Software Básico

Aula #05

# Código e Dados: Vetores

Como realizar a alocação de um vetor de tamanho estático? Como acessar as posições de um vetor em memória?

**Ciência da Computação - BCC2 - 2023/02**
Prof. Vinícius Fülber Garcia

# Relembrando Conceitos

O QUE É UM VETOR?

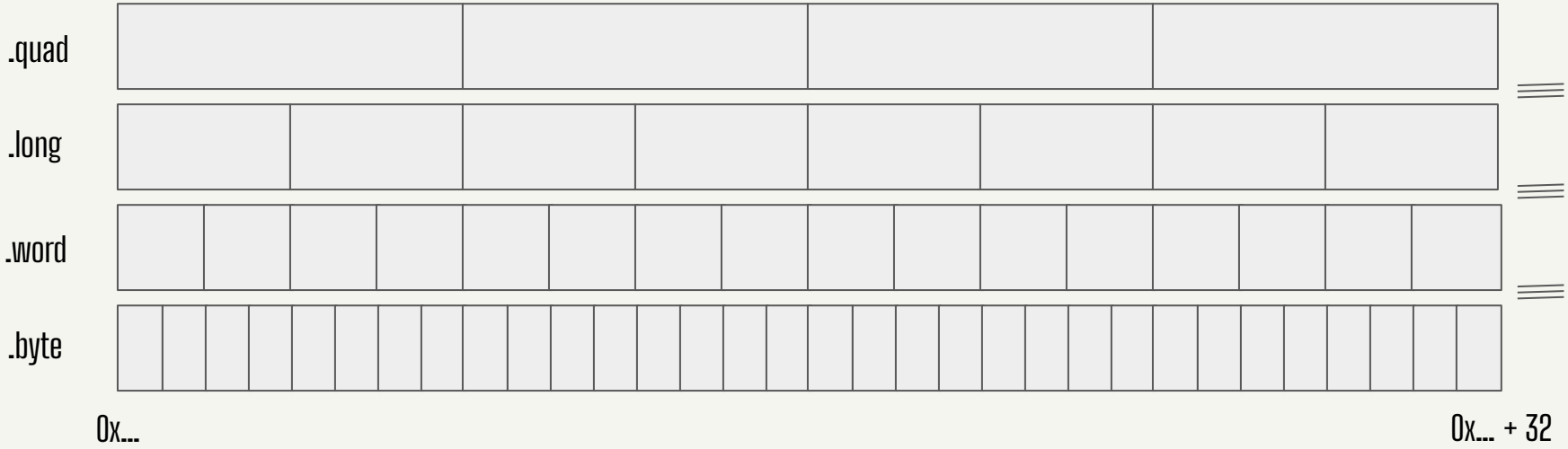
# Relembrando Conceitos

Um vetor pode ser compreendido como uma coleção de elementos idênticos dispostos de maneira sequencial na memória.

Mais do que isso, podemos abstrair a ideia de elementos: um elemento é representado por uma certa quantidade de bytes.

**Logo, um vetor representa uma determinada quantidade de bytes sequenciais em memória!**

# Relembrando Conceitos




Vale lembrar, entretanto, que a memória é indexada, em baixo nível, em **bytes**!

# Vetores em *Assembly*

```
1)   long int i = 0;
2)   long int m = 0;
3)   long int v[5] = {10, 30, 5, 10, 50};
4)   int main(){
5)     for(;i < 5; i++){
6)         if (v[i] > m)
7)             m = v[i];
8)     }
9)     return m;
10)  }
```

Considere o código em C ao lado. Analisando suas linhas, sabemos traduzir boa parte dele para *assembly*; porém, ainda nos falta:

- Entender como vetores globais são declarados
- Entender como acessar os dados em determinadas posições de um vetor

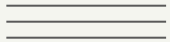
# Vetores em *Assembly*

Vamos por partes! Primeiro, vamos entender como **DECLARAR UM VETOR** em *assembly*...

- Os vetores globais e inicializados são declarados na seção *.data*, como qualquer variável;
- Na sua inicialização, os valores são determinados um após o outro, separados por vírgula;
- O número de valores é o número de elementos no vetor; a multiplicação de tal número pelo tamanho de dados utilizado define a quantidade de bytes no vetor.

