

# **Лабораторная работа №9**

**Понятие подпрограммы. Отладчик GDB.**

Хорошева Алёна Евгеньевна

# Содержание

<b>1</b>	<b>Цель работы</b>	<b>5</b>
<b>2</b>	<b>Задание</b>	<b>6</b>
<b>3</b>	<b>Теоретическое введение</b>	<b>7</b>
<b>4</b>	<b>Выполнение лабораторной работы</b>	<b>15</b>
<b>5</b>	<b>Задание для самостоятельной работы</b>	<b>26</b>
	<b>Список литературы</b>	<b>32</b>

## **Список иллюстраций**

## **Список таблиц**

# 1 Цель работы

Освоить работы с отладчиком GDB для исправления ошибок в программе. Приобрести практические навыки написания и использования подпрограмм внутри программ.

## 2 Задание

**\*\*Задание для самостоятельной работы\*\***

1. Преобразуйте программу из лабораторной работы №8 (Задание №1 для самостоятельной работы), реализовав вычисление значения функции  $f(x)$  как подпрограмму.
2. В листинге 9.3 приведена программа вычисления выражения  $(3 + 2) \cdot 4 + 5$ . При запуске данная программа дает неверный результат. Проверьте это. С помощью отладчика GDB, анализируя изменения значений регистров, определите ошибку и исправьте ее.

## 3 Теоретическое введение

### 1. Понятие об отладке

*Отладка* — это процесс поиска и исправления ошибок в программе. В общем случае его можно разделить на четыре этапа:

- обнаружение ошибки;
- поиск её местонахождения;
- определение причины ошибки;
- исправление ошибки.

Можно выделить следующие типы ошибок:

- *синтаксические ошибки* — обнаруживаются во время трансляции исходного кода и вызваны нарушением ожидаемой формы или структуры языка;
- *семантические ошибки* — являются логическими и приводят к тому, что программа запускается, отработывает, но не даёт желаемого результата;

- *ошибки в процессе выполнения* — не обнаруживаются при трансляции и вызывают прерывание выполнения программы (например, это ошибки, связанные с переполнением или делением на ноль).

Второй этап — *поиск местонахождения ошибки*. Некоторые ошибки обнаружить довольно трудно. Лучший способ найти место в программе, где находится ошибка, это разбить программу на части и произвести их отладку отдельно друг от друга.

Третий этап — *выяснение причины ошибки*. После определения местонахождения ошибки обычно проще определить причину неправильной работы программы.

Последний этап — *исправление ошибки*. После этого при повторном запуске программы, может обнаружиться следующая ошибка, и процесс отладки начнётся заново.

## 2. Методы отладки

Наиболее часто применяют следующие методы отладки:

- создание точек контроля значений на входе и выходе участка программы (например, вывод промежуточных значений на экран — так называемые *диагностические сообщения*);
- использование специальных программ-отладчиков.

Отладчики позволяют управлять ходом выполнения программы, контролировать и изменять да

• *Пошаговое выполнение* — это выполнение программы с остановкой после каждой строчки, ч

• *Точки останова* — это специально отмеченные места в программе, в которых программа-

отладчик приостанавливает выполнение программы и ждёт команд. Наиболее популярные виды

- Breakpoint — точка останова (остановка происходит, когда выполнение доходит до определённой строки, адреса или процедуры, отмеченной программистом);



- Watchpoint — точка просмотра (выполнение программы приостанавливается, если программа обратилась к определённой переменной: либо считала её значение, либо изменила его).

Точки останова устанавливаются в отладчике на время сеанса работы с кодом программы, т.е. они сохраняются до выхода из программы-отладчика или до смены отлаживаемой программы.

### 3. Основные возможности отладчика GDB

GDB (GNU Debugger — отладчик проекта GNU) работает на многих UNIX-подобных системах и умеет производить отладку многих языков программирования. GDB предлагает обширные средства для слежения и контроля за выполнением компьютерных программ. Отладчик не содержит собственного графического пользовательского интерфейса и использует стандартный текстовый интерфейс консоли. Однако для GDB существует несколько сторонних графических надстроек, а кроме того, некоторые интегрированные среды разработки используют его в качестве базовой подсистемы отладки.

