Практическое занятие по СПО №6

Густов Владимир Владимирович
gutstuf@gmail.com

Цитата дня: Если отладка — процесс удаления ошибок, то программирование должно быть процессом их внесения. (с) Эдсгер Дейкстра

Repeat it

1

```
begin
 for i := 0 to 2 * 6 do
 begin
   case i of
    0: a := i + 1;
    1: b := a;
    2: c := b + a - i;
    3: begin
      if c > a then
       d := c + a + b
      else
       d := c div (a + b) * i
     end;
   end;
 end;
end.
```

2

```
begin
 a := 0;
 if (a > 5) and (a < 7) then
  goto _end
 else
  a := 5 + 4 div 2 - 2 * 1;
_end:
end.
```

Метка/символ

Метка:
  <имя> : <код>

Символ:
  .set <имя>, <значение>

```
.data

some_label:             # метка
   .byte 1

.set some_value, 2      # символ
```

```
.text
.globl _main
_main:

ret
```

Значение метки – **адрес**, указывающий на сегмент кода/выделенную память;

Значение символа – указанное **значение**

NM

Утилита, выводящая информацию о бинарных файлах, прежде всего – таблицу имён. Используется в качестве отладки, в частности для разрешения конфликтов имён.

```
$ gcc -m32 -c lbl.S
$ nm lbl.o
00000000 b .bss
00000000 d .data
00000000 t .text
00000000 T _main
00000000 d some_label
00000002 a some_value

$ gcc -m32 -o test lbl.S
$ nm test.exe
...
00404004 d some_label
00000002 a some_value
```

a – absolute value
b – bss symbol (uninitialized data section)
d – data section
t – text section

Адресация

1) $ - непосредственная – значение берётся «буквально» (напрямую)

2) _ - абсолютная – значение считается как адрес, и производится попытка считывания этого адреса (за исключением обращения к регистрам)

3) ($) – косвенная – работает как и абсолютная; при обращении к регистру, пытается считать значение как адрес

$ - Непосредственная

```
.data
Label:          # метка – указатель на местоположение в
  .long 1234    #  памяти, значением является адрес

.set var, 5     # символ – «абсолютное значение», значением
                #   является число 5

.text
.global _main

_main:
  xor  %eax, %eax   # обнуляем регистр eax
  movl $1337, %eax  # в регистре теперь хранится число 1337

  movl $Label, %eax # в регистре хранится адрес метки (0x84…)

  movl $var, %eax   # в регистре eax хранится число 5
  ret
```

Абсолютная

```
.data
Label:
    .long 1234

.set var, 5

.text
.global _main

_main:
  xor  %eax, %eax  # обнуляем регистры
  xor  %ebx, %ebx

  movl Label, %eax # в eax хранится значение, на которое ссылается
                   # метка, т.е. в eax хранится число 1234

  movl %eax, %ebx  # в ebx будет хранится значение регистра eax — 1234

  movl var, %eax   # Ошибка Segmentation fault (SIGSEGV), т.к.
                   # производится обращение к значению, как к адресу,
                   # а по адресу 5 ничего нет
```

() - Косвенная

```
_main:
  xor  %eax, %eax  # обнуляем регистр eax
  xor  %ebx, %ebx

  movl $Label, %ebx # в ebx содержится адрес метки - 0x8040...
  movl (%ebx), %eax # в eax содержится значение 1234,
                    # считанное из памяти по адресу 0x8040...

  movl (Label), %eax # эквивалентно обращению без ().
                     # В eax хранится 1234

  movl (%eax), %ebx # Ошибка SIGSEGV, т.к. пытаемся считать
                    # из памяти значение по адресу 0x1234,
                    # который нам не доступен

  movl (var), %eax  # эквивалентно обращению без ().
                    # Ошибка Segmentation fault
  ret
```

Указатели

смещение(база, индекс, множитель)

*Вычисленный адрес = база + индекс * множитель + смещение*

Множитель может принимать значения 1, 2, 4 или 8 (кратные 2).

(%ecx) - адрес операнда находится в регистре %ecx.

4(%ecx) - адрес операнда равен %ecx + 4. Например, в %ecx адрес некоторой структуры, второй элемент которой находится «на расстоянии» 4 байта от её начала (говорят «по смещению 4 байта»);

-4(%ecx) - адрес операнда равен %ecx − 4;

foo(,%ecx,4) - адрес операнда равен foo + %ecx × 4, где foo — некоторый адрес. Если foo — указатель на массив, элементы которого имеют размер 4 байта, то мы можем заносить в %ecx номер элемента и таким образом обращаться к самому элементу.

