

13 Лабораторная работа

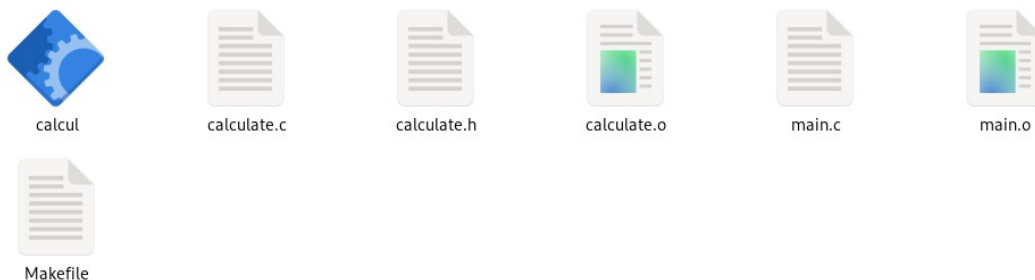
Прищепов Александр

Цель работы

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

Выполнение лабораторной работы

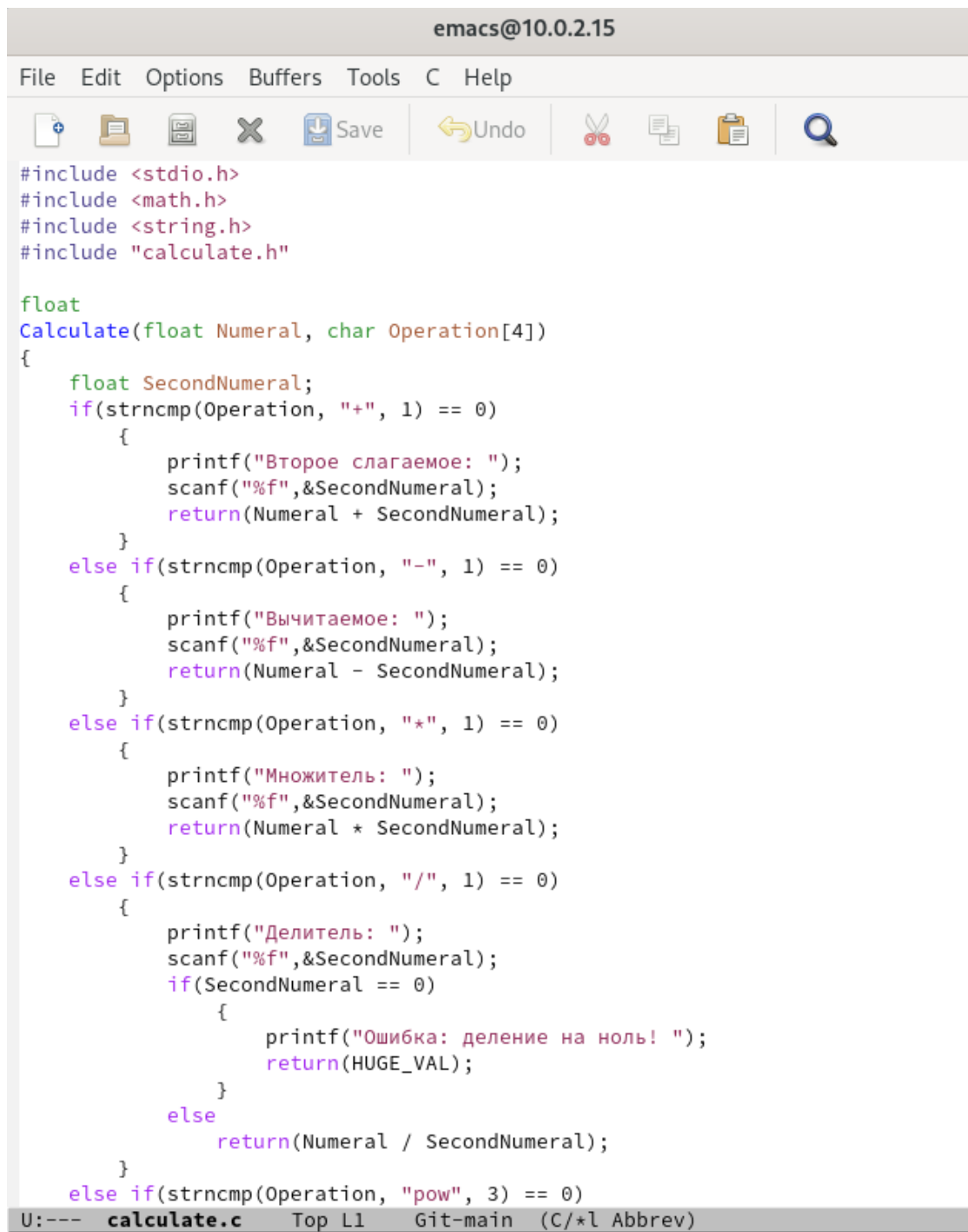
1. В домашнем каталоге создаем подкаталог `~/work/os/lab_prog` и в нем уже создаем три файла: `calculate.h`, `calculate.c`, `main.c` (рис. 1). Это будет примитивнейший калькулятор, способный складывать, вычитать, умножать и делить, возводить число в степень, брать квадратный корень, вычислять `sin`, `cos`, `tan`. При запуске он будет запрашивать первое число, операцию, второе число. После этого программа выведет результат и остановится.



