

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

Отчёт по лабораторной работе №14

Мурашко В.В.

Содержание

1	Теоретическое введение	5
2	Цель работы	6
3	Задание	7
4	Выполнение лабораторной работы	8
5	Выводы	16
6	Библиография	17
7	Контрольные вопросы	18

Список таблиц

Список иллюстраций

4.1	Создание подкаталога	8
4.2	Создание файлов	8
4.3	Реализация функций калькулятора в файле calculate.h	9
4.4	Реализация функций калькулятора в файле calculate.h	10
4.5	Интерфейсный файл calculate.h	10
4.6	Основной файл main.c	11
4.7	Команда gcc	11
4.8	Компиляция	11
4.9	Makefile	12
4.10	Отладчик GDB и run	12
4.11	Команда list	13
4.12	Команда list, точка останова и информация	13
4.13	Run, команда backtrace, Numeral и удаление точки останова	14
4.14	Анализ calculate.c	14
4.15	Анализ main.c	15

1 Теоретическое введение

Интегрированные средства (среды) разработки (IDE) не являются критически необходимым компонентом программной разработки. В традициях UNIX вполне достаточным для ведения программной разработки считается использование текстового редактора, обладающего дополнительными развитыми свойствами, такими как цветовая разметка текста, функции контекстного поиска и замены... Удовлетворяющих таким требованиям редакторов в Linux великое множество, начиная с традиционных `vim` и `Emacs`, и до простого редактирования в `mc` по F4. Опыт использования показывает, что этих средств вполне достаточно вплоть до средних размеров проектов.

Но использование IDE часто позволяет более производительнее вести отработку программного кода, оперативнее выполнять в связке цикл: редактирование кода — сборка проекта — отладка. Значительно возрастает роль IDE в разработке GUI приложений, потому как большинство IDE предполагают в своём составе визуальные построители графических экранов.

Под Linux доступно весьма много разных IDE, различной степени интегрированности. Их уже настолько много, что становится бессмысленным описывать все, или значительную их часть в деталях: использование тех или иных IDE становится, в значительной мере, вопросом субъективных предпочтений и привычек.

2 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.

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.cgcc -c main.cgcc calculate.o main.o -o calcul -lm`
4. При необходимости исправьте синтаксические ошибки.
5. Создайте `Makefile` со следующим содержанием.
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 я создала подкаталог ~/work/os/lab_prog.

```
vvmurashko1@dk8n70 ~ $ mkdir work
vvmurashko1@dk8n70 ~ $ cd work
vvmurashko1@dk8n70 ~/work $ mkdir os
vvmurashko1@dk8n70 ~/work $ cd ~/work/os
vvmurashko1@dk8n70 ~/work/os $ mkdir lab_prog
vvmurashko1@dk8n70 ~/work/os $ cd ~/work/os/lab_prog
vvmurashko1@dk8n70 ~/work/os/lab_prog $ █
```

Рис. 4.1: Создание подкаталога

2. Я создала в нём файлы: calculate.h, calculate.c, main.c.

```
vvmurashko1@dk8n70 ~/work/os/lab_prog $ touch calculate.h
vvmurashko1@dk8n70 ~/work/os/lab_prog $ touch calculate.c
vvmurashko1@dk8n70 ~/work/os/lab_prog $ touch main.c
vvmurashko1@dk8n70 ~/work/os/lab_prog $ ls
calculate.c calculate.h main.c
vvmurashko1@dk8n70 ~/work/os/lab_prog $ █
```

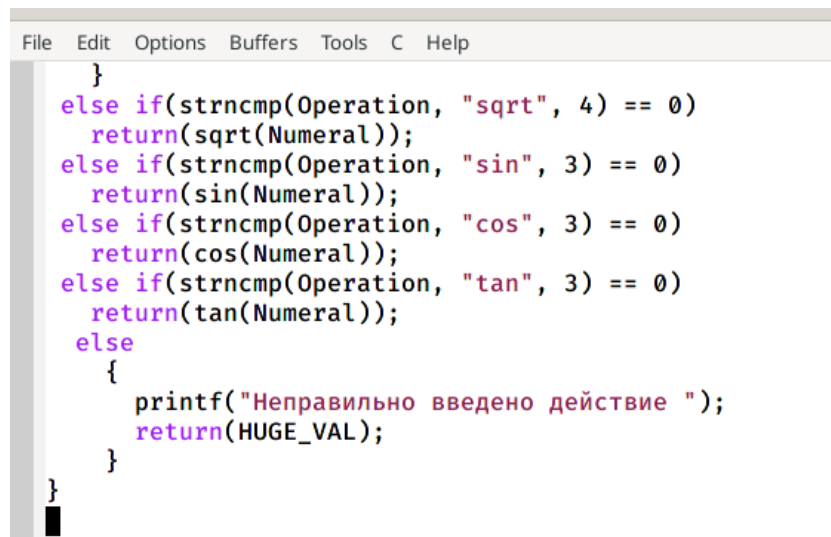
Рис. 4.2: Создание файлов


```

File Edit Options Buffers Tools C Help
#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)

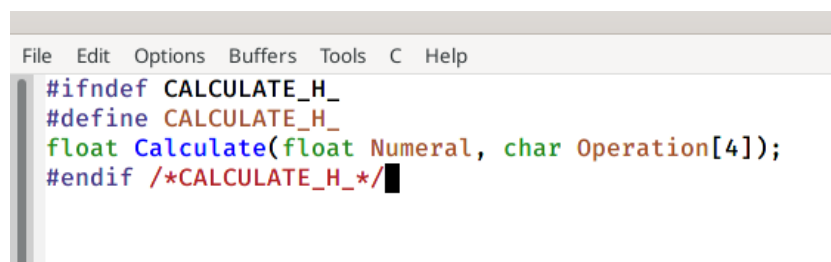
```

