcd.c` включает два способа компиляции и построения исполняемого модуля. Первый способ предусматривает

обычную компиляцию с построением исполняемого модуля с именем `abcd`. Второй способ позволяет включать в исполняемый модуль `testabcd` возможность выполнить процесс отладки на уровне исходного текста.

7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?
- Пошаговая отладка программ заключается в том, что выполняется один оператор программы и, затем контролируются те переменные, на которые должен был воздействовать данный оператор. Если в программе имеются уже отлаженные подпрограммы, то подпрограмму можно рассматривать, как один оператор программы и воспользоваться вторым способом отладки программ. Если в программе существует достаточно большой участок программы, уже отлаженный ранее, то его можно выполнить, не контролируя переменные, на которые он воздействует. Использование точек останова позволяет пропускать уже отлаженную часть программы. Точка останова устанавливается в местах, где необходимо проверить содержимое переменных или просто проконтролировать, передаётся ли управление данному оператору. Практически во всех отладчиках поддерживается это свойство (а также выполнение программы до курсора и выход из подпрограммы). Затем отладка программы продолжается в пошаговом режиме с контролем локальных и глобальных переменных, а также внутренних регистров микроконтроллера и напряжений на выводах этой микросхемы.
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;
- splist – 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. Выполнили компиляцию программы
2. Увидели ошибки в программе
3. Открыли редактор и исправили программу
4. Загрузили программу в отладчик gdb
5. run — отладчик выполнил программу, мы ввели требуемые значения.
6. программа завершена, gdb не видит ошибок.

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

1. отладчику не понравился формат %s для &Operation, т.к %s — символьный формат, а значит необходим только Operation.
11. Назовите основные средства, повышающие понимание исходного кода программы. Если вы работаете с исходным кодом, который не вами разрабатывался, то назначение различных конструкций может быть не совсем понятным. Система разработки приложений UNIX предоставляет различные средства, повышающие понимание исходного кода. К ним относятся:
 - cscope - исследование функций, содержащихся в программе;
 - splint — критическая проверка программ, написанных на языке Си.
12. Каковы основные задачи, решаемые программой slint?
 1. Проверка корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений;
 2. Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;
 3. Общая оценка мобильности пользовательской программы.