Отчёт по лабораторной работе №14

дисциплина: Операционные системы

Быстров Глеб Андреевич

Содержание

1 Цель работы

В данной лабораторной работе мне будет необходимо 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.

2 Теория

Bash — популярный командный интерпретатор, используемый в юниксоподобных системах, например, в GNU/Linux. Это программа, которую называют оболочка либо шелл (shell), а само название «bash» является сокращением от «Bourne Again Shell». Интерпретатор Bash принимает ваши команды, передавая их операционной системе. Чтобы осуществлялось взаимодействие с ОС, применяются терминалы (gnome-terminal, nxterm и прочие).

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 специфи- каций, 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.

Стандартным средством для компиляции программ в ОС типа UNIX является GCC (GNU Compiler Collection). Это набор компиляторов для разного рода языков программирования (C, C++, Java, Фортран и др.). Работа с GCC производится при помощи одноимённой управляющей программы gcc , которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла. 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.

3 Задание

1. В домашнем каталоге создайте подкаталог ~/work/os/lab_prog.
2. 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. Выполните компиляцию программы посредством gcc: gcc -c calculate.c gcc -c main.c gcc calculate.o main.o -o calcul -lm
4. При необходимости исправьте синтаксические ошибки.
5. Создайте Makefile со следующим содержанием: 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* ~ Поясните в отчёте его содержание.
6. С помощью gdb выполните отладку программы calcul (перед использованием gdb исправьте Makefile)
7. С помощью утилиты splint попробуйте проанализировать коды файлов calculate.c и main.c.

4 Выполнение лабораторной работы

1. В домашнем каталоге создал подкаталог ~/work/os/lab_prog. Использовал команды mkdir и cd. (рис. 4.1)

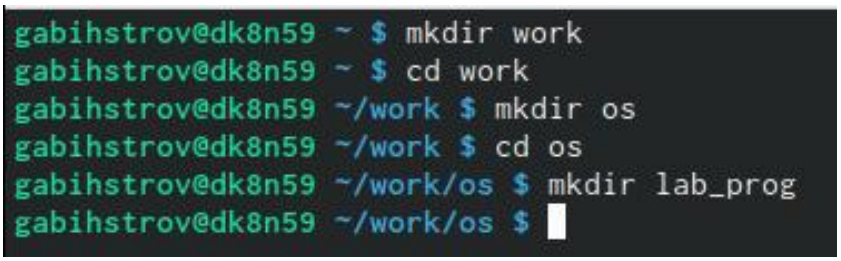


Figure 4.1: Создание подкаталога

2. Создал в нём файлы: calculate.h, calculate.c, main.c. (рис. 4.2)

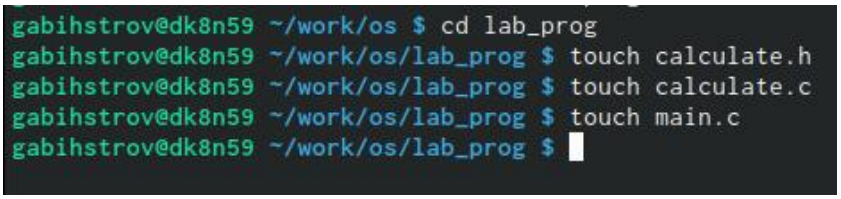
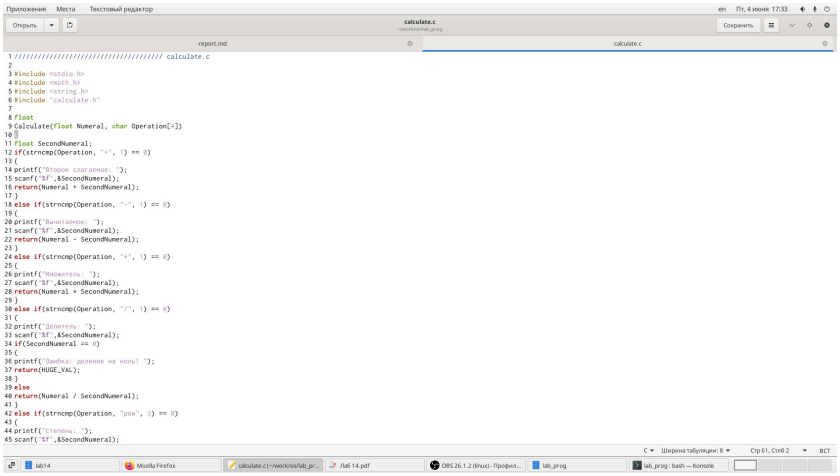


