Лабораторная работа №14

Российский университет дружбы народов

Андреев Владислав Владимирович

Содержание

Список таблиц

Список иллюстраций

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 Выполнение лабораторной работы

1.В домашнем каталоге создаём подкаталог ~/work/os/lab_prog.

2.Создаём 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.

Командный файл calculate.c и его код(Рисунок 2.1, 2.2, 2.3).

```c
////////////////////////////////////// calculate.c
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

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

Рис. 2.1: calculate.c

```
      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)
```

Рис. 2.2: calculate.c

```
    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. 2.3: calculate.c

Командный файл calculate.h и его код(Рисунок 2.4).

```
//////////////////////////////////////// calculate.h
#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
U:**-  calculate.h     All L7     (C/*l Abbrev)
```

Рис. 2.4: calculate.h

Командный файл main.c и его код(Рисунок 2.5).

```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;
}
```

Рис. 2.5: main.c

3.Выполняем компиляцию программы посредством gcc:

gcc -c calculate.c

gcc -c main.c

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

4.Исправляем синтаксические ошибки.

5.Создаём Makefile(Рисунок 2.6).

```
#
# 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. 2.6: Makefile

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.Исправляем Makefile(Рисунок 2.7).

```
#
# 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. 2.7: Исправленный Makefile

В переменную CFLAGS добавляем опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Делаем так, что утилита компиляции выбирается с помощью переменной CC. После этого удаляем исполняемые и объектные файлы из каталога с помощью команды «make clean». Выполняем компиляцию файлов, используя команды «make calculate.o», «make main.o», «make calcul».

Далее с помощью gdb выполняем отладку программы calcul. Запускаем отладчик GDB, загрузив в него программу для отладки, используя команду:«gdb./calcul»(Рисунок 2.8).

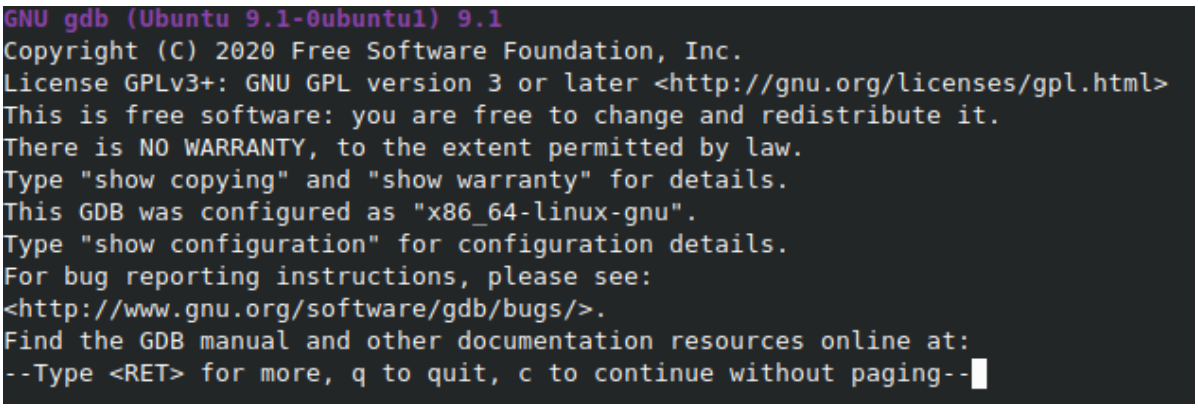

Рис. 2.8: GDB

Для запуска программы внутри отладчика вводим команду run(Рисунок 2.9).

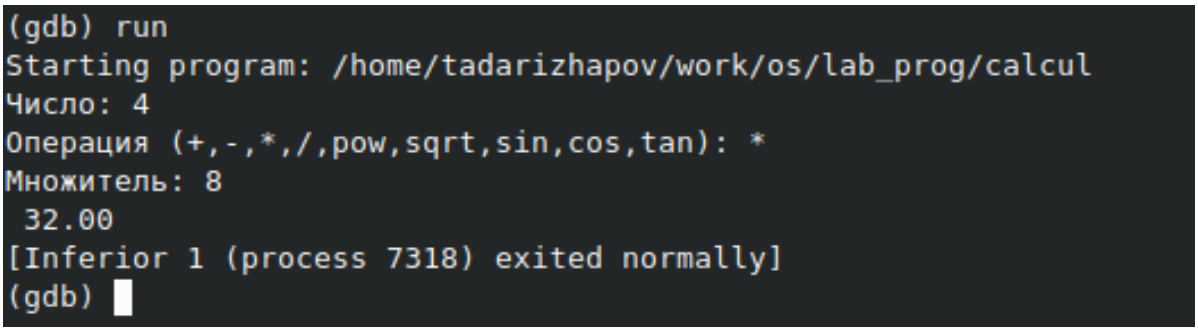

Рис. 2.9: run

Для постраничного(по 10 строк) просмотра исходного кода используем команду list(Рисунок 2.10).

```
(gdb) list
1       ////////////////////////////////////////// main.c
2
3       #include <stdio.h>
4       #include "calculate.h"
5       int
6       main (void)
7       {
8         float Numeral;
9         char Operation[4];
10        float Result;
```

Рис. 2.10: list

Для просмотра строк с 12 по 15 основного файла использовала команду list 12,15(Рисунок 2.11).

```
(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. 2.11: list2

Для просмотра определённых строк неосновного файла используем команду list calculate.c:20,29(Рисунок 2.12).

