Padrões Pedagógicos

19 de Abril de 2007

Lista de Padrões
Padrões: convenções da linguagem

1. PROGRAMA SIMPLES
2. DECLARAÇÃO DE VARIÁVEL
3. CONJUNTO DE AÇÕES
4. IMPRESSÃO
5. LEITURA
6. ATRIBUIÇÃO
7. EXPRESSÃO
8. CONDIÇÃO SIMPLES
9. CONDIÇÃO COMPOSTA
10. DECLARAÇÃO DE VETOR
11. FUNÇÕES LÓGICAS
12. DECLARAÇÃO DE VETOR

Padrões: Estratégias

1. SELEÇÃO SIMPLES
2. SELEÇÃO ALTERNATIVA
3. SELEÇÃO SEQUENCIAL
4. SELEÇÃO MULTIPLA
5. REPETIÇÃO
6. REPETIÇÃO CONTADA

7. REPETIÇÃO COM SENTINELA
8. REPETIÇÃO COM INDICADOR DE PASSAGEM
9. FUNÇÃO
10. BUSCA EM VETORES

nome do padrão *Programa Simples*

---

uso/aplicação Você quer criar um novo programa C.

---

padrões dependentes *declaração-de-variavel*, *conjunto-de-ações*.

---

sintaxe

```
/* <texto-de-comentarios> */
<incluir-bibliotecas>

int main() {
   <declarações-de-variáveis>
   <conjunto-de-ações>
   return 0;
}
```

---

semântica A execução do programa começa pelas declarações, no começo de `main` e passa pelo conjunto de ações, até atingir `return 0;`, onde o programa termina.

```
[ coment\´arios ]

#include <stdio.h>

int main() {
   [ declaraç\~oes
          ⇓
   [ conjunto de a
          ⇓
   return 0;
}
```

Para uma leitura clara do programa aconselha-se identar o código.

---

exemplo de uso Um programa que diz "Olá".

```
/*     Programa 'Ola'    */
/* Autor: Lucas Ferreira */
#include <stdio.h>

int main() {
   printf("Olá\n");

   return 0;
}
```

nome do padrão *Declaração de Variável*

---

uso/aplicação Você quer dar um nome a uma variável, para uso posterior.

---

padrões dependentes Nenhum.

---

sintaxe 1 Declaração de uma variável

```
[tipo] [nome da variável];
```

sintaxe 2 Declaração de `N` variáveis inteiras

```
[tipo] [nome da variável1], [nome da variável2], ... , [nome da variávelN];
```

ou

```
[tipo] [nome da variável1];
[tipo] [nome da variável2];
   ...
[tipo] [nome da variávelN];
```

sintaxe 3 Declaração e inicialização

```
[tipo] [nome da variável] = [valor];
```

[tipo] pode ser: `int` (para inteiro),`float` (para real), `char`(para um caracter)

---

semântica A declaração de uma variável, reserva um espaço na memória para armazenar um valor de um tipo de dado. Quando o nome da variável aparece em uma expressão, o valor dela é recuperado e substituido nessa expressão.

---

exemplo de uso

É importante dar um nome *mnemônico* à variável, ou seja, um nome cujo significado em português corresponda ao uso da variável no programa.
Criar uma variável para guardar a soma de números reais:

```
float soma;
```

Criar uma variável para contar os elementos de uma seqüência de números:

```
int cont = 0;
```

---

nome do padrão *Conjunto de Ações*

---

uso/aplicação Você quer efetuar uma ou mais ações, seqüencialmente.

---

padrões dependentes Nenhum.

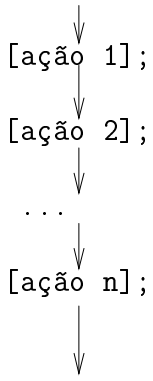
---

sintaxe

```
[ação 1];
[ação 2];
  ...
[ação n];
```

---

semântica As ações são executadas seqüencialmente, da "ação 1" até a "ação n".



---

exemplo de uso Atribuir 5 à variável `n` e imprimir o seu valor.

```
n = 5;
printf("%d\n", n);
```

---

nome do padrão *Impressão*

---

uso/aplicação Você quer exibir alguma informação na tela (texto e/ou valores).

---

padrões dependentes *expressão.*

---

sintaxe 1 Escrever um texto:

```
printf("[texto]\n");
```

sintaxe 2 Escrever o valor de uma expressão inteira.

```
printf("[texto] %[tipo] [texto]\n", [expressão]);
```

sintaxe 3 Escrever o valor de várias expressões-inteiras

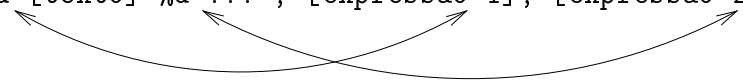
```
printf("[texto] %[tipo] [texto] %[tipo] [texto] ...\n",
[expressão 1], [expressão 2], ...);
```

Onde [texto] pode ser vazio. [tipo] pode ser *d* para valores inteiros, *f* para reais e *c* para caracter.

---

semântica O texto é exibido na tela. Ocorrências de `%[tipo]` serão substituidas pelo valor da expressão respectiva, dessa forma:

```
printf("[texto] %d [texto] %d ...", [expressão 1], [expressão 2], ...);
```



---

exemplo de uso

1. Exibir o texto “Bom dia !”.

```
printf("Bom dia !\n");
```

2. Exibir o texto “O valor de 2 * n é ” seguido do valor de `2 * n`.

```
printf("O valor de 2 * n é %d \n", 2 * n);
```

3. Exibir os valores de a, b e de a+b, na forma “a + b = a+b”.

```
printf("%d + %d = %d \n", a, b, a + b);
```

---

nome do padrão *Leitura*

---

uso/aplicação Você quer pedir para o usuário entrar com um número, e armazenar o número digitado em uma variável.

---

padrões dependentes Nenhum.

---

sintaxe

```
printf("[texto] : \n");
scanf("%[tipo]", &[nome da variável]);
```

[tipo] pode ser *d* para valores inteiros, *f* para reais e *c* para caracter.

---

semântica O texto, um pedido ao usuário, é exibido na tela. Em seguida o valor é lido do teclado e armazenado na variável.

---

exemplo de uso Ler um número inteiro.

```
printf("Digite um número : ");
scanf("%d", &numero);
```

---

nome do padrão *Atribuição*

---

uso/aplicação Você quer guardar um valor para uso posterior, dando-lhe um nome.

---

padrões dependentes *expressão*.

---

sintaxe

```
[nome-da-variável] = [expressão];
```

O tipo da expressão deve ser o mesmo que o da variável.

---

semântica A expressão é avaliada, e uma cópia do valor resultante é armazenada na posição de memória referenciada pelo nome da variável.

