Лабораторная работа № 13. Средства, применяемые при разработке программного обеспечения в ОС типа UNIX/Linux

13.1. 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.

13.2. Указания к лабораторной работе

13.2.1. Этапы разработки приложений

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.

13.2.2. Компиляция исходного текста и построение исполняемого файла

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.

Для компиляции файла `main.c`, содержащего написанную на языке C простейшую программу:

```
1  /*
2   * main.c
3   */
4  #include <stdio.h>
5  int main()
6  {
```

```
  printf("Hello World!\n");
  return 0;
}
```

достаточно в командной строке ввести:

```
gcc -c main.c
```

Таким образом, 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
```

Описание некоторых опций gcc приведено в табл. 13.1.

Таблица 13.1

Некоторые опции компиляции в gcc

Опция	Описание
-c	компиляция без компоновки — создаются объектные файлы file.o
-o file-name	задать имя file-name создаваемому файлу
-g	поместить в файл (объектный или исполняемый) отладочную информацию для отладчика gdb
-MM	вывести зависимости от заголовочных файлов C и/или C++ программ в формате, подходящем для утилиты make; при этом объектные или исполняемые файлы не будет созданы
-Wall	вывод на экран сообщений об ошибках, возникших во время компиляции

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.

В самом простом случае Makefile имеет следующий синтаксис:

```
<цель_1> <цель_2> ... : <зависимость_1> <зависимость_2> ...
        <команда 1>
        ...
        <команда n>
```

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.

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 *~

# End Makefile for abcd.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.

13.2.3. Тестирование и отладка

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
```

Затем можно использовать по мере необходимости различные команды gdb. Наиболее часто используемые команды `gdb` приведены в табл. 13.2.

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`.

13.2.4. Анализ исходного текста программы

Ещё одним средством проверки исходных кодов программ, написанных на языке C, 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 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.

Рассмотрим следующий небольшой пример:

```
float sum(float x, float y){
  return x + y;
}

int main(){
  int x, y;
  float z;
  x = 10;
  y = 12;
  z = x + y + sum(x, y);
  return z;
}
```

Сохраните код данного примера в файл `example.c`. Для анализа кода программы следует выполнить следующую команду:

Таблица 13.2

Некоторые команды gdb

Команда	Описание действия
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

```
splint example.c
```

В результате на экран будут выведены следующие пять предупреждений

```

Splint 3.1.2 --- 03 May 2009

example.c: (in function main)
example.c:10:18: Function sum expects arg 1 to be float gets int: x
  To allow all numeric types to match, use +relaxtypes.
example.c:10:21: Function sum expects arg 2 to be float gets int: y
example.c:10:2: Assignment of int to float: z = x + y + sum(x, y)
example.c:11:9: Return value type float does not match declared
  type int: z

Finished checking --- 4 code warnings
```

Первые два предупреждения относятся к функции `sum`, которая определена для двух аргументов вещественного типа, но при вызове ей передаются два аргумента `x` и `y` целого типа. Третье предупреждение относится к присвоению вещественной переменной `z` значения целого типа, которое получается в результате суммирования `x + y + sum(x, y)`. Далее вещественное число `z` возвращается в качестве результата работы функции `main`, хотя данная функция объявлена как функция, возвращающая целое значение.

В качестве упражнения попробуйте скомпилировать программу компилятором `gcc` и сравнить выдаваемые им предупреждения с приведёнными выше.

13.3. Последовательность выполнения работы

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.
   Реализация функций калькулятора в файле `calculate.h`:

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

Интерфейсный файл `calculate.h`, описывающий формат вызова функции-калькулятора:

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

#ifndef CALCULATE_H_
#define CALCULATE_H_

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

#endif /*CALCULATE_H_*/
```

Основной файл `main.c`, реализующий интерфейс пользователя к калькулятору:

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

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:

```
#
# 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
```

Поясните в отчёте его содержание.

6. С помощью gdb выполните отладку программы calcul (перед использованием gdb исправьте `Makefile`):
   - Запустите отладчик GDB, загрузив в него программу для отладки:

```
gdb ./calcul
```

– Для запуска программы внутри отладчика введите команду `run`:

```
run
```

– Для постраничного (по 9 строк) просмотра исходного код используйте команду `list`:

```
list
```

– Для просмотра строк с 12 по 15 основного файла используйте `list` с параметрами:

```
list 12,15
```

– Для просмотра определённых строк не основного файла используйте `list` с параметрами:

```
list calculate.c:20,29
```

– Установите точку останова в файле `calculate.c` на строке номер 21:

```
list calculate.c:20,27
break 21
```

– Выведите информацию об имеющихся в проекте точка останова:

```
info breakpoints
```

– Запустите программу внутри отладчика и убедитесь, что программа остановится в момент прохождения точки останова:

```
run
5
-
backtrace
```

– Отладчик выдаст следующую информацию:

```
#0  Calculate (Numeral=5, Operation=0x7fffffffd280 "-")
  at calculate.c:21
#1  0x0000000000400b2b in main () at main.c:17
```

а команда `backtrace` покажет весь стек вызываемых функций от начала программы до текущего места.

– Посмотрите, чему равно на этом этапе значение переменной `Numeral`, введя:

```
print Numeral
```

На экран должно быть выведено число 5.

- Сравните с результатом вывода на экран после использования команды:

```
display Numeral
```

- Уберите точки останова:

```
info breakpoints
delete 1
```

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

13.4. Содержание отчёта

1. Титульный лист с указанием номера лабораторной работы и ФИО студента.
2. Формулировка цели работы.
3. Описание результатов выполнения задания:
   - скриншоты (снимки экрана), фиксирующие выполнение лабораторной работы;
   - листинги (исходный код) программ (если они есть);
   - результаты выполнения программ (текст или снимок экрана в зависимости от задания).
4. Выводы, согласованные с целью работы.
5. Ответы на контрольные вопросы.

13.5. Контрольные вопросы

1. Как получить информацию о возможностях программ gcc, make, gdb и др.?
2. Назовите и дайте краткую характеристику основным этапам разработки приложений в UNIX.
3. Что такое суффикс в контексте языка программирования? Приведите примеры использования.
4. Каково основное назначение компилятора языка C в UNIX?
5. Для чего предназначена утилита make?
6. Приведите пример структуры `Makefile`. Дайте характеристику основным элементам этого файла.
7. Назовите основное свойство, присущее всем программам отладки. Что необходимо сделать, чтобы его можно было использовать?
8. Назовите и дайте основную характеристику основным командам отладчика `gdb`.
9. Опишите по шагам схему отладки программы, которую Вы использовали при выполнении лабораторной работы.
10. Прокомментируйте реакцию компилятора на синтаксические ошибки в программе при его первом запуске.
11. Назовите основные средства, повышающие понимание исходного кода программы.
12. Каковы основные задачи, решаемые программой `splint`?