

Relatório de trabalho

1. Objectivo

O objectivo do trabalho é a criação duma aplicação na linguagem C que resolva um puzzle do jogo Batalha Naval. O trabalho dividiu-se em 13 tarefas distintas (comandos **c**, **m**, **l**, **e**, **h**, **v**, **p**, **E**, **V**, **D**, **R**, **G**, **q**).

2. Estruturas

Como precisámos de actuar sobre um “tabuleiro”, que não é mais do que uma matriz com a informação do número de artefactos em cada linha/coluna, criámos um tipo de dados, `<TAB_BN>`. Este tipo, não é mais que uma *struct*, da qual fazem parte 5 variáveis: número de linhas; número de colunas; um vector com a informação das linhas e outro para as colunas; matriz de caracteres (que corresponde ao tabuleiro *per se*). Definimos os vectores e a matriz como tendo um tamanho fixo de 105. Além desta estrutura central (sobre a qual assentam todas as funcionalidades do jogo), foram criados mais dois tipos `<JOGO>` e `<STACK>` que são também structs, que vão guardando o estado do tabuleiro à medida que este evolui, permitindo assim funcionalidades como introduzir uma jogada aleatório ou reverter jogadas até determinado ponto.

As declarações destes três tipos foram colocadas num ficheiro header que depois foi incluído em todos os ficheiros `.c`.

3. Interpretação de comandos

Para que de facto os comandos implementados possam ser utilizados pelo utilizador, foi criada uma função `<interpretar>`, cujo valor de retorno é o inteiro que é retornado pela função-comando que o utilizador “chama” (e.g. “c”, “l”). Este inteiro é depois utilizado na função `<interpretador>`, que consoante o seu valor, ou não faz nada (comando inválido), ou acaba a execução do programa (resultado do comando “q”) ou modifica/dá a informação do tabuleiro, de acordo com o comando introduzido.

4. Comando <resolver>

A implementação do comando <resolver> tira partido do facto das estratégias retornarem o inteiro “1” ou “-1” caso de facto mudem ou não o estado do tabuleiro, respectivamente. Assim, foi criada uma função auxiliar que contém um ciclo que chama as estratégias e aplica-as ao tabuleiro, enquanto elas mudem o seu estado. Essa função retorna o inteiro “-1” caso nenhuma das 4 estratégias seja mais capaz de alterar o estado do tabuleiro. Assim, quando esta função retorna “-1”, ou o puzzle está resolvido, ou as estratégias são incapazes de avançar na resolução do mesmo.

Foi criada uma função auxiliar <esta_resolvido> que retorna “1” ou “0” caso o puzzle esteja resolvido ou esteja por resolver, respectivamente. Além disso foram criadas outras funções auxiliares como a <jogada_aleatoria> que faz uma jogada aleatória quando o puzzle não está resolvida e as estratégias não são capazes de avançar na resolução.

O comando “R” <cmd_R> tira partido das funções acima descritas para conseguir resolver correctamente o puzzle fornecido. É uma função recursiva que aplica a função <resolver>, faz jogadas aleatórias, desfaz jogadas, quando envereda por um caminho errado e pára quando o tabuleiro <esta_resolvido>.

5. Análise do código gerado

- Tabela de alocação de registos

Registos	Variáveis			
%edi	x	x	x	x
%esi				tam
%ebx				y
%ecx	y	y	y	
%edx	dy	dy	dy	
%eax	num	tam	tam	count
	1	2	3	4

Os números 1,2,3 e 4 correspondem a diferentes instantes ao longo da execução da função.

O instante 1 corresponde às instruções antes da verificação da condição if (lin). As instruções da condição if (lin) e da condição else () dizem respeito aos instantes 2 e 3, respectivamente. Por fim, as instruções do ciclo for () estão representadas no instante 4.

- Estudo da variável *tab* e indexação da matriz

Sendo a variável *tab* uma matriz 100 por 100 de caracteres, é expectável que ela ocupe cerca de 10 000 bytes. Isto porque a matriz contém cerca de 10 000 elementos (100 linhas x 100 colunas), e cada um desses elementos, sendo um char, ocupa 1 byte, perfazendo então o total de 10 000 bytes.