---

exemplo de uso Guardar o resultado de 2 + 3 na variável `total`.

```
total = 2 + 3;
```

---

nome do padrão *Expressão*

---

uso/aplicação Você quer avaliar uma expressão que contém números e/ou variáveis. Geralmente é usada como parte de uma atribuição ou condição.

---

padrões dependentes Nenhum.

---

sintaxe 1

```
[nome-da-variável ou número] [operador-aritmético]
[nome-da-variável ou número] [operador-aritmético]
[nome-da-variável ou número] ... ;
```

sintaxe 2

```
[nome-da-variável] [operador-lógico]
[nome-da-variável] [operador-lógico]
[nome-da-variável] ... ;
```

sintaxe 3

```
[nome-da-variável]
```

Operadores aritméticos podem ser:

`+ , - , * , / , %.`

O operador % devolve o resto de uma divisão de números inteiros. Os delimitadores de expressões '(' e ')' podem ser usados para priorizar operações.
Operadores lógicos podem ser:

`&&(e) ||(ou) !(não).`

---

semântica A expressão é avaliada e o resultado é armazenado em uma posição temporária de memória (ou acumulador). Operadores aritméticos são avaliados em C, de acordo com suas prioridades. Operadores de mesma prioridade são avaliados da esquerda para direita. A prioridade entre os operadores aritméticos diferentes é definida como:

```
maior prioridade: * , / , %
menor prioridade: + , -
```

---

exemplo de uso

```
    a = 2 * 3;
    total = a + 4 * 3 / 2;      /* o resultado e' 12 */
    total = (a + 4) * 3 / 2;    /* o resultado e' 15 */
```

---

nome do padrão *Condição simples*

---

uso/aplicação Você quer verificar se uma variável satisfaz uma dada propriedade. É usada como condição da seleção ou laço.

---

padrões dependentes expressão.

---

sintaxe 1

```
[expressão] [operador-relacional] [expressão]
```

sintaxe 2

```
[expressão]
```

Operadores relacionais podem ser:
> (maior), < (menor), >= (maior ou igual), <= (menor ou igual), == (igual) , ! = (diferente).

---

semântica A condição é verificada e o resultado, verdadeiro ou falso, é armazenado em uma posição temporária de memória (ou acumulador). Os valores, verdadeiro e falso, são representados na memória pelos valores inteiros '1' e '0', respectivamente.

---

exemplo de uso

```
    a >= 0
ou
    a == b
ou
    (soma + 1) != n
```

---

nome do padrão *Condição composta*

---

uso/aplicação Você quer verificar se um conjunto de variaveis satisfacem algumas dadas propriedades. É usada como condição da seleção ou laço.

---

padrões dependentes condição simples.

---

sintaxe

```
[condição-simples] [operador-lógico] [condição-simples]...
```

Operadores lógicos podem ser:

```
 &&(e) ||(ou) !(não).
```

---

semântica A condição composta é verificada e o resultado, verdadeiro ou falso, é armazenado em uma posição temporária de memória (ou acumulador). Os valores, verdadeiro e falso, são representados na memória pelos valores inteiros '1' e '0', respectivamente.
Operadores relacionais e lógicos são avaliados em C, de acordo com suas prioridades. Operadores de mesma prioridade são avaliados da esquerda para direita. A prioridade entre os operadores relacionais e lógicos é definida como:

```
maior prioridade: !
< , > , >= , <=
== , !=
&&
menor prioridade:||
```

---

exemplo de uso

```
ou
    a >= 0 && b < 100
ou
    (soma + 1) != n || n > 200
```

---

nome do padrão *Seleção Simples*

---

uso/aplicação Você quer selecionar um conjunto de ações para ser executado somente se uma condição for satisfeita.

---

padrões dependentes *Conjunto de ações.*

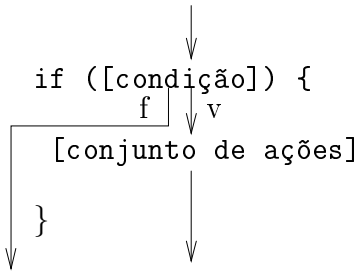
---

sintaxe

```
if ( [condição] ) {
   [conjunto de ações];
}
```

---

semântica A condição é testada. Se ela for satisfeita, então o conjunto de ações será executado. Caso contrário, nenhuma ação será executada.



---

exemplo de uso Dada a temperatura de uma pessoa, dizer se ela está com febre.

```
if ( temperatura >= 37 ) {
   printf("O paciente está com febre\n");
}
```

---

lista de usos (do aluno)

nome do padrão *Seleção Alternativa*

---

uso/aplicação Você quer selecionar para executar um entre dois conjuntos de ações alternativos, baseado em uma condição.

---

padrões dependentes *Conjunto de Ações.*

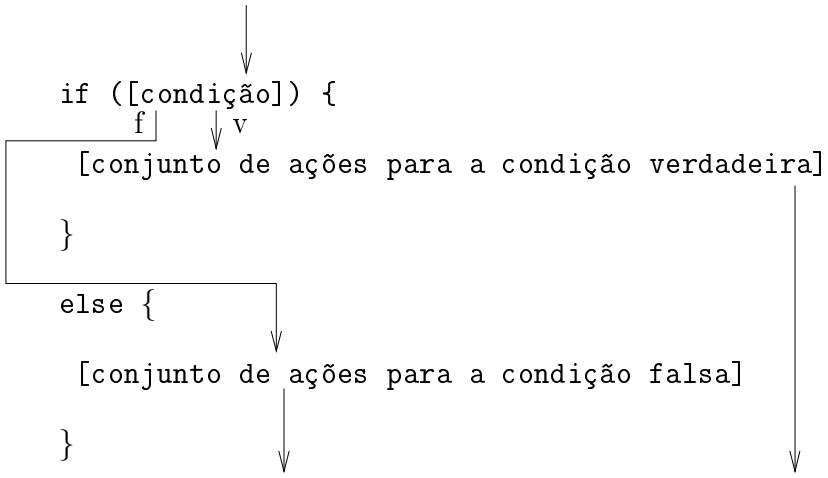
---

sintaxe

```
if ( [condição] ) {
   [ conjunto de ações para a condição verdadeira ]
}
else {
   [ conjunto de ações para a condição falsa ]
}
```

---

semântica A condição é testada. Se ela for satisfeita, então o conjunto de ações da primeira parte será executado. Caso contrário, o conjunto de ações da segunda parte é que será executado.



---

exemplo de uso Dada uma nota, dizer se o aluno foi aprovado (média >= 5) ou reprovado (caso contrário).

```
if ( nota >= 5 ) {
   printf("O aluno foi aprovado\n");
}
else
   printf("O aluno foi reprovado\n");
}
```

---

lista de usos (do aluno)

nome do padrão *Seleção Seqüencial*

---

uso/aplicação Você quer selecionar um ou nenhum conjunto de ações alternativo, baseado em diferentes condições.