Figure 4.2: Создание файлов

3. Реализация функций калькулятора в файле calculate.c. (рис. 4.3)



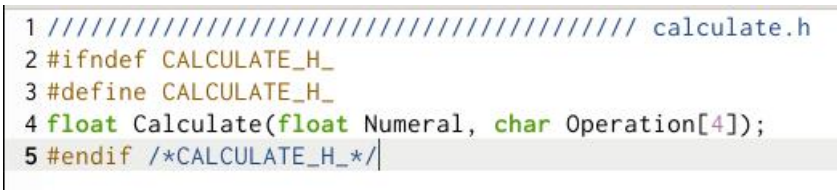
```
////////////////////////////////////// calculate.c

#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);
```

Figure 4.3: Файл calculate.c

4. Интерфейсный файл calculate.h, описывающий формат вызова функции калькулятора. (рис. 4.4)



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

Figure 4.4: Файл calculate.h

5. Основной файл main.c, реализующий интерфейс пользователя к калькулятору. (рис. 4.5)

```
1 ////////////////////////////////////// main.c
2
3 #include <stdio.h>
4 #include "calculate.h"
5
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 }
```

Figure 4.5: Файл main.c

6. Выполнил компиляцию программы посредством gcc: gcc -c calculate.c gcc -c main.c gcc calculate.o main.o -o calcul -lm (рис. 4.6)

```
gabihstrov@dk8n59 ~/work/os/lab_prog $ gcc -c calculate.c
gabihstrov@dk8n59 ~/work/os/lab_prog $ gcc -c main.c
main.c: В функции «main»:
main.c:15:9: предупреждение: формат «%s» ожидает аргумент типа «char *», но аргумент 2 имеет тип «char (*)[4]» [-Wformat=]
   15 | scanf("%s",&Operation);
      |        ~^  ~~~~~~~~~~
      |         |  |
      |         |  char (*)[4]
      |         char *
gabihstrov@dk8n59 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm
gabihstrov@dk8n59 ~/work/os/lab_prog $
```

Figure 4.6: Компиляция

7. Создал Makefile. (рис. 4.7)

