

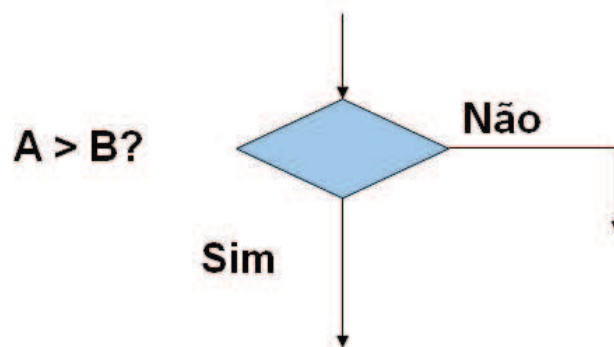
LINGUAGEM C: DESCOMPLICADA

Prof. André R. Backes

1 COMANDOS DE CONTROLE CONDICIONAL

Os programas escritos até o momento são programas sequenciais: um comando é executado após o outro, do começo ao fim do programa, na ordem em que foram declarados no código fonte. Nenhum comando é ignorado.

Entretanto, há casos em que é preciso que um bloco de comandos seja executado somente se uma determinada condição for verdadeira. Para isso, precisamos de uma estrutura de seleção, ou um comando de controle condicional, que permita selecionar o conjunto de comandos a ser executado. Isso é muito similar ao que ocorre em um fluxograma, onde o símbolo do losango permitia escolher entre diferentes caminhos com base em uma condição do tipo verdadeiro/falso:



Nesta seção iremos ver como funcionam cada uma das estruturas de seleção presentes na linguagem C.

1.1 COMANDO IF

Na linguagem C, o comando **if** é utilizado sempre que é necessário escolher entre dois caminhos dentro do programa, ou quando se deseja executar um ou mais comandos que estejam sujeitos ao resultado de um teste.

A forma geral de um comando **if** é:

```
if (condição) {  
    seqüência de comandos;  
}
```

Na execução do comando **if** a condição será avaliada e:

- se a condição for **diferente** de zero, ela será considerada **verdadeira** e a seqüência de comandos será executada;
- se a condição for zero, ela será considerada **falsa** e a seqüência de comandos não será executada.

Abaixo, tem-se um exemplo de um programa que lê um número inteiro digitado pelo usuário e informa se o mesmo é maior do que 10:

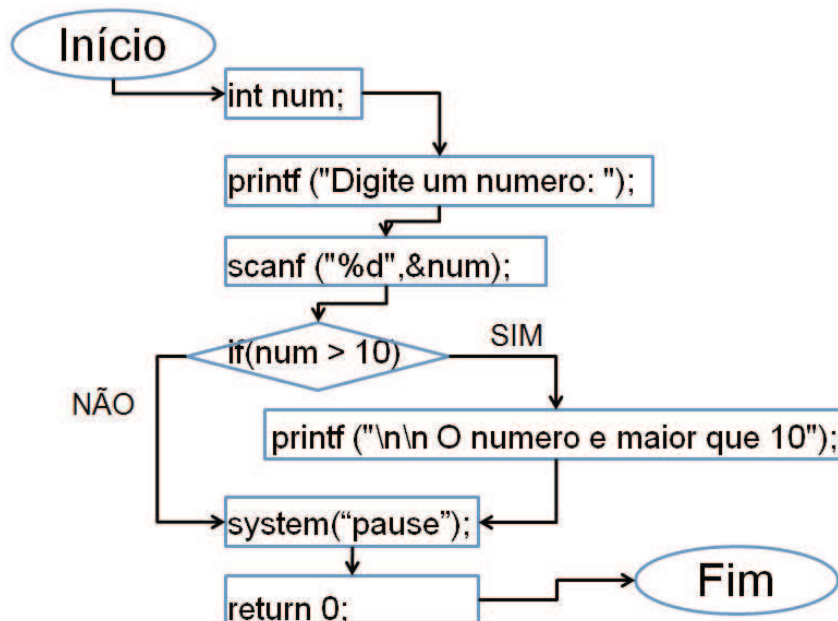
Exemplo: comando if

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int num;
5     printf ("Digite um numero: ");
6     scanf ("%d",&num);
7     if (num > 10)
8         printf ("O numero e maior do que 10\n");
9     system("pause");
10    return 0;
11 }

```

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



Por condição, entende-se qualquer expressão que resulte numa resposta do tipo falso (zero) ou verdadeiro (diferente de zero). A condição pode ser uma expressão que utiliza operadores dos tipos:

- Matemáticos : +, -, *, /, %
- Relacionais: >, <, >=, <=, ==, !=
- Lógicos: &&, ||



Diferente da maioria dos comandos, não se usa o ponto e vírgula (;) depois da condição do comando if.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int num;
5     printf (" Digite um numero: ");
6     scanf ("%d",&num);
7     if (num > 10); //ERRO
8         printf("O numero e maior que 10\n");
9     system("pause");
10    return 0;
11 }
```

Na linguagem C, o operador ponto e vírgula (;) é utilizado para separar as instruções do programa. Colocá-lo logo após o comando if, como exemplificado acima, faz com que o compilador entenda que o comando if já terminou e trate o comando seguinte (**printf**) como se o mesmo estivesse fora do if. No exemplo acima, a mensagem de que o número é maior do que 10 será exibida independente do valor do número.



O compilador não irá acusar um erro se colocarmos o operador ponto e vírgula (;) após o comando if, mas a lógica do programa poderá estar errada.

1.1.1 USO DAS CHAVES {}

No comando if, e em diversos outros comandos da linguagem C, usa-se os operadores de chaves { } para delimitar um bloco de instruções.



Por definição, comandos de condição (if e else) ou repetição (while, for,...) atuam apenas sobre o comando seguinte a eles.

Desse modo, se o programador deseja que mais de uma instrução seja executada por aquele comando if, esse conjunto de instruções deve estar contido dentro de um bloco delimitado por chaves { }.

```
if (condição) {  
    comando 1;  
    comando 2;  
    ...  
    comando n;  
}
```



As chaves podem ser ignoradas se o comando contido dentro do if for único.

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 int main () {  
4     int num;  
5     printf (" Digite um numero: ");  
6     scanf ("%d",&num);  
7     if (num > 10)  
8         printf("O numero e maior que 10\n");  
9  
10    /*OU  
11    if(num > 10){  
12        printf ("O numero e maior que 10\n");  
13    }  
14    */  
15    system("pause");  
16    return 0;  
17 }
```

1.1.2 EXPRESSÃO CONDICIONAL

Uma expressão condicional é qualquer expressão que resulte numa resposta do tipo falso (zero) ou verdadeiro (diferente de zero).



Uma expressão condicional pode utilizar operadores dos tipos: matemáticos, relacionais e/ou lógicos.

```
1 //x é maior ou igual a y?
2 if (x >= y)
3
4 //x é maior do que y+2?
5 if (x > y+2)
6
7 //x-5 é diferente de y+3?
8 if (x-5 != y+3)
9
10 //x é maior do que y e menor do que z?
11 if (x > y && x < z)
12 if (y < x < z) //ERRO!
```

Quando o compilador avalia uma condição, ele quer um valor de retorno (verdadeiro ou falso) para poder tomar a decisão. No entanto, esta expressão não necessita ser uma expressão no sentido convencional.



Uma variável sozinha pode ser uma "expressão condicional" e retornar o seu próprio valor.

É importante lembrar que o computador trabalha em termos de 0's e 1's, sendo a condição

- falsa: quando o valor da expressão é zero;
- verdadeira: quando o valor da expressão é **diferente** de zero.

Isto quer dizer que, dado uma variável inteira *num*, as seguintes expressões são equivalentes para o compilador:

if (num!=0)//Se a variável é diferente de zero...

if (num)//...ela sozinha retorna um valor que é verdadeiro.

if (num==0)//Se a variável é igual a zero (falso)...

e

if (!num)//...sua negação é um valor verdadeiro.

1.2 COMANDO ELSE

O comando else pode ser entendido como sendo um complemento do comando if. Ele auxilia o comando if na tarefa de escolher dentre os vários caminhos a ser seguido dentro do programa.

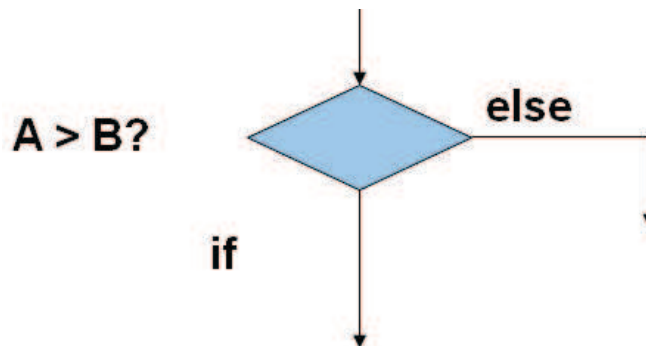
A forma geral de um comando else é:

```
if (condição) {  
    seqüência de comandos;  
}  
else{  
    seqüência de comandos;  
}
```



Se o comando if diz o que fazer quando a condição é verdadeira, o comando else trata da condição quando ela é falsa.

Isso fica bem claro quando olhamos a representação do comando else em um fluxograma:



Antes, na execução do comando if a condição era avaliada e:

- se a condição fosse **verdadeira** a seqüência de comandos seria executada;
- se a condição fosse **falsa** a seqüência de comandos não seria executada e o programa seguiria o seu fluxo padrão.

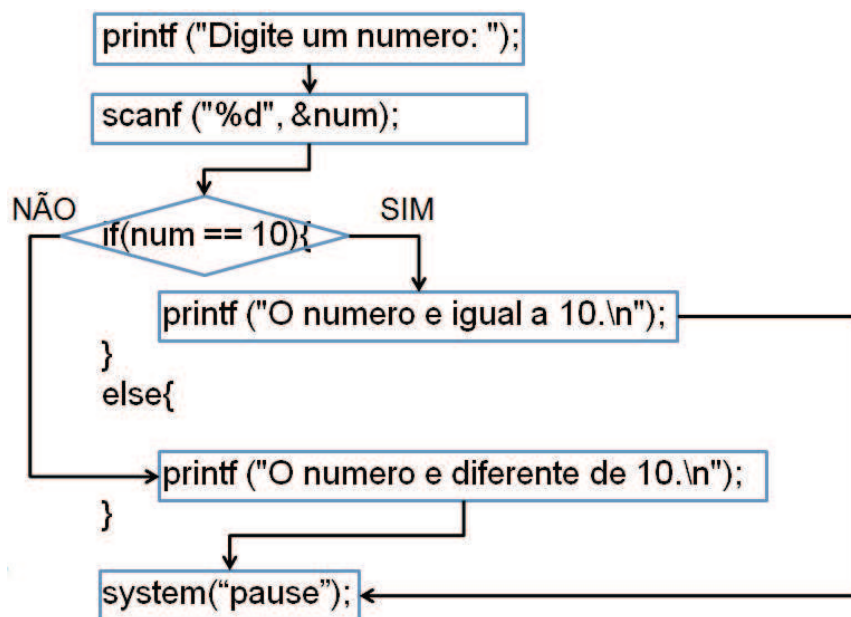
Com o comando else, temos agora que:

- se a condição for **verdadeira**, a sequência de comandos do bloco if será executada;
- se a condição for **falsa**, a sequência de comandos do bloco else será executada.

Abaixo, tem-se um exemplo de um programa que lê um número inteiro digitado pelo usuário e informa se o mesmo é ou não igual a 10:

Exemplo: comando if-else	
1	#include <stdio.h>
2	#include <stdlib.h>
3	int main () {
4	int num;
5	printf ("Digite um numero: ");
6	scanf ("%d", &num);
7	if (num == 10) {
8	printf ("O numero e igual a 10.\n");
9	} else {
10	printf ("O numero e diferente de 10.\n");
11	}
12	system("pause");
13	return 0;
14	}

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



O comando else não tem condição. Ele é o caso contrário da condição do if.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main (){
4     int num;
5     printf ("Digite um numero: ");
6     scanf ("%d", &num);
7     if(num == 10){
8         printf ("O numero e igual a 10.\n");
9     } else(num != 10){//ERRO
10        printf ("O numero e diferente de 10.\n");
11    }
12    system("pause");
13    return 0;
14 }
  
```

O comando else deve ser entendido como sendo um complemento do comando if. Ele diz quais comandos se deve executar se a condição do comando if for falsa. Portanto, não é necessário estabelecer uma condição para o comando else, ele é o oposto do if.