---

padrões dependentes *Conjunto de Ações.*

---

sintaxe 1
Executar um ou nenhum conjunto de ações:

```
if ( [condição 1] ) {
   [ conjunto de ações para a cond. 1 ]
}
else if ( [condição 2] ) {
   [ conjunto de ações para a cond. 2 ]
}
   ...
else if ( [condição n] ) {
   [ conjunto de ações para a cond. n ]
}
```

sintaxe 2
Executar exatamente um conjunto de ações:

```
if ( [condição 1] ) {
   [ conjunto de ações para a cond. 1 ]
}
else if ( [condição 2] ) {
   [ conjunto de ações para a cond. 2 ]
}
   ...
else if ( [condição n] ) {
   [ conjunto de ações para a cond. n ]
}
else {
   [ conjunto final de ações ]
}
```

---

semântica para a sintaxe 1 Cada condição é testada em seqüência, em busca de alguma que seja satisfeita. Se alguma for encontrada, o conjunto de ações correspondente será executado, e a busca termina. Caso contrário, nenhuma ação será executada.
semântica para a sintaxe 2 Cada condição é testada em seqüência, em busca de alguma que seja satisfeita. Se alguma for encontrada, o conjunto de ações correspondente será executado, e a busca termina. Caso contrário, o conjunto final de ações será executado.

exemplo de uso

Dada uma nota, dizer se o aluno foi aprovado (média >= 5), ficou de recuperação (média >= 3), ou foi reprovado (nenhum dos dois casos ocorreu).

```
if ( nota >= 5 ) {
   printf("O aluno foi aprovado\n");
}
else if ( nota >= 3 ) {
   printf("O aluno ficou de recuperação\n");
}
else {
   printf("O aluno foi reprovado\n");
}
```

---

lista de usos (do aluno)

nome do padrão *Seleção multipla*

---

uso/aplicação Você quer selecionar um ou nenhum conjunto de ações alternativo, baseado no valor de uma expressão.

---

padrões dependentes *Expressão , Conjunto de Ações.*

---

sintaxe 1
Executar um ou nenhum conjunto de ações:

```
switch(<expressão>){
   case <valor1>:
          < conjunto-de-ações-para-valor1>
          break;
   case <valor2>:
          < conjunto-de-ações-para-valor2>
          break;
   ...
   case <valorN>:
          < conjunto-de-ações-para-valorN>
          break;
   }
```

sintaxe 2
Executar exatamente um conjunto de ações:

```
switch(<expressão>){
   case <valor1>:
          < conjunto-de-ações-para-valor1>
          break;
   case <valor2>:
          < conjunto-de-ações-para-valor2>
          break;
   ...
   case <valorN>:
          < conjunto-de-ações-para-valorN>
          break;
   default:
          <conjunto final de ações>
   }
```

---

semântica para a sintaxe 1 A condição <expressão>==<valorI> é testada em seqüência, em busca de alguma que seja satisfeita. Se alguma for encontrada, o conjunto de ações correspondente será executado, e a busca termina. Caso contrário, nenhuma ação será executada.

semântica para a sintaxe 2 A condição <expressão>==<valorI> é testada em seqüência, em busca de alguma que seja satisfeita. Se alguma for encontrada, o conjunto de ações correspondente será executado, e a busca termina. Caso contrário, va para **default** e o conjunto final de ações será executado.
<valor> é uma constante inteira, um carater o uma expressão constante, no último caso o valor resultante tem que ser inteiro.
break termina a execução da sentença **switch**.

---

exemplo de uso
Dado o tipo de vehículo, determinar o monto a pagar. Si o vehículo é tipo 1 o monto a pagar é 150, se é tipo 2 o monto a pagar é 200, se é tipo 3 o monto a pagar é 300, se é qualquer outro tipo o monto a pagar é 350.

```
switch(tipo){
   case 1:
          monto=150;
          break;
   case 2:
          monto=200;
          break;

   case 3:
          monto=300;
          break;
   default:
          monto=350;
   }
   printf("O monto a pagar e' % d",monto);
```

---

lista de usos (do aluno)

nome do padrão *Repetição*

---

uso/aplicação Você quer repetir um conjunto de ações várias vezes, enquanto uma condição for verdadeira.

---

padrões dependentes *Condição, Conjunto de Ações.*

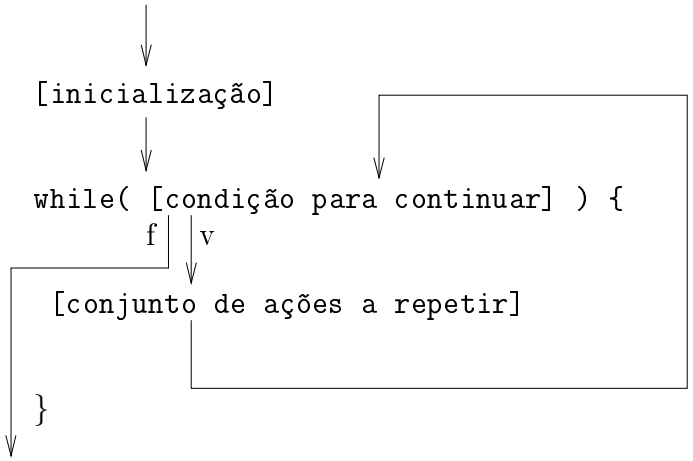
---

sintaxe

```
[inicializações]
while( [condição para continuar] ) {
   [conjunto de ações a repetir]
}
```

---

semântica A *inicialização* (um conjunto de ações) é executada uma única vez. Em seguida, a *condição para continuar* é verificada (passo 1); se verdadeira, o *conjunto de ações a repetir* é executado (passo 2). Os passos 1 e 2 se repetem enquanto a condição for verdadeira.
O processo pára quando a condição se tornar falsa. Para isso, alguma ação do *conjunto de ações a repetir* deve modificar uma ou mais variáveis da condição, caso contrário, dizemos que o programa entra em um "laço infinito".
Observação: é possível que o conjunto de ações nunca seja executado.



---

exemplo de uso Ler um número $n >= 0$ do teclado e calcular o seu fatorial (fat).

```
scanf("%d", &n);  /* inicializa n */
if (n>0){
   fat=n;         /* inicializa o fatorial para n>0*/
}
else {
   fat=1;         /* inicializa o fatorial para n=0*/
}
while( n > 1 ) {
```

```
        fat = fat * (n - 1);
        n = n - 1;
    }
```

---

lista de usos (do aluno)

nome do padrão *Repetição contada*

---

uso/aplicação
Você quer repetir um determinado número de vezes um conjunto de ações. Em geral, o conjunto de ações está relacionado ao processamento de uma seqüência de elementos ou números. A quantidade de elementos deve ser conhecida. Os elementos podem ser lidos ou gerados.