```
(gdb) list calculate.c:20,29
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);
28            }
29          else if(strncmp(Operation, "/", 1) == 0)
(gdb)
```

Рис. 2.12: list3

Установливаем точку останова в файле calculate.c на строке номер 21, используя команды list calculate.c:20,27 и break 21(Рисунок 2.13).

```
(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 0x555555555306: file calculate.c, line 21.
```

Рис. 2.13: Точка останова

Выводим информацию об имеющихся в проекте точках останова с помощью команды info breakpoints(Рисунок 2.14).

```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000555555555306 in Calculate at calculate.c:21
(gdb)
```

Рис. 2.14: info breakpoints

Запусткаем программу внутри отладчика и убеждаемся, что программа остановилась в момент прохождения точки останова. Используем команды run, 5, – и backtrace(Рисунок 2.15).

```
(gdb) run
Starting program: /home/tadarizhapov/work/os/lab_prog/calcul
Число: 5
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): -

Breakpoint 1, Calculate (Numeral=5, Operation=0x7fffffffdda4 "-") at calculate.c:21
21            printf("Вычитаемое: ");
(gdb) backtrace
#0  Calculate (Numeral=5, Operation=0x7fffffffdda4 "-") at calculate.c:21
#1  0x00005555555555bd in main () at main.c:15
(gdb)
```

Рис. 2.15: Запуск программы

Смотрим, чему равно на этом этапе значение переменной Numeral, введя команду print Numeral(Рисунок 2.16).

```
(gdb) print Numeral
$1 = 5
(gdb)
```

Рис. 2.16: print

Сравниваем с результатом вывода на экран после использования команды display Numeral. Значения совпадают(Рисунок 2.17).

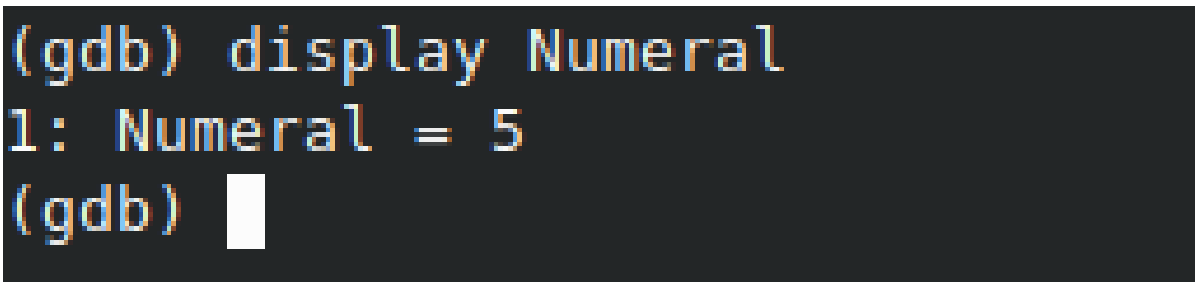

Рис. 2.17: display

Убираем точки останова с помощью команд info breakpoints и delete 1(Рисунок 2.18).

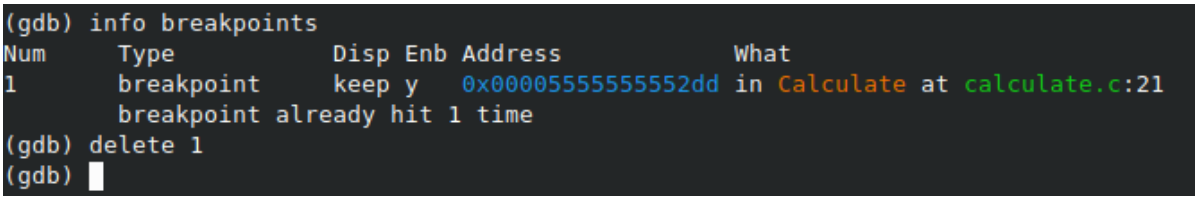

Рис. 2.18: Удаление

7.Далее воспользуемся 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(Рисунок 2.19, 2.20).

```
Splint 3.1.2 --- 20 Feb 2018

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:10:31: Function parameter Operation declared as manifest array
                     (size constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:16:7: 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:22:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:28:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:35:10: 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:38:10: 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:
```

Рис. 2.19: splint1

```
Splint 3.1.2 --- 20 Feb 2018

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:12:3: 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:14:3: Return value (type int) ignored: scanf("%s", Oper...

Finished checking --- 3 code warnings
```

Рис. 2.20: splint2

3 Выводы

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

4 Ответы на 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: 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.

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: … : … <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 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* ~ 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 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 UNIXвходит отладчик GDB(GNU Debugger). Для использования GDBнеобходимо 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 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 (в 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 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 Санализатор 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 значениямии типами и многое другое.