

Sistemas Hardware-Software

Aula 8 – Variáveis na pilha

Engenharia

Fabio Lubacheski

Maciel C. Vidal

Igor Montagner

Fábio Ayres

Funções e seus argumentos¶

Argumentos inteiros ou ponteiros são passados nos registradores (nesta ordem):

6 primeiros argumentos

%rdi
%rsi
%rdx
%rcx
%r8
%r9

Diane's

Silk

Dress

Costs

\$89

**retorno da
função**

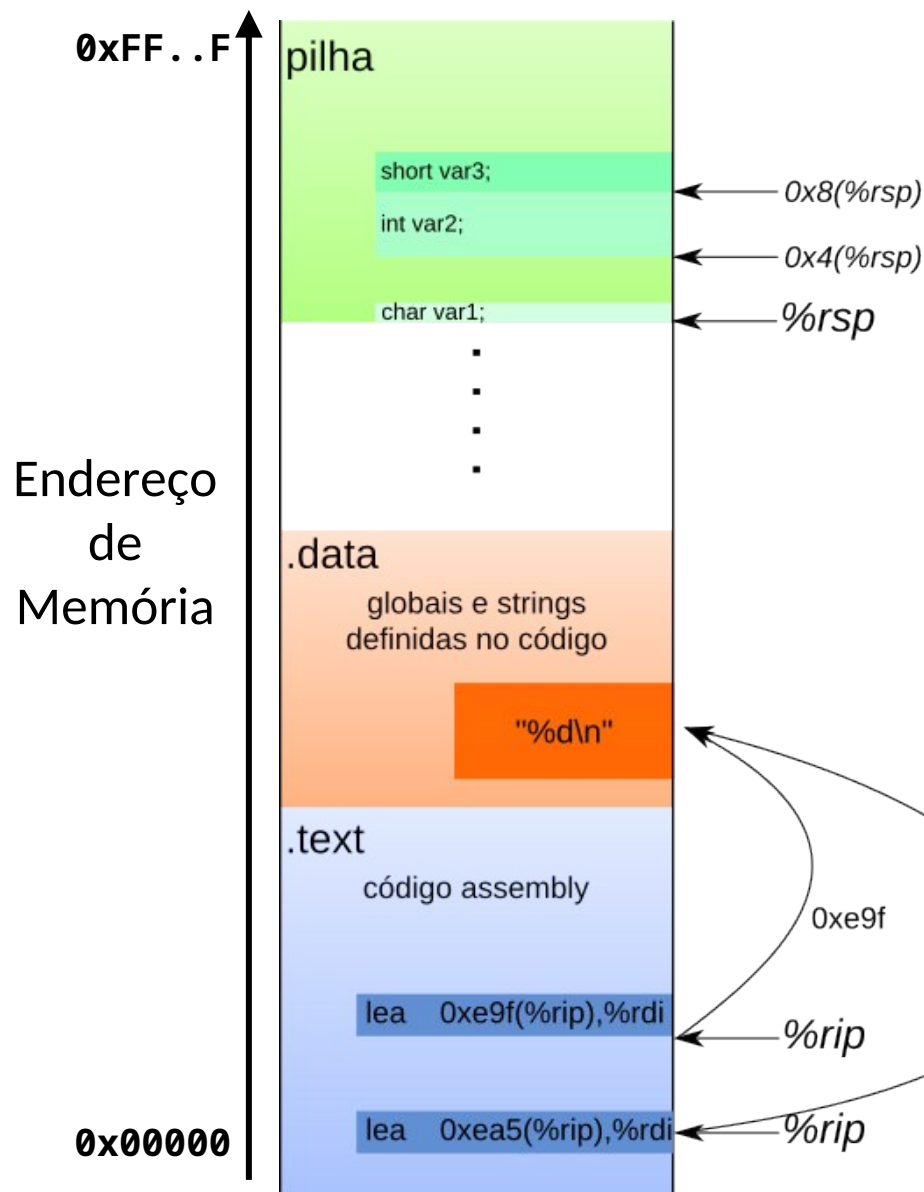
%rax

Os registradores **não estão na memória.**

Questões:

- Os ponteiros apontam para região de memória, mas qual região na memória?
- Só dá pra passar 6 argumentos para funções ? e se tiver mais que 6 ?