---

padrões dependentes *Condição, Conjunto de Ações.*

---

sintaxe 1

```
<inicializações>
<inicialização do contador>
while ( <condição do contador> ) {
   <leitura/geração de um elemento da seqüência>
   <processar elemento>
   <atualização do contador>
}
```

sintaxe 2

```
<inicializações>
for ( <inicialização do contador> ; <condição do contador> ; <atualização do contador> ) {
   <leitura/geração de um elemento da seqüência>
   <processar elemento>
}
```

O conjunto de ações é composto por <leitura/geração de um elemento da seqüência>, <processar elemento> e <atualização do contador>.

<inicialização do contador> é uma atribuição de um valor inicial para o contador e geralmente é 0 ou 1. <atualização do contador> é da forma:

```
[nome-da-variável]=[nome-da-variável]+[número]
```

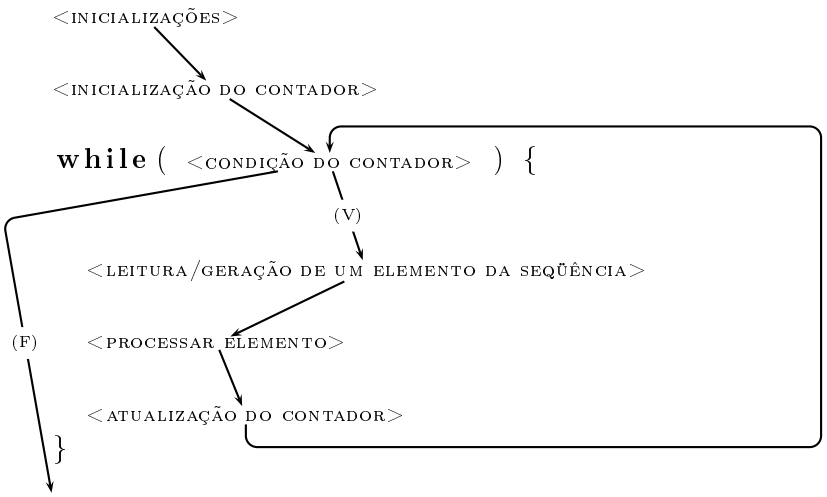
por exemplo c=c+1.
<processar elemento> poderia ter uma variável especial chamada acumulador da forma:

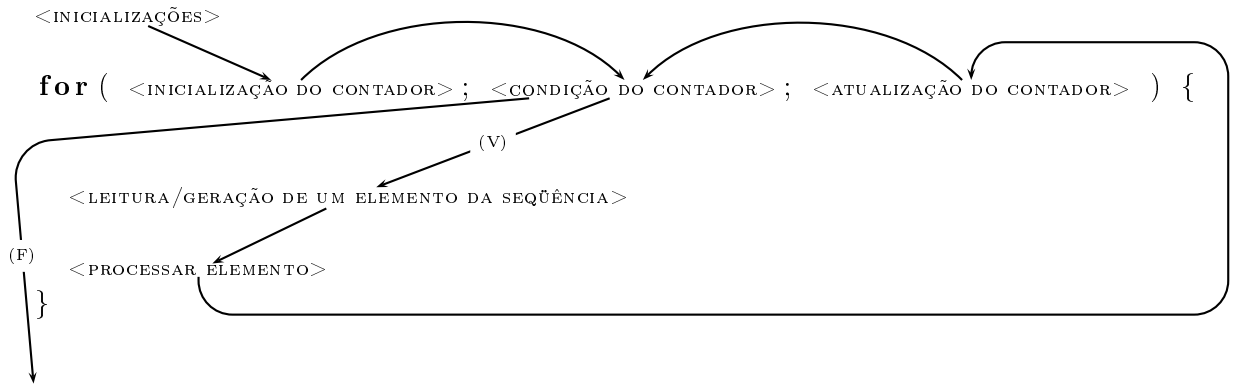
```
[nome-da-variável1]=[nome-da-variável1]+[nome-da-variável2]
```

por exemplo soma=soma+num; nesse caso em <inicializações> deve ter uma atribuição com um valor inicial para o acumulador que geralmente é 0 ou 1.

---

semântica A *inicialização* de variáveis do programa e do contador são executadas uma única vez. Em seguida, a *condição do contador* é verificada; se for verdadeira, o *conjunto de ações* é executado. ou seja, (1) um elemento da seqüência é lido/gerado, (2) processado e (3) o contador é atualizado. Em seguida, a condição do contador é novamente verificada. O processo pára quando a condição do contador se tornar falsa. Para isso, a *atualização do contador* e a *condição do contador* devem ser feitas corretamente, caso contrário, o programa pode entrar em um "laço infinito".





exemplo de uso Faça um programa que leia uma sequência de n números inteiros e calcule a sua soma.

Usando sintaxe 1:

```
printf("Digite␣o␣valor␣de␣n:␣");
scanf("%d", &n);        /* inicializa n */
contador = 0;           /* inicializa o contador */
while( contador < n ) {
  printf("Digite␣o␣numero␣inteiro:␣");
  scanf("%d", &numero);  /* leitura do seguinte numero */
  soma = soma + numero;
  contador = contador + 1;
}
printf ("A␣soma␣e´␣%d\n", soma);
```

Usando sintaxe 2:

```
printf("Digite␣o␣valor␣de␣n:␣");
scanf("%d", &n);        /* inicializa n */
for (contador = 0; contador < n;      contador = contador + 1) {
  printf("Digite␣o␣numero␣inteiro:␣");
  scanf("%d", &numero);  /* leitura do seguinte numero */
  soma = soma + numero;
}
printf ("A␣soma␣e´␣%d\n", soma);
```

---

lista de usos (do aluno)

soma de n números		

nome do padrão *Repetição com sentinela*

---

uso/aplicação Você quer repetir um conjunto de ações enquanto uma condição for verdadeira. Em geral, o conjunto de ações está relacionado ao processamento de uma seqüência de elementos ou números. A quantidade de elementos é desconhecida mas o fim da seqüência é indicado por um valor sentinela. Os elementos podem ser lidos ou gerados.

---

padrões dependentes *Condição*, *Conjunto de Ações*.