Como no caso do if, não se usa o ponto e vírgula (;) depois do comando else.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int num;
5     printf ("Digite um numero: ");
6     scanf ("%d", &num);
7     if (num == 10) {
8         printf ("O numero e igual a 10.\n");
9     } else; { //ERRO
10        printf ("O numero e diferente de 10.\n");
11    }
12    system("pause");
13    return 0;
14 }
```

Como no caso do if, colocar o operador de ponto e vírgula (;) logo após o comando else, faz com que o compilador entenda que o comando else já terminou e trate o comando seguinte (**printf**) como se o mesmo estivesse fora do else. No exemplo acima, a mensagem de que o número é diferente de 10 será exibida independente do valor do número.



A sequência de comandos do if é independente da sequência de comandos do else. Cada comando tem o seu próprio conjunto de chaves.

Se o comando if for executado em um programa, o seu comando else não será executado. Portanto, não faz sentido usar o mesmo conjunto de chaves {} para definir os dois conjuntos de comandos.

Uso das chaves no comando if-else	
Certo	Errado
<pre>1 if (condicao){ 2 seqüência de comandos; 3 } 4 else{ 5 seqüência de comandos; 6 }</pre>	<pre>1 if (condicao){ 2 seqüência de comandos; 3 else 4 seqüência de comandos; 5 }</pre>



Como no caso do comando if, as chaves podem ser ignoradas se o comando contido dentro do else for único.

1.3 ANINHAMENTO DE IF

Um if aninhado é simplesmente um comando if utilizado dentro do bloco de comandos de um outro if (ou else) mais externo. basicamente, é um comando if dentro de outro.

A forma geral de um comando if aninhado é:

```
if(condição 1) {  
    seqüência de comandos;  
    if(condição 2) {  
        seqüência de comandos;  
        if...  
    }  
    else{  
        seqüência de comandos;  
        if...  
    }  
} else{  
    seqüência de comandos;  
}
```

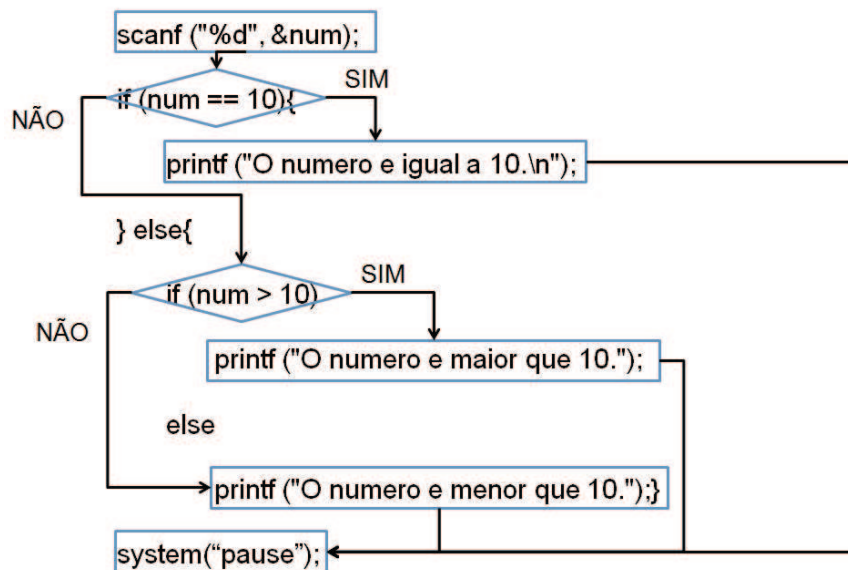
Em um aninhamento de if's, o programa começa a testar as condições começando pela **condição 1**. Se o resultado dessa condição for diferente de zero (verdadeiro), o programa executará o bloco de comando associados a ela. Do contrário, irá executar o bloco de comando associados ao comando else correspondente, se ele existir. Esse processo se repete para cada comando if que o programa encontrar dentro do bloco de comando que ele executar.

O aninhamento de if's é muito útil quando se tem mais do que dois caminhos para executar dentro de um programa. Por exemplo, o comando if é suficiente para dizer se um número é maior do que outro número ou não. Porém, ele sozinho é incapaz de dizer se esse mesmo número é maior, menor ou igual ao outro como mostra o exemplo abaixo:

Exemplo: aninhamento de if

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int num;
5     printf("Digite um numero: ");
6     scanf("%d", &num);
7     if (num == 10){
8         printf("O numero e igual a 10.\n");
9     } else{
10         if (num > 10)
11             printf("O numero e maior que 10.\n");
12         else
13             printf("O numero e menor que 10.\n");
14     }
15     system("pause");
16     return 0;
17 }
```

Isso fica bem claro quando olhamos a representação do aninhamento de if's em um fluxograma:



O único cuidado que devemos ter no aninhamento de if's é o de saber exatamente a qual if um determinado else está ligado.

Esse cuidado fica claro no exemplo abaixo: apesar do comando else estar alinhado com o primeiro comando if, ele está na verdade associado ao

segundo if. Isso acontece porque o comando else é sempre associado ao primeiro comando if encontrado antes dele dentro de um bloco de comandos.

```
if (cond1)
    if (cond2)
        seqüência de comandos;
else
    seqüência de comandos;
```

No exemplo anterior, para fazer com que o comando else fique associado ao primeiro comando if é necessário definir um novo bloco de comandos (usando os operadores de chaves { }) para isolar o comando if mais interno.

```
if (cond1) {
    if (cond2)
        seqüência de comandos;
} else
    seqüência de comandos;
```



Não existe aninhamento de else's.

O comando else é o caso contrário da condição do comando if. Assim, para cada else deve existir um if anterior, porém nem todo if precisa ter um else.

```
if (cond1)
    seqüência de comandos;
else
    seqüência de comandos;
else //ERRO!
    seqüência de comandos;
```

1.4 OPERADOR ?

O operador ? é também conhecido como *operador ternário*. Trata-se de uma simplificação do comando if-else na sua forma mais simples, ou seja, com apenas um comando e não blocos de comandos.

A forma geral do operador ? é:

expressão condicional ? expressão1 : expressão2;

O funcionamento do operador ? é idêntico ao do comando if-else: primeiramente, a *expressão condicional* será avaliada e

- se essa condição for **verdadeira**, o valor da *expressão1* será o resultado da *expressão condicional*;
- se essa condição for **falsa**, o valor da *expressão2* será o resultado da *expressão condicional*;



O operador ? é tipicamente utilizado para atribuições condicionais.

O exemplo abaixo mostra como uma expressão de atribuição pode ser simplificada utilizando o operador ternário:

Usando if-else	Usando o operador ternário
<pre>1 #include <stdio.h> 2 #include <stdlib.h> 3 int main () { 4 int x,y,z; 5 printf("Digite x:"); 6 scanf("%d",&x); 7 printf("Digite y:"); 8 scanf("%d",&y); 9 if (x > y) 10 z = x; 11 else 12 z = y; 13 printf("Maior = %d",z); 14 system("pause"); 15 return 0; 16 }</pre>	<pre>1 #include <stdio.h> 2 #include <stdlib.h> 3 int main () { 4 int x,y,z; 5 printf("Digite x:"); 6 scanf("%d",&x); 7 printf("Digite y:"); 8 scanf("%d",&y); 9 z = x > y ? x : y; 10 printf("Maior = %d",z); 11 system("pause"); 12 return 0; 13 }</pre>

O operador `?` é limitado e por isso não atende a uma gama muito grande de casos que o comando `if-else` atenderia. Porém, ele pode ser usado para simplificar expressões complicadas. Uma aplicação interessante é a do contador circular, onde uma variável é incrementada até um valor máximo e, sempre que atinge esse valor, a variável é zerada.

```
index = (index == 3) ? 0: ++index;
```



Apesar de limitado, o operador `?` não é restrito a atribuições apenas.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int num;
5     printf("Digite um numero: ");
6     scanf("%d", &num);
7     (num == 10)? printf("O numero e igual a 10.\n")
8                 : printf("O numero e diferente de 10.\n")
9                 ;
10    system("pause");
11    return 0;
12 }
```

1.5 COMANDO SWITCH

Além dos comandos `if` e `else`, a linguagem C possui um comando de seleção múltipla chamado **switch**. Esse comando é muito parecido com o aninhamento de comandos `if-else-if`.



O comando `switch` é muito mais limitado que o comando `if-else`: enquanto o comando `if` pode testar expressões lógicas ou relacionais, o comando `switch` somente verifica se uma variável é ou não igual a um certo valor constante.

A forma geral do comando switch é:

```
switch (variável) {  
    case valor1:  
        seqüência de comandos;  
        break;  
    case valor2:  
        seqüência de comandos;  
        break;  
    ...  
    case valorN:  
        seqüência de comandos;  
        break;  
    default:  
        seqüência de comandos; }
```



O comando switch é indicado quando se deseja testar uma variável em relação a diversos valores pré-estabelecidos.

Na execução do comando switch, o valor da *variável* é comparado, na ordem, com cada um dos valores definidos pelo comando **case**. Se um desses valores for igual ao valor da variável, a seqüência de comandos daquele comando case é executado pelo programa.

Abaixo, tem-se um exemplo de um programa que lê um caractere digitado pelo usuário e informa se o mesmo é um símbolo de pontuação:

Exemplo: comando switch

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char char_in;
5     printf("Digite um simbolo de pontuacao: ");
6     char_in = getchar();
7     switch( char_in ) {
8         case '.': printf("Ponto.\n" ); break;
9         case ',': printf("Virgula.\n" ); break;
10        case ':': printf("Dois pontos.\n" ); break;
11        case ';': printf("Ponto e virgula.\n"); break;
12        default : printf("Nao eh pontuacao.\n" );
13    }
14    system("pause");
15    return 0;
16 }
```

No exemplo acima, será pedido ao usuário que digite um caractere. O valor desse caractere será comparado com um conjunto de possíveis símbolos de pontuação, cada qual identificado em um comando **case**. Note que, se o caractere digitado pelo usuário não for um símbolo de pontuação, a sequência de comandos dentro do comando default será executada.



O comando *default* é opcional e sua sequência de comandos somente será executada se o valor da variável que está sendo testada pelo comando switch não for igual a nenhum dos valores dos comandos case.