Отладчик GDB (как и любой другой отладчик) позволяет увидеть, что происходит «внутри» программы в момент её выполнения или что делает программа в момент сбоя.

GDB может выполнять следующие действия:

- начать выполнение программы, задав всё, что может повлиять на её поведение;
- остановить программу при указанных условиях;
- исследовать, что случилось, когда программа остановилась;

- изменить программу так, чтобы можно было поэкспериментировать с устранением эффектов одной ошибки и продолжить выявление других.

#### 4. Запуск отладчика GDB; выполнение программы; выход

Синтаксис команды для запуска отладчика имеет следующий вид:

```
gdb [опции] [имя_файла | ID процесса]
```

После запуска gdb выводит текстовое сообщение — так называемое «nice GDB logo». В следующей строке появляется приглашение (gdb) для ввода команд.

Далее приведён список некоторых команд GDB.

Команда run (сокращённо r) — запускает отлаживаемую программу в оболочке GDB. Если точки останова не были установлены, то программа выполняется и выводятся сообщения:

```
(gdb) run
```

```
Starting program: test
```

```
Program exited normally.
```

```
(gdb)
```

Если точки останова были заданы, то отладчик останавливается на соответствующей команде и выдаёт номер точки останова, адрес и дополнительную информацию — текущую строку, имя процедуры, и др.

Команда kill (сокращённо k) прекращает отладку программы, после чего следует вопрос о прекращении процесса отладки:

```
Kill the program being debugged? (y or n) y
```

Если в ответ введено y (то есть «да»), отладка программы прекращается.

Командой run её можно начать заново, при этом все точки останова (breakpoints), точки просмотра (watchpoints) и точки отлова (catchpoints) сохраняются.

Для выхода из отладчика используется команда quit (или сокращённо q):

```
(gdb) q
```

## 5. Дизассемблирование программы

Если есть файл с исходным текстом программы, а в исполняемый файл включена информация о номерах строк исходного кода, то программу можно отлаживать, работая в отладчике непосредственно с её исходным текстом. Чтобы программу можно было отлаживать на уровне строк исходного кода, она должна быть откомпилирована с ключом -g.

Посмотреть дизассемблированный код программы можно с помощью команды `disassemble` :

```
(gdb) disassemble _start
```

Существует два режима отображения синтаксиса машинных команд: режим Intel, используемый в том числе в NASM, и режим ATT (значительно отличающийся внешне). По умолчанию в дизассемблере GDB принят режим ATT. Переключиться на отображение команд с привычным Intel'овским синтаксисом можно, введя команду `set disassembly-flavor intel`.

## 6. Точки останова

Установить точку останова можно командой `break` (кратко `b`). Типичный аргумент этой команды — место установки. Его можно задать как имя метки или как адрес. Чтобы не было путаницы с номерами, перед адресом ставится «звёздочка»:

```
(gdb) break *
```

```
(gdb) b
```

Информацию о всех установленных точках останова можно вывести командой `info` (кратко `i`):

```
(gdb) info breakpoints
```

```
(gdb) i b
```

Для того чтобы сделать неактивной какую-нибудь ненужную точку остано-

ва, можно воспользоваться командой `disable`:

```
disable breakpoint
```

Обратно точка останова активируется командой `enable`:

```
enable breakpoint
```

Если же точка останова в дальнейшем больше не нужна, она может быть удалена с помощью команды `delete`:

```
(gdb) delete breakpoint
```

Ввод этой команды без аргумента удалит все точки останова.

Информацию о командах этого раздела можно получить, введя `help breakpoints`.

## 7. Пошаговая отладка

Для продолжения остановленной программы используется команда `continue (c) (gdb) c [аргумент]`. Выполнение программы будет происходить до следующей точки останова.

В качестве аргумента может использоваться целое число `N`, которое указывает отладчику проигнорировать `N - 1` точку останова (выполнение остановится на `N`-й точке).

Команда `stepi` (кратко `si`) позволяет выполнять программу по шагам, т.е. данная команда выполняет ровно одну инструкцию:

```
(gdb) si [аргумент]
```

При указании в качестве аргумента целого числа `N` отладчик выполнит команду `step` `N` раз при условии, что не будет точек останова или выполнение программы не прервётся по другим причинам.

