

Лабораторная работа №13

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

Латыпова Диана. НФИбд-02-21

Содержание

1	Цель работы	5
2	Задание	6
3	Выполнение лабораторной работы	7
4	Контрольные вопросы	21
5	Выводы	26

Список иллюстраций

3.1	Создание каталога и файлов	7
3.2	Emacs	7
3.3	Код файла calculate.c	10
3.4	Код файла calculate.h	11
3.5	Код файла main.c	12
3.6	Компиляция программы	12
3.7	ls	13
3.8	Код файла Makefile	14
3.9	Исправленный код Makefile	15
3.10	Отладка программы	16
3.11	Запуск программы внутри отладчика	16
3.12	list	17
3.13	Просмотр определённых строк не основного файла	17
3.14	Точка останова на 21 строке	17
3.15	Точка останова на 16 строке	18
3.16	Проверка точки останова	18
3.17	Numerical	19
3.18	Удаление точек останова	19
3.19	Анализ кода файла calculate.c	20
3.20	Анализ кода файла main.c	20

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

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`.
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile` с содержанием...
6. С помощью `gdb` выполните отладку программы `calcul` (перед использованием `gdb` исправьте `Makefile`).
7. С помощью утилиты `slint` попробуйте проанализировать коды файлов `calculate.c` и `main.c`.

3 Выполнение лабораторной работы

Я создала в домашнем каталоге каталог lab_prog(рис. 3.1):

mkdir ~/work/os/lab_prog

В созданном каталоге создала еще 3 файла: calculate.h, calculate.c, main.c(рис. 3.1):

1 touch calculate.h

2 touch calculate.c

3 touch main.c

```
[dlatihpova@fedora ~]$ cd ~/work/os/lab_prog
[dlatihpova@fedora lab_prog]$ touch calculate.h
[dlatihpova@fedora lab_prog]$ touch calculate.c
[dlatihpova@fedora lab_prog]$ touch main.c
```

Рис. 3.1: Создание каталога и файлов

После чего открыла каждый файл в редакторе emacs и вписала содержимое(рис. 3.2):

```
[dlatihpova@fedora lab_prog]$ emacs calculate.h
[dlatihpova@fedora lab_prog]$ emacs calculate.c
[dlatihpova@fedora lab_prog]$ emacs main.c
```

Рис. 3.2: Emacs

- Листинг файла calculate.c(рис. 3.3):

```
////////////////////////////////////
// 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);
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);
    }
}

```

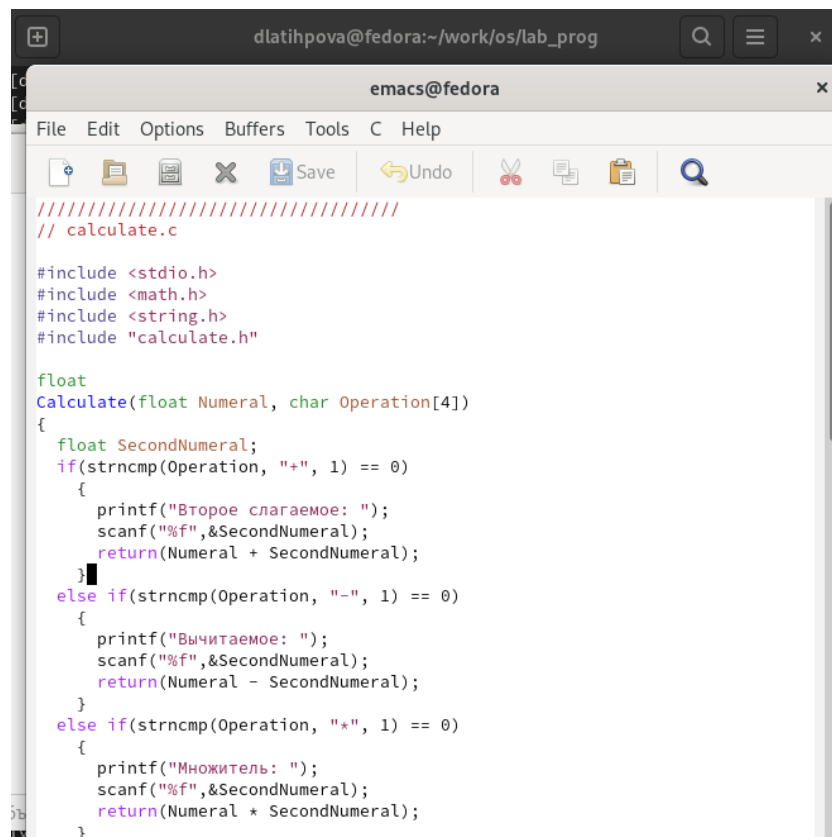


Рис. 3.3: Код файла calculate.c

- Листинг файла calculate.h(рис. 3.4):

```

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

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