O exemplo anterior do comando switch poderia facilmente ser reescrito com o aninhamento de comandos if-else-if como se nota abaixo:

Exemplo: simulando o comando switch com if-else-if

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char char_in;
5     printf("Digite um simbolo de pontuacao: ");
6     char_in = getchar();
7     if (char_in == '.')
8         printf("Ponto.\n" );
9     else
10        if (char_in == ',')
11            printf("Virgula.\n" );
12        else
13            if (char_in == ':')
14                printf("Dois pontos.\n" );
15            else
16                if (char_in == ';')
17                    printf("Ponto e virgula.\n");
18                else
19                    printf("Nao eh pontuacao.\n" );
20    system("pause");
21    return 0;
22 }
```

Como se pode notar, o comando switch apresenta uma solução muito mais elegante que o aninhamento de comandos if-else-if quando se necessita comparar o valor de uma variável.

Apesar das semelhanças entre os dois comandos, o comando switch e o aninhamento de comandos if-else-if, existe uma diferença muito importante entre esses dois comandos: o comando **break**.



Quando o valor associado a um comando case é igual ao valor da variável do switch a respectiva seqüência de comandos é executada até encontrar um comando break. Caso o comando break não exista, a seqüência de comandos do case seguinte também será executada e assim por diante

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char char_in;
5     printf("Digite um simbolo de pontuacao: ");
6     char_in = getchar();
7     switch( char_in ) {
8         case '.': printf("Ponto.\n" );
9         case ',': printf("Virgula.\n" );
10        case ':': printf("Dois pontos.\n");
11        case ';': printf("Ponto e virgula.\n");
12        default : printf("Nao eh pontuacao.\n" );
13    }
14    system("pause");
15    return 0;
16 }
```

Note, no exemplo acima, que caso o usuário digite o símbolo de ponto (.) todas as mensagens serão escritas na tela de saída.



O comando break é **opcional** e faz com que o comando switch seja interrompido assim que uma das seqüência de comandos seja executada.

De modo geral, é quase certo que se venha a usar o comando break dentro do switch. Porém a sua ausência pode ser muito útil em algumas situações. Por exemplo, quando queremos que uma ou mais seqüências de comandos sejam executadas a depender do valor da variável do switch.

Exemplo: comando switch sem break

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int num;
5     printf("Digite um numero inteiro de 0 a 9: ");
6     scanf("%d",&num);
7     switch(num){
8         case 9: printf("Nove\n");
9         case 8: printf("Oito\n");
10        case 7: printf("Sete\n");
11        case 6: printf("Seis\n");
12        case 5: printf("Cinco\n");
13        case 4: printf("Quatro\n");
14        case 3: printf("Tres\n");
15        case 2: printf("Dois\n");
16        case 1: printf("Um\n");
17        case 0: printf("Zero\n");
18    }
19    system("pause");
20    return 0;
21 }
```

2 COMANDOS DE REPETIÇÃO

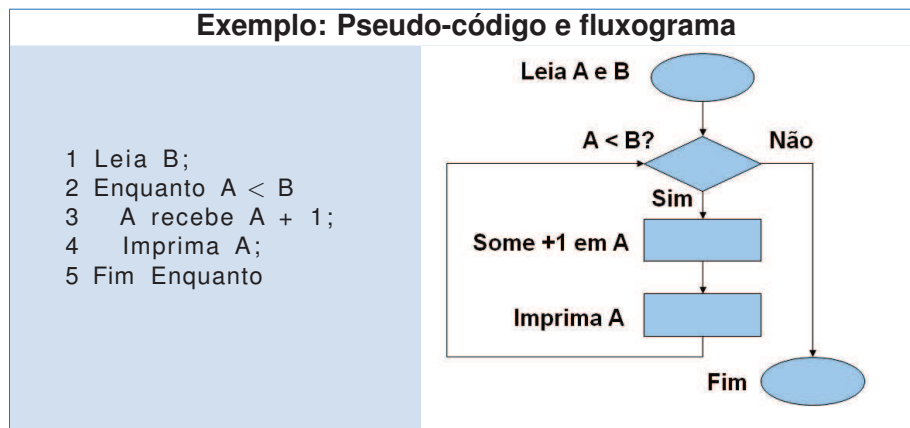
2.1 REPETIÇÃO POR CONDIÇÃO

Na seção anterior, vimos como realizar desvios condicionais em um programa. Desse modo, criamos programas em que um bloco de comandos é executado somente se uma determinada condição é verdadeira.

Entretanto, há casos em que é preciso que um bloco de comandos seja executado mais de uma vez se uma determinada condição for verdadeira:

```
enquanto condição faça  
    sequência de comandos;  
fim enquanto
```

Para isso, precisamos de uma estrutura de repetição que permita executar um conjunto de comandos quantas vezes forem necessárias. Isso é muito similar ao que ocorre em um fluxograma, onde o símbolo do losango permitia escolher entre diferentes caminhos com base em uma condição do tipo verdadeiro/falso, com a diferença de que agora o fluxo do programa é desviado novamente para a condição ao final da sequência de comandos:



De acordo com a condição, os comandos serão repetidos zero (se falsa) ou mais vezes (enquanto a condição for verdadeira). Essa estrutura normalmente é denominada laço ou loop.

Note que a sequência de comandos a ser repetida está subordinada a uma condição. Por condição, entende-se qualquer expressão que resulte numa resposta do tipo falso (zero) ou verdadeiro (diferente de zero). A condição pode ser uma expressão que utiliza operadores dos tipos:

- Matemáticos : +, -, *, /, %
- Relacionais: >, <, >=, <=, ==, !=
- Lógicos: &&, ||

Na execução do comando enquanto, a condição será avaliada e:

- se a condição for **diferente** de zero, ela será considerada **verdadeira** e a sequência de comandos será executada. Ao final da sequência de comandos, o fluxo do programa é desviado novamente para a condição;
- se a condição for zero, ela será considerada **falsa** e a sequência de comandos não será executada.

2.2 LAÇO INFINITO

Um laço infinito (ou loop infinito) é uma sequência de comandos em um programa de computador que se repete infinitamente. Isso geralmente ocorre por algum erro de programação, quando

- não definimos uma condição de parada;
- a condição de parada existe, mas nunca é atingida.

Basicamente, um laço infinito ocorre quando cometemos algum erro ao especificar a condição lógica que controla a repetição ou por esquecer de algum comando dentro da sequência de comandos.

Exemplo: loop infinito	
O valor de X é sempre diminuído em uma unidade, portanto nunca atinge a condição de parada.	O valor de X nunca é modificado, portanto a condição é sempre verdadeira.
<pre> 1 X recebe 4; 2 enquanto (X < 5) faça 3 X recebe X - 1; 4 Imprima X; 5 fim enquanto </pre>	<pre> 1 X recebe 4; 2 enquanto (X < 5) faça 3 Imprima X; 4 fim enquanto </pre>

2.3 COMANDO WHILE

O comando while equivale ao comando "enquanto" utilizado nos pseudocódigos apresentados até agora.

A forma geral de um comando while é:

```

while (condição){
    seqüência de comandos;
}

```

Na execução do comando while, a condição será avaliada e:

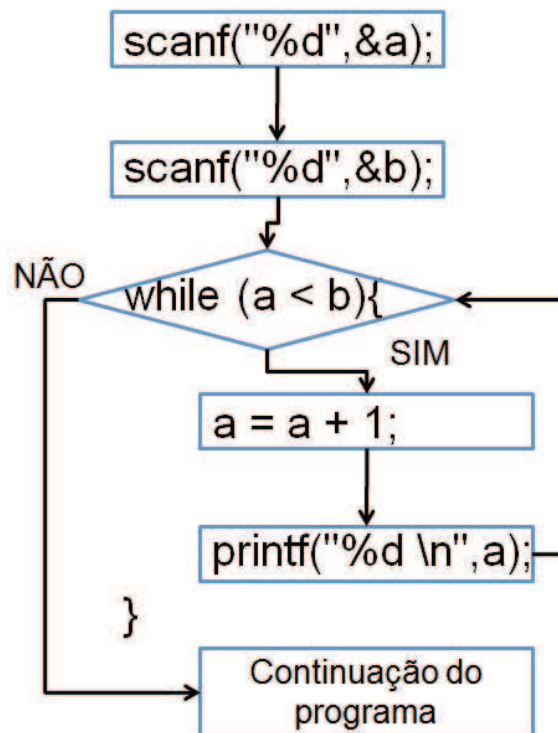
- se a condição for **diferente** de zero, ela será considerada **verdadeira** e a seqüência de comandos será executada. Ao final da seqüência de comandos, o fluxo do programa é desviado novamente para a condição;
- se a condição for zero, ela será considerada **falsa** e a seqüência de comandos não será executada.

Abaixo, tem-se um exemplo de um programa que lê dois números inteiros *a* e *b* digitados pelo usuário e imprime na tela todos os números inteiros entre *a* e *b*:

Exemplo: comando while

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     while (a < b){
10         a = a + 1;
11         printf("%d \n",a);
12     }
13     system("pause");
14     return 0;
15 }
```

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



O comando while segue todas as recomendações definidas para o comando if quanto ao uso das chaves e definição da condição usada.

Isso significa que a condição pode ser qualquer expressão que resulte numa resposta do tipo falso (zero) ou verdadeiro (diferente de zero), e que utiliza operadores dos tipos *matemáticos*, *relacionais* e/ou *lógicos*.

Como nos comandos condicionais, o comando while atua apenas sobre o comando seguinte a ele. Se quisermos que ele execute uma sequência de comandos, é preciso definir essa sequência de comandos dentro de chaves {}.



Como no comando if-else, não se usa o ponto e vírgula (;) depois da condição do comando while.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int a,b;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     while (a < b); { //ERRO!
10         a = a + 1;
11         printf("%d \n",a);
12     }
13     system("pause");
14     return 0;
15 }
```

Como no caso dos comandos condicionais, colocar o operador de ponto e vírgula (;) logo após o comando while, faz com que o compilador entenda que o comando while já terminou e trate o comando seguinte (**a = a + 1**) como se o mesmo estivesse fora do while. No exemplo acima, temos um laço infinito (o valor de *a* e *b* nunca mudam, portanto a condição de parada nunca é atingida).



É responsabilidade do programador modificar o valor de algum dos elementos usados na condição para evitar que ocorra um laço infinito.

2.4 COMANDO FOR

O comando for é muito similar ao comando while visto anteriormente. Basicamente, o comando for é usado para repetir um comando, ou uma

sequência de comandos, diversas vezes.

A forma geral de um comando for é:

```
for (inicialização; condição; incremento) {  
    sequência de comandos;  
}
```

Na execução do comando for, a seguinte sequência de passo é realizada:

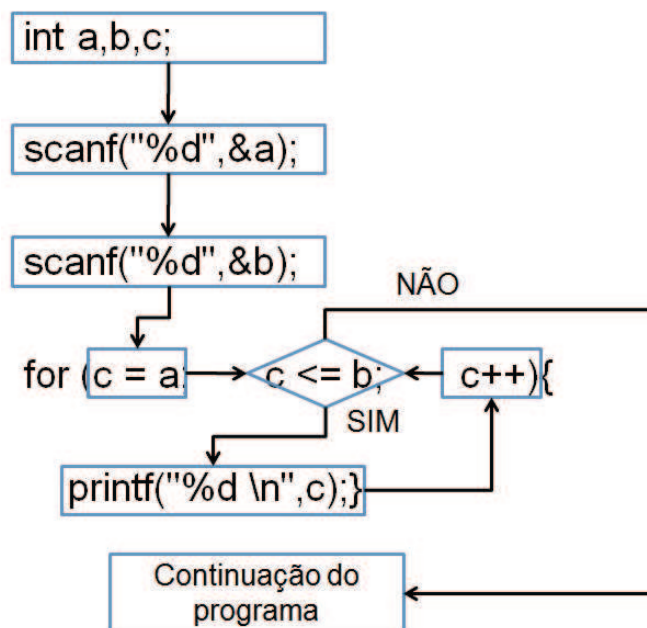
- a cláusula inicialização é executada: nela as variáveis recebem um valor inicial para usar dentro do for.
- a condição é testada:
 - se a condição for **diferente** de zero, ela será considerada **verdadeira** e a sequência de comandos será executada. Ao final da sequência de comandos, o fluxo do programa é desviado para o incremento;
 - se a condição for zero, ela será considerada **falsa** e a sequência de comandos não será executada (fim do comando for).
- incremento: terminada a execução da sequência de comandos, ocorre a etapa de incremento das variáveis usadas no for. Ao final dessa etapa, o fluxo do programa é novamente desviado para a condição.

Abaixo, tem-se um exemplo de um programa que lê dois números inteiros *a* e *b* digitados pelo usuário e imprime na tela todos os números inteiros entre *a* e *b* (incluindo *a* e *b*):

Exemplo: comando for

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 int main(){  
4     int a,b,c;  
5     printf("Digite o valor de a: ");  
6     scanf("%d",&a);  
7     printf("Digite o valor de b: ");  
8     scanf("%d",&b);  
9     for (c = a; c <= b; c++){  
10         printf("%d \n",c);  
11     }  
12     system("pause");  
13     return 0;  
14 }
```

No exemplo acima, a variável c é inicializada como valor de a ($c = a$). Em seguida, o valor de c é comparado com o valor de b ($c \leq b$). Por fim, se a sequência de comandos foi executada, o valor da variável c será incrementado em uma unidade ($c++$). Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



O comando `for` segue todas as recomendações definidas para o comando `if` e `while` quanto ao uso das chaves e definição da condição usada.

Isso significa que a condição pode ser qualquer expressão que resulte numa resposta do tipo falso (zero) ou verdadeiro (diferente de zero), e que utiliza operadores dos tipos *matemáticos*, *relacionais* e/ou *lógicos*.

Como nos comandos condicionais, o comando `while` atua apenas sobre o comando seguinte a ele. Se quisermos que ele execute uma sequência de comandos, é preciso definir essa sequência de comandos dentro de chaves `{}`.

Exemplo: for versus while	
for	while
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main(){ 4 int i, soma = 0; 5 for (i = 1; i <= 10; i ++){ 6 soma = soma + i; 7 } 8 printf("Soma = %d \n", soma); 9 system("pause"); 10 return 0; 11 }</pre>	<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main(){ 4 int i, soma = 0; 5 i = 1 6 while (i <= 10){ 7 soma = soma + i; 8 i++; 9 } 10 printf("Soma = %d \n", soma); 11 system("pause"); 12 return 0; 13 }</pre>

Dependendo da situação em que o comando for é utilizado, podemos omitir qualquer uma de suas cláusulas:

- inicialização;
- condição;
- incremento.



Independente de qual cláusula é omitida, o comando for exige que se coloque os dois operadores de ponto e vírgula (;).

O comando for exige que se coloque os dois operadores de ponto e vírgula (;) pois é este operador que indica a separação entre as cláusulas de inicialização, condição e incremento. Sem elas, o compilador não tem certeza de qual cláusula foi omitida.

Abaixo, são apresentados três exemplos de comando for onde, em cada um deles, uma das cláusulas é omitida.

Exemplo: comando for sem inicialização

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     for (; a <= b; a++){
10         printf("%d \n",a);
11     }
12     system("pause");
13     return 0;
14 }
```

No exemplo acima, a variável *a* é utilizada nas cláusulas de condição e incremento do comando *for*. Como a variável *a* teve seu valor inicial definido através de um comando de leitura do teclado (**scanf**), não é necessário a etapa de inicialização do comando *for* para definir o seu valor.



Ao omitir a condição do comando *for*, criamos um laço infinito.

Para o comando *for*, a ausência da cláusula de condição é considerada como uma condição que é sempre verdadeira. Sendo a condição sempre verdadeira, não existe condição de parada para o comando *for*, o qual vai ser executado infinitamente.

Exemplo: comando for sem condição

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     //o comando for abaixo é um laço infinito
10    for (c = a; ; c++){
11        printf("%d \n",c);
12    }
13    system("pause");
14    return 0;
15 }
```

Por último, temos um exemplo de comando for sem a cláusula de incremento. Nessa etapa do comando for, um novo valor é atribuído para uma (ou mais) variáveis utilizadas.

Exemplo: comando for sem incremento

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     for (c = a; c <= b; ){
10         printf("%d \n",c);
11         c++;
12     }
13     system("pause");
14     return 0;
15 }
```

No exemplo acima, a cláusula de incremento foi omitida da declaração do comando for. Para evitar a criação de uma laço infinito (onde a condição de parada existe, mas nunca é atingida), foi colocado um comando de incremento (c++) dentro da sequência de comandos do for. Perceba que, desse modo, o comando for fica mais parecido com o comando while, já que agora se pode definir em qual momento o incremento vai ser executado, e não apenas no final.



A cláusula de incremento é utilizada para **atribuir** um novo valor a uma ou mais variáveis durante o comando for. Essa atribuição não está restrita a apenas o operador de incremento (++).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int a,b,c;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9
10    //incremento de duas unidades
11    for (c = a; c <= b; c=c+2){
12        printf("%d \n",c);
13    }
14
15    //novo valor é lido do teclado
16    for (c = a; c <= b; scanf("%d",&c)){
17        printf("%d \n",c);
18    }
19    system("pause");
20    return 0;
21 }
```

Nesse exemplo, fica claro que a cláusula de incremento pode conter qualquer comando que altere o valor de uma das variáveis utilizadas pelo comando for.



O operador de vírgula (,) pode ser usado em qualquer uma das cláusulas.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int i,j;
5     for (i = 0, j = 100; i < j; i++, j--){
6         printf("i = %d e j = %d \n",i,j);
7     }
8     system("pause");
9     return 0;
10 }
```

No exemplo acima, foram definidos dois comandos para a cláusula de

inicialização: $i = 0$ e $j = 100$. Cada comando na inicialização é separado pelo operador de vírgula (,). A cláusula de inicialização só termina quando o operador de ponto e vírgula (;) é encontrado. Na fase de incremento, novamente o valor das duas variáveis é modificado: o valor de i é incrementado ($i++$) enquanto o de j é decrementado ($j--$). Novamente, cada comando na cláusula de incremento é separado pelo operador de vírgula (,).

2.5 COMANDO DO-WHILE

O comando do-while é bastante semelhante ao comando while visto anteriormente. Sua principal diferença é com relação a avaliação da condição: enquanto o comando while avalia a condição para depois executar uma seqüência de comandos, o comando do-while executa uma seqüência de comandos para depois testar a condição.

A forma geral de um comando do-while é:

```
do{  
    seqüência de comandos;  
} while(condição);
```

Na execução do comando do-while, a seguinte ordem de passos é executada:

- a seqüência de comandos é executada;
- a condição é avaliada:
 - se a condição for **diferente** de zero, ela será considerada **verdadeira** e o fluxo do programa é desviado novamente para o comando do, de modo que a seqüência de comandos seja executada novamente;
 - se a condição for zero, ela será considerada **falsa** e o laço termina.



O comando do-while é utilizado sempre que se desejar que a seqüência de comandos seja executada pelo menos uma vez.

No comando `while`, a condição é sempre avaliada antes da sequência de comandos. Isso significa que a condição pode ser falsa logo na primeira repetição do comando `while`, o que faria com que a sequência de comandos não fosse executada nenhuma vez. Portanto, o comando `while` pode repetir uma sequência de comandos **zero** ou mais vezes.

Já no comando `do-while`, a sequência de comandos é executada primeiro. Mesmo que a condição seja falsa logo na primeira repetição do comando `do-while`, a sequência de comandos terá sido executada pelo menos uma vez. Portanto, o comando `do-while` pode repetir uma sequência de comandos **uma** ou mais vezes.



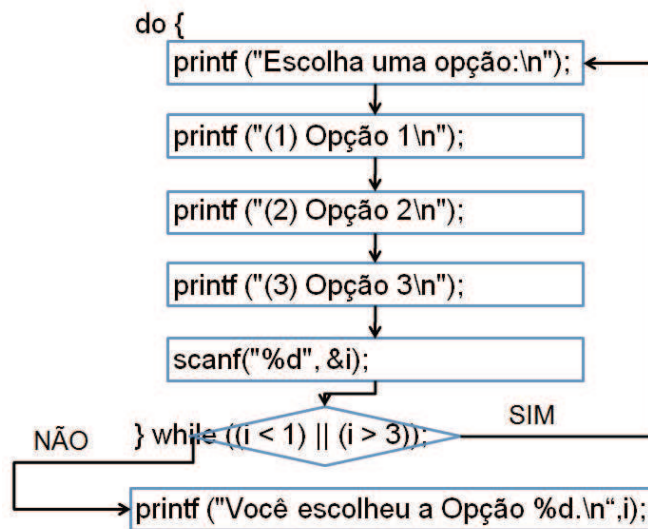
O comando `do-while` segue todas as recomendações definidas para o comando `if` quanto ao uso das chaves e definição da condição usada.

Abaixo, tem-se um exemplo de um programa que exibe um menu de opções para o usuário e espera que ele digite uma das suas opções:

Exemplo: comando `do-while`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i;
5     do {
6         printf ("Escolha uma opção:\n");
7         printf ("(1) Opção 1\n");
8         printf ("(2) Opção 2\n");
9         printf ("(3) Opção 3\n");
10        scanf("%d", &i);
11    } while ((i < 1) || (i > 3));
12    printf ("Você escolheu a Opção %d.\n", i);
13    system("pause");
14    return 0;
15 }
```

Relembrando a idéia de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados durante a execução do programa:



Diferente do comando if-else, é necessário colocar um ponto e vírgula (;) depois da condição do comando do-while.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int i = 0;
5     do{
6         printf("Valor %d\n", i);
7         i++;
8     }while(i < 10); //Esse ponto e vírgula é necessário!
9     system("pause");
10    return 0;
11 }
  
```

No comando do-while, a sequência de comandos é definida antes do teste da condição, diferente dos outros comando condicionais e de repetição. Isso significa que o teste da condição é o último comando da repetição do-while. Sendo assim, o compilador entende que a definição do comando do-while já terminou e exige que se coloque o operador de ponto e vírgula (;) após a condição.



É responsabilidade do programador modificar o valor de algum dos elementos usados na condição para evitar que ocorra um laço infinito.

2.6 COMANDO BREAK

Vimos, anteriormente, que o comando `break` pode ser utilizado em conjunto com o comando `switch`. Basicamente, sua função era interromper o comando `switch` assim que uma das seqüências de comandos da cláusula `case` fosse executada. Caso o comando `break` não existisse, a seqüência de comandos do `case` seguinte também seria executada e assim por diante.

Na verdade, o comando `break` serve para quebrar a execução de um comando (como no caso do `switch`) ou interromper a execução de qualquer comando de laço (`for`, `while` ou `do-while`). O `break` faz com que a execução do programa continue na primeira linha seguinte ao laço ou bloco que está sendo interrompido.



O comando `break` é utilizado para terminar abruptamente uma repetição. Por exemplo, se estivermos em uma repetição e um determinado resultado ocorrer, o programa deverá sair da iteração.

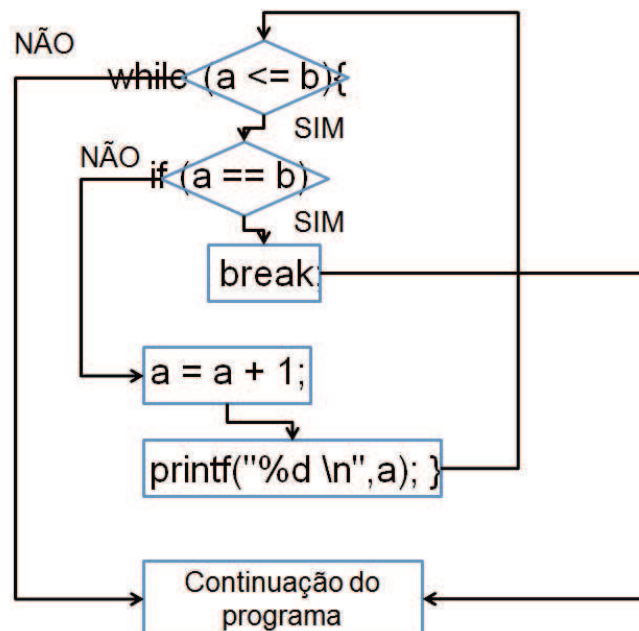
Abaixo, tem-se um exemplo de um programa que lê dois números inteiros *a* e *b* digitados pelo usuário e imprime na tela todos os números inteiros entre *a* e *b*. Note que no momento em que o valor de *a* atinge o valor de *b*, o comando `break` é chamado e o laço terminado:

Exemplo: comando break

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int a,b;
5     printf("Digite o valor de a: ");
6     scanf("%d",&a);
7     printf("Digite o valor de b: ");
8     scanf("%d",&b);
9     while (a <= b){
10         if (a == b)
11             break;
12         a = a + 1;
13         printf("%d \n",a);
14     }
15     system("pause");
16     return 0;
17 }
```

Relembrando o conceito de fluxogramas, é possível ter uma boa representação

de como os comandos do exemplo anterior são um-a-um executados pelo programa:



2.7 COMANDO CONTINUE

O comando `continue` é muito parecido com o comando `break`. Tanto o comando `break` quanto o comando `continue` ignoram o restante da sequência de comandos da repetição que os sucedem. A diferença é que, enquanto o comando `break` termina o laço de repetição, o comando `break` interrompe apenas aquela repetição e passa para a próxima repetição do laço (se ela existir).

Por esse mesmo motivo, o comando `continue` só pode ser utilizado dentro de um laço.



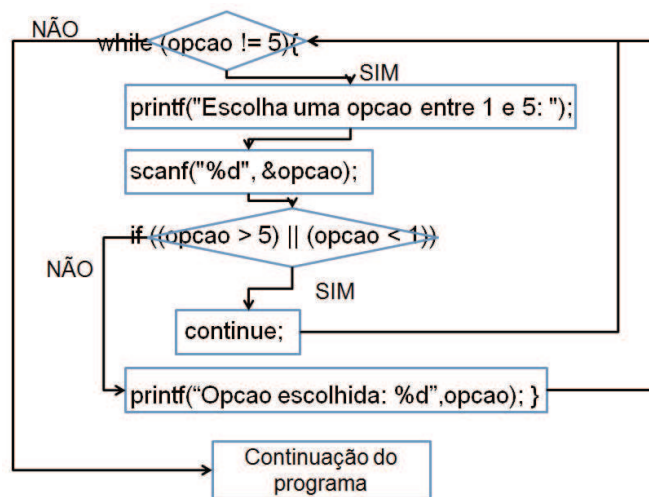
Os comandos que sucedem o comando `continue` no bloco não são executados.

Abaixo, tem-se um exemplo de um programa que lê, repetidamente, um número inteiro do usuário e a imprime apenas se ela for maior ou igual a 1 e menor ou igual a 5. Caso o número não esteja nesse intervalo, essa repetição do laço é desconsiderada e reiniciada:

Exemplo: comando continue

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int opcao = 0;
5     while (opcao != 5){
6         printf("Escolha uma opcao entre 1 e 5: ");
7         scanf("%d", &opcao);
8         if ((opcao > 5) || (opcao < 1))
9             continue;
10        printf("Opcao escolhida: %d", opcao);
11    }
12    system("pause");
13    return 0;
14 }
```

Relembrando o conceito de fluxogramas, é possível ter uma boa representação de como os comandos do exemplo anterior são um-a-um executados pelo programa:



2.8 GOTO E LABEL

O comando goto é um salto condicional para um local especificado por uma palavra chave no código. A forma geral de um comando goto é:

destino:

goto **destino**;

Na sintaxe acima, o comando **goto** (do inglês go to, literalmente "ir para") muda o fluxo do programa para um local previamente especificado pela expressão **destino**, onde **destino** é uma palavra definida pelo programador. Este local pode ser a frente ou atrás no programa, mas deve ser dentro da mesma função.

O teorema da programação estruturada prova que a instrução goto não é necessária para escrever programas; alguma combinação das três construções de programação (comandos sequenciais, condicionais e de repetição) são suficientes para executar qualquer cálculo. Além disso, o uso de goto pode deixar o programa muitas vezes ilegível.

Exemplo: goto versus for	
goto	for
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main(){ 4 int i = 0; 5 inicio: 6 if (i < 5){ 7 printf("Numero %d\n" 8 ,i); 9 i++; 10 goto inicio; 11 } 12 system("pause"); 13 return 0; 14 }</pre>	<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main(){ 4 int i; 5 for(i = 0; i < 5; i++) 6 printf("Numero %d\n" 7 ,i); 8 system("pause"); 9 return 0; 10 }</pre>

Como se nota no exemplo acima, o mesmo programa feito com o comando for é muito mais fácil de entender do que o mesmo programa feito com o comando goto.



Apesar de banido da prática de programação, o comando goto pode ser útil em determinadas circunstâncias. Ex: sair de dentro de laços aninhados.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i,j,k;
5     for(i = 0; i < 5; i++)
6         for(j = 0; j < 5; j++)
7             for(k = 0; k < 5; k++)
8                 if(i == 2 && j == 3 && k == 1)
9                     goto fim;
10                else
11                    printf("Posicao [%d,%d,%d]\n",i,j,k);
12
13
14 fim:
15 printf("Fim do programa\n");
16
17 system("pause");
18 return 0;
19 }
```

3 VETORES E MATRIZES - ARRAYS

3.1 EXEMPLO DE USO

Um array ou "vetor" é a forma mais comum de dados estruturados da linguagem C. Um array é simplesmente um conjunto de variáveis do mesmo tipo, igualmente acessíveis por um índice.



Imagine o seguinte problema: dada uma relação de 5 estudantes, imprimir o nome de cada estudante, cuja nota é maior do que a média da classe.

Um algoritmo simples para resolver esse problema poderia ser o pseudo-código apresentado abaixo:

```
Leia(nome1, nome2, nome3, nome4, nome5);  
Leia(nota1, nota2, nota3, nota4, nota5);  
media = (nota1+nota2+nota3+nota4+nota5) / 5,0;  
Se nota1 > media então escreva (nome1)  
Se nota2 > media então escreva (nome2)  
Se nota3 > media então escreva (nome3)  
Se nota4 > media então escreva (nome4)  
Se nota5 > media então escreva (nome5)
```

O algoritmo anterior representa uma solução possível para o problema. O grande inconveniente dessa solução é a grande quantidade de variáveis para gerenciarmos e o uso repetido de comandos praticamente idênticos.



Essa solução é inviável para uma lista de 100 alunos.

Expandir o algoritmo anterior para trabalhar com um total de 100 alunos significaria, basicamente, aumentar o número de variáveis para guardar os dados de cada aluno e repetir, ainda mais, um conjunto de comandos praticamente idênticos. Desse modo, teríamos:

- Uma variável para armazenar cada nome de aluno: 100 variáveis;
- Uma variável para armazenar a nota de cada aluno: 100 variáveis;
- Um comando de teste e impressão na tela para cada aluno: 100 testes.

O pseudo-código abaixo representa o algoritmo anterior expandido para poder trabalhar com 100 alunos:

```

Leia(nome1, nome2, ..., nome100);
Leia(nota1, nota2,..., nota100);
media = (nota1+nota2+...+nota100) / 100,0;
Se nota1 > media então escreva (nome1)
Se nota2 > media então escreva (nome2)
...
Se nota100 > media então escreva (nome100)

```

Como se pode notar, temos uma solução extremamente engessada para o nosso problema. Modificar o número de alunos usado pelo algoritmo implica em reescrever todo o código, repetindo comandos praticamente idênticos. Além disso, temos uma grande quantidade de variáveis para gerenciar, cada uma com o seu próprio nome, o que torna essa tarefa ainda mais difícil de ser realizada sem a ocorrência de erros.



Como estes dados têm uma relação entre si, podemos declará-los usando um ÚNICO nome para todos os 100 elementos.

Surge então a necessidade de usar um array.

3.2 ARRAY COM UMA DIMENSÃO - VETOR

A idéia de um array ou "vetor" é bastante simples: criar um conjunto de variáveis do mesmo tipo utilizando apenas um nome.

Relembrando o exemplo anterior, onde as variáveis que guardam as notas dos 100 alunos são todas do mesmo tipo, essa solução permitiria usar apenas um nome (**notas**, por exemplo) de variável para representar todas as notas dos alunos, ao invés de um nome para cada variável.

Em linguagem C, a declaração de um array segue a seguinte forma geral:

```
tipo_dado nome_array[tamanho];
```

O comando acima define um array de nome *nome_array* contendo *tamanho* elementos adjacentes na memória. Cada elemento do array é do tipo *tipo_dado*. Pensando no exemplo anterior, poderíamos usar uma array de inteiros contendo 100 elementos para guardar as notas dos 100 alunos:

```
int notas[100];
```

Como cada nota do aluno possui agora o mesmo nome que as demais notas dos outros alunos, o acesso ao valor de cada nota é feito utilizando um índice.



Para indicar qual índice do array queremos acessar, utiliza-se o operador de colchetes [].