Команда `nexthi` (или `ni`) аналогична `stepi`, но вызов процедуры (функции) трактуется отладчиком как одна инструкция:

```
(gdb) ni [аргумент]
```

Информацию о командах этого раздела можно получить, введя `(gdb) help running`.

## 8. Работа с данными программы в GDB

Как уже упоминалось, отладчик может показывать содержимое ячеек памяти и регистров, а при необходимости позволяет вручную изменять значения регистров и переменных.

Посмотреть содержимое регистров можно с помощью команды `info registers` (или `ir`):

```
(gdb) info registers
```

Для отображения содержимого памяти можно использовать команду `x/NFU`, выдаёт содержимое ячейки памяти по указанному адресу. `NFU` задает формат, в котором выводятся данные.

Например, `x/4uh 0x63450` — это запрос на вывод четырёх полуслов (`h`) из памяти в формате беззнаковых десятичных целых (`u`), начиная с адреса `0x63450`.

Чтобы посмотреть значения регистров используется команда `print /F` (сокращенно `p`). Перед именем регистра обязательно ставится префикс `$`. Например, команда `p/x $ecx` выводит значение регистра в шестнадцатеричном формате. Изменить значение для регистра или ячейки памяти можно с помощью команды `set`, задав ей в качестве аргумента имя регистра или адрес. При этом перед именем регистра ставится префикс `$`, а перед адресом нужно указать в фигурных скобках тип данных (размер сохраняемого значения; в качестве типа данных можно использовать типы языка Си).

Справку о любой команде `gdb` можно получить, введя

```
(gdb) help [имя_команды]
```

## 9. Понятие подпрограммы

*Подпрограмма* — это, как правило, функционально законченный участок

кода, который можно многократно вызывать из разных мест программы. В отличие от простых переходов из подпрограмм существует возврат на команду, следующую за вызовом.

Если в программе встречается одинаковый участок кода, его можно оформить в виде подпрограммы, а во всех нужных местах поставить её вызов. При этом подпрограмма будет содержаться в коде в одном экземпляре, что позволит уменьшить размер кода всей программы.

#### 10. Инструкция **call** и инструкция **ret**

Для вызова подпрограммы из основной программы используется инструкция **call**, которая заносит адрес следующей инструкции в стек и загружает в регистр **еір** адрес соответствующей подпрограммы, осуществляя таким образом переход. Затем начинается выполнение подпрограммы, которая, в свою очередь, также может содержать подпрограммы.

Подпрограмма завершается инструкцией **ret**, которая извлекает из стека адрес, занесённый туда соответствующей инструкцией **call**, и заносит его в **еір**. После этого выполнение основной программы возобновится с инструкции, следующей за инструкцией **call**.

Подпрограмма может вызываться как из внешнего файла, так и быть частью основной программы.

Важно помнить, что если в подпрограмме занести что-то в стек и не извлечь, то на вершине стека окажется не адрес возврата и это приведёт к ошибке выхода из подпрограммы. Кроме того, надо помнить, что подпрограмма без команды возврата не вернётся в точку вызова, а будет выполнять следующий за подпрограммой код, как будто он является её продолжением.

## 4 Выполнение лабораторной работы

1. Создаём файл lab9-1.asm и записываем туда программу из листинга 9.1 для вычисления выражения  $f(x) = 2x+7$  с помощью подпрограммы \_calcul.

Первые строки программы отвечают за вывод сообщения на экран (call sprint), чтение данных введенных с клавиатуры (call sread) и преобразования введенных данных из символьного вида в численный (call atoi).

После следующей инструкции call \_calcul, которая передает управление подпрограмме \_calcul, будут выполнены инструкции подпрограммы:

```
mov ebx,2
mul ebx
add eax,7
mov [res],eax
ret
```

Инструкция ret является последней в подпрограмме и ее исполнение приводит к возвращению в основную программу к инструкции, следующей за инструкцией call, которая вызвала данную подпрограмму.

Последние строки программы реализуют вывод сообщения (call sprint), результата вычисления (call iprintLF) и завершение программы (call quit).

Проверяем работу программы:

```
alyona@aeckhorosheva:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 1
2x+7=9
```

2. Теперь изменим текст программы, добавив в подпрограмму `_calcul` вызов другой подпрограммы `_subcalcul`, которая будет вычислять значение  $g(x) = 3x - 1$ . Таким образом, нам нужно получить программу, которая вычислит значение  $f(g(x))$  для  $x$ , введенного с клавиатуры.