# Vetores em *Assembly*

```
1)   long int i = 0;
2)   long int m = 0;
3)   long int v[5] = {10, 30, 5, 10, 50};
```



```
1)   .section .data
2)    i: .quad 0
3)    m: .quad 0
4)    v: .quad 10, 30, 5, 10, 50
```

Considerando a notação de declaração de vetores em *assembly* AMD64, temos a tradução ao lado.

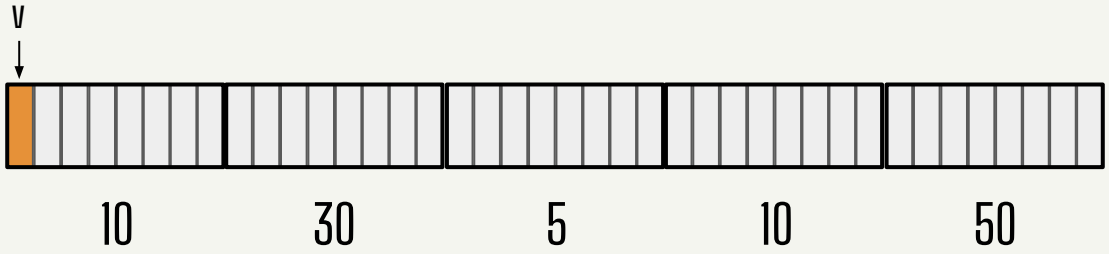
Sendo esta a inicialização, quantos bytes a frente está o início do vetor v da...

- Variável i?
- Variável m?

# Vetores em *Assembly*

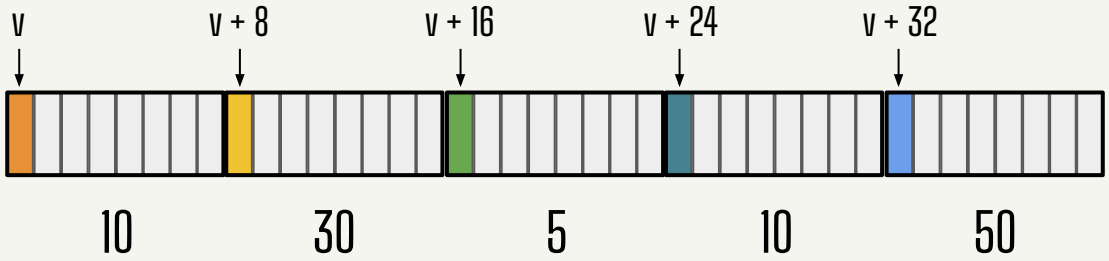
Outro ponto que vale a pena ressaltar é que **RÓTULOS SÃO APENAS MNEMÔNICOS PARA POSIÇÕES DE MEMÓRIA** em *assembly*.

Sendo assim, o rótulo do vetor nada mais é que a posição do primeiro byte do seu primeiro elemento!