Executável na memória - variáveis



Variáveis locais

- Na maioria do tempo são colocadas em **registradores**
- Se não for possível colocamos na pilha
- Uso **&var** requer uso da pilha.
- Acessadas via deslocamentos relativos a **%rsp** (*stack pointer*) – **topo da pilha**

Variáveis globais / strings constantes

- Acessadas usando pulos relativos a **%rip** (*instruction pointer*)
- Como **%rip** muda a cada instrução, o deslocamento usado muda também
- É necessário fazer o cálculo para chegar ao endereço final

Uso da pilha para &variável

```
long incremento(long *p, long val) {  
    long x = *p;  
    long y = x + val;  
    *p = y;  
    return x;  
}
```

```
incremento:  
    movq    (%rdi), %rax // x = *p;  
    addq    %rax, %rsi // y = x + val  
    movq    %rsi, (%rdi) // *p = y;  
    ret
```

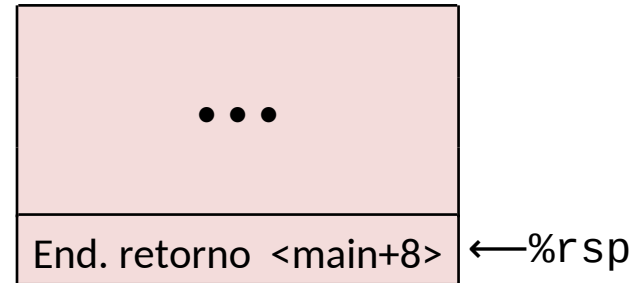
registrador	valor
%rdi	1o arg (p)
%rsi	2o arg (val), y
%rax	X, retorno

Chamada de funções (Estado inicial)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Estado inicial Stack (pilha)



O valor apontado por **%rsp** é o endereço da instrução imediatamente após a chamada para **call_incr** na função **main()**.

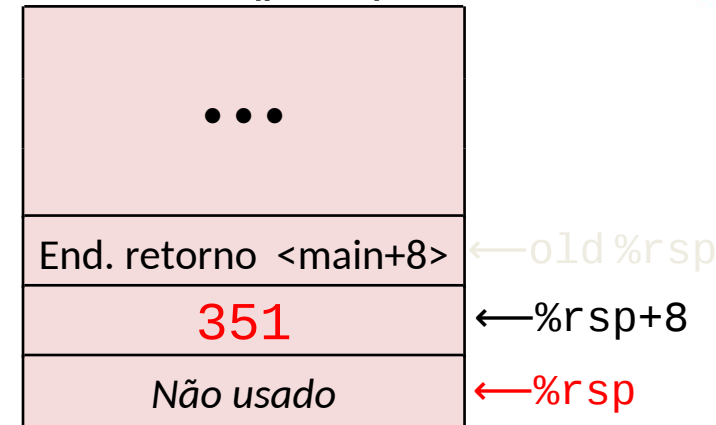
Execução da função **call_incr** (passo 1)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

} aloca espaço
na pilha para
variável v1

Stack (pilha)



Apenas a variável local **v1** precisa de espaço na pilha.

Compilador alocou espaço extra, isso pode acontecer por vários motivos, como por alinhamento na memória.

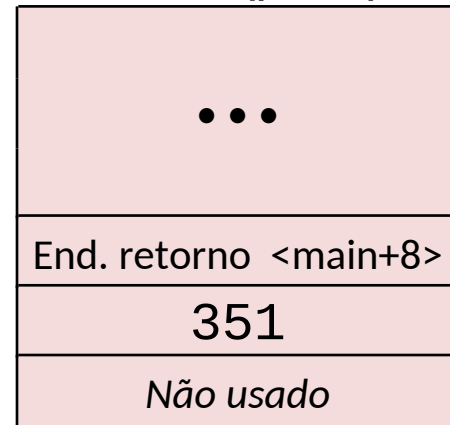
Execução da função **call_incr** (passo 2)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

} prepara os argumentos para
chamar a função incremento

Stack (pilha)



movl é usado porque 100 é um valor que cabe em um registrador de 32 bits **%esi**. Registrador **%rdi** recebe o endereço da variável **v1** na Stack.

