# Arrays

João Marcelo Uchôa de Alencar

18 de maio de 2023



1/35

2/35

# Introdução

- Até agora, não mostramos nada de *arrays*.
- Vamos mostrar:
  - Declaração.
  - Acesso.
  - Indexação.
  - Entrada e Saída.
- Existem várias maneiras de indexar um *array*, vamos mostrar apenas duas.
- Neste capítulo, tratamos de *arrays* de *signed double words* (sdword).
- No capítulo de *strings* falaremos de *arrays* de *bytes*.

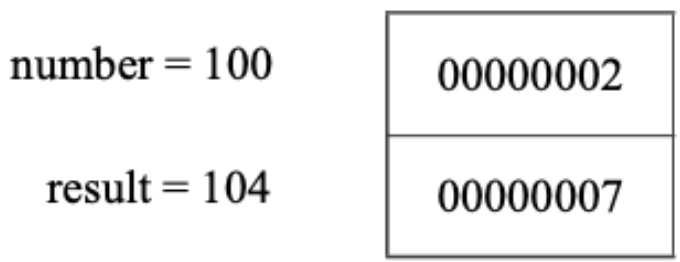

3/35

# Endereçamento e Declaração de *Arrays*

- A maneira mais simples de declarar um *array* é listar localização de memória após localização de memória.
- É possível, a partir de uma variável, acessar outra adicionando uma constante ao endereço.

```
.data
number sdword 2
result sdword 7
```

- *mov eax, result* e *mov eax, number+4* são a mesma coisa (*aliasing*).
- Veja que *number+4* não é a mesma coisa que *add number, 4*.





4/35

# Endereçamento e Declaração de *Arrays*

- *Aliasing* não é uma boa técnica, pois pode deixar o código confuso.

```
.data
number sdword 2
answer sdword 5
result sdword 7
```

- Toda referência a *number* + *4* agora está invalidada.
- Para criar *arrays*, não usamos novos nomes para os endereços de memória seguintes:

```
.data
numary sdword 2
       sdword 5
       sdword 7
```

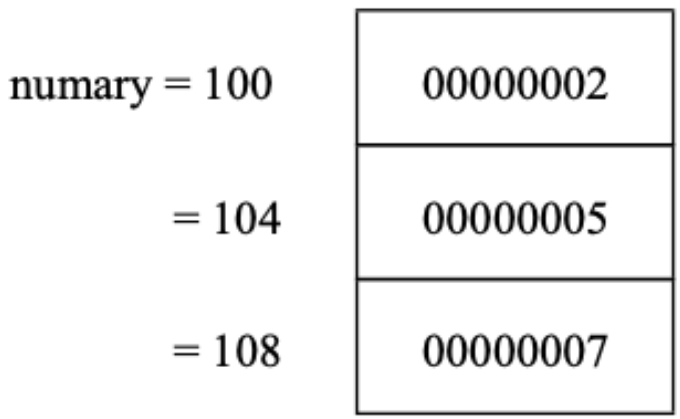

5/35

# Endereçamento e Declaração de *Arrays*

- Uma maneira mais simples de declaração usa apenas uma linha:

```
.data
number sdword 2, 5, 7
```

- Em ambos os casos, a organização de memória é a mesma:





6/35

# Endereçamento e Declaração de *Arrays*

- Inicializando com um mesmo valor ou sem valores definidos:

```
.data
zeroary sdword 0,0,0
empary  sdword ?,?,?
```

- E se forem centenas de elementos? Usamos o operador *dup*:

```
.data
zeroary sdword 100 dup(0)
empary  sdword 100 dup(?)
```


7/35

# Endereçamento e Declaração de *Arrays*

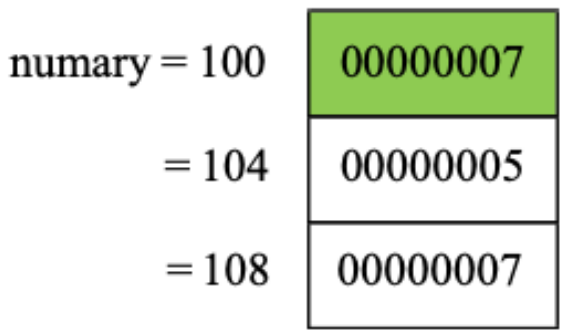
- Como faríamos em *assembly*?

```
numary[0] = numary[2];
```

- Lembrando que em C o *array* começa em 0:

```
mov eax, numary + 8 ; load eax with third element
mov numary + 0, eax ; store eax in first element
```

- Transferências entre posições de memória ainda precisam passar por registradores.