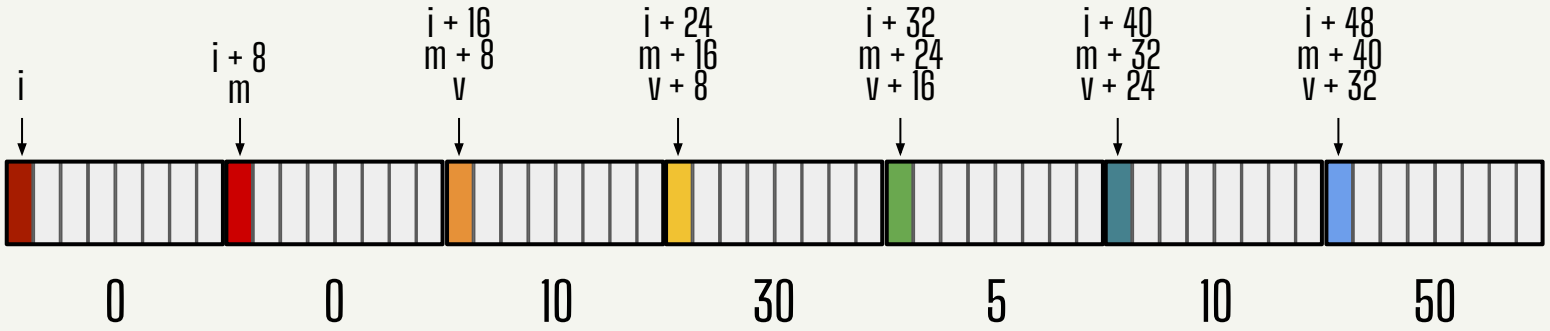
# Vetores em *Assembly*

Então, nós podemos percorrer o vetor considerando v como o nosso endereço base e adicionando ao mesmo um determinado valor de deslocamento (em bytes) a depender do tamanho do dado utilizado: *.quad*, por exemplo, tem oito (8) bytes.



# Vetores em *Assembly*

Um ponto interessante de notar é que, na seção *.data*, **todas as variáveis são alocadas de maneira sequencial**. Ou seja, qualquer uma pode servir de base para acessar as demais: basta conhecermos o deslocamento correto!



# Vetores em *Assembly*

Mas a pergunta em que chegamos é:

**Como nós representamos esse deslocamento e acessamos um dado específico em *assembly*?**

# Endereçamento em *Assembly*

Para falarmos de acesso a elementos específicos de um vetor, precisamos entender os modelos de endereçamento em assembly AMD64.

O primeiro e mais simples de todos é o **ENDEREÇAMENTO DIRETO**, que consiste em acessar um dado contido em um endereço fornecido de maneira explícita:

```
movq v %rax
```

(O conteúdo em v é movido para o registrador %rax)

Mas... o que determina a quantidade de bytes lidos?

# Endereçamento em *Assembly*

Também existe a alternativa de **ENDEREÇAMENTO INDIRETO**, que consiste em acessar um dado contido em um endereço fornecido em um registrador:

```
movq $v %rax
```

(O endereço de v é movido para o registrador %rax)

```
movq (%rax) %rbx
```

(O conteúdo da posição de memória em %rax é movido para %rbx)

Essa alternativa já nos permite fazer o acesso a diferentes elementos de um vetor... algum palpite de como isso ocorre?

# Endereçamento em *Assembly*

Usando o valor de **%rax (endereço de memória) como uma base**, podemos simplesmente **adicionar o deslocamento desejado (em bytes)** utilizando uma expressão aritmética:

```
movq $v %rax
```

(O endereço de v é movido para o registrador %rax)

```
addq $8 %rax
```

(Um deslocamento de oito bytes é aplicado em %rax)

```
movq (%rax) %rbx
```

(O conteúdo da posição de memória em %rax é movido para %rbx)

# Endereçamento em *Assembly*

O uso de endereçamento indireto funciona muito bem para percorrer vetores, sendo tipicamente a única opção para tal tarefa em processadores RISC.

Porém, percorrer um vetor via endereçamento indireto tem uma inconveniência:

**O endereço base não é mantido! Ele é deslocado a cada operação.**

Em processadores CISC, entretanto, é comum haver uma terceira alternativa de endereçamento que facilita a nossa vida nesse sentido!

# Endereçamento em *Assembly*

A terceira alternativa de endereçamento é chamada de **ENDEREÇAMENTO INDEXADO**. Nesse contexto, indexamos um elemento específico em memória através da seguinte notação:

```
displacement(base, index, scale)
```

Mas, para simplificar e ainda assim satisfazer nossas necessidades, usaremos:

```
displacement(, index, scale)
```

# Endereçamento em *Assembly*

`displacement(, index, scale)`

- *displacement*: o endereço base, considerando o endereço 0x0 como referência
- *index*: o índice do elemento do vetor que queremos acessar
- *scale*: a quantidade de bytes ocupada por um elemento do vetor

Sabendo desta notação, como fazemos para acessar o primeiro elemento do vetor v? E o terceiro?

# Vetores em *Assembly*

Acessando o **primeiro** elemento:

```
v(, 0, 8)
```

Acessando o **terceiro** elemento:

```
v(, 2, 8)
```

E se eu...

```
v(, 0, 4)
```

# Vetores em *Assembly*

Temos que ter muito cuidado com acessos parciais aos elementos, pois isso não resulta em erro no programa, mas pode gerar resultados errados!