---

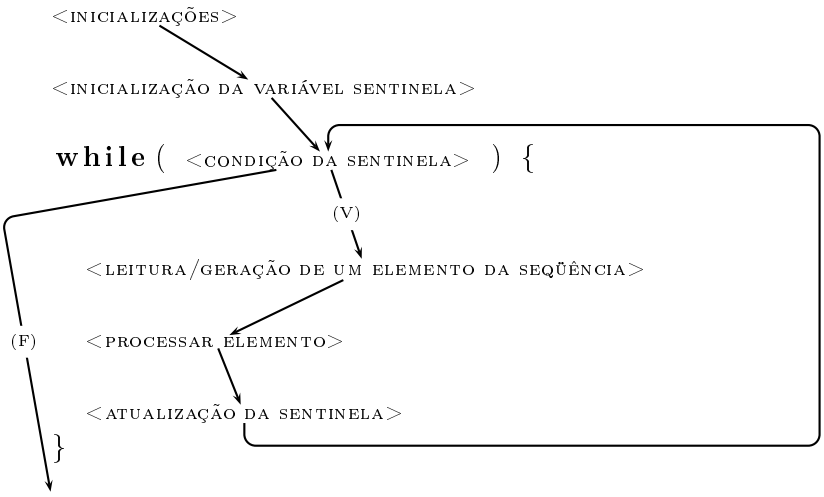
sintaxe

```
<INICIALIZAÇÕES>
<INICIALIZAÇÃO DA VARIÁVEL SENTINELA>
while ( <CONDIÇÃO DA VARIÁVEL SENTINELA> ) {
   <LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQÜÊNCIA>
   <PROCESSAR ELEMENTO>
   <ATUALIZAÇÃO DA VARIÁVEL SENTINELA>
}
```

O conjunto de ações é composto por <LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQÜÊNCIA>, <PROCESSAR ELEMENTO> e <ATUALIZAÇÃO DA VARIÁVEL SENTINELA>.

---

semântica A *inicialização* de variáveis e *inicialização da sentinela para condição verdadeira* são executadas uma única vez. Em seguida, a *condição da sentinela* é verificada; (1) um elemento da seqüência é lido/gerado, (2) processado e, eventualmente, (3) a variável sentinela é atualizada. Em seguida, a condição da variável sentinela é novamente verificada. O processo pára quando a *condição da variável sentinela* se tornar falsa. Para isso, a *atualização da variável sentinela* e a *condição da variável sentinela* devem ser feitas corretamente, caso contrário, o programa pode entrar em um "laço infinito".



---

exemplo de uso 1 Faça un programa que leia uma sequência de números inteiros e calcule a sua soma até que não se deseje inserir mais dados.

```
soma=0;
continua=1;
while( continua == 1 ) {
   printf("Digite o numero inteiro: ");
   scanf("%d", &numero);
   soma = soma + numero;
   printf("Deseja continuar se sim digite: 1, não digite: 0 ");
   scanf("%d", &continua);
}
```

exemplo de uso 2 Faça un programa que leia uma sequência de números inteiros terminada por zero e calcule a sua soma.

```
soma=0;
printf("Digite o numero inteiro: ");
scanf("%d", &numero); /* leitura do primero numero */
while( numero != 0 ) {
   soma = soma + numero;
   printf("Digite o numero inteiro: ");
   scanf("%d", &numero);  /* leitura do numero seguinte */
}
```

No exemplo <LEITURA/GERAÇÃO DE UM ELEMENTO DA SEQÜÊNCIA> e <ATUALIZAÇÃO DA VARIÁVEL SENTINELA> coincidem e estão representados pela linha que faz a leitura do número seguinte.

---

lista de usos (do aluno)

nome do padrão *Repetição com Indicador de Passagem*

---

uso/aplicação Você quer repetir um conjunto de ações enquanto uma condição for verdadeira. Em geral, o conjunto de ações está relacionado ao processamento de uma seqüência de elementos ou números. Porém, durante esse processamento, você precisa verificar uma determinada propriedade da seqüência para: (1) interromper o processamento e/ou (2) para, após o término da repetição, realizar uma ação que dependa daquela propriedade foi satisfeita. Para isso, uma variável, chamada de **indicador de passagem**, é usada para indicar se a propriedade foi satisfeita.
Para verificar a propiedadade da seqüência, basta que um certo fato ocorra uma única vez para dizer que a seqüência não cumpre a propiedade; já para dizer sim é preciso que tal fato não tenha ocorrido nenhuma vez considerando todos os dados.
Note que a quantidade de elementos da seqüência pode ser conhecida ou não, e os elementos podem ser lidos ou gerados.

---

padrões dependentes *Conjunto de Ações*, *Atribuição*, *Repetição*, *Repetição Contada*, *Repetição com Sentinela*

---

sintaxe 1: interromper o processamento

```
...
int indica;
<inicializações>
indica = 0;

while( <condição da repetição> &&indica == 0) {
  <leitura/geração de um elemento da seqüência>
  <processar elemento>
  if(<condição da variável indicador de passagem>)
     indica = 1;
}
if (indica == 1){
     <conjunto-de-ações>
}
else{
     <conjunto-de-ações>
}
```

sintaxe 2: após o término da repetição, realizar uma ação que dependa daquela propriedade foi satisfeita

```
...
int indica;
<inicializações>
indica = 0;
```

```
while ( <condição da repetição>) {
   <leitura/geração de um elemento da seqüência>
   <processar elemento>
   if (<condição da variável indicador de passagem>)
       indica = 1;
}
if (indica == 1){
     <conjunto-de-ações>
}
else{
     <conjunto-de-ações>
}
```

O conjunto de ações dentro do **while** é composto por <leitura/geração de um elemento da seqüência>, <processar elemento> e a seleção simples.

---

semântica A variável indicadora de passagem - *indica* é inicializada. Em seguida, a <condição da repetição> e a condição da variável *indica* são verificadas. Caso a condição conjuntiva (&&) seja verdadeira (1) um elemento da seqüência é lido/gerado, (2) processado e, eventualmente, (3) a variável *indica* é atualizada se a <condição da variável indicador de passagem> for verdadeira. Em seguida, a <condição da repetição> e a condição da variável *indica* são novamente verificadas. O processo pára quando a <condição da repetição> e a condição da variável *indica* se tornar falsa. Para isso as atualizaçoes devem ser feitas corretamente caso contrário, o programa pode entrar em um "laço infinito".
Para a sintaxe 2 só a <condição da repetição> é verificada.