8/35

# Indexando Usando o Registrador Base

- *ebx* é conhecido com o registrador base.
- Lembrando do exemplo:

```
.data
numary sdword 2,5,7
sum    sdword ?
```

- A soma de todos os elementos ficaria assim em C:

```
sum = 0;
for (i = 0; i < 3; i++)
   sum += numary[i];
```


9/35

# Indexando Usando o Registrador Base

- Soma de todos os elementos em *assembly*:

```
mov sum, 0              ; initialize sum to 0
mov ecx, 3              ; initialize ecx to 3
mov ebx, 0              ; initialize ebx to 0
.repeat
mov eax, numary[ebx] ; load eax with element of numary
add sum, eax            ; add eax to sum
add ebx, 4              ; increment ebx by 4
.untilcxz
```

- *ebx* precisa ser incrementado em 4 para acessar o próximo elemento.
- Devemos preservar *ecx* pois ele controla o laço.


10/35

# Indexando Usando o Registrador Base


- *ebx* tem o valor hexadecimal 0000000C (12 em decimal) após o código acima ser executado.
- Ele poderia acabar sendo usado para acessar *sum* de forma equivocada.
- O programador precisa ser cuidadoso para evitar que isso ocorra.


11/35

# Indexando Usando o Registrador Base

```
int array[20], n, i;
printf("\n%s",
            "Enter the number of integers to be input: ");
scanf("%d", &n);
if (n > 0) {
        for (i = 0; i < n; i++) {
                printf("\n%s", "Enter an integer: ");
                scanf("%d", &array[i]);
        }
        printf("\n%s\n\n", "Reversed");
        for (i = n - 1; i >= 0; i--)
                printf("   %d\n\n", array[i]);
} else
        printf("\n%s\n\n", "No data entered.");
```


12/35

# Indexando Usando o Registrador Base

```
        .data
msg1fmt   byte    0Ah, "%s", 0
msg2fmt   byte    0Ah, "%s", 0Ah, 0Ah, 0
msg3fmt   byte    "   %d", 0Ah, 0Ah, 0
in1fmt    byte    "%d", 0
msg1      byte    "Enter the number of
                   integers to be input: ", 0
msg2      byte    "Enter an integer: ", 0
msg3      byte    "Reversed", 0
msg4      byte    "No data entered."
n         sdword  ?
array     sdword  20 dup (?)
        .code
```

```
main proc
     INVOKE printf, ADDR msg1fmt, ADDR msg1
     INVOKE scanf, ADDR in1fmt, ADDR n
     mov ecx, n     ; initialize ecx to n
     mov ebx, 0     ; initialize ebx to 0
     .if ecx > 0
     .repeat
     push ecx       ; save ecx
     INVOKE printf, ADDR msg1fmt, ADDR msg2
     INVOKE scanf, ADDR in1fmt, ADDR array[ebx]
     pop ecx        ; restore ecx
     add ebx, 4     ; increment ebx by 4
     .untilcxz
     INVOKE printf, ADDR msg2fmt, ADDR msg3
     mov ecx, n     ; initialize ecx to n
     sub ebx, 4     ; subtract 4 from ebx
     .repeat
     push ecx       ; save ecx
     INVOKE printf, ADDR msg3fmt, array[ebx]
     pop ecx        ; restore ecx
     sub ebx, 4     ; decrement ebx by 4
     .untilcxz
     .else
     INVOKE printf, ADDR msg2fmt, ADDR msg4
     .endif
     ret
main endp
     end
```


13/35

# Busca

- Busca sequencial (dados desordenados) e busca binária (dados ordenados).
- Vamos mostrar a busca sequencial:
  - Dados em um *array*.
  - Não há duplicatas.
  - O tamanho do *array* é conhecido.