registrador	valor
%rdi	&v1
%rsi	100

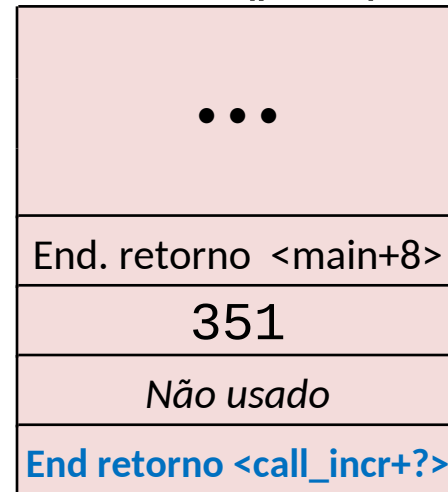
Execução da função **call_incr** (passo 3)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

```
incremento:  
    movq    (%rdi), %rax  
    addq    %rax, %rsi  
    movq    %rsi, (%rdi)  
    ret
```

Stack (pilha)



Execução dentro da função **incremento**
O endereço de retorno no topo da pilha é o endereço da instrução **addq** imediatamente após a chamada da função **incremento**

registrador	valor
%rdi	&v1
%rsi	100
%rax	

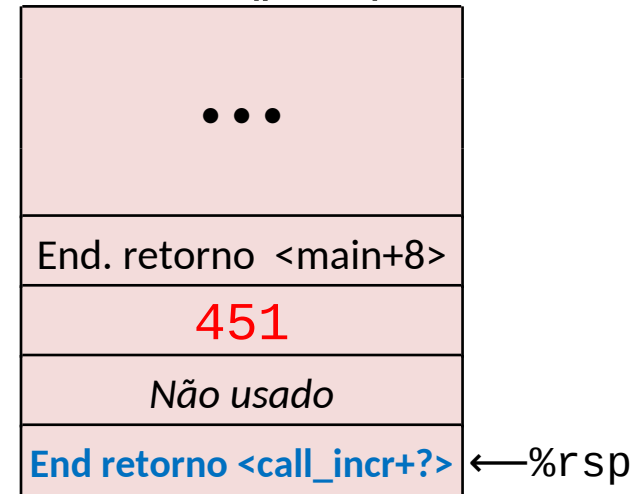
Execução da função **incremento** (passo 4)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

```
incremento:  
    movq    (%rdi), %rax // x = *p  
    addq    %rax, %rsi // y = x+100  
    movq    %rsi, (%rdi) // *p = y  
    ret
```

Stack (pilha)



Estado da execução após executar o corpo da função **incremento**.

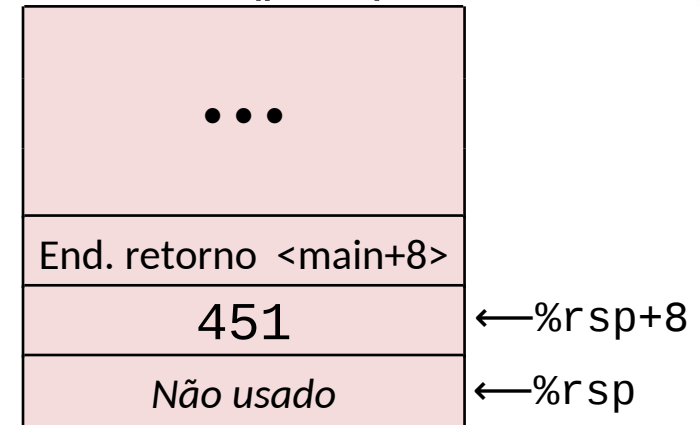
registrador	valor
%rdi	&v1
%rsi	451
%rax	351

Execução da função **call_incr** (passo 5)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = increment(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack (pilha)



Depois de retornar da função para **incremento**
Os registros e a memória foram modificados e o **endereço de retorno foi retirado da pilha**

registrador	valor
%rdi	&v1
%rsi	451
%rax	351

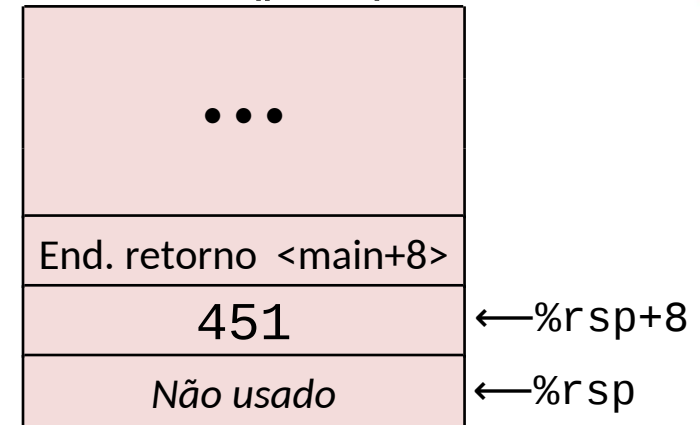
Execução da função **call_incr** (passo 6)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremenot(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