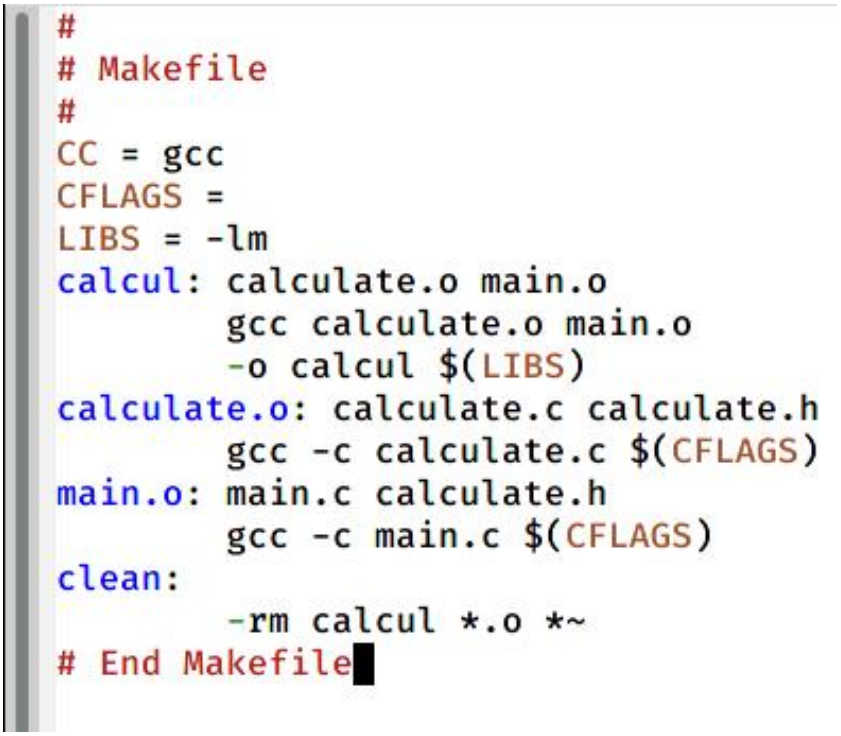


Figure 4.7: Makefile

8. С помощью gdb выполнил отладку программы calcul (перед использованием gdb исправил Makefile). Запустил отладчик GDB, загрузив в него программу для отладки gdb ./calcul. Для запуска программы внутри отладчика ввёл команду run. (рис. 4.8)

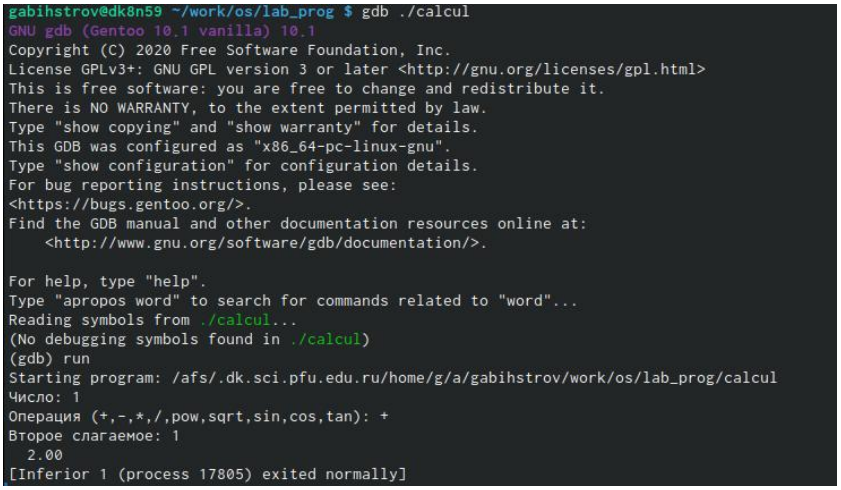


Figure 4.8: Отладка программы calcul

9. Для постраничного (по 9 строк) просмотра исходного код использовал команду list. (рис. 4.9)

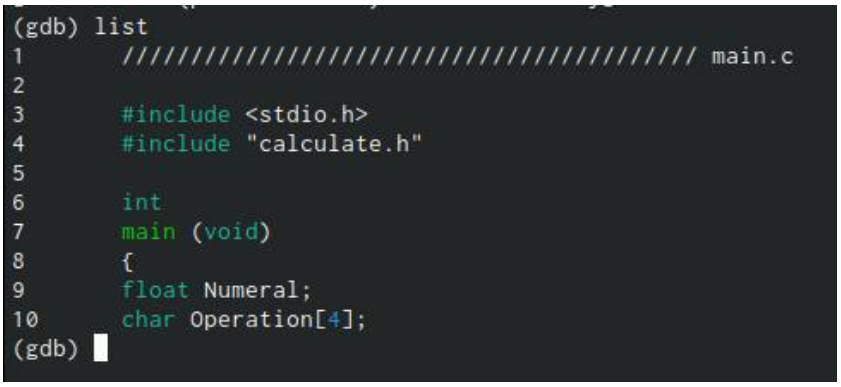


Figure 4.9: Команда list

10. Для просмотра строк с 12 по 15 основного файла использовал list с параметрами list 12,15. (рис. 4.10)