14/35

# Busca

```
int array[20], n = 20, number, found;
printf("\n%s", "Enter the integer to be found: ");
scanf("%d", &number);
i = 0;
found = 0;
while (i < n && !found)
        if (number == array[i])
                found = -1;
        else
                i++;
if (found)
        printf("\n%s\n\n", "The integer was found");
else
        printf("\n%s\n\n", "The integer was not found");
```


15/35

# Busca

```
                .686
                .model  flat, c
                .stack  100h
printf    PROTO   arg1:Ptr Byte, printlist:VARARG
scanf     PROTO   arg2:Ptr Byte, inputlist:VARARG
                .data
msg1fmt   byte    "%s", 0
msg2fmt   byte    0Ah, "%s", 0Ah, 0Ah, 0
in1fmt    byte    "%d", 0
msg1      byte    "Enter the integer to be found: ", 0
msg2      byte    "The integer was found ", 0
msg3      byte    "The integer was not found", 0
array     sdword  20,19,18,17,16,15,14,13,12,11,
                  1,2,3,4,5,6,7,8,9
n         sdword  20
number    sdword  ?
found     sbyte   ?
```

```
                .code
main      proc
          INVOKE printf, ADDR msg1fmt, ADDR msg1
          INVOKE scanf, ADDR in1fmt, ADDR number
          mov ebx, 0          ; initialize ebx to 0
          mov ecx, 0          ; initialize ecx to 0
          mov edx, number     ; load edx with number
          mov found, 0        ; initialize found to 0
          .while (ecx < n && !found)
          .if (edx == array[ebx])
          mov found, -1       ; set found to -1
          .else
          add ebx, 4          ; increment ebx by 4
          .endif
          inc ecx             ; increment ecx by 1
          .endw
          .if (found)
          INVOKE printf, ADDR msg2fmt, ADDR msg2
          .else
          INVOKE printf, ADDR msg2fmt, ADDR msg3
          .endif
          ret
main      endp
          end
```


16/35

# Indexando Usando os Registradores *esi* e *edi*

- *esi*: *source index register*.
- *edi*: *destination index register*.
- Funcionam de maneira semelhante a ponteiros.
- O endereço do *array* é carregado em um registrador e o nome da variável não é mais usado.
- Colocar o segundo elemento do *array numary* em *eax*:

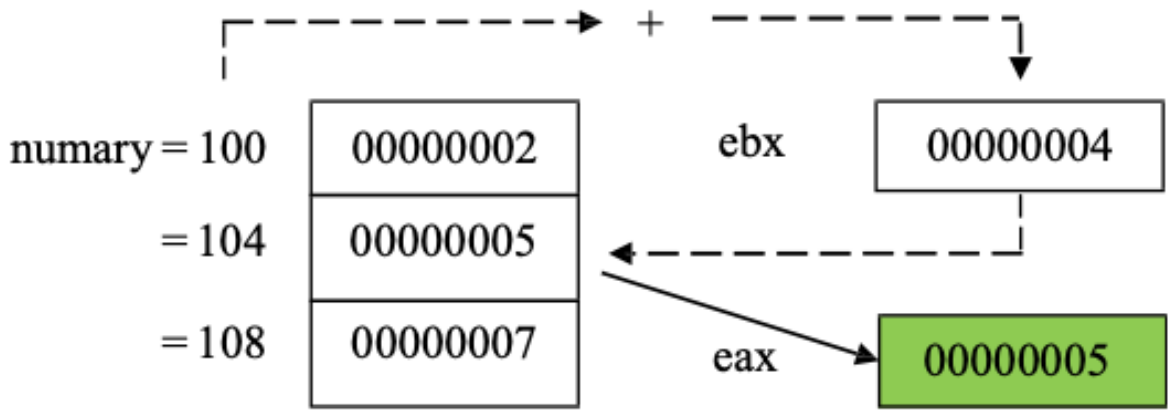
```
mov eax, numary + 4
```

- Podemos fazer a mesma coisa usando um registrador como índice:

```
mov ebx, 4
mov eax, numary[ebx]
```