```

```
#endif /*CALCULATE_H_*/
```

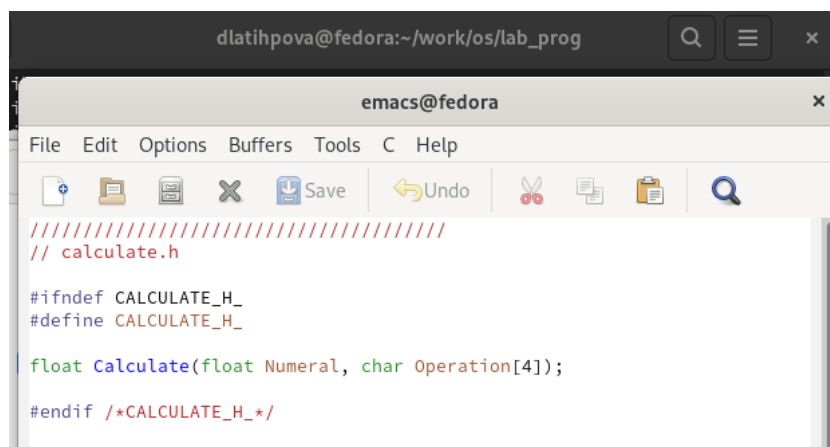


Рис. 3.4: Код файла calculate.h

- Листинг файла main.c(рис. 3.5):

```
////////////////////////////////////  
// 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("%.2f\n",Result);

return 0;

}

```

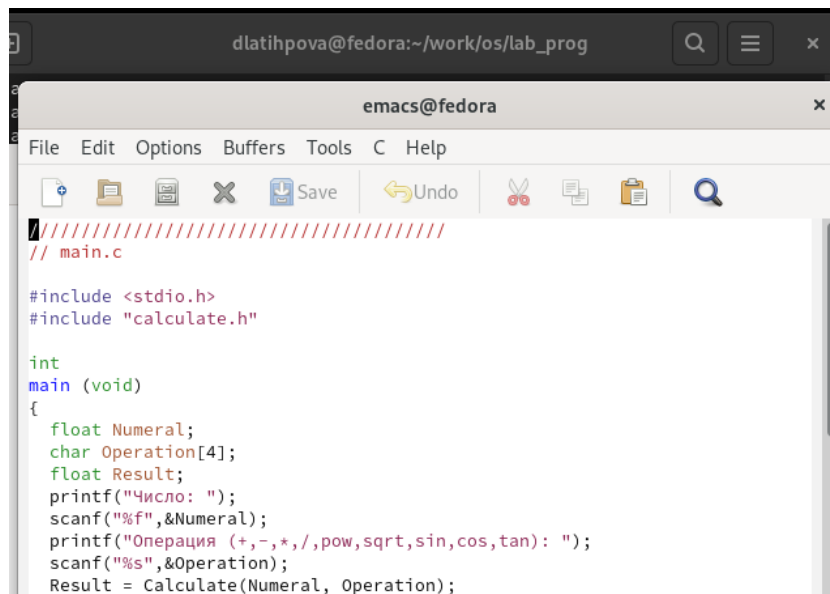


Рис. 3.5: Код файла main.c

Далее я выполнила компиляцию программы посредством gcc(рис. 3.6):

- 1 **gcc -c calculate.c**
- 2 **gcc -c -g main.c**
- 3 **gcc calculate.o main.o -o calcul -lm**

```

[dlatihpova@fedora lab_prog]$ gcc -c calculate.c
[dlatihpova@fedora lab_prog]$ gcc -c -g main.c
[dlatihpova@fedora lab_prog]$ gcc calculate.o main.o -o calcul -lm

```

Рис. 3.6: Компиляция программы

И просмотрели, что созданся исполняемый файл calcul(рис. 3.7):

```
[dlatihpova@fedora lab_prog]$ ls
calcul      calculate.c~ calculate.h~  main.c  main.o
calculate.c calculate.h  calculate.o  main.c~ Makefile~
```

Рис. 3.7: ls

Синтаксических ошибок не оказалось, поэтому перешла к следующему заданию.