; Подпрограмма вычисления выражения  $g(x) = 3x - 1$

`_subcalcul:`

`mov ebx,3`

`mul ebx`

`dec eax`

`ret`

Вызов данной подпрограммы мы производим в самом начале `_calcul`:

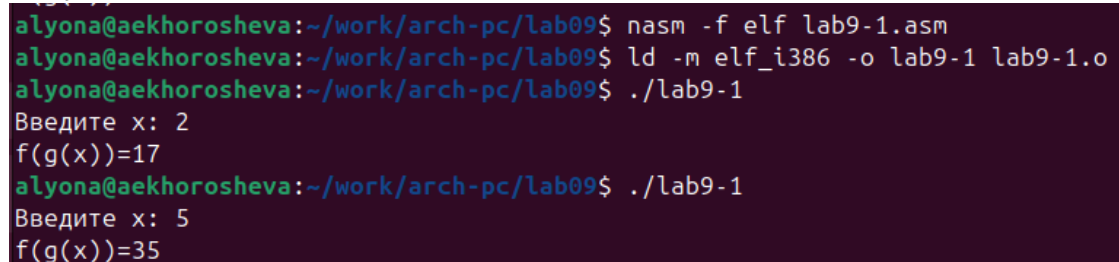
`_calcul:`

`call _subcalcul`

`mov ebx,2`

...

Запускаем исполняемый файл:



```
alyona@aeckhorosheva:~/work/arch-pc/lab09$ nasm -f elf lab9-1.asm
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab9-1 lab9-1.o
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 2
f(g(x))=17
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ./lab9-1
Введите x: 5
f(g(x))=35
```

3. Создадим новый файл `lab09-2.asm` с текстом из листинга 9.2 - это программа печати сообщения "Hello, world!". Будем проверять работы этой программы с помощью отладчика *GDB*. Для этого при создании исполняемого файла и предшествующих этому инструкций, нужно использовать ключ `"-g"`.

`nasm -f elf -g -l lab09-2.lst lab09-2.asm`

`ld -m elf_i386 -o lab09-2 lab09-2.o`

`gdb lab09-2`



Теперь можно запустить программу в gdb для проверки её работы командой `run`. Для более же подробного анализа работы программы поставим брейк-поинт на метку `_start`, т.е. на начало выполнения программы. Запускаем всё той же командой `run` или `r`.

```
(gdb) r
Starting program: /home/alyona/work/arch-pc/lab09/lab9-2

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
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.
Downloading separate debug info for system-supplied DSO at 0xf7ffc000
Hello, world!
[Inferior 1 (process 8621) exited normally]
(gdb) b _start
Breakpoint 1 at 0x8049000: file lab9-2.asm, line 12.
(gdb) r
Starting program: /home/alyona/work/arch-pc/lab09/lab9-2

Breakpoint 1, _start () at lab9-2.asm:12
12      mov     eax, 4
(gdb) █
```

4. Посмотрим *дизассемблированный код* программы(т.е. *отображение инструкций, которые создал компилятор*) с помощью команды `disassemble _start`(начинаем с метки `_start`).

```
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     $0x4,%eax
    0x08049005 <+5>:      mov     $0x1,%ebx
    0x0804900a <+10>:     mov     $0x804a000,%ecx
    0x0804900f <+15>:     mov     $0x8,%edx
    0x08049014 <+20>:     int     $0x80
    0x08049016 <+22>:     mov     $0x4,%eax
    0x0804901b <+27>:     mov     $0x1,%ebx
    0x08049020 <+32>:     mov     $0x804a008,%ecx
    0x08049025 <+37>:     mov     $0x7,%edx
    0x0804902a <+42>:     int     $0x80
    0x0804902c <+44>:     mov     $0x1,%eax
    0x08049031 <+49>:     mov     $0x0,%ebx
    0x08049036 <+54>:     int     $0x80
```