17/35

# Indexando Usando os Registradores *esi* e *edi*





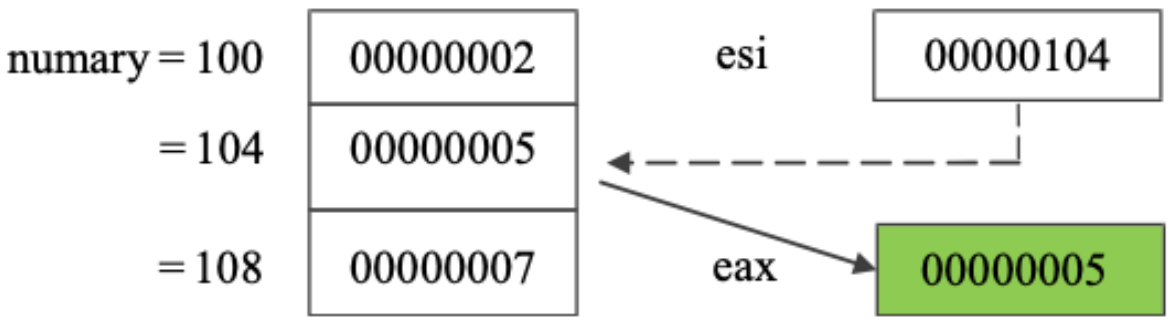

18/35

# Indexando Usando os Registradores *esi* e *edi*

- Usando o registrador *esi* e o operador *offset*:

```
mov esi, offset numary+4
mov eax, [esi]
```

- No lugar de copiar o conteúdo de *numary+4* no registrador *esi*, o operador *offset* carrega o endereço de *numary+4*.
- O endereço em *esi* pode ser usado como um ponteiro.





19/35

# Indexando Usando os Registradores *esi* e *edi*

- Instrução *lea* - *load effective address*:

```
lea esi, numary+4
mov eax, [esi]
```

- É uma alternativa ao operador *offset*.
- Quando usamos *offset*, o endereço é calculado na montagem, com *lea* o endereço é calculado em tempo de execução.
- Se no lugar de *numary* tivéssemos um registrador, teríamos que usar *lea*.


20/35

# Indexando Usando os Registradores *esi* e *edi*

- Nos exemplos anteriores, tanto o *edi* quanto o *edi* poderiam ser usados.
- Como regra geral:
  - Quando for para recuperar dados da memória, use o *esi*.
  - Quando for para salvar dados na memória, use o *edi*.
- Movendo dados de uma posição de memória para outra:

```
mov eax, num1
mov num2, eax
```

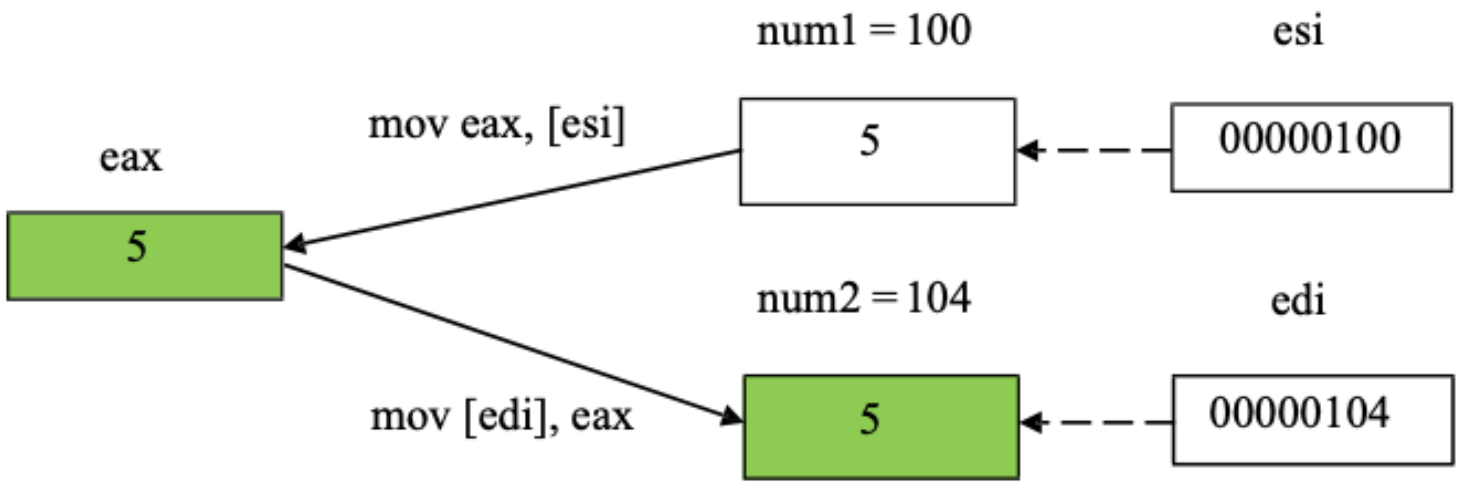
- Usando *esi* e *edi*:

```
lea esi, num1  ; load the address of num1 into esi
lea edi, num2  ; load the address of num2 into edi
mov eax, [esi] ; move contents of where esi
               ; is pointing to eax
mov [edi], eax ; move contents of eax to
               ; where edi is pointing
```


21/35

# Indexando Usando os Registradores *esi* e *edi*

- *mov [edi], [esi]* é memória para memória.