Pela análise do código é possível verificar que esta variável faz parte de uma estrutura. Analisando essa mesma estrutura, constata-se que a variável *tab* é a primeira variável a ser declarada, sendo a variável *lins* a segunda. Como em memória as variáveis da estrutura são armazenadas consecutivamente, uma forma para poder verificar qual o endereço onde está armazenada a variável *tab* passa por verificar onde está armazenada a variável *lins* e, subtrair 10 000 bytes.

Pela análise do código gerado em Assembly, a instrução <contar_segs+54> tem por finalidade colocar o valor de *lins* no registo %eax. Desta forma é possível concluir que a variável *lins* está armazenada no endereço de memória dado por %ebp + 0x2718 (Fig.1).

Register group: general			
eax	0x1	1	
edx	0x0	0	
esp	0xbfff9bb0	0xbfff9bb0	
esi	0xbfffea48	-1073747384	
eip	0x8048436	0x8048436 <contar_segs+54>	
cs	0x73	115	
ds	0x7b	123	
fs	0x0	0	
ecx	0x0	0	
ebx	0x1	1	
ebp	0xbfff9bc8	0xbfff9bc8	
edi	0x0	0	
eflags	0x202	[IF]	
ss	0x7b	123	
es	0x7b	123	
gs	0x33	51	

B+	0x8048406	<contar_segs+6>	sub	\$0xc,%esp
	0x8048409	<contar_segs+9>	xor	%edi,%edi
	0x804840b	<contar_segs+11>	xor	%ecx,%ecx
	0x804840d	<contar_segs+13>	xor	%edx,%edx
	0x804840f	<contar_segs+15>	cmpl	\$0x0,0x2720(%ebp)
	0x8048416	<contar_segs+22>	movl	0x2724(%ebp),%eax
	0x804841c	<contar_segs+28>	movl	\$0x0,-0x10(%ebp)
	0x8048423	<contar_segs+35>	movl	\$0x0,-0x14(%ebp)
	0x804842a	<contar_segs+42>	je	0x8048486 <contar_segs+134>
	0x804842c	<contar_segs+44>	leal	-0x1(%eax),%ecx
	0x804842f	<contar_segs+47>	movl	\$0x1,-0x10(%ebp)
	0x8048436	<contar_segs+54>	movl	0x2718(%ebp),%eax
	0x804843c	<contar_segs+60>	test	%eax,%eax
	0x804843e	<contar_segs+62>	jle	0x804847b <contar_segs+123>

FIGURA 1 - CONTEÚDO DOS REGISTOS APÓS A EXECUÇÃO DA INSTRUÇÃO <CONTAR_SEGS+47>

Pela figura 1 tem-se que o registo %ebp tem o valor 0xbfff9bc8, e somando este com 0x2718 obtém-se o endereço 0xbfffc2e0. A este endereço, subtraindo-se 1000 (0x2710) obtém-se 0xbfff9bd0 que corresponde ao endereço onde está armazenada a variável *tab*.

Uma forma de comprovar este valor passa por verificar qual a posição de memória do primeiro carácter. Na primeira vez que o código é executado, este carácter é passado como argumento para a função <e_seg> assim, analisando a linha <contar_segs+91>,

Register group: general			
eax	0xbfff9bd0	-1073767472	ecx 0x0 0
edx	0x0	0	ebx 0x0 0
esp	0xbfff9bb0	0xbfff9bb0	ebp 0xbfff9bc8 0xbfff9bc8
esi	0xa	10	edi 0x0 0
eip	0x804845b	0x804845b <contar_segs+91>	eflags 0x246 [PF ZF IF]
cs	0x73	115	ss 0x7b 123
ds	0x7b	123	es 0x7b 123
fs	0x0	0	gs 0x33 51

