

Lista 4 – Felipe Melo – Thalles Nonato



DRE Felipe: 119093752

DRE Thalles: 119058809

Questão 1) a) Compile o código com a opção -m32 e gere um executável. Procure explicar a saída impressa. Seja detalhado, justificando cada valor impresso:

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ ls -lHs
total 8
4 -rw-rw-r-- 1 felipe felipe 211 mai 15 13:16 questao1.c
4 -rw-rw-r-- 1 felipe felipe 1152 mai 15 13:17 questao1.s
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ cat questao1.c
main() {
    short int array[] = {0, 1, 2, 3, 4};
    #define TOTAL_ELEMENTS (sizeof(array) / sizeof(array[0]))
    printf("%d\n", sizeof(TOTAL_ELEMENTS));
    printf("%d\n", sizeof(array) / sizeof(array[0]));
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ gcc -m32 -fno-PIC -o questao1.out questao1.c
questao1.c:1:1: warning: return type defaults to 'int' [-Wimplicit-int]
1 | main() {
  | ~~~~~
questao1.c: In function 'main':
questao1.c:4:5: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
4 |     printf("%d\n", sizeof(TOTAL_ELEMENTS));
  |     ~~~~~
questao1.c:4:5: warning: incompatible implicit declaration of built-in function 'printf'
questao1.c:1:1: note: include '<stdio.h>' or provide a declaration of 'printf'
+++ |+#include <stdio.h>
1 | main() {
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ ./questao1.out
4
5
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$
```



O primeiro valor impresso é 4. Isso pois, apesar de a expressão `sizeof(array) / sizeof(array[0])` resultar em 5, graças ao comando `#define`, ocorreu um cast desse valor para `int`, ocupando 4 bytes.

Já no segundo `printf`, o programa imprime o valor 5 pois se trata da razão entre o tamanho total do nosso array do tipo `short int` e o tamanho de um elemento isolado do array (`array[0]`). Dessa forma, teremos $10 / 2 = 5$.

Questão 1) b) Compile, sem includes como dado, usando as instruções -E -m32 e liste como resposta o que foi obtido, verificando se houve alguma substituição de sizeof no pré-processamento:

```
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ ls -lHs
total 24
4 -rw-rw-r-- 1 felipe felipe 211 mai 15 13:16 questao1.c
16 -rwxrwxr-x 1 felipe felipe 15600 mai 15 13:28 questao1.out
4 -rw-rw-r-- 1 felipe felipe 1152 mai 15 13:17 questao1.s
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ gcc -E -m32 -o questao1v2.out questao1.c
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ ls -lHs
total 28
4 -rw-rw-r-- 1 felipe felipe 211 mai 15 13:16 questao1.c
16 -rwxrwxr-x 1 felipe felipe 15600 mai 15 13:28 questao1.out
4 -rw-rw-r-- 1 felipe felipe 1152 mai 15 13:17 questao1.s
4 -rw-rw-r-- 1 felipe felipe 328 mai 15 20:27 questao1v2.out
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$ cat questao1v2.out
# 1 "questao1.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "questao1.c"
main() {
    short int array[] = {0, 1, 2, 3, 4};