22/35

# Indexando Usando os Registradores *esi* e *edi*

```
sum = 0;
for(i = 0; i < 3; i++)
        sum += numary[i];
```

- Versão em MASM:

```
mov sum, 0          ; initialize sum to zero
mov ecx, 3          ; initialize ecx to 3
lea esi, numary + 0 ; load the address of numary into esi
.repeat
mov eax, [esi]      ; move contents of where esi is poiting to eax
add sum, eax        ; add eax to sum
add esi, 4          ; increment esi by 4 to next element
.untilcxz
```


23/35

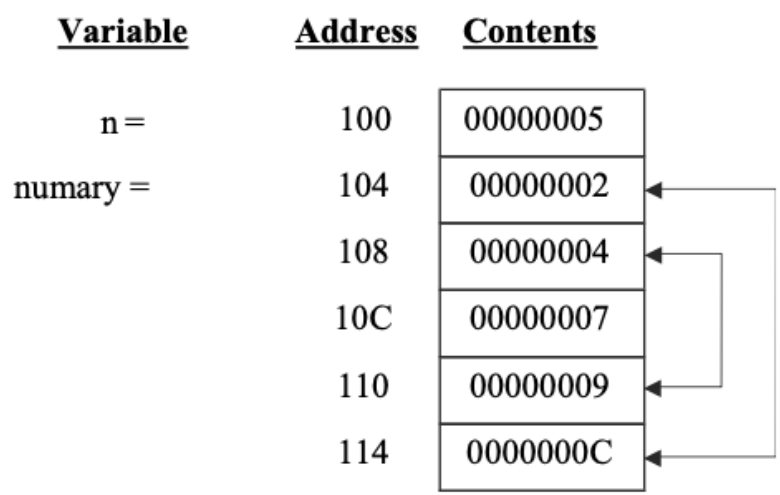
# Indexando Usando os Registradores *esi* e *edi*

- Inverter o seguinte *array*:

```
n       sdword 5
numary  sdword 2,4,7,9,12
```

- Versão em C:

```
j = n - 1;
for (i = 0; i < n / 2; i++) {
    temp = numary[i];
    numary[i] = numary[j];
    numary[j] = temp;
    j--;
}
```





24/35

# Indexando Usando os Registradores *esi* e *edi*

```
mov ecx, n              ; load ecx with contents of n
sar ecx, 1              ; divide ecx by 2, number of times to loop
lea esi, numary + 0     ; load address of numary into esi
mov edi, esi            ; move contents of esi to edi
mov eax, n              ; load eax with contents of n
dec eax                 ; decrement eax by one
sal eax, 2              ; multiply eax by 4
add edi, eax            ; add eax to edi for ending address of array
.repeat
mov eax, [esi]          ; move contents where esi is pointing to eax
xchg eax, [edi]         ; exchange eax and where edi is pointing
mov [esi], eax          ; move eax to where edi is pointing
add esi, 4              ; add four to esi for next element
sub edi, 4              ; subtract four from edi for next element
.untilcxz
```


25/35

# Operadores *Lengthof* e *Sizeof*

- *lengthof* retorna a quantidade de elementos de um *array*.
- *sizeof* retorna quantos *bytes* o *array* ocupa.
- Esses operadores são acionados em tempo de montagem.
- Considere o problema de encontrar o último elemento do *array*:

```
mov eax, n          ; load eax with contents of n
dec eax             ; decrement eax by one
sal eax, 2          ; multiply eax by 4
```

- Poderia ser escrito como:

```
mov eax, sizeof numary ; load eax with the size of numary
sub eax, 4             ; decrement eax by four
```