0x804843c	<contar_segs+60>	test	%eax,%eax
0x804843e	<contar_segs+62>	jle	0x804847b <contar_segs+123>
0x8048440	<contar_segs+64>	mov	%eax,%esi
0x8048442	<contar_segs+66>	lea	0x0(,%edx,4),%eax
0x8048449	<contar_segs+73>	add	%edx,%eax
0x804844b	<contar_segs+75>	lea	(%ecx,%ecx,4),%ebx
0x804844e	<contar_segs+78>	mov	%eax,-0x18(%ebp)
0x8048451	<contar_segs+81>	lea	0x0(%esi),%esi
0x8048454	<contar_segs+84>	lea	(%ebx,%ebx,4),%eax
0x8048457	<contar_segs+87>	lea	0x8(%ebp,%eax,4),%eax
0x804845b	<contar_segs+91>	sub	\$0xc,%esp
0x804845e	<contar_segs+94>	movsbl	(%edi,%eax,1),%eax
0x8048462	<contar_segs+98>	push	%eax
0x8048463	<contar_segs+99>	call	0x80483e4 <e_seg>

FIGURA 2 - CONTEÚDO DOS REGISTOS APÓS A EXECUÇÃO DA INSTRUÇÃO <CONTAR_SEGS+87>

constata-se que o registo %eax contém o endereço do primeiro caracter, fruto da execução da instrução anterior.

Pela figura 2, comprova-se que o endereço %eax contém o valor previamente calculado.

Posto isto, tem-se que esta zona de memória se encontra organizada da forma representada na Tabela 1:

Tabela 1 - Organização em memória da variável tab

Endereços Memória	
0xbfff9bd0	tab [1][1]
0xbfff9bd1	tab [1][2]
...	...
...	tab[1][100]
...	tab[2][1]
...	...
0xbfffc2df	tab[100][100]
0xbfffc2e0	lins

A matriz é armazenada em memória linha a linha. Assim, a forma de aceder a um elemento da matriz é feito do seguinte modo:

$$\text{endereço_elemento} = (\text{linha_elemento}) \cdot (\text{coluna_elemento}) \cdot (\text{n}^\circ_elementos_por_linha) + (\text{endereço_da_matriz}).$$

As linhas de código *Assembly* responsáveis por fazer a indexação da matriz são então:

```
<contar_segs+64>:  mov  %eax,%esi
<contar_segs+66>:  lea   0x0(%edx,4),%eax
<contar_segs+73>:  add   %edx,%eax
<contar_segs+75>:  lea   (%ecx,%ecx,4),%ebx
<contar_segs+78>:  mov   %eax,-0x18(%ebp)
<contar_segs+81>:  lea   0x0(%esi),%esi
<contar_segs+84>:  lea   (%ebx,%ebx,4),%eax
<contar_segs+87>:  lea   0x8(%ebp,%eax,4),%eax
<contar_segs+91>:  sub   $0xc,%esp
<contar_segs+94>:  movsbl (%edi,%eax,1),%eax
<contar_segs+91>:  sub   $0xc,%esp
<contar_segs+94>:  movsbl (%edi,%eax,1),%eax
```