Теперь изменим отображение команд на синтаксис Intel с помощью команды `set disassembly-flavor intel`.

```
(gdb) set disassembly-flavor intel
(gdb) disassemble _start
Dump of assembler code for function _start:
=> 0x08049000 <+0>:      mov     eax,0x4
    0x08049005 <+5>:      mov     ebx,0x1
    0x0804900a <+10>:     mov     ecx,0x804a000
    0x0804900f <+15>:     mov     edx,0x8
    0x08049014 <+20>:     int     0x80
    0x08049016 <+22>:     mov     eax,0x4
    0x0804901b <+27>:     mov     ebx,0x1
    0x08049020 <+32>:     mov     ecx,0x804a008
    0x08049025 <+37>:     mov     edx,0x7
    0x0804902a <+42>:     int     0x80
    0x0804902c <+44>:     mov     eax,0x1
    0x08049031 <+49>:     mov     ebx,0x0
    0x08049036 <+54>:     int     0x80
End of assembler dump.
```

Рассмотрим основные *различия отображения* синтаксиса машинных команд в режимах *ATT* и *Intel*:

- расположение значения регистра и его названия: в режиме ATT сначала идёт значение, потом название регистра, а в Intel - наоборот;
- в синтаксисе Intel нет префиксов перед регистром и его значением, а в ATT они есть: “\$” перед значением, “%” перед названием регистра.

5. Далее включаем режим псевдографики - это позволит удобнее анализировать изменения в процессе работы программы. Для этого используем две программы: `layout asm` и `layout regs`.

```
alyona@aeckhorosheva: ~/work/arch-pc/lab09
Register group: general
eax      0x0      0
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd020 0xffffd020
ebp      0x0      0x0

B->0x8049000 <_start>    mov     eax,0x4
0x8049005 <_start+5>    mov     ebx,0x1
0x804900a <_start+10>   mov     ecx,0x804a000
0x804900f <_start+15>   mov     edx,0x8
0x8049014 <_start+20>   int     0x80
0x8049016 <_start+22>   mov     eax,0x4

native process 8635 (asm) In: _start          L12    PC: 0x8049000
(gdb) layout regs
```

Как видно на скриншоте, в режиме есть три окна(сверху вниз):

- названия регистров с их текущими значениями;
- результат дизассемблирования кода;
- поле для ввода команд.

6. Продолжим работу с lab9-2.asm. Командой i b (info breakpoints) посмотрим, где установлены точки останова:

```
(gdb) i b
Num      Type          Disp Enb Address      What
1        breakpoint    keep y   0x08049000  lab9-2.asm:12
        breakpoint already hit 1 time
```

Установим ещё одну такую точку командой break \*

. Адрес инструкции можно узнать в средней части экрана в левом столбце. После установки новой точки останова проверяем информацию о всех точках командой i b.

```

b+ 0x8049031 <_start+49>    mov     ebx,0x0
    0x8049036 <_start+54>    int     0x80
    0x8049038                add     BYTE PTR [eax],al

native process 9130 (asm) In: _start
1      breakpoint      keep y  0x08049000 lab9-2.asm:12
      breakpoint already hit 1 time
(gdb) b *0x8049031
Breakpoint 2 at 0x8049031: file lab9-2.asm, line 25.
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint     keep y  0x08049000 lab9-2.asm:12
      breakpoint already hit 1 time
2        breakpoint     keep y  0x08049031 lab9-2.asm:25
(gdb)

```

7. Выполним 5 инструкций программы, чтобы посмотреть на изменение значений регистров. Для этого используем команду stepi(si).

- В регистре eax находится значение 4. Тем временем в средней части экрана мы видим, что выполняется уже следующая инструкция, которая помещает значение 1 в регистр ebx:

```

Register group: general
eax      0x4      4
ecx      0x0      0
edx      0x0      0
ebx      0x0      0
esp      0xffffd020 0xffffd020
ebp      0x0      0x0
esi      0x0      0
edi      0x0      0

D B+ 0x8049000 <_start>    mov     eax,0x4
    >0x8049005 <_start+5>  mov     ebx,0x1

```

- Теперь мы наблюдаем результат предыдущей инструкции - в регистре ebx значение 1. Переход к следующей инструкции - перемещение значения с определённого адреса в регистр ecx:

eax	0x4	4
ecx	0x0	0
edx	0x0	0
ebx	0x1	1
esp	0xffffd020	0xffffd020
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0