26/35

# Indexando Usando os Registradores *esi* e *edi*

```
mov ecx, lenghtof numary ; load ecx with length of numary
sar ecx, 1               ; divide ecx by 2, number of times to loop
lea esi, numary + 0      ; load address of numary into esi
mov edi, esi             ; move contents of esi to edi
mov eax, sizeof numary   ; load eax with size of numary
sub eax, 4               ; decrement eax by four
add edi, eax             ; add eax to edi for ending address of array
.repeat
mov eax, [esi]           ; move contents where esi is pointing to eax
xchg eax, [edi]          ; exchange eax and where edi is pointing
mov [esi], eax           ; move eax to where edi is pointing
add esi, 4               ; add four to esi for next element
sub edi, 4               ; subtract four from edi for next element
.untilcxz
```


27/35

# Programa Completo: Implementando uma Fila

- Filas - Estrutura FIFO.
- É uma estrutura presente em várias partes de um sistema operacional.
- Em geral, é usada quando um elemento mais rápido precisa enviar dados para um elemento mais lento.
- Operações para adicionar e remover elementos.
- Veja as versões em C e MASM no repositório.


28/35

# Programa Completo: Implementando *Selection Sort*

- Algoritmos de ordenação *in-place*:
  - Dois laços aninhados.
  - Complexidade $O(n^2)$.
- Vamos mostrar o *Selection Sort*:
  - Dois laços, um *if* e um procedimento *swap*.
  - Pode ser aprimorado para evitar a troca desnecessária de elementos.
  - Percorrer o *array* $n - 1$ vezes.
  - Na primeira passagem, o primeiro elemento é comparado com todos os outros.
  - Se um elemento for menor, há a troca. Ao final, o menor elemento estará na primeira posição.
  - O processo continua no segundo, terceiro, etc, elemento.

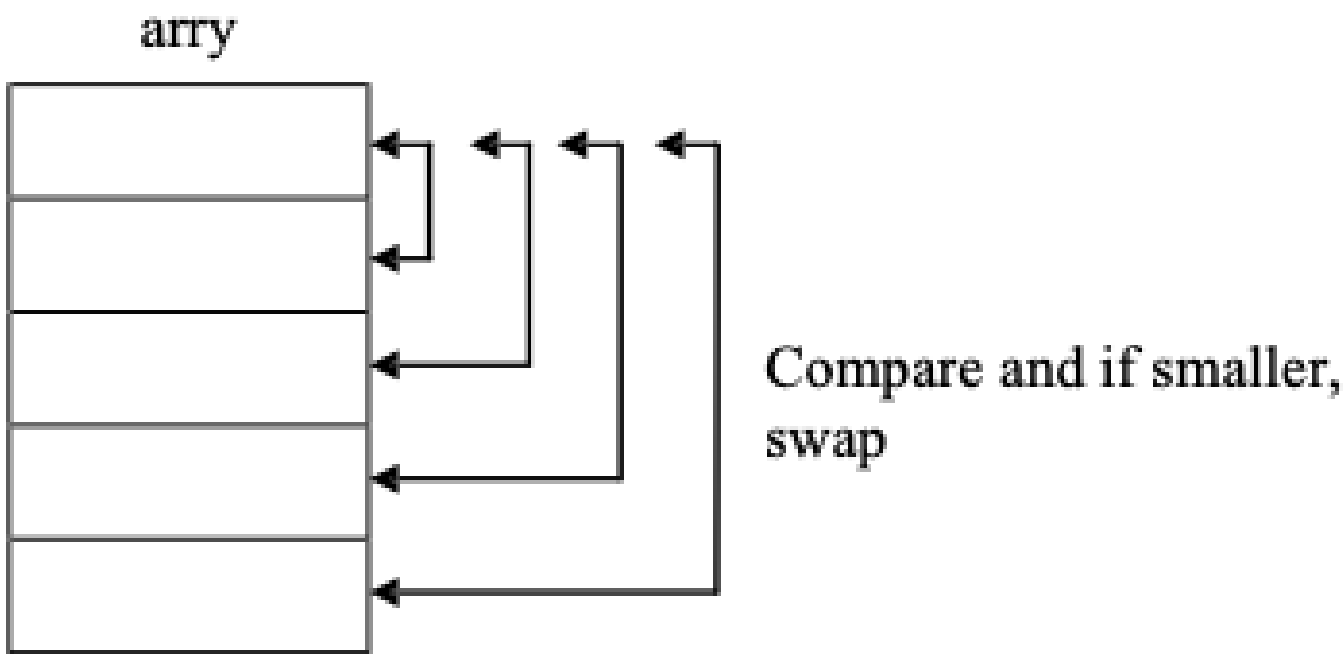

29/35

# Programa Completo: Implementando *Selection Sort*

```
// number of passes
for (i = 0; i < n - 1; i++)
   // number of comparisons
   for (j = i + 1; j < n; j++)
      // compare the ith and jth element
         if (array[j] < array[i]) {
               // swap the elements
               temp = array[j];
               array[j] = array[i];
               array[i] = temp;
         }
```