С помощью touch создала Makefile и вписала следующее содержимое(рис. 3.8):

```
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 *~
```

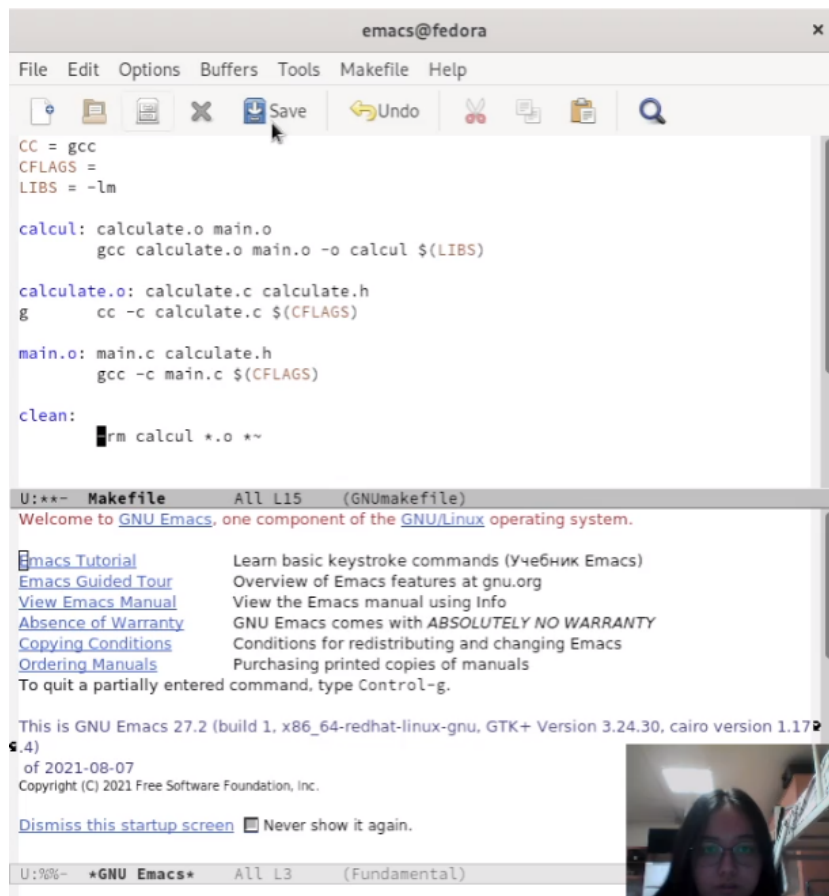


Рис. 3.8: Код файла Makefile

Перед использованием gdb исправила Makefile(рис. 3.9):

```
CC = gcc
```

```
CFLAGS = -g
```

```
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
```