    printf("%d\n", sizeof((sizeof(array) / sizeof(array[0]))));
    printf("%d\n", sizeof(array) / sizeof(array[0]));
}
felipe@Felipe-VirtualBox:~/Desktop/Lista 4/Questao1$
```

Não houve substituição.

Questão 1) c) Justifique cada linha no código gerado, associando ao código C mostrando ou a procedimento explicado:

main:	
01 endbr32	20 call printf
02 leal 4(%esp), %ecx	21 addl \$16, %esp
03 andl \$-16, %esp	22 subl \$8, %esp
04 pushl -4(%ecx)	23 pushl \$5
05 pushl %ebp	24 pushl \$.LC0
06 movl %esp, %ebp	25 call printf
07 pushl %ecx	26 addl \$16, %esp
08 subl \$20, %esp	27 movl \$0, %eax
09 movl %gs:20, %eax	30 movl -12(%ebp), %edx
10 movl %eax, -12(%ebp)	31 xorl %gs:20, %edx
11 xorl %eax, %eax	32 je .L3
12 movw \$0, -22(%ebp)	33 call __stack_chk_fail
13 movw \$1, -20(%ebp)	.L3:
14 movw \$2, -18(%ebp)	34 movl -4(%ebp), %ecx
15 movw \$3, -16(%ebp)	35 leave
16 movw \$4, -14(%ebp)	36 leal -4(%ecx), %esp
17 subl \$8, %esp	37 ret
18 pushl \$4	
19 pushl \$.LC0	



L2: copia o primeiro argumento `argc` para `%ecx`

L3: faz o topo da pilha se mover para o próximo endereço múltiplo de 16 após zerar os 4 bits menores do topo

L4: salva o RIP na pilha

L5 e L6: cria registro de ativação

L7: copia `argc` para o topo da pilha

L8: adiciona 5 posições na pilha

L9: prepara teste de corrupção

L10: copia `%eax` para `(%ebp - 12)`

L11: zera o registrador `%eax`

L12: copia o valor do índice 0 do array para `(%ebp - 22)`

L13: copia o valor do índice 1 do array para `(%ebp - 20)`

L14: copia o valor do índice 2 do array para `(%ebp - 18)`

L15: copia o valor do índice 3 do array para `(%ebp - 16)`

L16: copia o valor do índice 4 do array para `(%ebp - 14)`

L17: adiciona 2 posições na pilha

L18: salva o valor 4 no topo da pilha

L19: salva a string no topo da pilha

L20: chama a função `printf`

L21: diminui 4 posições da pilha

L22: adiciona 2 posições na pilha

L23: salva o valor 5 no topo da pilha

L24: salva a string no topo da pilha

L25: chama a função `printf`

L26: diminui 4 posições da pilha

L27: zera o registrador `%eax`

L30: copia `(%ebp - 12)` para `%edx`

L31: compara com valor original

L32: se for igual, não houve corrupção e faz um desvio para .L3 para retornar

L33: rotina para tratar corrupção da pilha

L34: copia `(%ebp - 4)` para `%ecx`

L35: copia %ebp para %esp e em seguida restaura o conteúdo anterior de %ebp na pilha

L36: faz o topo apontar para o RIP original

L37: transfere o controle para o endereço de retorno da função



Questão 2) a) Comente cada linha do código, associando ao código C. Identifique os elementos das estruturas que estão sendo acessados e/ou manipulados. Justifique a ocorrência das instruções:

```
soma:
01  endbr32
02  pushl %ebp
03  movl %esp, %ebp
04  subl $16, %esp
05  movl 12(%ebp), %edx
06  movl 16(%ebp), %eax
07  movl (%eax), %eax
08  addl %edx, %eax
09  movl %eax, -8(%ebp)
10  movl 12(%ebp), %edx
11  movl 16(%ebp), %eax
12  movl (%eax), %eax
13  subl %eax, %edx
14  movl %edx, %eax
15  movl %eax, -4(%ebp)
16  movl 8(%ebp), %ecx
17  movl -8(%ebp), %eax
18  movl -4(%ebp), %edx
19  movl %eax, (%ecx)
20  movl %edx, 4(%ecx)
21  movl 8(%ebp), %eax
22  leave
23  ret $4