B+	0x8049000	<_start>	mov	eax,0x4
	0x8049005	<_start+5>	mov	ebx,0x1
	>0x804900a	<_start+10>	mov	ecx,0x804a000
	0x804900f	<_start+15>	mov	edx,0x8

- Регистр ecx получил нужное значение с указанным ранее адресом. Следующая инструкция помещает 8 в регистр edx:

eax	0x4	4
ecx	0x804a000	134520832
edx	0x0	0
ebx	0x1	1
esp	0xffffd020	0xffffd020
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0

B+	0x8049000	<_start>	mov	eax,0x4
	0x8049005	<_start+5>	mov	ebx,0x1
	0x804900a	<_start+10>	mov	ecx,0x804a000
	>0x804900f	<_start+15>	mov	edx,0x8

- В edx уже находится 8, также начинается выполнение инструкции int:

ecx	0x804a000	134520832
edx	0x8	8
ebx	0x1	1
esp	0xffffd020	0xffffd020
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0

B+	0x8049000	<_start>	mov	eax,0x4
	0x8049005	<_start+5>	mov	ebx,0x1
	0x804900a	<_start+10>	mov	ecx,0x804a000
	0x804900f	<_start+15>	mov	edx,0x8
	>0x8049014	<_start+20>	int	0x80

- Последнее выполнение - в eax значение 8, а следующая инструкция задаст значение 4 этому регистру:

eax	0x8	8
ecx	0x804a000	134520832
edx	0x8	8
ebx	0x1	1
esp	0xffffd020	0xffffd020
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0

B+	0x8049000	<_start>	mov	eax,0x4
	0x8049005	<_start+5>	mov	ebx,0x1
	0x804900a	<_start+10>	mov	ecx,0x804a000
	0x804900f	<_start+15>	mov	edx,0x8
	0x8049014	<_start+20>	int	0x80
	>0x8049016	<_start+22>	mov	eax,0x4

8. Теперь нужно отобразить содержимое памяти - для этого нужна команда `x` или `x/NFU` (чтобы задать формат вывода данных). Смотреть значение переменных можно как по имени, так и по адресу.

Для начала узнаем адрес переменной `msg2`. Используя дизассемблированный код находим инструкцию `mov ecx,msg2` и определяем адрес:

```
0x8049020 <_start+32>  mov     ecx,0x804a008
```

Можно тестировать команды: по имени переменной посмотрим значение `msg1`, а по адресу - `msg2`:

```
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "Hello, "
(gdb) x/1sb 0x804a008
0x804a008 <msg2>:      "world!\n\034"
```

9. Командой `set` можно изменять значения для ячейки памяти. Используем префикс `&` перед именем переменной, а в фигурных скобках указываем тип данных из языка Си:

```
(gdb) set {char}&msg1='h'
(gdb) x/1sb &msg1
0x804a000 <msg1>:      "hello, "
(gdb) set {char}&msg2='W'
(gdb) x/1sb &msg2
0x804a008 <msg2>:      "World!\n\034"
```

Таким образом, мы изменили надпись “Hello, world!” на “hello, world!”.

10. Чтобы посмотреть значения регистров используется команда `print /F $`. Выведем в различных форматах значение `edx`.

```

(gdb) p/s $edx
$1 = 8
(gdb) p/t $edx
$2 = 1000
(gdb) p/x $edx
$3 = 0x8

```

Мы видим, что p/s - это вывод в символьном виде, а p/t и p/x - в двоичной и шестнадцатеричной системах счисления соответственно.

11. Поменяем значение ebx на '2' и 2 по порядку. Посмотрим, как будет меняться вывод команды p/s(т.е. символьный вид вывода).

```

(gdb) set $ebx='2'
(gdb) p/s $ebx
$4 = 50
(gdb) set $ebx=2
(gdb) p/s $ebx
$5 = 2

```

Наглядно заметно, что вывод в этих двух случаях отличается. Сначала мы ввели "2" в кавычках, поэтому компьютер воспринимает это как символ, а не само число 2. Символу "2" соответствует число 50 в десятичной системе счисления согласно таблице символов ASCII (Приложение 2.Лабораторная работа №6).

Во втором случае мы ввели 2 именно как число, поэтому такой же результат и выводится командой p/s.