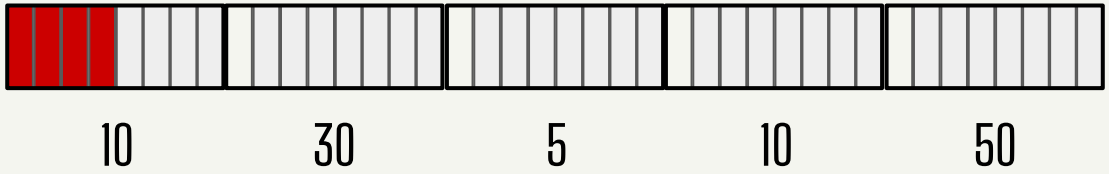
Essa é uma situação que também pode ocorrer facilmente quando usamos **endereçamento indireto**:

```
movq $v %rax
addq $12 %rax
movq (%rax) %rbx
```

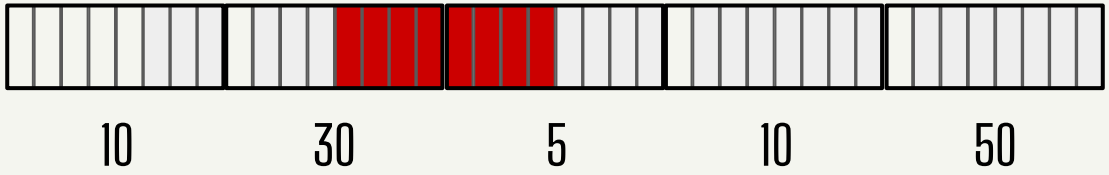
**O que constará em %rbx?**

# Vetores em *Assembly*

`v(, 0, 4):`




`movq $v %rax;addq $12 %rax;movq (%rax) %rbx:`



# Vetores em *Assembly*

Para facilitar um pouco a memorização, podemos criar paralelos entre os modelos de endereçamento em *assembly* e o acesso à vetores em C!

## ENDEREÇAMENTO DIRETO

```c
long int v[5] = {10, 30, 5, 10, 50};
int main(){
 return *v;
}
```

≡

```asm
.section .data
 v: .quad 10, 30, 5, 10, 50
.section .text
.global _start
_start:
 movq v, %rdi
 movq $60, %rax
 syscall
```

# Vetores em *Assembly*

Para facilitar um pouco a memorização, podemos criar paralelos entre os modelos de endereçamento em *assembly* e o acesso à vetores em C!

## ENDEREÇAMENTO INDIRETO

```c
long int v[5] = {10, 30, 5, 10, 50};
int main(){
 return *(v+1);
}
```

≡

```asm
.section .data
 v: .quad 10, 30, 5, 10, 50
.section .text
.global _start
_start:
 movq $v, %rax
 addq $8, %rax
 movq (%rax), %rdi
 movq $60, %rax
 syscall
```

# Vetores em *Assembly*

Para facilitar um pouco a memorização, podemos criar paralelos entre os modelos de endereçamento em *assembly* e o acesso à vetores em C!

## ENDEREÇAMENTO INDEXADO

```c
long int v[5] = {10, 30, 5, 10, 50};
int main(){
 return v[2];
}
```

=

```assembly
.section .data
v: .quad 10, 30, 5, 10, 50
 .section .text
 .global _start
_start:
 movq $2, %rax
 movq v(, %rax, 8), %rdi
 movq $60, %rax
 syscall
```

# Vetores em *Assembly*

```
1)   long int i = 0;
2)   long int m = 0;
3)   long int v[5] = {10, 30, 5, 10, 50};
4)   int main(){
5)     for(;i < 5; i++){
6)         if (v[i] > m)
7)             m = v[i];
8)     }
9)     return m;
10)  }
```

Sabendo como manipular vetores em *assembly*, agora podemos **traduzir o restante do nosso programa** inicial!

Vamos fazer esse processo usando tanto ENDEREÇAMENTO INDIRETO quanto ENDEREÇAMENTO INDEXADO.