← Atualiza %rax com v1+v2

Stack (pilha)



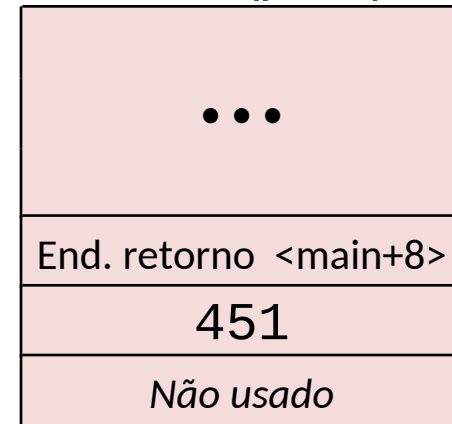
registrador	valor
%rdi	&v1
%rsi	451
%rax	451+351

Execução da função **call_incr** (passo 7)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack (pilha)



← %rsp

← old %rsp

← Desaloca espaço para variável local (v1)

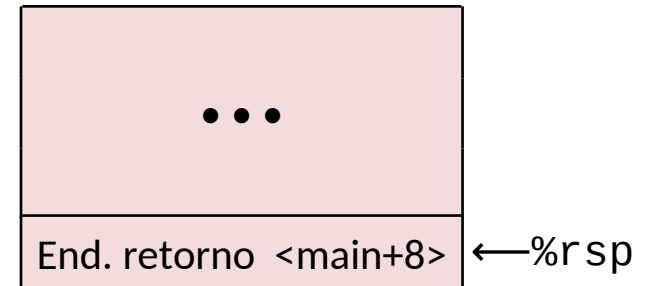
registrador	valor
%rdi	&v1
%rsi	451
%rax	802

Execução da função **call_incr** (passo 8)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack (pilha)



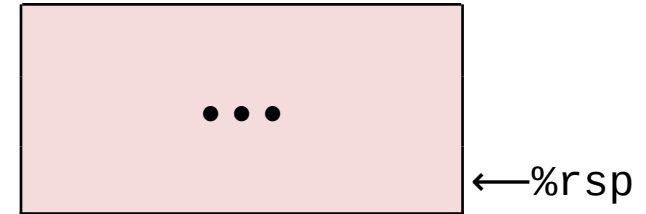
Estado antes de retornar da chamada para **call_incr**

registrador	valor
%rdi	&v1
%rsi	451
%rax	802

Execução da função **call_incr** (passo 9)

```
long call_incr() {  
    long v1 = 351;  
    long v2 = incremento(&v1, 100);  
    return v1+v2;  
}
```

Stack (pilha)



```
call_incr:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    incremento  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Estado imediatamente após retornar da chamada da função **call_incr**
O endereço de retorno foi retirado da pilha
O controle retornou para a instrução imediatamente após a chamada para **call_incr**

registrador	valor
%rdi	&v1
%rsi	451
%rax	802

Uso da pilha para função com mais de 6 argumentos

Quando uma função possui mais de 6 argumentos, também é usada a pilha para passar os valores dos argumentos.

```
int call_proc()  
{  
    int v1=1, v2=2,  
        v3=3, v4=4,  
        v5=5, v6=6,  
        v7=7, v8=8;  
  
    return  proc(v1, v2, v3, v4, v5, v6, v7, v8);  
}
```

```
call_proc:  
    pushq $8  
    pushq $7  
    movl  $6, %r9d  
    movl  $5, %r8d  
    movl  $4, %ecx  
    movl  $3, %edx  
    movl  $2, %esi  
    movl  $1, %edi  
    call  proc  
    addq  $16, %rsp  
    ret
```

Exemplo de função com mais 6 argumentos

```
int proc(int r1, int r2, int r3, int r4,  
         int r5, int r6, int p7, int p8)  
{  
    int resp;  
    resp = r1+r2+r3+r4+r5+r6+p7+p8;  
    return resp;  
}
```