Рис. 4.3: Реализация функций калькулятора в файле calculate.h



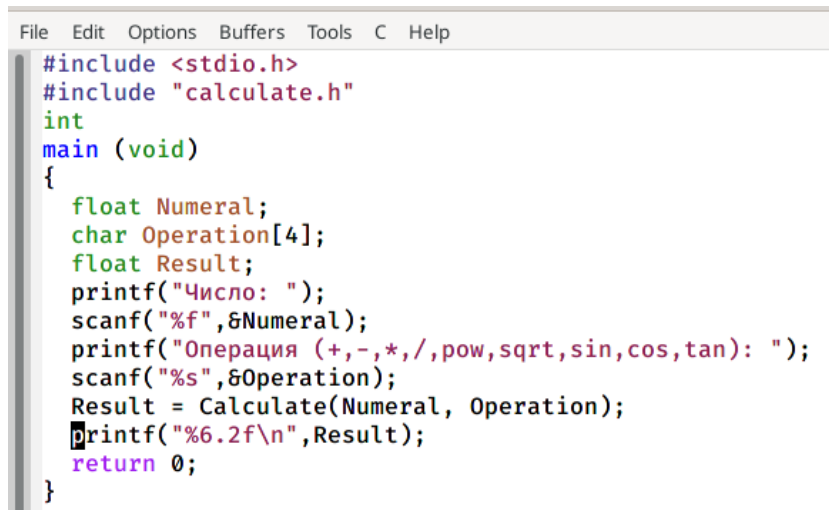
```
File Edit Options Buffers Tools C Help
}
else if(strncmp(Operation, "sqrt", 4) == 0)
    return(sqrt(Numeral));
else if(strncmp(Operation, "sin", 3) == 0)
    return(sin(Numeral));
else if(strncmp(Operation, "cos", 3) == 0)
    return(cos(Numeral));
else if(strncmp(Operation, "tan", 3) == 0)
    return(tan(Numeral));
else
{
    printf("Неправильно введено действие ");
    return(HUGE_VAL);
}
}
```

Рис. 4.4: Реализация функций калькулятора в файле calculate.h



```
File Edit Options Buffers Tools C Help
#ifndef CALCULATE_H_
#define CALCULATE_H_
float Calculate(float Numeral, char Operation[4]);
#endif /*CALCULATE_H_*/
```

Рис. 4.5: Интерфейсный файл calculate.h



```

File Edit Options Buffers Tools C Help
#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. 4.6: Основной файл main.c

3. Я выполнила компиляцию программы посредством gcc: gcc -c calculate.cgcc
-c main.c gcc calculate.o main.o -o calcul -lm

```

vvmurashko1@dk8n70 ~/work/os/lab_prog $ gcc -c calculate.c
vvmurashko1@dk8n70 ~/work/os/lab_prog $ gcc -c main.c

```

Рис. 4.7: Команда gcc

```

vvmurashko1@dk8n70 ~/work/os/lab_prog $ gcc calculate.o main.o -o calcul -lm

```

Рис. 4.8: Компиляция

4. Я исправила синтаксические ошибки.
5. Я создала Makefile со следующим содержанием.

```

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

```

Рис. 4.9: Makefile

В содержании файла указаны флаги компиляции, тип компилятора и файлы, которые должен собрать сборщик.

6. С помощью gdb я выполнила отладку программы calcul (перед использованием gdb исправила Makefile).