---

exemplo de uso
Dado um número inteiro X, verificar se X é primo

```
int eprimo;
printf("\nDigite um inteiro: ");
scanf("%d", &n);
/* inicializacoes */
if (n <= 1)
  {
    eprimo = 0; /* nenhum numero <= 1 e' primo */
  }
else
  {
    eprimo = 1;  /* o numero e' primo ate que se prove o contrario */
  }

/* os candidatos a divisores positivos de n sao 1,2,...,n/2 */
divisor = 2;
while (divisor <= n/2 && eprimo == 1)
  {
    if (n % divisor == 0)
      {
        eprimo = 0; /* n nao e' primo! */
      }
    divisor=divisor+1;
  }

if (eprimo == 1)
  {
    printf("e' primo\n");
  }
else
  {
    printf("nao e' primo\n");
  }
```

Faça um programa que leia uma sequência de números inteiros terminada por zero e verifique se a seqüência é crescente ou não.

```
/* crescente é a variável do tipo indicador de passagem */
crescente = 1;
printf("Digite o numero inteiro: ");
scanf("%d", &numero); /* leitura do primero numero */
while( numero != 0 ) {
   printf("Digite o proximo numero inteiro: ");
   scanf("%d", &prox_numero);  /* leitura do numero seguinte */
   if(numero > prox_numero)
         crescente = 0;
```

```
        numero = prox_numero;
    }
    if (crescente == 1)
        printf("A sequencia e' crescente");
    else
        printf("A sequencia nao e' crescente");
```

---

lista de usos (do aluno)

nome do padrão *Funções propriamente ditas*

---

uso/aplicação Você necessita repetir várias vezes uma mesma seqüência de ações e quer agrupá-las para organizar melhor o código; melhorar a clareza do programa e reutilizar esse código em várias situações . Como resultado da execução dessa seqüência de ações obtem-se um único valor e para o cálculo desse único valor, pode-se necessitar vários valores como entrada.

---

padrões dependentes *conjunto-de-ações.*

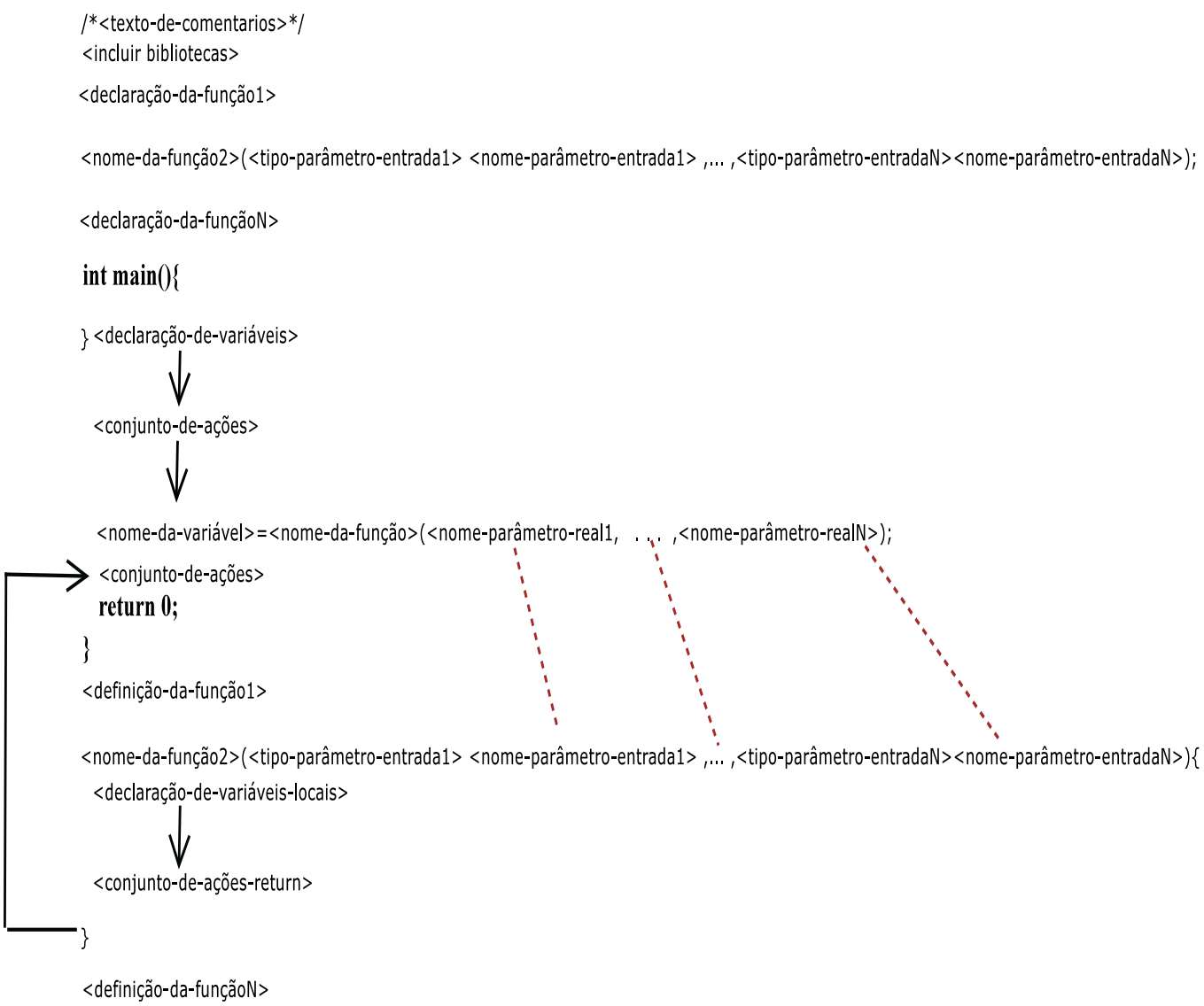
---

sintaxe

```
Declaração ou protótipo da função:

 <tipo-valor-devolvido> <nome-da-função>
(<tipo-parâmetro-entrada1>  <nome-parâmetro-entrada1>,
 <tipo-parâmetro-entrada2>  <nome-parâmetro-entrada2> ,... ,
 <tipo-parâmetro-entradaN><nome-parâmetro-entradaN>);

Definição da função:

 <tipo-valor-devolvido> <nome-da-função>
(<tipo-parâmetro-entrada1>  <nome-parâmetro-entrada1>,
 <tipo-parâmetro-entrada2>  <nome-parâmetro-entrada2> ,... ,
 <tipo-parâmetro-entradaN><nome-parâmetro-entradaN>){
     <declarações-de-variáveis-locais>
     <conjunto-de-ações-return>
 }

Chamada ou uso da função:

<nome-da-variável>=<nome-da-função>(<nome-parâmetro-real1>,
<nome-parâmetro-real2> ,... <nome-parâmetro-realN>);
```

---

semântica A declaração ou protótipo da função especifica ao compilador o número e o tipo dos parâmetros, bem como, o tipo devolvido pela função.
A definição da função especifica o que a função faz.
A chamada da função causa que o corpo da função seja executado.
<DECLARAÇÕES-DAS-VARIÁVEIS-LOCAIS> que permite declarar variáveis que serão necessárias apenas dentro da função.
O <CONJUNTO-DE-AÇÕES-RETURN> é um conjunto de acões que tem pelo menos un comando return <valor-devolvido> o qual retorna o valor como resultado da chamada à função.
Quando, durante o fluxo de execução do programa, uma chamada à função é encontrada (1) os parâmetros reais são associados aos parâmetros da definição da função, (2) as sentenças do corpo da definição da função são executadas, começando pelas <DECLARAÇÕES-DAS-VARIÁVEIS-LOCAIS> e passando pelo <CONJUNTO-DE-AÇÕES-RETURN> até atingir o comando return <valor-devolvido>,



onde a execução da função termina e o controle retorna ao ponto imediato após a chamada da função.