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int notas[100];
5     int i;
6     for (i = 0; i < 100; i++) {
7         printf("Digite a nota do aluno %d", i);
8         scanf("%d", &notas[i]);
9     }
10    system("pause");
11    return 0;
12 }
```

No exemplo acima, percebe-se que cada posição do array possui todas as características de uma variável. Isso significa que ela pode aparecer em comandos de entrada e saída de dados, expressões e atribuições. Por exemplo:

```
scanf("%d",&notas[5]);  
notas[0] = 10;  
notas[1] = notas[5] + notas[0];
```



O tempo para acessar qualquer uma das posições do array é o mesmo.

Lembre-se, cada posição do array é uma variável. Portanto, todas as posições do array são igualmente acessíveis, isto é, o tempo e o tipo de procedimento para acessar qualquer uma das posições do array são iguais ao de qualquer outra variável.



Na linguagem C a numeração começa sempre do ZERO e termina em N-1, onde N é o número de elementos do array.

Isto significa que, no exemplo anterior, as notas dos alunos serão indexadas de 0 a 99:

```
notas[0]  
notas[1]  
...  
notas[99]
```

Isso acontece pelo seguinte motivo: um array é um agrupamento de dados, do mesmo tipo, adjacentes na memória. O nome do array indica onde esses dados começam na memória. O índice do array indica quantas posições se deve pular para acessar uma determinada posição. A figura abaixo exemplifica como o array está na memória:



Num array de 100 elementos, índices menores do que 0 e maiores do que 99 também podem ser acessados. Porém, isto pode resultar nos mais variados erros durante a execução do programa.

Como foi explicado, um array é um agrupamento de dados adjacentes na memória e o seu índice apenas indica quantas posições se deve pular para

Memória			
#	var	conteúdo	
123	notas	81	notas[0]
124		55	notas[1]
125		63	notas[2]
126		67	notas[3]
127		90	notas[4]
128		...	
129			
.			
.			
.			

acessar uma determinada posição. Isso significa que se tentarmos acessar o índice 100, o programa tentará acessar a centésima posição a partir da posição inicial (que é o nome do array). O mesmo vale para a posição de índice -1. Nesse caso o programa tentará acessar uma posição anterior ao local onde o array começa na memória. O problema é que, apesar dessas posições existirem na memória e serem acessíveis, elas não pertencem ao array. Pior ainda, elas podem pertencer a outras variáveis do programa, e a alteração de seus valores pode resultar nos mais variados erros durante a execução do programa.



É função do programador garantir que os limites do array estão sendo respeitados.

Deve-se tomar cuidado ao se trabalhar com arrays. Principalmente ao se usar a operação de atribuição (=).



Não se pode fazer atribuição de arrays.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int v[5] = {1,2,3,4,5};
5     int v1[5];
6     v1 = v; //ERRO!
7
8     system("pause");
9     return 0;
10 }
```

Isso ocorre porque a linguagem C não suporta a atribuição de um array para outro. Para atribuir o conteúdo de um array a outro array, o correto é copiar seus valores elemento por elemento para o outro array.

3.3 ARRAY COM DUAS DIMENSÕES - MATRIZ

Os arrays declarados até o momento possuem apenas uma dimensão. Há casos, em que uma estrutura com mais de uma dimensão é mais útil. Por exemplo, quando trabalhamos com matrizes, onde os valores são organizados em uma estrutura de linhas e colunas.

Em linguagem C, a declaração de uma matriz segue a seguinte forma geral:

```
tipo_dado nome_array[nro_linhas][nro_colunas];
```

O comando acima define um array de nome *nome_array* contendo *nro_linhas* \times *nro_colunas* elementos adjacentes na memória. Cada elemento do array é do tipo *tipo_dado*.

Por exemplo, para criar um array de inteiros que possua 100 linhas e 50 colunas, isto é, uma matriz de inteiros de tamanho 100×50 , usa-se a declaração abaixo:

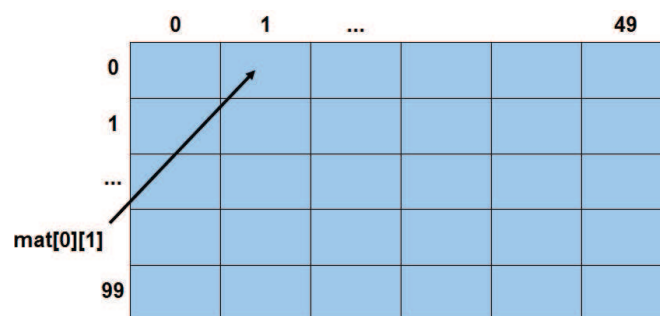
```
int mat[100][50];
```

Como no caso dos arrays de uma única dimensão, cada posição da matriz possui todas as características de uma variável. Isso significa que ela

pode aparecer em comandos de entrada e saída de dados, expressões e atribuições:

```
scanf("%d",&mat[5][0]);  
mat[0][0] = 10;  
mat[1][2] = mat[5][0] + mat[0][0];
```

Perceba, no entanto, que o acesso ao valor de uma posição da matriz é feito agora utilizando dois índices: um para a linha e outro para a coluna.



Lembre-se, cada posição do array é uma variável. Portanto, todas as posições do array são igualmente acessíveis, isto é, o tempo e o tipo de procedimento para acessar qualquer uma das posições do array são iguais ao de qualquer outra variável.

3.4 ARRAYS MULTIDIMENSIONAIS

Vimos até agora como criar arrays com uma ou duas dimensões. A linguagem C permite que se crie arrays com mais de duas dimensões de maneira fácil.



Na linguagem C, cada conjunto de colchetes [] representa uma dimensão do array.

Cada par de colchetes adicionado ao nome de uma variável durante a sua declaração adiciona uma nova dimensão àquela variável, independente do seu tipo:

```
int vet[5]; // 1 dimensão
```

float mat[5][5]; // 2 dimensões

double cub[5][5][5]; // 3 dimensões

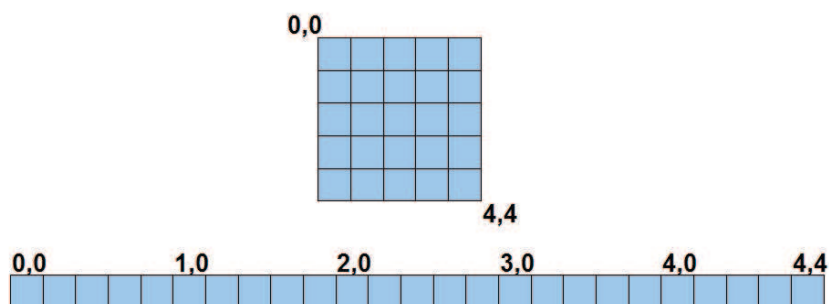
int X[5][5][5][5]; // 4 dimensões



O acesso ao valor de uma posição de um array multidimensional é feito utilizando um índice para cada dimensão do array.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int cub[5][5][5];
5     int i, j, k;
6     //preenche o array de 3 dimensões com zeros
7     for (i=0; i < 5; i++){
8         for (j=0; j < 5; j++){
9             for (k=0; k < 5; k++){
10                 cub[i][j][k] = 0;
11             }
12         }
13     }
14 }
15 system("pause");
16 return 0;
17 }
```

Apesar de terem o comportamento de estruturas com mais de uma dimensão, os dados dos arrays multidimensionais são armazenados linearmente na memória. É o uso dos colchetes que cria a impressão de estarmos trabalhando com mais de uma dimensão.



Por esse motivo, é importante ter em mente qual a dimensão que se move mais rapidamente na memória: sempre a mais a direita, independente do tipo ou número de dimensões do array, como se pode ver abaixo marcado em vermelho:

```
int vet[5]; // 1 dimensão
float mat[5][5]; // 2 dimensões
double cub[5][5][5]; // 3 dimensões
int X[5][5][5][5]; // 4 dimensões
```



Basicamente, um array multidimensional funciona como qualquer outro array. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

3.5 INICIALIZAÇÃO DE ARRAYS

Um array pode ser inicializado com certos valores durante sua declaração. Isso pode ser feito com qualquer array independente do tipo ou número de dimensões do array.

A forma geral de inicialização de um array é:

```
tipo_dado nome_array[tam1][tam2]...[tamN] = {dados };
```

Na declaração acima, dados é uma lista de valores (do mesmo tipo do array) separados por vírgula e delimitado pelo operador de chaves {}. Esses valores devem ser colocados na mesma ordem em que serão colocados dentro do array.



A inicialização de uma array utilizando o operador de chaves {} só pode ser feita durante sua declaração.

A inicialização de uma array consiste em atribuir um valor inicial a cada posição do array. O operador de chaves apenas facilita essa tarefa, como mostra o exemplo abaixo:

Exemplo: inicializando um array	
Com o operador de {}	Sem o operador de {}
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main() { 4 int vet[5] = {15,12,91,35}; 5 6 system("pause"); 7 return 0; 8 }</pre>	<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main() { 4 int vet[5]; 5 vet[0] = 15; 6 vet[1] = 12; 7 vet[2] = 9; 8 vet[3] = 1; 9 vet[4] = 35; 10 11 system("pause"); 12 return 0; 13 }</pre>

Abaixo são apresentados alguns exemplos de inicialização de arrays de diferentes tipos e número de dimensões:

Exemplos: inicializando um array
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 int main() { 4 int matriz1 [3][4] = {1,2,3,4,5,6,7,8,9,10,11,12}; 5 int matriz2 [3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}}; 6 7 char str1 [10] = {'J','o','a','o','\0'}; 8 char str2 [10] = "Joao"; 9 10 char str_matriz [3][10] = {"Joao","Maria","Jose"}; 11 12 system("pause"); 13 return 0; 14 }</pre>

Note no exemplo acima que a inicialização de um array de 2 dimensões pode ser feita de duas formas distintas. Na primeira matriz (**matriz1**) os valores iniciais da matriz são definidos utilizando um único conjunto de chaves {}, igual ao que é feito com vetores. Nesse caso, os valores são atribuídos para todas as colunas da primeira linha da matriz, para depois passar para as colunas da segunda linha e assim por diante. Lembre-se, a dimensão que se move mais rapidamente na memória é sempre a mais a direita, independente do tipo ou número de dimensões do array. Já na segunda matriz (**matriz2**) usa-se mais de um conjunto de chaves {} para definir cada uma das dimensões da matriz.

Para a inicialização de um array de caracteres, pode-se usar o mesmo princípio definido na inicialização de vetores (**str1**). Percebe-se que essa forma de inicialização não é muito prática. Por isso, a inicialização de um array de caracteres também pode ser feita por meio de "aspas duplas", como mostrado na inicialização de **str2**. O mesmo princípio é válido para iniciar um array de caracteres de mais de uma dimensão.



Na inicialização de um array de caracteres não é necessário definir todos os seus elementos.

3.5.1 INICIALIZAÇÃO SEM TAMANHO

A linguagem C também permite inicializar um array sem que tenhamos definido o seu tamanho. Nesse caso, simplesmente não se coloca o valor do tamanho entre os colchetes durante a declaração do array:

```
tipo_dado nome_array[ ] = {dados };
```

Nesse tipo de inicialização, o compilador da linguagem C vai considerar o tamanho do dado declarado como sendo o tamanho do array. Isto ocorre durante a compilação do programa. Depois disso, o tamanho do array não poderá mais ser modificado durante o programa.

Abaixo são apresentados alguns exemplos de inicialização de arrays sem tamanhos:

Exemplos: inicializando um array sem tamanho

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     //A string texto terá tamanho 13
5     //(12 caracteres + o caractere '\0')
6     char texto [ ] = "Linguagem C.";
7
8     // O número de posições do vetor será 10.
9     int vetor[ ] = {1,2,3,4,5,6,7,8,9,10};
10
11    //O número de linhas de matriz será 5.
12    int matriz [][2] = {1,2,3,4,5,6,7,8,9,10};
13
14    system("pause");
15    return 0;
16 }
```

Note no exemplo acima que foram utilizados 12 caracteres para iniciar o array de char "texto". Porém, o seu tamanho final será 13. Isso ocorre por que arrays de caracteres sempre possuem o elemento seguinte ao último caractere como sendo o caractere '\0'. Mais detalhes sobre isso podem ser vistos na seção seguinte.