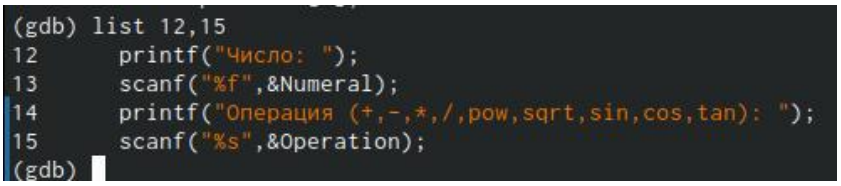


Figure 4.10: Просмотр строк

11. Для просмотра определённых строк не основного файла используйте list с параметрами: list calculate.c:20,29. (рис. 4.11)

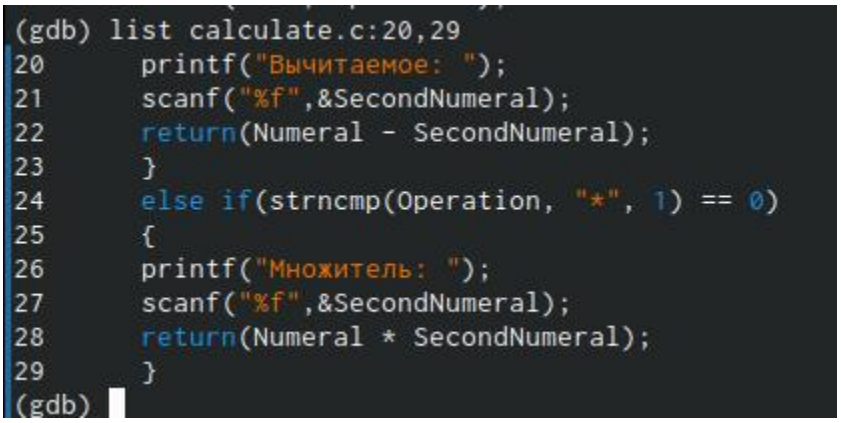


Figure 4.11: Просмотр определённых строк

12. Установите точку останова в файле calculate.c на строке номер 21: list calculate.c:20,27 break 21 (рис. 4.12)

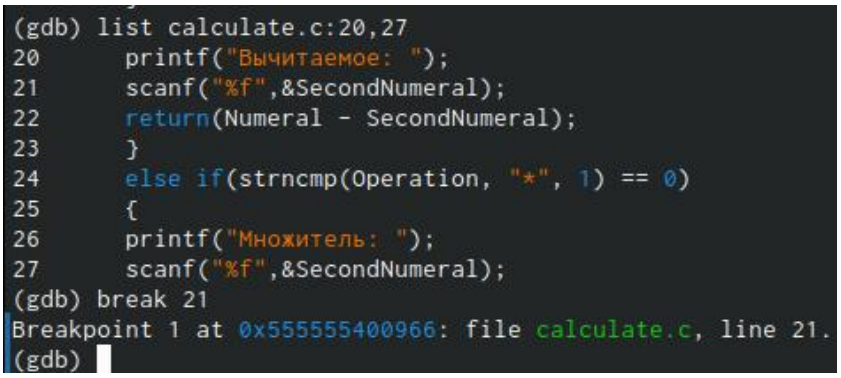


Figure 4.12: Установил точку останова

13. Выведите информацию об имеющихся в проекте точка останова: info breakpoints. (рис. 4.13)