---

exemplo de uso /Faça um programa que leia dois números reais x, y e dois inteiros positivos a, b, calculando em seguida o valor da expressão $x^a + y^b + (x - y)^{a+b}$

```
/*     Programa com funções /*

#include <stdio.h>

float elevado(float base, int exp);
int main() {
```

```
      int a,b;
      float x,y,soma;
      printf(‘‘Digite os valores de x,y,a e b: ‘‘);
      scanf(‘‘%f %f % d %d’’,&x,&y,&a,&b);
      soma=elevado(x,a)+elevado(y,b)+elevado(x-y,a+b);
      printf(‘‘\n O valor da expressao e’ %f:’’, soma);

      return 0;
   }

float elevado(float base, int exp){
   int cont;
   float pot;
   cont=0;
   pot=1;
   while(cont<exp){
      pot=pot*base;
      cont=cont+1;
   }
   return pot;
}
```

---

lista de usos (do aluno)

fatorial		

nome do padrão *Declaração de Vetores*

---

uso/aplicação Você precisa de um conjunto de variáveis de um mesmo tipo.

---

padrões dependentes Nenhum.

---

sintaxe 1 Declaração estática de um vetor

```
<tipo> <nome da variável>[<tamanho do vetor>];
```

<tipo> pode ser: `int` (para inteiro),`float` (para real), `char`(para um caracter) <tamanho do vetor> é um inteiro que corresponde ao tamanho do vetor

---

semântica A declaração de um vetor reserva um espaço na memória para armazenar <tamanho do vetor> valores de um tipo de dado. Cada posição do vetor possui um índice e, através dele, é possível recuperar o valor armazenado em qualquer posição do vetor. Os índices do vetor variam de 0 a <tamanho do vetor> - 1.

---

exemplo de uso
É importante dar um nome *mnemônico* à variável, ou seja, um nome cujo significado em português corresponda ao uso da variável no programa.
Criar um vetor com as notas dos alunos de MAC110, supondo que existem 30 alunos na sala:

```
float notas[30];
```

---

nome do padrão *Percorrimento em Vetores*

---

uso/aplicação Queremos inicializar o vetor e/ou explorar os dados nele contidos.

---

padrões dependentes Conjunto de Ações.

---

sintaxe Percorrendo o vetor

```
for ( <inicialização do contador>;
<condição do contador>;
<atualização do contador> ) {
        [conjunto de ações];
}
```

<inicialização do contador> o contador geralmente é inicializado com valor é 0(zero). O contador deve ser: `int` (para inteiro). <condição do contador> é a condição de parada, geralmente é o <tamanho do vetor>. <atualização do contador> geralmente soma-se 1 ao contador. [conjunto de ações] representa as ações que queremos realizar com o elemento da determinada posição do vetor.

---

semântica Percorrer vetores é essencial para que possamos realizar operações com seus elementos. Note que os limites do vetor tem que estar bem definidos para que o acesso a posições inválidas do vetor não aconteça.

---

exemplo de uso 1
Ler 30 notas dos alunos de MAC110 da entrada padrão e guardá-las em um vetor:

```
    int i;
    float notas[30];
    for (i = 0; i < 30; i++){
        scanf (``%f'', &notas[i])
}
```

exemplo de uso 2
Somar 1 a cada elemento do vetor:

```
    int i;
    int valores[20];
    for (i = 0; i < 20; i++){
       valores[i] = valores[i] + 1;
}
```

---

nome do padrão *Busca em Vetores*

---

uso/aplicação Você precisa saber se um determinado valor, ou um valor com certa característica está contido no vetor.

---

padrões dependentes Nenhum.

---

sintaxe 1

```
for ( <inicialização do contador> ;
<condição do contador> ;
<atualização do contador> ) {
    if ( [condição] ) {
       [conjunto de ações];
    }
}
```

<inicialização do contador> o contador deve ser inicializado com o índice do vetor com o qual deseja-se iniciar a busca, geralmente este valor é 0(zero). O contador deve ser: `int` (para inteiro). <condição do contador> é a condição de parada da busca, geralmente é o índice no qual deseja-se parar a busca no vetor. Na maioria dos casos é <tamanho do vetor>. <atualização do contador> geralmente soma-se 1 ao contador. [condição] representa a expressão lógica que te permite encontrar o elemento procurado. [conjunto de ações] representa as ações que queremos realizar com o elemento encontrado.

semântica A busca em vetor nos possibilita explorar os dados contidos dentro do mesmo. Note que ao procurar determinado valor, esta busca retorna apenas a primeira ocorrência deste valor no vetor.

---

exemplo de uso
É importante ter certeza que as condições de parada estão corretas, para que o acesso a posições inexistentes do vetor não aconteça.
Vamos contar quantos inteiros divisíveis por 3 estão contidos no vetor:

```
for (i = 0; i < tamanho; i++) {
  if ( (v[i]% 3) == 0){
    nNumeros = nNumeros++;
  }
}
```

Queremos apenas saber se a *9,5* está contida no vetor de notas dos alunos de MAC-110:

```
for (i = 0; i < tamanho; i++) {
  if ( v[i] == 9.5){
    break;
  }
}
/* Se i atingiu o tamanho do vetor significa que percorremos o vetor inteiro
   e não encontramos a nota . Caso contrário, encontramos a nota.*/
 if ( i >= tamanho) printf (``A nota 9.5 não está no vetor'');
 else printf (``A nota 9.5 está contida no vetor'');
```

---

nome do padrão *Declaração de Matrizes*

---

uso/aplicação Você precisa trabalhar com um conjunto de valores do mesmo tipo tais que sua representação se assemelha a uma tabela.

---

padrões dependentes Nenhum.