изображение

2. В созданных файлах напишем программы для работы калькулятора, которые нам предоставили (рис. 2), (рис. 3), (рис. 4).

рис 2:



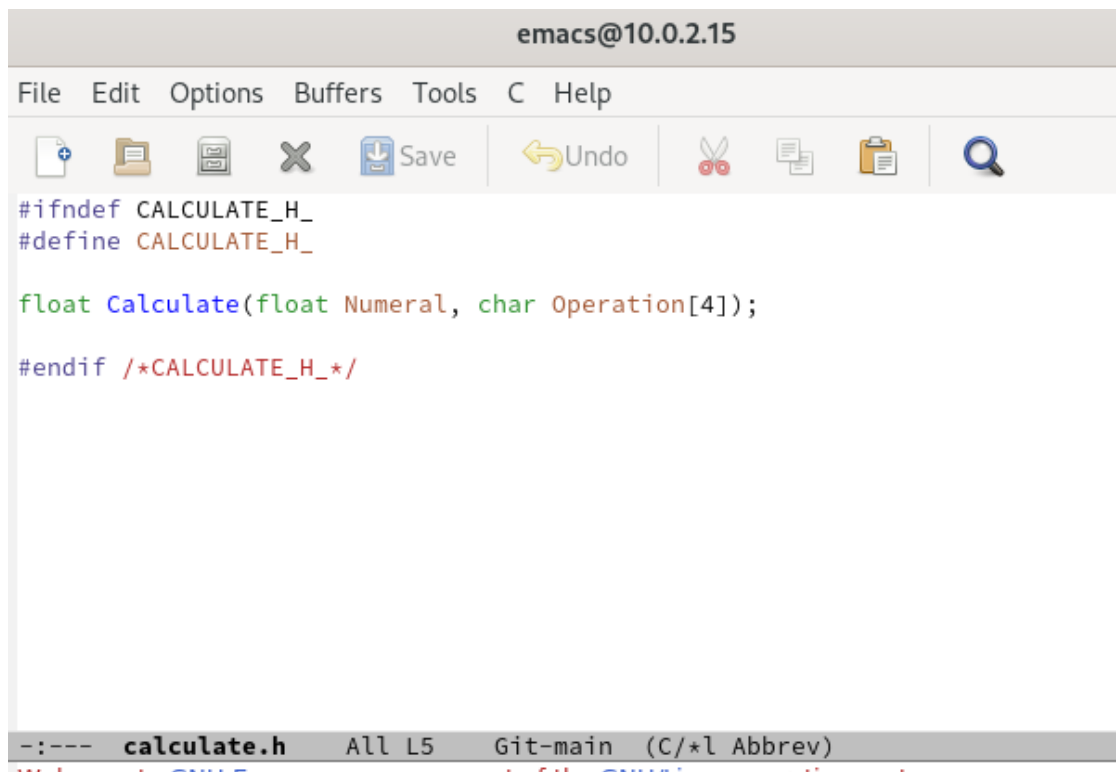
```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "calculate.h"

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("Повышение в степень: ");
        scanf("%f",&SecondNumeral);
        return(pow(Numeral, SecondNumeral));
    }
}
```

U:--- calculate.c Top L1 Git-main (C/*l Abbrev)

изображение

рис 3:



The screenshot shows the Emacs editor interface with the title bar 'emacs@10.0.2.15'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations and editing. The main text area displays the following C code:

```
#ifndef CALCULATE_H_
#define CALCULATE_H_

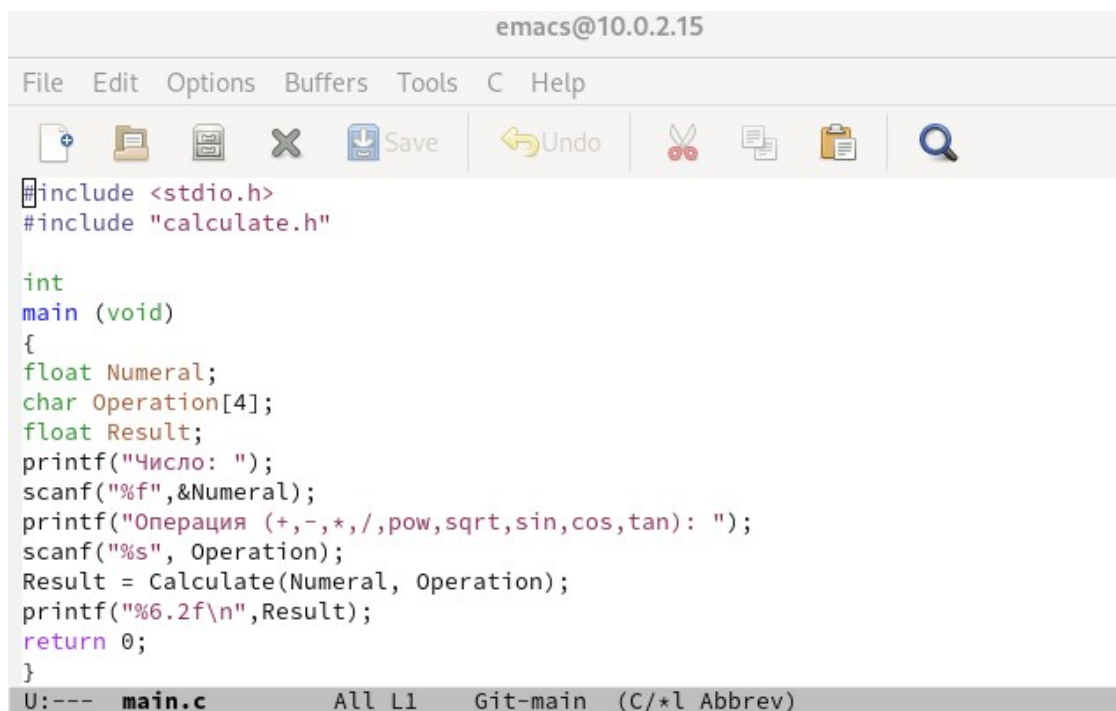
float Calculate(float Numeral, char Operation[4]);

#endif /*CALCULATE_H_*/
```

The status bar at the bottom indicates the current file is 'calculate.h', it is on line 5, and the buffer is 'Git-main (C/*l Abbrev)'.

изображение

рис 4:



The screenshot shows the Emacs editor interface with the title bar 'emacs@10.0.2.15'. The menu bar includes 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C', and 'Help'. The toolbar contains icons for file operations and editing. The main text area displays the following C code:

```
#include <stdio.h>
#include "calculate.h"

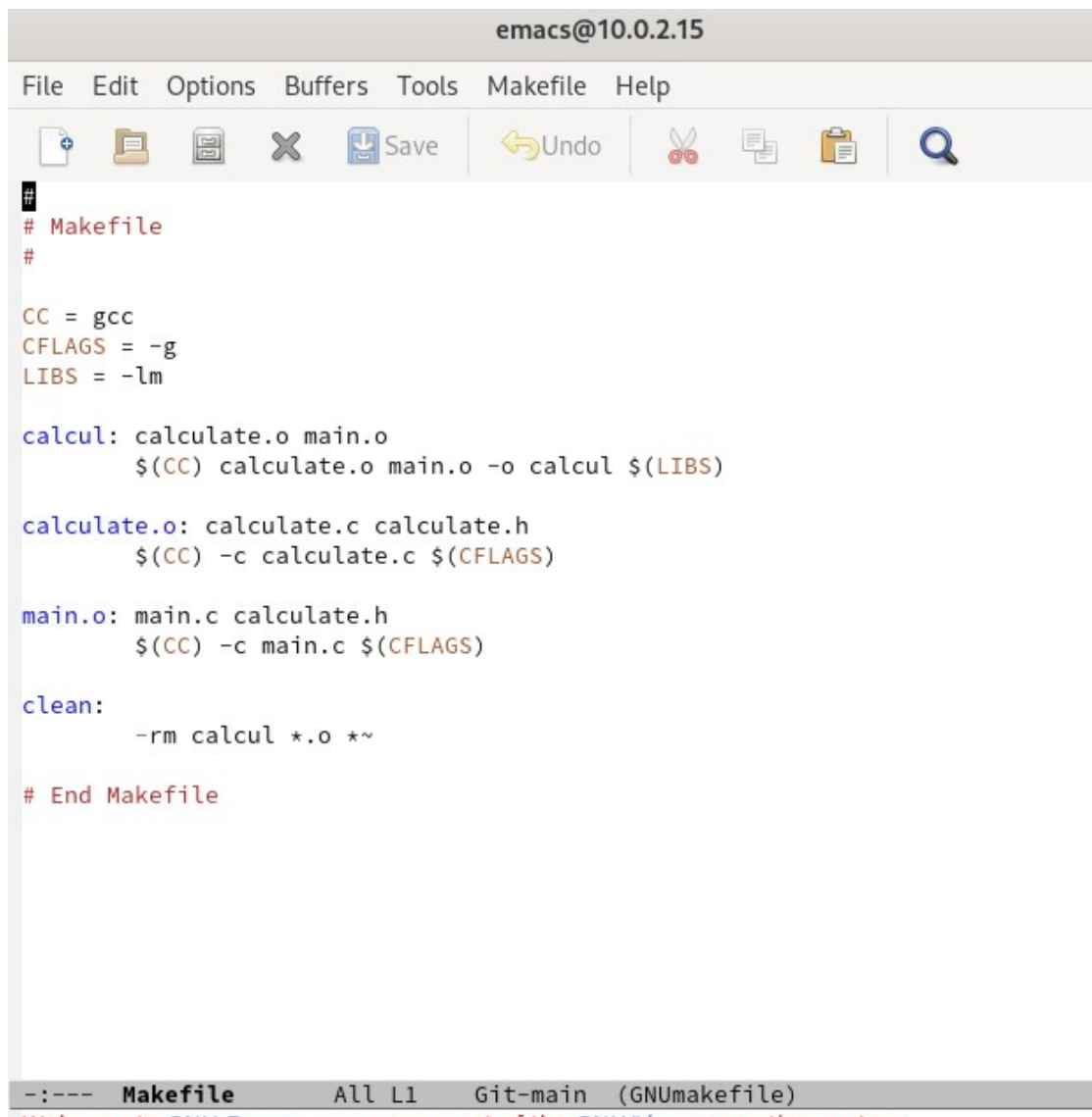
int
main (void)
{
    float Numeral;
    char Operation[4];
    float Result;
    printf("Число: ");
    scanf("%f",&Numeral);
    printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");
    scanf("%s", Operation);
    Result = Calculate(Numeral, Operation);
    printf("%6.2f\n",Result);
    return 0;
}
```

The status bar at the bottom indicates the current file is 'main.c', it is on line 1, and the buffer is 'Git-main (C/*l Abbrev)'.

изображение

3. Выполним компиляцию программы посредством gcc и при необходимости исправим синтаксические ошибки
4. Создадим Makefile и введем в него предложенное содержимое (рис. 5).

рис 5:



```
#
# 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
```

изображение

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.

5. Далее исправим Makefile. В переменную CFLAGS добавил опцию -g, необходимую для компиляции объектных файлов и их использования в программе отладчика GDB. Сделаем так, что утилита компиляции выбирается с помощью переменной CC.

После этого удалим исполняемые и объектные файлы из каталога с помощью команды `make clean`. Выполним компиляцию файлов, используя команды `make calculate.o`, `make main.o`, `make calcul`.

6. Далее с помощью команды `gdb ./calcul` запустим отладку программы
 - Для запуска программы внутри отладчика введем команду `run`
 - Для постраничного (по 9 строк) просмотра исходного кода используем команду `list`
 - Для просмотра строк с 12 по 15 основного файла используем `list` с параметрами
 - Для просмотра определённых строк не основного файла используем `list` с параметрами
 - Установим точку останова в файле `calculate.c` на строке номер 18 и выведем информацию об имеющихся в проекте точка останова
 - Запустим программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова
 - Введем команду `backtrace`, которая покажет весь стек вызываемых функций от начала программы до текущего места
 - Посмотрим, чему равно на этом этапе значение переменной `Numeral`, введя команду `print Numeral` и сравним с результатом команды `display Numeral`
 - Уберем точки останова
7. С помощью утилиты `splint` проанализируем коды файлов `calculate.c` и `main.c`. 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 сохраняются. 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 потере данных.

Выводы

Здесь кратко описываются итоги проделанной работы.

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.

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 C 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.