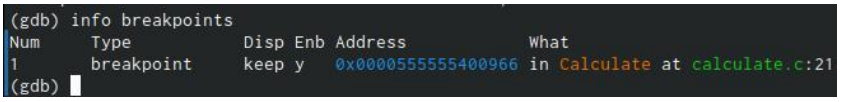


Figure 4.13: Информация о точке останова

14. Запустил программу внутри отладчика и убедился, что программа остановилась в момент прохождения точки останова (рис. 4.14)

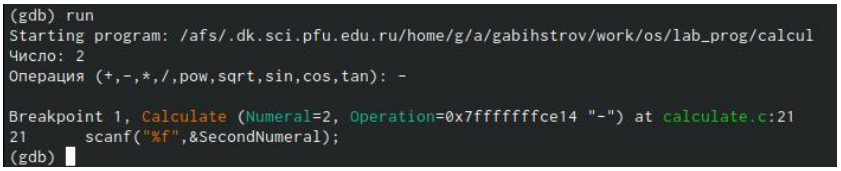


Figure 4.14: Остановка программы

15. Посмотрите, чему равно на этом этапе значение переменной Numeral, введя: print Numeral. На экран выведено число 2. (рис. 4.15)

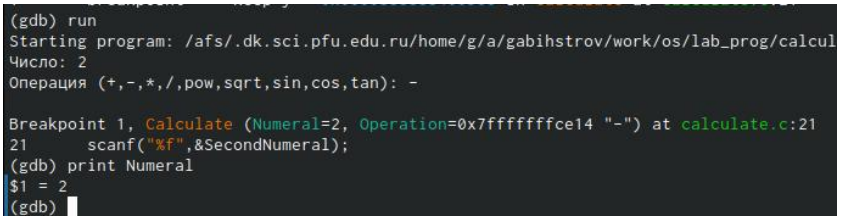


Figure 4.15: Переменная Numeral

16. Сравнил с результатом вывода на экран после использования команды: display Numeral. Цифры сходятся. (рис. 4.16)

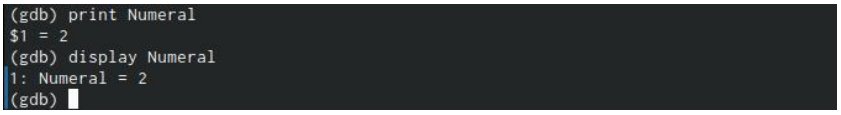


Figure 4.16: Сравнение цифр

17. Убрал точки останова: info breakpoints delete 1. (рис. 4.17)

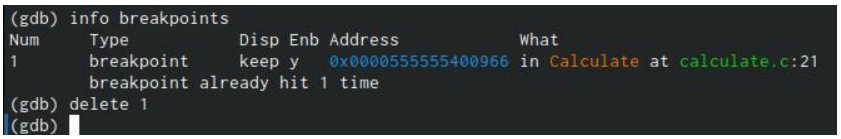


Figure 4.17: Убрал точки

18. С помощью утилиты splint проанализировал коды файлов calculate.c и main.c. (рис. 4.18) (рис. 4.19)

```
gabihstrov@dk8n59 ~/work/os/lab_prog $ splint calculate.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:4:37: 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:9:31: Function parameter Operation declared as manifest array (size
                    constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:15:1: 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:21:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:27:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:33:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:34:4: 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:37:7: Return value type double does not match declared type float:
                    (HUGE_VAL)
  To allow all numeric types to match, use +relaxtypes.
calculate.c:45:1: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:46:7: Return value type double does not match declared type float:
                    (pow(Numeral, SecondNumeral))
calculate.c:49:7: Return value type double does not match declared type float:
                    (sqrt(Numeral))
calculate.c:51:7: Return value type double does not match declared type float:
                    (sin(Numeral))
calculate.c:53:7: Return value type double does not match declared type float:
                    (cos(Numeral))
calculate.c:55:7: Return value type double does not match declared type float:
                    (tan(Numeral))
calculate.c:59:7: Return value type double does not match declared type float:
                    (HUGE_VAL)

Finished checking --- 15 code warnings
gabihstrov@dk8n59 ~/work/os/lab_prog $
```