---

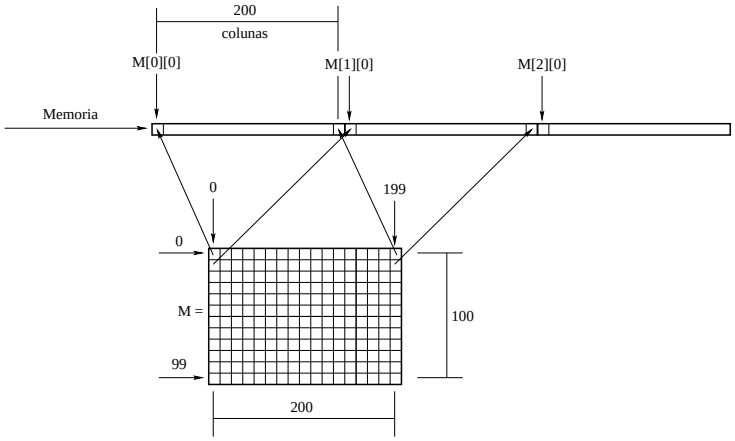
sintaxe Declaração estática de uma matriz

```
<tipo> <nome da variável>[<numero de linhas>][<numero de colunas>];
```

<tipo> pode ser: `int` (para inteiro),`float` (para real), `char`(para um caracter) <número de linhas> é um inteiro que corresponde a quantidade de linhas que a matriz terá <número de colunas> é um inteiro que corresponde a quantidade de colunas que a matriz terá.

---

semântica A declaração de uma matriz reserva um espaço que tem capacidade de armazenar *<número de linhas> x <número de colunas>* valores. Na realidade, uma matriz é um vetor de vetores. Sua representação na memória:



---

exemplo de uso

É importante dar um nome *mnemônico* à variável, ou seja, um nome cujo significado em português corresponda ao uso da variável no programa.
Criar uma matriz com as notas dos alunos de MAC110 de todos os eps. Supondo que existem 30 alunos e foram dados 4 eps ao longo do semestre. alunos na sala:

```
float notas[30][4];
```

---

nome do padrão *Percorrimento de Matrizes*

---

uso/aplicação Queremos inicializar a matriz e/ou explorar os dados nela contidos.

---

padrões dependentes Conjunto de Ações.

---

sintaxe 1 Percorrendo a matriz por linhas

```
for ( <inicialização do contLinha>;
<condição do contLinha>;
<atualização do contLinha> ) {
    for ( <inicialização do contColuna>;
    <condição do contColuna>;
    <atualização do contColuna> ) {
        [conjunto de ações];
    }
}
```

<inicialização do contador> o contador geralmente é inicializado com valor é 0(zero). O contador deve ser: `int` (para inteiro). <condição do contador> é a condição de parada, geralmente é o <tamanho do vetor>. <atualização do contador> geralmente soma-se 1 ao contador. [conjunto de ações] representa as ações que queremos realizar com o elemento da determinada posição do vetor.

sintaxe 2 Percorrendo a matriz por colunas

```
for ( <inicialização do contColuna>;
<condição do contColuna>;
<atualização do contColuna> ) {
    for ( <inicialização do contLinha>;
    <condição do contLinha>;
    <atualização do contLinha> ) {
        [conjunto de ações];
    }
}
```

Note que é muito parecido com o percorrimento de matrizes por linha. A mudança está relacionada a ordem dos comandos `for`. Ao percorrer por linha, devemos fixar uma linha (`for` exterior) e percorrer todas as suas colunas (`for` interior). Já no perocrrimento por colunas devemos fixar uma coluna (`for` exterior) e percorrer todas as suas linhas (`for` interior). Não devemos esquecer que os índices de uma matriz M são M[linha][coluna].

---

sintaxe 3 Percorrendo a diagonal principal de uma matriz

```
for ( contLinha = 0,
      contColuna = 0;
      contLinha < <numero de linhas>,
      contColuna < <numero de colunas>;
      contLinha++,
      contColuna++){
          [conjunto de ações];
}
```

Note que no percorrimento da diagonal principal de uma matriz não fixamos uma linha/coluna e percorremos uma linha/coluna. Neste caso as linhas e colunas estão dentro do mesmo comando `for` pois a cada iteração precisamos atualizar ambos os índices.

---

sintaxe 4 Percorrendo a diagonal secundária de uma matriz

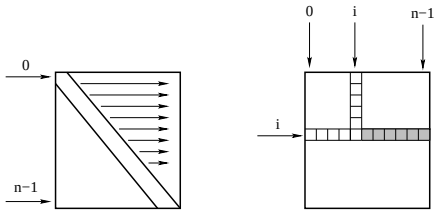
```
for ( contLinha = 0,
      contColuna = <numero de colunas>-1;
      contLinha < <numero de linhas>,
      contColuna >= 0 ;
      contLinha++,
      contColuna--){
          [conjunto de ações];
}
```

---

sintaxe 5 Percorrendo o triângulo superior de uma matriz

```
for ( contLinha = 0;
contLinha < <numero de linhas>;
contLinha++ ) {
      for ( contColuna = contLinha + 1;
      contColuna < <numero de colunas>;
      contColuna++ ) {
           [conjunto de ações];
      }
 }
```

Note que para uma linha fixa (`cont_linha`), inicializamos o índice da coluna com o número da linha que está sendo percorrida mais 1 pois isso permite que o elemento a ser lido está um índice a frente da diagonal principal. Para melhor compreensão, olhe a figura.



---

semântica Percorrer matrizes é essencial para que possamos realizar operações com seus elementos. As várias formas de percorrer nos dá flexibilidade para realizar todas as operações matemáticas envolvidas com matrizes. Assim como em vetores, os limites da matriz devem estar bem definidos para que o acesso a posições inválidas não aconteça.

---

exemplo de uso 1
Imprimir o conteúdo de uma matriz M de n linhas e m colunas:

```
    for (i = 0; i < n; i++){
     for (j = 0; j < m; j++){
printf (‘‘%d’’, M[i][j]);
}
printf("\n");
}
```

---

exemplo de uso 2
Multiplicação de duas matrizes (A[n][n] e B[n][n]) guardando o resultado na matriz C. Note que percorremos a matriz A por linhas e a matriz B por colunas.

```
for (i = 0; i < n; i++){
     for (j = 0; j < n; j++){
C[i][j] = 0;
```

```
for (k = 0; k < n; k++){
C[i][j] = C[i][j] + A[i][k] * B[k][j];
}
}
}
```

---

exemplo de uso 3 Verificar se a diagonal principal de uma matriz M[n][m] é nula.

```
for (i = 0, j = 0; i < n, j < m; i++, j++){
     if ( C[i][j] != 0){
printf ("A matriz tem diagonal nula.\n");
  break;
}
}
```

---

exemplo de uso 5 Verificar se a matriz M[n][n] é simétrica.

```
simetrica = 1;
for (i = 0; i < n; i++){
for (j = i + 1; j < n; j++){
     if ( M[i][j] != M[j][i]){
simetrica = 0;
}
}
```