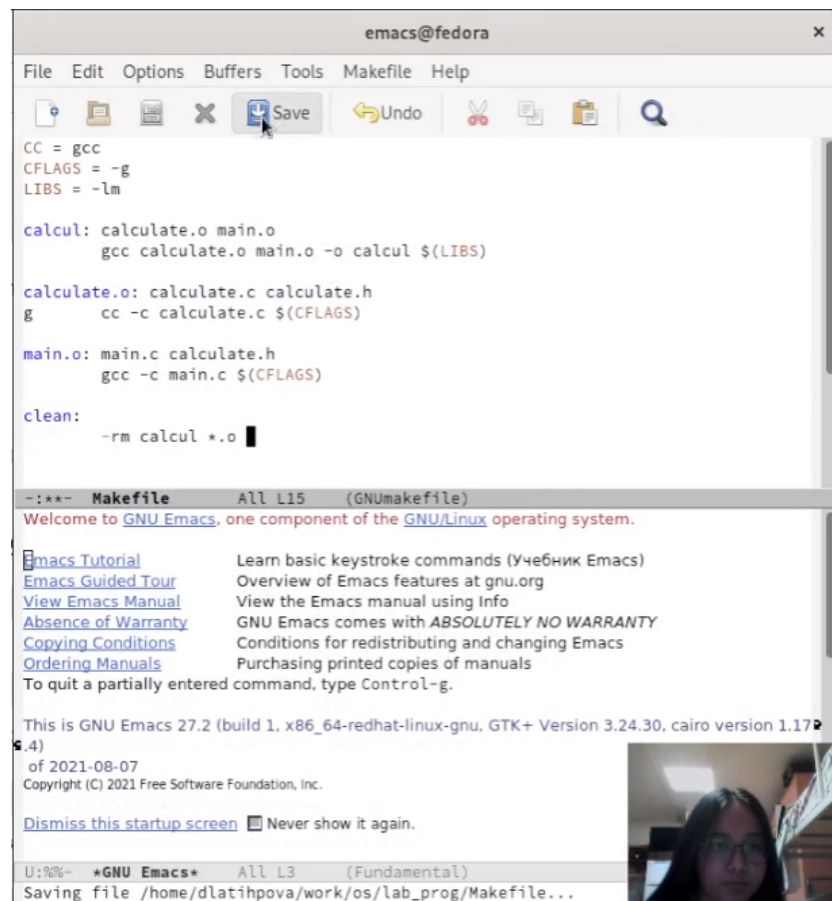


Рис. 3.9: Исправленный код Makefile

С помощью gdb выполнила отладку программы calcul(рис. 3.10):
`gdb ./calcul`

```
[dlatihpova@fedora lab_prog]$ gdb ./calcul
GNU gdb (GDB) Fedora 12.1-1.fc35
Copyright (C) 2022 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...
```

Рис. 3.10: Отладка программы

Запустила внутри отладчика программу(рис. 3.11):

run

```
(gdb) run
Starting program: /home/dlatihpova/work/os/lab_prog/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.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 9
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 3
27.00
```

Рис. 3.11: Запуск программы внутри отладчика

Сначала постранично просмотрела код, затем с 12 по 15 строку(рис. 3.12):

1 list

2 list 12,15


```

(gdb) list
1      ///////////////////////////////////////////////////
2      // main.c
3
4      #include <stdio.h>
5      #include "calculate.h"
6
7      int
8      main (void)
9      {
10         float Numeral;
(gdb) list 12,15
12         float Result;
13         printf("Число: ");
14         scanf("%f",&Numeral);
15         printf("Операция (+,-,*,/,pow,sqrt,sin,cos,tan): ");

```

Рис. 3.12: list

Затем попробовала посмотреть определённые строки не основного файла. Но не вышло(рис. 3.13):

list calculate.c:20,29

```

(gdb) list calculate.c:20,29
No source file named calculate.c.

```

Рис. 3.13: Просмотр определённых строк не основного файла

Попробовала поставить точку останова на строке номер 21, однако такой строки нет(рис. 3.14):

break 21

```

(gdb) break 21
No line 21 in the current file.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (21) pending.

```

Рис. 3.14: Точка останова на 21 строке

Поэтому поставила точку останова на 16 строке(рис. 3.15):

break 16

```
(gdb) break 16
Breakpoint 2 at 0x4014c2: file main.c, line 16.
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
1        breakpoint      keep y   <PENDING>                21
2        breakpoint      keep y   0x00000000004014c2 in main at main.c:16
```

Рис. 3.15: Точка останова на 16 строке

Запустила программу внутри отладчика и убедилась, что программа останавливается в момент прохождения точки останова. Команда `backtrace` показала нам весь стек вызываемых функций от начала программы до текущего места(рис.

3.16):

1 **run**

2 5

3 **backtrace**

```
(gdb) run
Starting program: /home/dlatihpova/work/os/lab_prog/calcul
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
Число: 5

Breakpoint 2, main () at main.c:16
16      scanf("%s",&operation);
(gdb) backtrace
#0  main () at main.c:16
```

Рис. 3.16: Проверка точки останова

Далее посмотрела, чему равно на этом этапе значение переменной `Numeral` и сравнила с результатом вывода на экран после использования команды(рис.