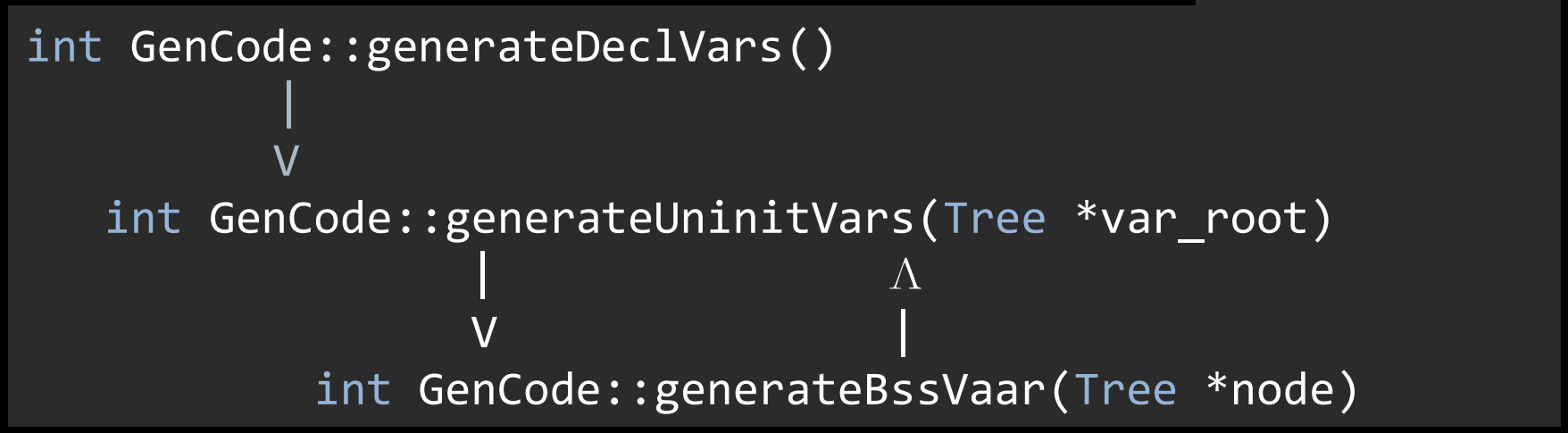
Указатели

```
.data
arr:    # 0  1  2  3
   .byte 1, 4, 8, 7

.text
.globl _main

_main:
  movl $2, %ecx          # используем, как номер эл-та, к-му хотим
                         # обратиться (в данном случае, ко 2 эл-ту)
  movb arr(,%ecx, 1), %al # в регистре al будет хранится число 8,
                         # т.к. arr + 2 * 1 == адрес указывающий на 2 эл.

  movl $arr, %ebx        # сохраняем адрес массива в регистре ebx
  movb 3(%ebx), %al      # в регистре al будет хранится число 7, т.к.
                         # %ebx + 3 == arr + 3 = адрес указ-ий на 3 эл.

  movb arr + 1, %al      # в регистре al будет хранится число 4, т.к.
                         # arr + 1 = адрес указывающему на первый элемент
  ret
```

Пример генерации GNU ASM из дерева

```
program exmpl;
var
  b    : boolean;
  a, d : integer;
  c : array [0..3] of integer;
begin
end.
```

```
# файл exmpl.S
.bss
b: .space 1
a: .space 4
d: .space 4
c: .space 16

.text
.globl main
main:
ret
```



0x11a9df8	exmpl	0x11a9b40	0x11a9c08
0x11a9b40	var	0x11a1410	0x11a1470
0x11a1410	b	0	0x11a1440
0x11a1440	boolean	0	0
0x11a1470	$	0x11a15f8	0x11a1658
0x11a15f8	a	0	0x11a1628
0x11a1628	integer	0	0
0x11a1658	$	0x11a1688	0x11a16e8
0x11a1688	d	0	0x11a16b8
0x11a16b8	integer	0	0
0x11a16e8	$	0x11a17d8	0
0x11a17d8	c	0x11a1808	0x11a1898
0x11a1808	array	0x11a1838	0x11a1868
0x11a1838	0	0	0
0x11a1868	3	0	0
0x11a1898	integer	0	0
0x11a9c08	begin	0	0x11a04b0
0x11a04b0	end	0	0

Источник/приёмник +-*/

```
inc операнд
dec операнд

add источник, приёмник
sub источник, приёмник

mul множитель
div делитель
```

add: приёмник + источник = приёмник

sub: приёмник – источник = приёмник

mul: умножает значение в регистре eax на операнд-множитель

div: нужно очистить edx. Делимое в eax

Команда	Второй сомножитель	Результат
mulb	%al	16 бит: %ax
mulw	%ax	32 бита: младшая в %ax, старшая в %dx
mull	%eax	64 бита: младшая в %eax, старшая в %edx