produto:
24  endbr32
25  pushl %ebp
26  movl %esp, %ebp
27  subl $40, %esp
28  movl %gs:20, %eax
29  movl %eax, -12(%ebp)
30  xorl %eax, %eax
31  movl 8(%ebp), %eax
32  movl %eax, -28(%ebp)
33  leal 12(%ebp), %eax
34  movl %eax, -24(%ebp)
35  leal -20(%ebp), %eax
36  pushl -24(%ebp)
37  pushl -28(%ebp)
38  pushl %eax
39  call soma
40  addl $8, %esp
41  movl -20(%ebp), %edx
42  movl -16(%ebp), %eax
43  imull %edx, %eax
44  movl -12(%ebp), %ecx
45  xorl %gs:20, %ecx
46  je .L5
47  call __stack_chk_fail
.L5:
48  leave
49  ret
```

L2 e L3: cria registro de ativação

L4: abre 4 posições na pilha

L5: copia s1.a para %edx

L6: copia s1.p para %eax

L7: faz *s1.p para %eax

L8: soma s1.a e *s1.p

L9: copia o resultado da soma para a variável resultado.soma

L10: copia s1.a para %edx

L11: copia s1.p para %eax

L12: pega o valor de s1.p, copiando *s1.p para %eax

L13: subtrai s1.a e *s1.p, armazenando o resultado em %edx (resultado.dif)

L14: copia resultado.dif para %eax

L15: salva resultado.dif na pilha

L16: copia (%ebp + 8) para %ecx

L17: copia resultado.soma para %eax

L18: copia (%ebp - 4) para %edx

L19: salva resultado.soma na pilha

L20: salva s1.a na pilha

L21: salva *s1.p na pilha

L22: copia %ebp para %esp e em seguida restaura o conteúdo anterior de %ebp na pilha

L23: transfere o controle para o endereço de retorno da função e incrementa o topo da pilha de 8

L25 e L26: cria registro de ativação

L27: abre 10 posições na pilha

L28: move canário para %eax

L29: salva o canário na pilha

L30: zera %eax

L31: copia x para %eax

L32: salva x na pilha

L33: copia y para %eax

L34: salva o endereço de y (s1.p) na pilha

L35: copia (%ebp - 20), ou seja, s2, para %eax

L36: coloca s1.p no topo da pilha (preparando para a chamada da função soma)

L37: coloca s1.a no topo da pilha (preparando para a chamada da função soma)

L38: coloca s1 no topo da pilha (preparando para a chamada da função soma)

L39: chama a função soma

L40: diminui 2 posições da pilha

- L41: copia s2.soma para %edx
- L42: copia s2.dif para %eax
- L43: computa (s2.soma * s2.dif) e armazena o resultado em %eax
- L44: move o canário para %ecx
- L45: compara com valor original
- L46: se for igual, não houve corrupção e faz um desvio para .L5 para retornar
- L47: rotina para tratar corrupção da pilha
- L48: copia %ebp para %esp e em seguida restaura o conteúdo anterior de %ebp na pilha
- L49: transfere o controle para o endereço de retorno da função



Questão 2) b) Faça um desenho inicial da pilha no estado imediatamente antes de entrar na execução da função produto, mostrando os parâmetros que foram passados para a função, bem como o RIP de quem chamou produto. Mostre que você conhece a passagem de parâmetros para a rotina produto. Todos os endereços que forem x16 devem ser identificados:

Endereço	Pilha	Comentário
%ebp + 16	&y	*s1.p
%ebp + 12	x	s1.a
%ebp + 8		
%ebp + 4	RIP	endereço de retorno ao sair produto
%esp = %ebp	OFP	base de produto
%ebp - 4	resultado.dif	s1.a - *s1.p
%ebp - 8	resultado.soma	s1,a + *s1.p
%ebp - 12		
%ebp - 16		



Questão 2) c) Complete o desenho da pilha, usando o modelo ex12-editado, a partir do início de execução da função produto, passando pela execução da função soma, mostrando todo o conteúdo da pilha até o retorno de soma, após execução da L23. Ao alterar o topo da pilha para retornar espaços, mantenha o conteúdo da memória e apenas atualize o ponteiro para o topo da pilha. O desenho da pilha deverá apresentar simultaneamente os registros de ativação de produto e o de soma. Identifique no campo descrição a base de cada um dos registros de forma clara (e.g., %ebp de prod, %ebp de soma):

Endereço	Pilha	Comentário
%ebp + 12	y	1º parâmetro
%ebp + 8	x	2º parâmetro
%ebp + 4	RIP	endereço de retorno ao sair produto
%esp = %ebp	OFP	base anterior
%ebp - 4		
%ebp - 8		
%ebp - 12	canário	
%ebp - 16 (n * 16)	resultado.dif	
%ebp - 20	resultado.soma	
%ebp - 24	&y	s1.p
%ebp - 28	x	struct s1
%ebp - 32 (n * 16)		
%ebp - 36		
%ebp - 40		
%ebp - 44	&y	s1.a
%ebp - 48 (n * 16)	x	s1.p
%ebp - 52		
%ebp - 56	RIP de soma	
%ebp (soma)	OFP	base anterior
%ebp (soma) - 4 (n * 16)	resultado.dif	s1.a - *s1.p
%ebp (soma) - 8	resultado.soma	s1,a + *s1.p

Questão 2) d) Quais os endereços das estruturas s1 e s2, manuseadas por produto? Indique isso no desenho da pilha e justifique:

Verificamos que a estrutura s1, pelo desenho da pilha, estará no endereço (%ebp - 28) e a estrutura s2 em (%ebp - 20).

Questão 2) e) O que a instrução na linha 23 faz? Pesquise e seja preciso:

A instrução `ret` sem acompanhar nenhum valor apenas transfere o controle para o endereço de retorno da rotina, enquanto a instrução da linha 23 além disso também incrementa o topo da pilha de 8.



Questão 2) f) Existe uma estratégia geral para o retorno de estruturas por uma função. Analise como a função soma sabe onde montar a estrutura resultado na pilha da função produto, o que a função soma retorna em `%eax` e descreva a estratégia. Dê argumentos sólidos baseados no exemplo acima.

A função recebe um parâmetro camuflado, que é um ponteiro para a estrutura de retorno. Quando a função soma retorna, a produto consegue acessar a estrutura de retorno, pois ela está em seu registro de ativação.



Questão 3) a) Faça a engenharia reversa, associando as linhas ao código C. Identifique o que está sendo calculado, quais variáveis e ponteiros estão sendo manipulados:

```
test:
1    movl    4(%esp), %eax
2    movl    8(%esp), %edx
3    movl    %eax, %ecx
4    sall    $4, %ecx
5    sall    $3, %eax
6    addl    16(%edx,%ecx), %eax
7    movl    52(%edx), %ecx
8    addl    (%edx), %ecx
9    movw    %cx, 4(%edx,%eax,2)
10   ret
```

L1: copia o argumento i para %eax

L2: copia o segundo argumento p para %edx

L3: copia i para %ecx

L4: computa $16*i$ e salva o resultado em %ecx

L5: computa $8*i$ e salva o resultado em %eax

L6: computa $[(\%ecx + \%edx + 16) + \%eax]$, ou seja, $[(16i + p + 16) + 8i] = (24i + p + 16)$

L7: copia $(\%edx + 52)$, ou seja, $(p + 52)$, que é $p \rightarrow right$, em %ecx

L8: soma o conteúdo de %edx com %ecx

L9: copia n para $q \rightarrow x[p \rightarrow idx]$

L10: transfere o controle para o endereço de retorno da função



Questão 3) b) Encontre o valor de, apresentando argumentos sólidos para a dimensão do vetor a:

A linha 7 do código de montagem nos dá o limite superior do tamanho da nossa struct1 a[D] (p + 52). Considerando que a primeira instrução int left está em (p + 0) e que ela é do tipo inteiro, verificamos que ela varia de (p + 0) a (p + 4), onde começa a struct. A partir daí, ela está entre os limites (p + 4) a (p + 52), totalizando 48 bytes de tamanho. A linha 6 nos indica que


$$[8i + (16i + p + 16)] = [8i + (16i + p + 4 + 12)] = [8i + (16i + \text{struct1 a}[D] + 12)] \\ = [8i + (q \rightarrow \text{idx})].$$



Além disso, sabemos que em C não há nenhum tipo primitivo de tamanho 12 bytes. Sendo assim, sabemos que anteriormente à idx havia um array. A linha 8 nos ajuda a descobrir o tamanho de idx, uma vez que ela consiste de um addl, o qual, por padrão, só pode operar com inteiros (4 bytes). Logo, sizeof(idx) = 4 e, analogamente, sizeof(x[]) = 12. Assim, D = 48 / (sizeof(x[]) + sizeof(idx)) = 48 / 16 = 3

Questão 3) c) Determine sizeof(idx) e sizeof(a) com justificativas claras:

Como explicado na alternativa B:

A linha 8 nos ajuda a descobrir o tamanho de idx, uma vez que ela consiste de n addl, o qual, por padrão, só pode operar com inteiros (4 bytes). Logo, sizeof(idx) = 4.

A linha 7 do código de montagem nos dá o limite superior do tamanho da nossa struct1 a[D] (p + 52). Considerando que a primeira instrução int left está em (p + 0) e que ela é do tipo inteiro, verificamos que ela varia de (p + 0) a (p + 4), onde começa a struct. A partir daí, ela está entre os limites (p + 4) a (p + 52), totalizando 48 bytes de tamanho.

Questão 3) d) Determine o tipo do vetor x e sua dimensão com justificativas claras:

Na linha 9, percebemos que, por se tratar da instrução movw, precisamos apenas de 2 bytes. Por isso, o vetor x, necessariamente deve conter dados de 2 bytes, ou seja, é do tipo short int. Além disso, pela análise feita no item A, sabemos que a struct1 a[3] possui 48 bytes, os quais representam o somatório de x[i] com idx.

$$3 * [\text{sizeof}(\text{idx}) + i * \text{sizeof}(x[])] = 48 \quad \rightarrow \quad 3 * [4 + 2i] = 48 \quad \rightarrow \quad i = 6.$$

Dessa forma, descobrimos que a dimensão do vetor é 6.



Questão 3) e) Identifique as possíveis declarações da estrutura struct1, sabendo que os únicos campos nesta estrutura são idx e o vetor x. Você tem que justificar os tipos das variáveis e a dimensão dos vetores de forma clara. Ao final, indique as declarações viáveis para struct1:

Na struct1 a[D], como analisado acima, percebemos que seus campos idx e x[] se tratam, respectivamente, de dados do tipo int e short int, assim como o valor de D é 3, a dimensão de x[] é 6.



Questão 4) a) Quais registradores o GCC escolheu inicialmente para as variáveis x e z? Justifique:

%esi para a variável x e %ebx para z. Podemos ver claramente nas linhas 10 e 11 do código de montagem que são atribuídos 7 e 3, que são justamente os valores passados no trecho de código em C, aos registradores em questão.



Questão 4) b) Quais os registradores escolhidos pelo programador no comando ASM para armazenar as variáveis x, y e z?

%esi, %edx e %ebx, respectivamente. Não há mudança nos registradores que já possuem valores e, portanto, os registradores nos quais estavam salvos x e z (%esi e %ebx) continuam com esses valores. Portanto, y está armazenado em %edx, que é o único registrador dentro de #APP que não recebeu valores anteriormente.



Questão 4) c) Dê argumentos sólidos para configurar a lista de saída do comando ASM:

A lista de saída é composta pelos elementos que estão em função printf (x e y, respectivamente).



Questão 4) d) Dê argumentos sólidos para configurar a lista de entrada do comando ASM:

A lista de entrada é composta pelos elementos que além de declarados, são também inicializados no programa (x e z, respectivamente).



Questão 4) e) Escreva o comando ASM apagado, usando tanto notação posicional como a nominal:

Notação posicional:

```
asm  (“leal 5(%1), %0  \n\t”  
      “addl %3, %1  \n\t”  
      : “=d”(y),   “=S”(x)  
      : “S”(x),    “b”(z)  
      :  
      );
```



Notação nominal:

```
asm  (“leal 5(%%[x]), %%[y]  \n\t”  
      “addl %%[z], %%[x]  \n\t”  
      : [y] “=d”(y),   [x] “=S”(x)  
      : [x] “S”(x),    [z] “b”(z)  
      :  
      );
```