```

vvmurashko1@dk8n70 ~/work/os/lab_prog $ gdb ./calcul
GNU gdb (Gentoo 10.1 vanilla) 10.1
Copyright (C) 2020 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-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://bugs.gentoo.org/>.
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...
(No debugging symbols found in ./calcul)
(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/v/v/vvmurashko1/work/os/lab_prog/calcul
Число: 3
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): *
Множитель: 4
12.00
[Inferior 1 (process 19068) exited normally]
(gdb) █

```

Рис. 4.10: Отладчик GDB и run

```

(gdb) list
1      #include <stdio.h>
2      #include "calculate.h"
3      int
4      main (void)
5      {
6          float Numeral;
7          char Operation[4];
8          float Result;
9          printf("Число: ");
10         scanf("%f",&Numeral);
(gdb) list 12,15
12         scanf("%s",&Operation);
13         Result = Calculate(Numeral, Operation);
14         printf("%6.2f\n",Result);
15         return 0;

```

Рис. 4.11: Команда list

```

...
(gdb) list calculate.c:20,29
20     }
21     else if(strncmp(Operation, "*", 1) == 0)
22     {
23         printf("Множитель: ");
24         scanf("%f",&SecondNumeral);
25         return(Numeral * SecondNumeral);
26     }
27     else if(strncmp(Operation, "/", 1) == 0)
28     {
29         printf("Делитель: ");
(gdb) list calculate.c:20,27
20     }
21     else if(strncmp(Operation, "*", 1) == 0)
22     {
23         printf("Множитель: ");
24         scanf("%f",&SecondNumeral);
25         return(Numeral * SecondNumeral);
26     }
27     else if(strncmp(Operation, "/", 1) == 0)
(gdb) break 21
Breakpoint 1 at 0x991: file calculate.c, line 21.
(gdb) info breakpoints
Num   Type             Disp Enb Address            What
1     breakpoint       keep y   0x0000000000000991 in Calculate at calculate.c:21
(gdb) █

```

Рис. 4.12: Команда list, точка останова и информация

```

(gdb) run
Starting program: /afs/.dk.sci.pfu.edu.ru/home/v/vvmurashko1/work/os/lab_prog/calcul
Число: 7
Операция (+,-,*,/,pow,sqrt,sin,cos,tan): pow

Breakpoint 1, Calculate (Numeral=7, Operation=0x7fffffff14 "pow") at calculate.c:21
21     else if(strcmp(Operation, "*") == 0)
(gdb) backtrace
#0 Calculate (Numeral=7, Operation=0x7fffffff14 "pow") at calculate.c:21
#1 0x000055555400c31 in main () at main.c:13
(gdb) print Numeral
$1 = 7
(gdb) display Numeral
1: Numeral = 7
(gdb) info breakpoints
Num  Type      Disp Enb Address      What
1     breakpoint keep y  0x000055555400991 in Calculate at calculate.c:21
      breakpoint already hit 1 time
(gdb) delete 1

```

Рис. 4.13: Run, команда backtrace, Numeral и удаление точки останова

7. С помощью утилиты splint попробуйте проанализировать коды файлов calculate.c и main.c.

```

vvmurashko1@dk8n70 ~/work/os/lab_prog $ splint calculate.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:3:37: 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:6:31: Function parameter Operation declared as manifest array (size
      constant is meaningless)
calculate.c: (in function Calculate)
calculate.c:12:7: 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:18:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:24:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:30:7: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:31:10: 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:34:10: Return value type double does not match declared type float:
      (HUGE_VAL)
  To allow all numeric types to match, use +relaxtypes.
calculate.c:42:6: Return value (type int) ignored: scanf("%f", &Sec...
calculate.c:43:10: Return value type double does not match declared type float:

```

Рис. 4.14: Анализ calculate.c

```

vvmurashko1@edk8n70 ~/work/os/lab_prog $ splint main.c
Splint 3.1.2 --- 13 Jan 2021

calculate.h:3:37: 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:10:3: 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:12:14: 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:12:11: Corresponding format code
main.c:12:3: Return value (type int) ignored: scanf("%s", &Op...

Finished checking --- 4 code warnings _

```

Рис. 4.15: Анализ main.c

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.

6 Библиография

http://rus-linux.net/MyLDP/BOOKS/Linux-tools/IDE_01.html

7 Контрольные вопросы

1. Информацию об этих программах можно получить с помощью функций `info` и `man`.
2. 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. 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 заключается в компиляции всей программы в целом и получении исполняемого модуля.
5. 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. 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. Пример можно найти в задании 5.

7. 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. 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. 1) Выполнила компиляцию программы 2) Увидела ошибки в программе
 3) Открыла редактор и исправила программу 4) Загрузила программу в отладчик `gdb` 5) `run` — отладчик выполнил программу, ввела требуемые значения. 6) Использовала другие команды отладчика и проверила работу программы
10. Отладчику не понравился формат `%s` для `&Operation`, т.к `%s` — символьный формат, а значит необходим только `Operation`.
11. 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. 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.