12. Копируем файл из лабораторной работы №8 с программой вывода на экран аргументов командной строки(листинг 8.2). Копию помещаем в файл



lab09-3.asm и создаём исполняемый файл.

Затем загружаем полученный файл в отладчик с указанием аргументов. Я указала три аргумента: 1, 2, "3". Устанавливаем точку останова - b \_start. Запускаем командой run.

Адрес вершины стека храниться в регистре esp и по этому адресу располагается число равное количеству аргументов командной строки (включая имя программы):

```
Breakpoint 1, _start () at lab09-3.asm:8
8      pop ecx ; Извлекаем из стека в `ecx` количество
(gdb) x/x $esp
0xffffcf50:      0x00000004
(gdb) x/s *(void**)(esp + 4)
0xffffd11f:      "/home/alyona/work/arch-pc/lab09/lab09-3"
(gdb) x/s *(void**)(esp + 8)
0xffffd147:      "1"
(gdb) x/s *(void**)(esp + 12)
0xffffd149:      "2"
(gdb) x/s *(void**)(esp + 16)
0xffffd14b:      "3"
(gdb) x/s *(void**)(esp + 20)
0x0:      <error: Cannot access memory at address 0x0>
```

Вывод первой команды x/x - число 4. Это 3 аргумента + 1 имя программы.

Далее идут 5 команд x/s: из них выполняются первые 4 команды (расположение файла с программой и введённые ранее с клавиатуры аргументы), а последняя выводит ошибку, так как аргументы в регистре esp закончились.

Шаг изменения адреса равен 4([esp+4], [esp+8], [esp+12] и т.д.), потому что размер операнда равен 4 байтам (целое число - int32).

## 5 Задание для самостоятельной работы

1. Создаём файл lab09-4.asm и копируем текст программы для вычисления значения функции  $f(x)$  для введённого с клавиатуры  $x$  (из лабораторной работы №8). Необходимо реализовать вычисление функции как подпрограмму:

`_calculateTheFunction:`

`mov ebx, 2`

`mul ebx`

`add esi, eax`

`add esi, 15 ; добавляем к промежуточной сумме`

`ret`

Из изменений этого участка кода: добавление название подпрограммы и команда `ret` для её окончания. Также изменения коснулись цикла `next:` теперь мы вызываем функцию для вычисления выражения, а не считаем внутри цикла:

`next:`

`cmp ecx, 0h ; проверяем, есть ли еще аргументы`

`jz _end ; если аргументов нет выходим из цикла`

`; (переход на метку _end)`

`pop eax ; иначе извлекаем следующий аргумент из стека`

`call atoi ; преобразуем символ в число`

\*call \_calculateTheFunction\* ; вызов функции для вычисления  $2x + 15$

loop next ; переход к обработке следующего аргумента

Запускаем исполняемый файл для значений аргумента 1,2,3. Результатом должно стать число 57:

```
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ./lab09-4 1 2 3
Функция: 2x+15
Результат: 57
```

Всё работает корректно.

2. Создаём файл lab09-5.asm и пишем туда текст программы из листинга 9.3. Эта программа должна вычислять значение  $(3+2)*4+5$ . Однако при запуске программа выдаёт неверный результат, а именно 10 вместо 25:

```
alyona@aeckhorosheva:~/work/arch-pc/lab09$ nasm -f elf lab09-5.asm
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-5 lab09-5.o
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ./lab09-5
Результат: 10
```

С помощью отладчика gdb необходимо исправить ошибку в данной программе. Для этого переходим в режим псевдографики для удобства анализа изменения регистров в процессе работы программы.

Используем команду si(stepi) до тех пор, пока не выявим ошибку:

- В регистре ebx находится значение 3. Число 2 передаётся в регистр eax.

ebx	0x3	3
esp	0xffffcf70	0xffffcf70
ebp	0x0	0x0
esi	0x0	0

B+	0x80490e8	<_start>	mov	ebx,0x3
	>0x80490ed	<_start+5>	mov	eax,0x2

- Мы видим, что значение в eax теперь стало равно 2. Следующей выполняется инструкция для сложения регистров ebx и eax, при этом их сумма остаётся в регистре ebx.