```
proc:  
    addl    %esi, %edi  
    addl    %edx, %edi  
    addl    %ecx, %edi  
    addl    %r8d, %edi  
    leal    (%rdi,%r9), %eax  
    addl    8(%rsp), %eax  
    addl    16(%rsp), %eax
```




Atividade prática

Exercícios de aula

1. Identificar funções que usem variáveis locais
2. Listar todas as variáveis locais de uma função que foram alocadas na pilha



Atividade prática

Exercícios para entrega

1. Identificar funções que usem variáveis locais
2. Listar todas as variáveis locais de uma função que foram alocadas na pilha
3. Está no seu repositório de atividade

Correção dos exercícios 2 e 3

Ex2

Dump of assembler code for function func1:

```
0x05fe <+0>:      sub    $0x10,%rsp
0x0602 <+4>:      movl   $0xa,0xc(%rsp)
0x060a <+12>:     movl   $0xb,0x8(%rsp)
0x0612 <+20>:     lea    0xc(%rsp),%rdi
0x0617 <+25>:     callq  0x5fa <func2>
0x061c <+30>:     addl   $0x1,0x8(%rsp)
0x0621 <+35>:     lea    0x8(%rsp),%rdi
0x0626 <+40>:     callq  0x5fa <func2>
0x062b <+45>:     add    $0x10,%rsp
0x062f <+49>:     retq
```

Ex2

Variáveis auxiliares:
int *p1, *p2;

Dump of assembler code for function func1:

0x05fe	<+0>:	sub	\$0x10,%rsp	
0x0602	<+4>:	movl	\$0xa,0xc(%rsp)	→ int a = 10;
0x060a	<+12>:	movl	\$0xb,0x8(%rsp)	→ int b = 11;
0x0612	<+20>:	lea	0xc(%rsp),%rdi	→ p1 = &a;
0x0617	<+25>:	callq	0x5fa <func2>	→ func2(p1);
0x061c	<+30>:	addl	\$0x1,0x8(%rsp)	→ b++;
0x0621	<+35>:	lea	0x8(%rsp),%rdi	→ p2 = &b;
0x0626	<+40>:	callq	0x5fa <func2>	→ func2(p2);
0x062b	<+45>:	add	\$0x10,%rsp	
0x062f	<+49>:	retq		

Ex3

Dump of assembler code for function main:

```
0x1149 <+0>:      sub     $0x18,%rsp
0x114d <+4>:      lea     0xc(%rsp),%rsi
0x1152 <+9>:      lea     0xeab(%rip),%rdi      # 0x2004
0x1159 <+16>:     mov     $0x0,%eax
0x115e <+21>:     callq   0x1040 <__isoc99_scanf@plt>
0x1163 <+26>:     cmpl    $0x0,0xc(%rsp)
0x1168 <+31>:     js      0x1180 <main+55>
0x116a <+33>:     lea     0xe9f(%rip),%rdi      # 0x2010
0x1171 <+40>:     callq   0x1030 <puts@plt>
0x1176 <+45>:     mov     $0x0,%eax
0x117b <+50>:     add     $0x18,%rsp
0x117f <+54>:     retq
0x1180 <+55>:     lea     0xe80(%rip),%rdi      # 0x2007
0x1187 <+62>:     callq   0x1030 <puts@plt>
0x118c <+67>:     jmp     0x1176 <main+45>
```

Ex3

Dump of assembler code for function main:

```
0x1149 <+0>:      sub     $0x18,%rsp
0x114d <+4>:      lea     0xc(%rsp),%rsi
0x1152 <+9>:      lea     0xeab(%rip),%rdi      # 0x2004
0x1159 <+16>:     mov     $0x0,%eax
0x115e <+21>:     callq   0x1040 <__isoc99_scanf@plt>
0x1163 <+26>:     cmpl    $0x0,0xc(%rsp)
0x1168 <+31>:     ---js     0x1180 <main+55>
0x116a <+33>:     lea     0xe9f(%rip),%rdi      # 0x2010
0x1171 <+40>:     callq   0x1030 <puts@plt>
0x1176 <+45>:     mov     $0x0,%eax ←-----
0x117b <+50>:     add     $0x18,%rsp
0x117f <+54>:     retq
0x1180 <+55>:     ->lea     0xe80(%rip),%rdi      # 0x2007
0x1187 <+62>:     callq   0x1030 <puts@plt>
0x118c <+67>:     jmp     0x1176 <main+45> -----
```

```
} int n;
} scanf("%d", &n);
} if (n<0) {
}   goto negativo;
} }
} printf("Positivo\n");
}
} retorno:
}   return 0;
} negativo:
}   printf("Negativo\n");
}   goto retorno;
```

Insper

www.insper.edu.br