Figure 4.18: calculate.c

```
gabihstrov@dk8n59 ~/work/os/lab_prog $ splint main.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:4:37: 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:13:1: 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:15:12: Format argument 1 to scanf (%s) expects char * gets char [4] *:
                &Operation
  Type of parameter is not consistent with corresponding code in format string.
  (Use -formattype to inhibit warning)
   main.c:15:9: Corresponding format code
main.c:15:1: Return value (type int) ignored: scanf("%s", &Ope...

Finished checking --- 4 code warnings
gabihstrov@dk8n59 ~/work/os/lab_prog $
```

Figure 4.19: main.c

5 Контрольные вопросы

1. Как получить информацию о возможностях программ gcc, make, gdb и др.?

Чтобы получить информацию о возможностях программ gcc, make, gdb необходимо использовать команды man, info или help. Также можно воспользоваться интернетом и узнать необходимую информацию.

2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.

Библиографический список ссылка №1

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 специфи- каций, 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.

3. Что такое суффикс в контексте языка программирования? Приведите примеры использования.

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.

Таким образом, gcc по расширению (суффиксу) .c распознает тип файла для компиляции и формирует объектный модуль — файл с расширением .o.

4. Каково основное назначение компилятора языка C в UNIX?

Стандартным средством для компиляции программ в ОС типа UNIX является GCC (GNU Compiler Collection). Это набор компиляторов для разного рода языков программирования (C, C++, Java, Фортран и др.). Работа с GCC производится при помощи одноимённой управляющей программы gcc, которая интерпретирует аргументы командной строки, определяет и осуществляет запуск нужного компилятора для входного файла. 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.

5. Для чего предназначена утилита make?

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. 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.

6. Приведите пример структуры Makefile. Дайте характеристику основным элементам этого файла.

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.

Рассмотрим пример Makefile для написанной выше простейшей программы, выводящей на экран приветствие 'Hello World!': hello: main.c gcc -o hello main.c Здесь в первой строке hello — цель, main.c — название файла, который мы хотим скомпилировать; во второй строке, начиная с табуляции, задана команда компиляции gcc с опциями. Для запуска программы необходимо в командной строке набрать команду make: make 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. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

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 -g73

После этого для начала работы с gdb необходимо в командной строке ввести одноимённую команду, указав в качестве аргумента анализируемый бинарный

файл: gdb file.o Затем можно использовать по мере необходимости различные команды gdb.

8. Назовите и дайте основную характеристику основным командам 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.

9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.

- Запустил отладчик для программы;
- установил точку остановки;
- запустил программу;
- узнал необходимые данные путём ввода команд;
- завершил программу.

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

В строке scanf(“%s”,&Operation) необходимо убрать &, так как им массива уже является указателем на первый элемент. Также компилятор писал, что программа принимает указатель на char массив.

11. Назовите основные средства, повышающие понимание исходного кода программы.

Библиографический список ссылка №2

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;
- lint — критическая проверка программ, написанных на языке Си.

Для постраничного (по 9 строк) просмотра исходного код используйте команду list : list Для просмотра строк с 12 по 15 основного файла используйте list с пара- метрами: list 12,15

12. Каковы основные задачи, решаемые программой splint?

Библиографический список ссылка №3

Ещё одним средством проверки исходных кодов программ, написанных на языке С, является утилита splint . Эта утилита 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.

6 Выводы

В данной лабораторной работе мне успешно удалось 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.

7 Библиографический список

1. Разработка Unix-приложений (https://qna.habr.com/q/17069)
2. Статический анализ кода (https://www.jetinfo.ru/staticheskij-analiz-koda-chto-mogut-instrumentalnye-sredstva/)
3. Sprint layout аналог для linux (https://a174.ru/sprint-layout-analog-dlya-linux/)