3.17):

1 **print Numeral**

2 **display Numeral**

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

Рис. 3.17: Numeral

Удалила точки останова(рис. 3.18):

1 **info breakpoints**

2 **delete 1**

3 **delete 2**

4 **info breakpoints**

```
(gdb) info breakpoints
Num   Type       Disp Enb Address                What
1      breakpoint keep y   <PENDING>              21
2      breakpoint keep y   0x00000000004014c2 in main at main.c:16
      breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info breakpoints
No breakpoints or watchpoints.
```

Рис. 3.18: Удаление точек останова

И наконец, с помощью утилиты splint проанализировала коды файлов calculate.c и main.c(рис. 3.19)(рис. 3.20):

1 **splint calculate.c**

2 **splint main.c**

```
[dlatihpova@fedora lab_prog]$ splint calculate.c
Splint 3.1.2 --- 23 Jul 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:16:7: 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
```

Рис. 3.19: Анализ кода файла calculate.c

```
[dlatihpova@fedora lab_prog]$ splint main.c
Splint 3.1.2 --- 23 Jul 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)
main.c: (in function main)
main.c:14:3: 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:16:14: 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:16:11: Corresponding format code
main.c:16:3: Return value (type int) ignored: scanf("%s", &Ope...

Finished checking --- 4 code warnings
[dlatihpova@fedora lab_prog]$
```

Рис. 3.20: Анализ кода файла main.c

4 Контрольные вопросы

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

Дополнительную информацию о этих программах можно получить с помощью функций `info` и `man`.

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

- 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 суффикс в контексте языка программирования? Приведите примеры использования.

Суффикс это составная часть имени файла. Система сборки каких-либо программ (например язык java) требует, чтобы имена файлов исходного кода заканчивались на .java.

Компиляторы C и компилятор C++ одинаково относятся к суффиксам, но каждый раз давать файлам заголовков имена с .h (расширение) настолько общепринято, что надоедает. Есть недостаток строгих правил, это к примеру несколько стилей именования файлов реализации в языке C++, например стандартные суффиксы .C, .crr, .cxx, .c++, и .cc. Иногда встречаются файлы заголовков C++ с суффиксом .hrr.

Главное — это соблюдать единообразие при выборе суффикса.

4. Каково основное назначение компилятора языка C в UNIX?

Основное назначение компиляторов – служить для разработки новых прикладных и системных программ с помощью языков высокого уровня.

5. Для чего предназначена утилита make?

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

6. Приведите пример структуры Makefile. 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.

7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?

Отладка программы — это процесс устранения ошибок из текста программы. Все ошибки делятся на синтаксические и логические. При наличии синтаксических ошибок (ошибок в написании операторов) программа не запускается. Подобные ошибки исправляются проще всего. Логические ошибки — это ошибки, при которых программа работает, но неправильно. В этом случае программа выдаёт не те результаты, которые ожидает разработчик или пользователь.

8. Назовите и дайте основную характеристику основным командам отладчика gdb.

Основные команды gdb:

- break или b - создание точки останова;
- info или i - вывести информацию, доступные значения: break, registers, frame, locals, args;

- run или r - запустить программу;
- continue или c - продолжить выполнение программы после точки останова;
- step или s - выполнить следующую строчку программы с заходом в функцию;
- next или n - выполнить следующую строчку без захода в функцию;
- print или p - вывести значение переменной;
- backtrace или bt - вывести стек вызовов;
- x - просмотр содержимого памяти по адресу;
- ptype - просмотр типа переменной;
- h или help - просмотр справки по команде;
- q или quit - выход из программы.

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

- Выполнили компиляцию программы
- Увидели ошибки в программе
- Открыли редактор и исправили программу
- Загрузили программу в отладчик gdb
- run — отладчик выполнил программу, мы ввели требуемые значения.
- программа завершена, gdb не видит ошибок.

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

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 C.

12. Каковы основные задачи, решаемые программой splint?

- Проверка корректности задания аргументов всех использованных в программе функций, а также типов возвращаемых ими значений;
- Поиск фрагментов исходного текста, корректных с точки зрения синтаксиса языка Си, но малоэффективных с точки зрения их реализации или содержащих в себе семантические ошибки;
- Общая оценка мобильности пользовательской программы.

5 Выводы

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.