умножение/деление

`div источник`
`mul источник`

Результат умножения хранит в ***edx:eax*** регистрах

```
.text
.globl _main
_main:
  xorl %eax, %eax
  xorl %ebx, %ebx

  movw $32000, %ax
  movw $32000, %bx
  mulw %bx
ret
```

Перед использованием ***div*** необходимо очистить регистр **edx**:

```
xorl %edx, %edx
```

Результат деления 64 битных значений хранит в ***edx:eax*** регистрах

```
.text
.globl _main
_main:
  xorl %eax, %eax
  xorl %ebx, %ebx
  xorl %edx, %edx

  movw $32000, %ax
  movb $2, %bl
  divw %bx
ret
```

cmp

```
cmp операнд_2, операнд_1
```

Выполняет:

```
операнд_1 – операнд_2
```

и устанавливает флаги.

Jump-команды:
j<cc> <метка>

Мнемоника <cc>	Значение	Смысл
e	equal	Равенство
n	not	Инверсия
g	greater	Больше
l	less	Меньше
a	above	Больше (без знака)
b	below	Меньше (без знака)

```
cmpl %eax, %ebx
jl less # проверяет операнд_1 < операнд_2
jne not_equal    # операнд_1 <> операнд_2
jge greate_equal # операнд_1 >= операнд_2
```

loop

loop <метка>

уменьшает значение регистра %ecx на 1
если %ecx == 0, передаёт управление следующей за loop команде
если %ecx != 0, передаёт управление на метку

```
.text
.globl _main
_main:
  xorl %eax, %eax  # обнуляем регистр eax
  movl $5,   %ecx  # устанавливаем в счётчик значение 5

sum:
  addl %ecx, %eax  # считаем сумму чисел от 5 до 1
  loop sum         # если ecx > 0, повторяем операцию

ret
```

loop

```
loop_start:     # начало цикла

  # тут находится тело цикла

  cmpl  ...       # что-то с чем-то сравнить для
                  # принятия решения о выходе из цикла

  jne loop_start # подобрать соответствующую
# команду условного перехода
                  # для повторения цикла
```

break/goto

jmp <адрес>

```
program ohoho;
var a : integer;
label bad;
begin
bad:
     a := 4;
     goto bad
end.
```

```
.bss
a: .space 4

.text
.globl _main

_main:
  xorl %eax, %eax

  kek:
    movl $4, a
    jmp kek
ret
```

```
program okko;
var a, i : integer;
begin
  for i := 0 to 5 do
  begin
    a := 3 + i;
    break;
  end;
end.
```

```
.bss
a: .space 4
i: .space 4

.text
.globl _main
_main:
  xorl %eax, %eax

op1.0:
  movl $0, i
  jmp check
loop:

op2.0:
  movl $3, %eax
  addl i,  %eax
  movl %eax, a
op2.0_end:

op2.1:
  jmp op1.0_end
op2.1_end:

  incl i
check:
  cmpl $5, i
  jle loop
op1.0_end:
ret
```

```
program okno;
var a : array[0..3] of integer;
    b : integer = 5;
begin
    a[1] := 1;
    a[0] := b + a[1];
    a[2] := 3 * b;
    a[3] := a[0] + a[1] - a[2];
end.
```

1

```
movl $arr, %esi
movl $1, 1*4(%esi)
```

2

```
movl $1, %ecx
movl $1, arr(, %ecx, 4)
```

Для вывода содержимого массива в gdb:

```
p/x (long[3])arr
```

где 3 – кол-во элементов в массиве

Try It

```
program okno;
var a : array[0..2] of integer;
    b : integer = 5;
begin
    a[0] := 1;
    a[1] := b + a[0];
    a[2] := 3 * b;
end.
```

Стек

1 (esp == 0xFFFF0A0C)

Адрес	Значение
0xFFFF0A0C	XXX
0xFFFF0A08	XXX
0xFFFF0A04	XXX
0xFFFF0A00	XXX

```
1) nothing
2) pushl $5
3) pushl %ebp
4) popl  %ebp
```

2 (esp == 0xFFFF0A08)

Адрес	Значение
0xFFFF0A0C	XXX
0xFFFF0A08	5
0xFFFF0A04	XXX
0xFFFF0A00	XXX

3 (esp == 0xFFFF0A04)

Адрес	Значение
0xFFFF0A0C	XXX
0xFFFF0A08	5
0xFFFF0A04	0
0xFFFF0A00	XXX

esp — указатель на «вершину» стека;
ebp — указатель на фрейм стека;
фрейм — отдельный фрагмент памяти стека.