# Vetores em *Assembly*

```c
long int i = 0;
long int m = 0;
long int v[5] = {10, 30, 5, 10, 50};
int main(){
  for(;i < 5; i++){
      if (v[i] > m)
          m = v[i];
  }
  return m;
}
```

≡

```asm
.section .data
i: .quad 0
m: .quad 0
v: .quad 10, 30, 5, 10, 50
.section .text
.global _start
_start:
movq i, %rax
movq m, %rdi
movq $v, %rbx
_loop_i:
cmp $5, %rax
je _loop_e
cmp %rdi, (%rbx)
jle _if_e
movq (%rbx), %rdi
_if_e:
addq $8, %rbx
addq $1, %rax
jmp _loop_i
_loop_e:
movq $60, %rax
syscall
```

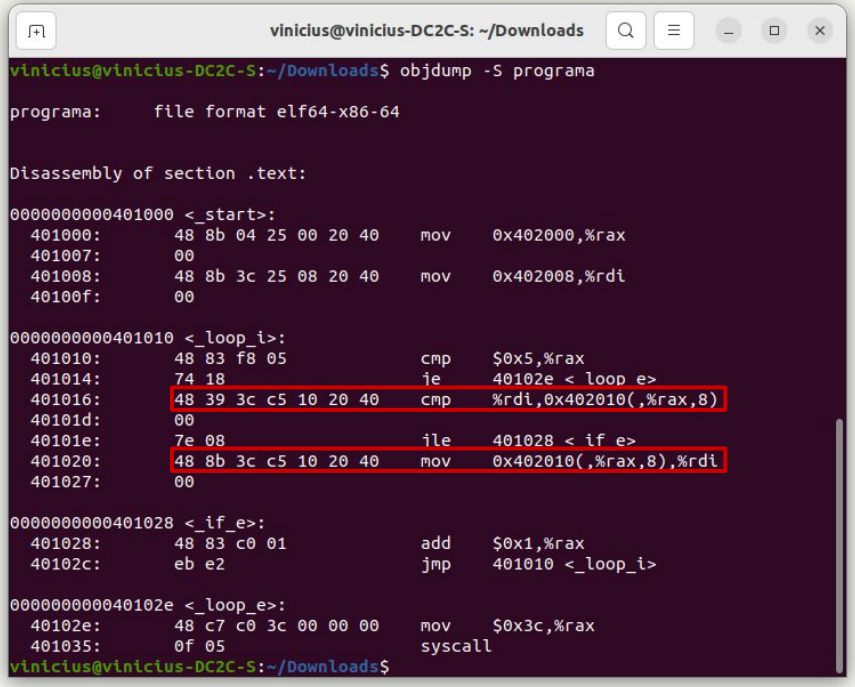
# Vetores em *Assembly*

```
1)   long int i = 0;
2)   long int m = 0;
3)   long int v[5] = {10, 30, 5, 10, 50};
4)   int main(){
5)     for(;i < 5; i++){
6)         if (v[i] > m)
7)             m = v[i];
8)     }
9)     return m;
10)  }
```

≡

```
1)   .section .data
2)   i: .quad 0
3)   m: .quad 0
4)   v: .quad 10, 30, 5, 10, 50
5)   .section .text
6)   .global _start
7)   _start:
8)   movq i, %rax
9)   movq m, %rdi
10)  _loop_i:
11)  cmp $5, %rax
12)  je _loop_e
13)  cmp %rdi, v(, %rax, 8)
14)  jle _if_e
15)  movq v(, %rax, 8), %rdi
16)  _if_e:
17)  addq $1, %rax
18)  jmp _loop_i
19)  _loop_e:
20)  movq $60, %rax
21)  syscall
```

# Laços de Repetição




## Vamos analisar o binário...

Analisando as instruções, temos que:

- 48: código para reg. 64 bits
- 39 3c: *compare with* %rdi
- 8b 3c: *move to* %rdi
- C5 10 20 40: memória (base + deslocamento)

# Exercício #04

Faça um programa em *assembly* AMD64 que contemple um **vetor global inicializado com dez (10) elementos (.quad) quaisquer**. Percorra o vetor verificando se os elementos são ímpares ou pares. Ao final, o programa deve **retornar número de elementos pares** presentes no vetor.

Para resolver esse problema, considere utilizar o operador “and” (realiza uma conjunção binária entre seus argumentos):

```
and arg1 arg2
```

# Obrigado!

Vinícius Fülber Garcia
inf.ufpr.br/vinicius/
viniciusfulber@ufpr.br