Anexo - código assembly comentado

```
0x08048400 <contar_segs+0>: push  %ebp
--salvaguarda o antigo base pointer. (endereço de regresso)
0x08048401 <contar_segs+1>: mov   %esp, %ebp
--igual a o stack pointer com o base pointer
0x08048403 <contar_segs+3>: push  %edi
--salvaguarda o registo (3ºarg)
0x08048404 <contar_segs+4>: push  %esi
--salvaguarda o registo (2ºarg)
0x08048405 <contar_segs+5>: push  %ebx
--salvaguarda o registo (1ºarg)
0x08048406 <contar_segs+6>: sub   $0xc,%esp
--%esp-0xc(aumentar 12 bytes na stack)
0x08048409 <contar_segs+9>: xor   %edi,%edi
--igual a 0 (x=0)
0x0804840b <contar_segs+11>:      xor   %ecx,%ecx
--igual a 0 (y=0)
0x0804840d <contar_segs+13>:      xor   %edx,%edx
--igual a 0 (dy=0)
0x0804840f <contar_segs+15>: cmpb  $0x0,0x2720(%ebp)
--if(lin)
0x08048416 <contar_segs+22>:      mov   0x2724(%ebp),%eax
--chama o valor de num e coloca em %eax
0x0804841c <contar_segs+28>:      movl  $0x0,-0x10(%ebp)
--dx=0
0x08048423 <contar_segs+35>:      movl  $0x0,-0x14(%ebp)
--count=0
0x0804842a <contar_segs+42>:      je    0x8048486 <contar_segs+134>
--caso seja igual a 0, salta para o endereço 0x8048486 (avança para o else)
0x0804842c <contar_segs+44>:      lea   -0x1(%eax),%ecx
--y=num-1
0x0804842f <contar_segs+47>: movl  $0x1,-0x10(%ebp)
--dx=1
0x08048436 <contar_segs+54>:      mov   0x2718(%ebp),%eax
--tam=t.lins
0x0804843c <contar_segs+60>:      test  %eax,%eax
--Verifica se tamanho é maior ou menor que 0
0x0804843e <contar_segs+62>:      jle   0x804847b <contar_segs+123>
--salta se %eax for 0 ou menor, ou seja, salta para depois do ciclo for
0x08048440 <contar_segs+64>:      mov   %eax,%esi
--passa o valor de tam para esi
0x08048442 <contar_segs+66>:      lea   0x0(,%edx,4),%eax
--sendo %edx=dy,o valor (4*dy) é colocado em %eax (se dy=1 -> eax=4; se dy=0 -> eax=0)
0x08048449 <contar_segs+73>:      add   %edx,%eax
--%eax=eax+dy
0x0804844b <contar_segs+75>:      lea   (%ecx,%ecx,4),%ebx
--5*y->ebx
0x0804844e <contar_segs+78>:      mov   %eax,-0x18(%ebp)
--Guarda o valor de dy em memoria
0x08048451 <contar_segs+81>:      lea   0x0(%esi),%esi
--Verifica o valor de tam
0x08048454 <contar_segs+84>:      lea   (%ebx,%ebx,4),%eax
--5*(5*y)->eax
0x08048457 <contar_segs+87>:      lea   0x8(%ebp,%eax,4),%eax
--ebp+(25*y*4)+8->eax
```

```

0x0804845b <contar_segs+91>:    sub    $0xc,%esp
--sobe o stack pointer mais 12 bytes
0x0804845e <contar_segs+94>:    movsbl (%edi,%eax,1),%eax
--vai buscar à memória o caracter que será passado como argumento para %eax (t.tab[y][x])
0x08048462 <contar_segs+98>:    push   %eax
--salv guarda registo %eax (que contém o caracter de t.tab[y][x])
0x08048463 <contar_segs+99>:    call   0x80483e4 <e_seg>
--chamada da função e_seg
0x08048468 <contar_segs+104>:    add    $0x10,%esp
--desce o stack pointer 16 bytes
0x0804846b <contar_segs+107>:    test   %al,%al
--verifica se o resultado de e_seg é 0 ou 1
0x0804846d <contar_segs+109>:    je     0x8048472 <contar_segs+114>
--saltar para o endereço 0x8048472 se o resultado de e_seg for 0
0x0804846f <contar_segs+111>:    incl   -0x14(%ebp)
--count++
0x08048472 <contar_segs+114>:    add    -0x10(%ebp),%edi
--x+=dx
0x08048475 <contar_segs+117>:    add    -0x18(%ebp),%ebx
--y+=dy
0x08048478 <contar_segs+120>:    dec    %esi
--tam--
0x08048479 <contar_segs+121>:    jne    0x8048454 <contar_segs+84>
--salta para endereço 0x8048454 atualização dos parametros do ciclo for
0x0804847b <contar_segs+123>:    mov    -0x14(%ebp),%eax
--eax=count
0x0804847e <contar_segs+126>:    lea    -0xc(%ebp),%esp
--recupera o endereço do stack pointer inicial
0x08048481 <contar_segs+129>:    pop    %ebx
--recupera registo
0x08048482 <contar_segs+130>:    pop    %esi
--recupera registo
0x08048483 <contar_segs+131>:    pop    %edi
--recupera registo
0x08048484 <contar_segs+132>:    leave
0x08048485 <contar_segs+133>:    ret
0x08048486 <contar_segs+134>:    lea    -0x1(%eax),%edi
--x=num-1
0x08048489 <contar_segs+137>:    mov    $0x1,%edx
--dy=1
0x0804848e <contar_segs+142>:    mov    0x271c(%ebp),%eax
--tam=t.cols
0x08048494 <contar_segs+148>:    jmp    0x804843c <contar_segs+60>
--salto para o endereço 0x804843c, inicio do ciclo for()

```