Esse tipo de inicialização é muito útil quando não queremos contar quantos caracteres serão necessários para inicializarmos uma string (array de caracteres).

No caso da inicialização de arrays de mais de uma dimensão, é necessário sempre definir as demais dimensões. Apenas a primeira dimensão pode ficar sem tamanho definido.

3.6 EXEMPLO DE USO DE ARRAYS

Nesta seção são apresentados alguns exemplos de operações básicas de manipulação de vetores e matrizes em C.

Somar os elementos de um vetor de 5 inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int i, lista[5] = {3,51,18,2,45};
5     int soma = 0;
6     for(i=0; i < 5; i++)
7         soma = soma + lista[i];
8     printf("Soma = %d",soma);
9     system("pause");
10    return 0;
11 }
```

Encontrar o maior valor contido em um vetor de 5 inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i, lista[5] = {3,18,2,51,45};
5     int Maior = lista[0];
6     for(i=1; i<5; i++){
7         if (Maior < lista[i])
8             Maior = lista[i];
9     }
10    printf("Maior = %d", Maior);
11    system("pause");
12    return 0;
13 }
```

Calcular a média dos elementos de um vetor de 5 inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int i, lista[5] = {3,51,18,2,45};
5     int soma = 0;
6     for(i=0; i < 5; i++)
7         soma = soma + lista[i];
8     float media = soma / 5.0;
9     printf("Media = %f", media);
10    system("pause");
11    return 0;
12 }
```

Somar os elementos de uma matriz de inteiros

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int mat[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
5     int i, j, soma = 0;
6     for(i=0; i < 3; i++)
7         for(j=0; j < 3; j++)
8             soma = soma + mat[i][j];
9     printf("Soma = %d", soma);
10    system("pause");
11    return 0;
12 }
```

Imprimir linha por linha uma matriz

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     int mat[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
5     int i, j;
6     for(i=0; i < 3; i++){
7         for(j=0; j < 3; j++){
8             printf("%d ",mat[i][j]);
9             printf("\n");
10        }
11    system("pause");
12    return 0;
13 }
```

4 ARRAYS DE CARACTERES - STRINGS

4.1 DEFINIÇÃO E DECLARAÇÃO DE STRINGS

String é o nome que usamos para definir uma seqüência de caracteres adjacentes na memória do computador. Essa seqüência de caracteres, que pode ser uma palavra ou frase, é armazenada na memória do computador na forma de um array do tipo **char**.

Sendo a string um array de caracteres, sua declaração segue as mesmas regras da declaração de um array convencional:

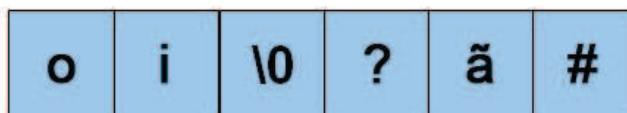
```
char str[6];
```

A declaração acima cria na memória do computador uma string (array de caracteres) de nome **str** e tamanho igual a **6**. No entanto, apesar de ser um array, devemos ficar atentos para o fato de que as strings têm no elemento seguinte a última letra da palavra/frase armazenada um caractere `'\0'`.



O caractere `'\0'` indica o fim da seqüência de caracteres.

Isso ocorre por que podemos definir uma string com um tamanho maior do que a palavra armazenada. Imagine uma string definida com um tamanho de 50 caracteres, mas utilizada apenas para armazenar a palavra "oi". Nesse caso, temos 48 posições não utilizadas e que estão preenchidas com **lixo de memória** (um valor qualquer). Obviamente, não queremos que todo esse lixo seja considerado quando essa string for exibida na tela. Assim, o caractere `'\0'` indica o fim da seqüência de caracteres e o início das posições restantes da nossa string que não estão sendo utilizadas nesse momento.



Ao definir o tamanho de uma string, devemos considerar o caractere `'\0'`.

Como o caractere `'\0'` indica o final de nossa string, isso significa que numa string definida com um tamanho de 50 caracteres, apenas 49 estarão disponíveis para armazenar o texto digitado pelo usuário.

Uma string pode ser lida do teclado ou já ser definida com um valor inicial. Para sua inicialização, pode-se usar o mesmo princípio definido na inicialização de vetores e matrizes:

```
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
```

Percebe-se que essa forma de inicialização não é muito prática. Por isso, a inicialização de strings também pode ser feita por meio de "aspas duplas":

```
char str [10] = "Joao";
```

Essa forma de inicialização possui a vantagem de já inserir o caractere `'\0'` no final da string.

Outro ponto importante na manipulação de strings é que, por se tratar de um array, cada caractere pode ser acessado individualmente por indexação como em qualquer outro vetor ou matriz:

```
char str[6] = "Teste";  
str[0] = 'L';
```

T	e	s	t	e	\0
---	---	---	---	---	----

L	e	s	t	e	\0
---	---	---	---	---	----



Na atribuição de strings usa-se "aspas duplas", enquanto que na de caracteres, usa-se 'aspas simples'.

4.2 TRABALHANDO COM STRINGS

O primeiro cuidado que temos que tomar ao se trabalhar com strings é na operação de atribuição.



Strings são arrays. Portanto, não se pode fazer atribuição de strings.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char str1[20] = "Hello World";
5     char str2[20];
6
7     str1 = str2; //ERRO!
8
9     system("pause");
10    return 0;
11 }
```

Isso ocorre porque uma string é um array e a linguagem C não suporta a atribuição de um array para outro. Para atribuir o conteúdo de uma string a outra, o correto é copiar a string elemento por elemento para a outra string.

Exemplo: Copiando uma string

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main () {
4     int count;
5     char str1[20] = "Hello World", str2[20];
6     for (count = 0; str1[count] != '\0'; count++)
7         str2[count] = str1[count];
8     str2[count] = '\0';
9     system("pause");
10    return 0;
11 }
```

O exemplo acima permite copiar uma string elemento por elemento para outra string. Note que foi utilizada a mesma forma de indexação que seria feita com um array de qualquer outro tipo (int, float, etc). Infelizmente, esse tipo de manipulação de arrays não é muito prática quando estamos trabalhando com palavras.



Felizmente, a biblioteca padrão da linguagem C possui funções especialmente desenvolvidas para a manipulação de strings na biblioteca `<string.h>`.

A seguir, serão apresentadas algumas das funções mais utilizadas para a leitura, escrita e manipulação de strings.

4.2.1 LENDO UMA STRING DO TECLADO

Existem várias maneiras de se fazer a leitura de uma sequência de caracteres do teclado. Uma delas é utilizando o já conhecido comando **scanf()** com o formato de dados `"%s"`:

```
char str[20];  
scanf("%s",str);
```



Quando usamos o comando `scanf()` para ler uma string, o símbolo de `&` antes do nome da variável não é utilizado.

Infelizmente, para muitos casos, o comando `scanf()` não é a melhor opção para se ler uma string do teclado.



O comando `scanf()` lê apenas strings digitadas sem espaços, ou seja palavras.

No caso de ter sido digitada uma frase (uma sequência de caracteres contendo espaços) apenas os caracteres digitados antes do primeiro espaço encontrado serão armazenados na string.

Uma alternativa mais eficiente para a leitura de uma string é a função **gets()**, a qual faz a leitura do teclado considerando todos os caracteres digitados (incluindo os espaços) até encontrar uma tecla enter:

```
char str[20];  
gets(str);
```

às vezes, podem ocorrer erros durante a leitura de caracteres ou strings do teclado. Para resolver esses pequenos erros, podemos limpar o buffer do teclado (entrada padrão) usando a função **setbuf(stdin, NULL)** antes de realizar a leitura de caracteres ou strings:

Exemplo: limpando o buffer do teclado	
leitura de caracteres	leitura de strings
<pre>1 char ch; 2 setbuf(stdin, NULL); 3 scanf("%c", &ch);</pre>	<pre>1 char str[10]; 2 setbuf(stdin, NULL); 3 gets(str);</pre>

Basicamente, a função **setbuf** preenche um buffer (primeiro parâmetro) com um determinado valor (segundo parâmetro). No exemplo acima, o buffer da entrada padrão (**stdin**) é preenchido com o valor vazio (**NULL**). Na linguagem C a palavra **NULL** é uma constante padrão que significa um valor nulo. Um buffer preenchido com **NULL** é considerado limpo/vazio.

Basicamente, para se ler uma string do teclado utilizamos a função **gets()**. No entanto, existe outra função que, utilizada de forma adequada, também permite a leitura de strings do teclado. Essa função é a **fgets()**, cujo protótipo é:

```
char *fgets (char *str, int tamanho, FILE *fp);
```

A função **fgets()** recebe 3 parâmetros de entrada

- **str**: a string a ser lida;
- **tamanho**: o limite máximo de caracteres a serem lidos;
- **fp**: a variável que está associado ao *arquivo* de onde a string será lida.

e retorna

- **NULL**: no caso de erro ou fim do arquivo;
- O ponteiro para o primeiro caractere da string recuperada em **str**.

Note que a função **fgets** utiliza uma variável **FILE *fp**, que está associado ao arquivo de onde a string será lida.



Para ler do teclado, basta substituir **FILE *fp** por **stdin**, o qual representa o dispositivo de entrada padrão (geralmente o teclado).

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char nome[30];
5     printf("Digite um nome: ");
6     fgets (nome, 30, stdin);
7     printf("O nome digitado foi: %s", nome);
8     system("pause");
9     return 0;
10 }
```

Como a função `gets()`, a função `fgets()` lê a string do teclado até que um caractere de nova linha (enter) seja lido. Apesar de parecerem iguais, a função `fgets` possui algumas diferenças e vantagens sobre a `gets`



Se o caractere de nova linha ('\n') for lido, ele fará parte da string, o que não acontecia com `gets`.

A função `gets()` armazena tudo que for digitado até o comando de enter. Já a função `fgets()` armazena tudo que for digitado, incluindo o comando de enter ('\n').



A função `fgets()` especifica o tamanho máximo da string de entrada.

Diferente da função `gets()`, a função `fgets()` lê a string até que um caractere de nova linha seja lido ou tamanho-1 caracteres tenham sido lidos. Isso evita o estouro do buffer, que ocorre quando se tenta ler algo maior do que pode ser armazenado na string.

4.2.2 ESCRIVENDO UMA STRING NA TELA

Basicamente, para se escrever uma string na tela utilizamos a função **`printf()`** com o formato de dados `"%s"`:

```
char str[20] = "Hello World";  
printf("%s",str);
```



Para escrever uma string, utilizamos o tipo de saída "%s".

No entanto, existe uma outra função que, utilizada de forma adequada, também permite a escrita de strings. Essa função é a **fputs()**, cujo protótipo é:

```
int fputs (char *str, FILE *fp);
```

A função **fputs()** recebe 2 parâmetros de entrada

- str: a string (array de caracteres) a ser escrita na tela;
- fp: a variável que está associado ao *arquivo* onde a string será escrita.

e retorna

- a constante EOF (em geral, -1), se houver erro na escrita;
- um valor diferente de ZERO, se o texto for escrito com sucesso.