eax	0x2	2
ecx	0x0	0
edx	0x0	0
ebx	0x3	3
esp	0xffffcf70	0xffffcf70
ebp	0x0	0x0
esi	0x0	0

B+	0x80490e8	<_start>	mov	ebx,0x3
	0x80490ed	<_start+5>	mov	eax,0x2
	>0x80490f2	<_start+10>	add	ebx,ebx

- В регистре ebx имеем значение 5, т.е. пока что программа работает верно. Следующая инструкция задает значение 4 регистру ecx.

ebx	0x5	5
esp	0xffffcf70	0xffffcf70
ebp	0x0	0x0
esi	0x0	0

B+	0x80490e8	<_start>	mov	ebx,0x3
	0x80490ed	<_start+5>	mov	eax,0x2
	0x80490f2	<_start+10>	add	ebx,ebx
	>0x80490f4	<_start+12>	mov	ecx,0x4

- Регистр ecx теперь равен 4. Выполняется инструкция по умножению регистра ecx. Как нам известно, результат умножения будет в регистре eax в данном случае. Значит и умножаются значения eax и ecx. Однако сумма, которая нам нужно умножить на 4, помещена в регистр ebx, а не eax(см. второе выполнение инструкции).

ecx	0x4	4
edx	0x0	0
ebx	0x5	5
esp	0xffffcf70	0xffffcf70
ebp	0x0	0x0
esi	0x0	0

0x80490ed	<_start+5>	mov	eax,0x2
0x80490f2	<_start+10>	add	ebx,ebx
0x80490f4	<_start+12>	mov	ecx,0x4
B+>0x80490f9	<_start+17>	mul	ecx

- В регистре eax значение 8 - это ошибка. Операция умножения производилась лишь на одно из слагаемых, а не на необходимую сумму, поэтому результат получили неверный. Далее инструкция сложения прибавляет к ebx число 5. При этом мы не учитываем результат умножения, который при этом тоже работает некорректно.

eax	0x8	8
ecx	0x4	4
edx	0x0	0
ebx	0x5	5
esp	0xffffcf70	0xffffcf70
ebp	0x0	0x0
esi	0x0	0

0x80490f2	<_start+10>	add	ebx,ebx
0x80490f4	<_start+12>	mov	ecx,0x4
B+ 0x80490f9	<_start+17>	mul	ecx
>0x80490fb	<_start+19>	add	ebx,0x5

- В регистре ebx значение 10. Это не тот конечный результат, который нам нужен.

Он помещается в регистр edi для дальнейшего вывода на экран.

ebx	0xa	10
esp	0xffffcf70	0xffffcf70
ebp	0x0	0x0
esi	0x0	0

0x80490f4	<_start+12>	mov	ecx,0x4
B+ 0x80490f9	<_start+17>	mul	ecx
0x80490fb	<_start+19>	add	ebx,0x5
>0x80490fe	<_start+22>	mov	edi,ebx

Итак, мы выявили ошибки в коде программы благодаря отладчику. Теперь приступим к изменениям текста программы.

Во-первых, нужно сохранить сумму (3+2) в регистр eax, чтобы затем умножение происходило именно на неё.

```
mov ebx,3
mov eax,2
add eax,ebx
```

Во-вторых, прибавлять 5 в конце необходимо к регистру eax, так как именно там хранится результат умножения.

```
mov ecx,4
mul ecx
add eax,5
mov edi,eax
```

Теперь создадим исполняемый файл и проверим работу программы.

```
alyona@aeckhorosheva:~/work/arch-pc/lab09$ nasm -f elf lab09-5.asm
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ld -m elf_i386 -o lab09-5 lab09-5.o
alyona@aeckhorosheva:~/work/arch-pc/lab09$ ./lab09-5
Результат: 25
```

Получен корректный результат.

# Выводы

В результате выполнения лабораторной работы получены теоретические зна-

ния о методах отладки, типах ошибок, понятии подпрограмм и их значимости. На практике были получены навыки работы с отладчиком GDB, а именно: нахождение ошибок в коде, установка точек останова, дизассемблизация кода для нахождения адресов ячеек памяти, обработка аргументов командной строки и т.д..

В итоге, написана программа для нахождения значения  $f(g(x))$ , исправлена программа вычисления выражения  $(3+2)*4+5$ .

## **Список литературы**