30/35

# Programa Completo: Implementando *Selection Sort*





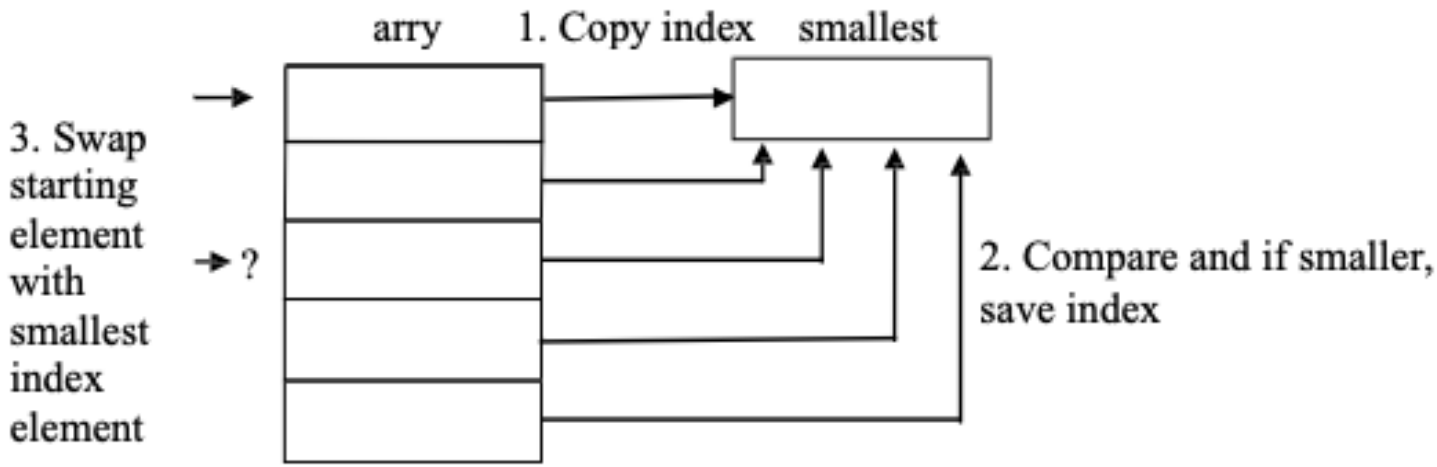
31/35

# Programa Completo: Implementando *Selection Sort*

```
// number of passes
for (i = 0; i < n - 1; i++)
   // save index of first element of pass in smallest
   smallest = i;
   // number of comparisons
   for (j = i + 1; j < n; j++)
      // compare the jth with smallest
         if (array[j] < array[smallest])
               // save the new smallest element
                     smallest = j;
         // swap first element of pass with smallest
         temp = array[j];
         array[j] = array[i];
         array[i] = temp;
```


32/35

# Programa Completo: Implementando *Selection Sort*





33/35

# Programa Completo: Implementando *Selection Sort*

- Os índices *i* e *j* podem utilizar os registradores *esi* e *edi*.
- Para termos dois laços, teremos que preserver *ecx* na pilha para podermos usar as diretivas *.repeat-.untilcxz*.
- O código MASM está no repositório.


34/35

# Resumo

- O operador *dup* ajuda na declaração de *arrays* grandes.
- O registrador *ebx* pode ser usado como índice de um *array*.
- Os registradores *esi* e *edi* podem ser usados como ponteiros.
- Quando lidando com *arrays* de *sdword*, o incremento entre os elementos é 4.
- Os operadores *offset* e *lea* permitem recuperar o endereço de uma variável.
- Use colchetes ao redor dos registradores *ebx*, *esi* e *edi* para recuperar o conteúdo da localização de memória.
- O operador *lengthof* retorna o tamanho do *array*, enquanto o operador *sizeof* retorna a quantidade de *bytes*.


35/35