4 (esp == 0xFFFF0A08)

Адрес	Значение
0xFFFF0A0C	XXX
0xFFFF0A08	5
0xFFFF0A04	XXX
0xFFFF0A00	XXX

```
main:
  xorl  %eax, %eax
  pushl %ebp        # сохраним текущий указатель стека
  movl  %esp, %ebp # сохраним новый (текущий) адрес вершины стека
op1:
  movl $5, a
op2:
  pushl $5              # 1
  pushl $2              # 2
  popl %eax
  addl %eax, -4(%ebp) # обратимся ко второму элементу в стеке

  pushl a
  popl %eax             # 3
  subl %eax, -4(%ebp)
  popl %eax

  leave             # вернём стек обратно на адрес в ebp
  ret
```

ebp == 0xFFFF0A08

esp == 0xFFFF0A0C

2

esp == 0xFFFF0A00

3

esp == 0xFFFF0A04

Адрес	Значение
0xFFFF0A0C	XXX
0xFFFF0A08	XXX
0xFFFF0A04	XXX
0xFFFF0A00	XXX

0xFFFF0A0C	XXX
0xFFFF0A08	0
0xFFFF0A04	5
0xFFFF0A00	2
0xFFFF09FC	XXX

0xFFFF0A0C	XXX
0xFFFF0A08	0
0xFFFF0A04	7
0xFFFF0A00	XXX
0xFFFF09FC	XXX

Подпрограммы

Соглашения о вызове:

cdecl (see-deck_ll) — стек очищается вызывающим. Аргументы, меньше 4х байт, расширяются до 4х байт.

stdcall — стек очищается вызванным;

fastcall — стек очищается вызванным; Аргументы передаются в регистрах.

thiscall — стек очищается вызванным; В регистр ecx записывается указатель на объект для которого вызывается метод.

Google it: ABI, ISA (instruction set architecture)

cdecl

```
.data
fmt:
  .string "Some long text, size: %d\n"
fmt_length:
   .long . - fmt  # . возвращает текущий адрес

fmt2:
  .string "Also please print %d\n"

.text
.globl _main
_main:
  # пролог
  pushl %ebp
  movl  %esp, %ebp
  ##################
  pushl fmt_length
  pushl $fmt
  call _printf

  pushl $1337
  pushl $fmt2
  call _printf

  # эпилог
  movl %ebp, %esp  # можно заменить
  popl %ebp        #  оператором leave
  ret
```

gcc -S

```
void main() {
  int a = 0;
  int b;
  b = a + 3 – 1 * 2;
}
```

gcc -m32 -o - -S main.c

```
# (MinGW, Windows 10)
andl $-16, %esp
subl $16, %esp
call ___main
movl $0, 12(%esp)
movl 12(%esp), %eax
addl $1, %eax
movl %eax, 8(%esp)
nop
```

```
        .file   "main.c"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        subl    $16, %esp

        movl    $0, -8(%ebp)
        movl    -8(%ebp), %eax
        addl    $1, %eax
        movl    %eax, -4(%ebp)
        nop
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 7.2.0-
ubuntu3.1~16.04.york0) 7.2.0"
        .section .note.GNU-stack,"",@progbits
```

case

```
  movl  $30, %eax    # получить в %eax
                     # некоторое интересующее
                     # нас значение

  cmpl  $5, %eax
  je    case_5

  cmpl  $30, %eax
  je    case_30

  cmpl  $120, %eax
  je    case_120

case_default:
  movl $100, %ecx
  jmp switch_end

case_5
  movl $5, %ecx
  jmp switch_end

case_30
  movl $30, %ecx
  jmp switch_end

case_120
  movl $120, %ecx
  jmp switch_end
```

Логические (побитовые) операторы

```
and  источник, приёмник
or   источник, приёмник
xor  источник, приёмник
not  операнд
test операнд_2, операнд_1
```

Работают аналогично операторам в Си:
`&` (and), `|` (or), `^` (xor), `~` (not)

`test` – команда логического сравнения. Позволяет проверить наличие определённых битов. Например:

```
  testl %eax, %eax
  je is_zero

  testl $0x10, %eax    # eax != 0
  je isnt_exist
  nop                  # если в eax есть бит 0001 0000

isnt_exist:            # если в eax нет бита 0001 0000
  nop
```

Google it

sal/sar, shl/shr, rol/ror, rcl/rcr
rep, repe/repz, repne/repnz
movs, cmps, scas
lods, stos
cld, std
setcc

Try It

```
program math;
var a : integer = 4 + 2 - 3 * 1;
    c : integer;
const b : integer = 3;
begin
  c := 20 div a;
  a := a - (a * 2 + 3) + c * 2;
end.
```

Compile time

```
program test_me;
var i : integer;
const a : integer = 2;
  x : array [-0..3] of integer = (0, 1, a, 3);
label _end;
begin
  for i := 0 to 3 do
    writeln(x[i]);

  x[0] = 5;
end.
```