Note que a função fputs utiliza uma variável FILE *fp, que está associado ao arquivo onde a string será escrita.



Para escrever no monitor, basta substituir *FILE *fp* por *stdout*, o qual representa o dispositivo de saída padrão (geralmente a tela do monitor).

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 int main() {  
4     \textbf{char} texto[30] = "Hello World\n";  
5     fputs (texto , stdout);  
6     system("pause");  
7     return 0;  
8 }
```

4.3 FUNÇÕES PARA MANIPULAÇÃO DE STRINGS

A biblioteca padrão da linguagem C possui funções especialmente desenvolvidas para a manipulação de strings na biblioteca `<string.h>`. A seguir são apresentadas algumas das mais utilizadas.

4.3.1 TAMANHO DE UMA STRING

Para se obter o tamanho de uma string, usa-se a função **strlen()**:

```
char str[15] = "teste";  
printf("%d",strlen(str));
```

Neste caso, a função retornará 5, que é o número de caracteres na palavra "teste" e não 15, que é o tamanho do array de caracteres.



A função **strlen()** retorna o número de caracteres até o caractere `'\0'`, e não o tamanho do array onde a string está armazenada.

4.3.2 COPIANDO UMA STRING

Vimos que uma string é um array e que a linguagem C não suporta a atribuição de um array para outro. Nesse sentido, a única maneira de atribuir o conteúdo de uma string a outra é a cópia, elemento por elemento, de uma string para outra. A linguagem C possui uma função que realiza essa tarefa para nós: a função **strcpy()**:

```
strcpy(char *destino, char *origem)
```

Basicamente, a função **strcpy()** copia a sequência de caracteres contida em *origem* para o array de caracteres *destino*:

Exemplo: strcpy()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char str1[100], str2[100];
5     printf("Entre com uma string: ");
6     gets(str1);
7     strcpy(str2, str1);
8     system("pause");
9     return 0;
10 }
```



Para evitar estouro de buffer, o tamanho do array *destino* deve ser longo o suficiente para conter a sequência de caracteres contida em *origem*.

4.3.3 CONCATENANDO STRINGS

A operação de concatenação é outra tarefa bastante comum ao se trabalhar com strings. Basicamente, essa operação consiste em copiar uma string para o final de outra string. Na linguagem C, para se fazer a concatenação de duas strings, usa-se a função **strcat()**:

strcat(char *destino, char *origem)

Basicamente, a função **strcat()** copia a sequência de caracteres contida em *origem* para o final da string *destino*. O primeiro caractere da string contida em *origem* é colocado no lugar do caractere '\0' da string *destino*:

Exemplo: strcat()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(){
4     char str1[15] = "bom ";
5     char str2[15] = "dia";
6     strcat(str1, str2);
7     printf("%s", str1);
8     system("pause");
9     return 0;
10 }
```



Para evitar estouro de buffer, o tamanho do array *destino* deve ser longo o suficiente para conter a sequência de caracteres contida em ambas as strings: *origem* e *destino*.

4.3.4 COMPARANDO DUAS STRINGS

Da mesma maneira como o operador de atribuição não funciona para strings, o mesmo ocorre com operadores relacionais usados para comparar duas strings. Desse modo, para saber se duas strings são iguais usa-se a função **strcmp()**:

```
int strcmp(char *str1, char *str2)
```

A função **strcmp()** compara posição a posição as duas strings (str1 e str2) e retorna um valor inteiro igual a zero no caso das duas strings serem iguais. Um valor de retorno diferente de zero significa que as strings são diferentes:

Exemplo: strcmp()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     char str1[100], str2[100];
5     printf("Entre com uma string: ");
6     gets(str1);
7     printf("Entre com outra string: ");
8     gets(str2);
9     if (strcmp(str1, str2) == 0)
10         printf("Strings iguais\n");
11     else
12         printf("Strings diferentes\n");
13     system("pause");
14     return 0;
15 }
```



A função **strcmp()** é case-sensitive. Isso significa que letras maiúsculas e minúsculas tornam as strings diferentes.

5 TIPOS DEFINIDOS PELO PROGRAMADOR

Os tipos de variáveis vistos até agora podem ser classificados em duas categorias:

- tipos básicos: char, int, float, double e void;
- tipos compostos homogêneos: array.

Dependendo da situação que desejamos modelar em nosso programa, esses tipos existentes podem não ser suficientes. Por esse motivo, a linguagem C permite criar novos tipos de dados a partir dos tipos básicos. Para criar um novo tipo de dado, um dos seguintes comandos pode ser utilizado:

- Estruturas: comando **struct**
- Uniões: comando **union**
- Enumerações: comando **enum**
- Renomear um tipo existente: comando **typedef**

Nas seções seguintes, cada um desses comandos será apresentado em detalhes.

5.1 ESTRUTURAS

Uma estrutura pode ser vista como uma lista de variáveis, sendo que cada uma delas pode ter qualquer tipo. A idéia básica por trás da estrutura é criar apenas um tipo de dado que contenha vários membros, que nada mais são do que outras variáveis.

A forma geral da definição de uma nova estrutura é utilizando o comando **struct**:

```
struct nomestruct{  
    tipo1 campo1;  
    tipo2 campo2;  
    ...  
    tipon campoN;  
};
```

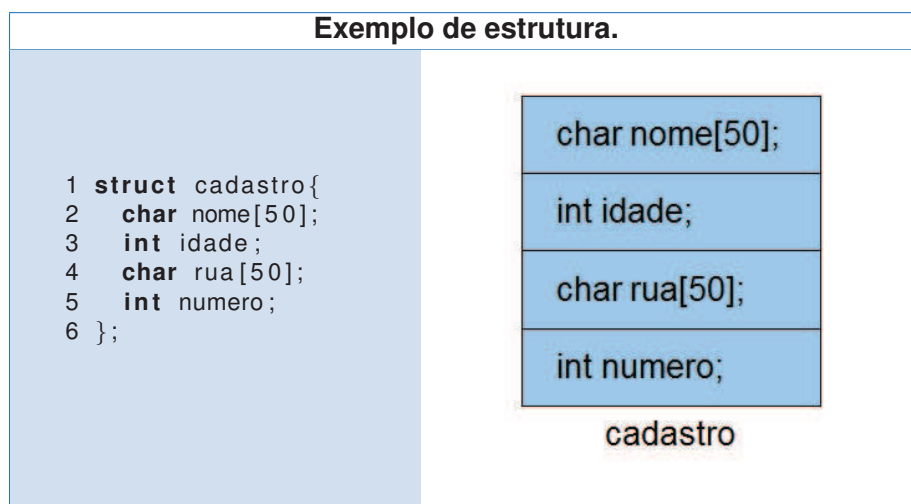

A principal vantagem do uso de estruturas é que agora podemos agrupar de forma organizada vários tipos de dados diferentes dentro de uma única variável.



As estruturas podem ser declaradas em qualquer escopo do programa (global ou local).

Apesar disso, a maioria das estruturas são declaradas no escopo global. Por se tratar de um novo tipo de dado, muitas vezes é interessante que todo o programa tenha acesso a estrutura. Daí a necessidade de usar o escopo global.

Abaixo, tem-se um exemplo de uma estrutura declarada para representar o cadastro de uma pessoa:



Note que os campos da estrutura são definidos da mesma forma que variáveis. Como na declaração de variáveis, os nomes dos membros de uma estrutura devem ser diferentes um do outro. Porém, estruturas diferentes podem ter membros com nomes iguais:

```
struct cadastro{
    char nome[50];
    int idade;
    char rua[50];
    int numero; };
```

```

struct aluno{
    char nome[50];
    int matricula
    float nota1,nota2,nota3;
};

```



Depois do símbolo de fecha chaves (}) da estrutura é necessário colocar um ponto e vírgula (;).

Isso é necessário uma vez que a estrutura pode ser também declarada no escopo local. Por questões de simplificações, e por se tratar de um novo tipo, é possível logo na definição da struct definir algumas variáveis desse tipo. Para isso, basta colocar os nomes das variáveis declaradas após o comando de fecha chaves (}) da estrutura e antes do ponto e vírgula (;):

```

struct cadastro{
    char nome[50];
    int idade;
    char rua[50];
    int numero;
} cad1, cad2;

```

No exemplo acima, duas variáveis (*cad1* e *cad2*) são declaradas junto com a definição da estrutura.

Uma vez definida a estrutura, uma variável pode ser declarada de modo similar aos tipos já existente:

```

struct cadastro c;

```



Por ser um tipo definido pelo programador, usa-se a palavra *struct* antes do tipo da nova variável declarada.

O uso de estruturas facilita muito a vida do programador na manipulação dos dados do programa. Imagine ter que declarar 4 cadastros, para 4 pessoas diferentes:

```
char nome1[50], nome2[50], nome3[50], nome4[50];
int idade1, idade2, idade3, idade4;
char rua1[50], rua2[50], rua3[50], rua4[50];
int numero1, numero2, numero3, numero4;
```

Utilizando uma estrutura, o mesmo pode ser feito da seguinte maneira:

```
struct cadastro c1, c2, c3, c4;
```

Uma vez definida uma variável do tipo da estrutura, é preciso poder acessar seus campos (ou variáveis) para se trabalhar.



Cada campo (variável) da estrutura pode ser acessada usando o operador "."(ponto).

O operador de acesso aos campos da estrutura é o ponto (.). Ele é usado para referenciar os campos de uma estrutura. O exemplo abaixo mostra como os campos da estrutura *cadastro*, definida anteriormente, podem ser facilmente acessados:

Exemplo: acessando as variáveis de dentro da estrutura

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main (){
10     struct cadastro c;
11     //Atribui a string "Carlos" para o campo nome
12     strcpy(c.nome,"Carlos");
13
14     //Atribui o valor 18 para o campo idade
15     c.idade = 18;
16
17     //Atribui a string "Avenida Brasil" para o campo rua
18     strcpy(c.rua,"Avenida Brasil");
19
20     //Atribui o valor 1082 para o campo numero
21     c.numero = 1082;
22
23     system("pause");
24     return 0;
25 }
```

Como se pode ver, cada campo da estrutura é tratado levando em consideração o tipo que foi usado para declará-la. Como os campos *nome* e *rua* são **strings**, foi preciso usar a função **strcpy()** para copiar o valor para esses campos.



E se quiséssemos ler os valores dos campos da estrutura do teclado?

Nesse caso, basta ler cada variável da estrutura independentemente, respeitando seus tipos, como é mostrado no exemplo abaixo:

Exemplo: lendo do teclado as variáveis da estrutura

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main (){
10     struct cadastro c;
11     //Lê do teclado uma string e armazena no campo nome
12     gets(c.nome);
13
14     //Lê do teclado um valor inteiro e armazena no campo idade
15     scanf("%d",&c.idade);
16
17     //Lê do teclado uma string e armazena no campo rua
18     gets(c.rua);
19
20     //Lê do teclado um valor inteiro e armazena no campo numero
21     scanf("%d",&c.numero);
22     system("pause");
23     return 0;
24 }
```

Note que cada variável dentro da estrutura pode ser acessada como se apenas ela existisse, não sofrendo nenhuma interferência das outras.



Lembre-se: uma estrutura pode ser vista como um simples agrupamento de dados.

Como cada campo é independente um do outro, outros operadores podem ser aplicados a cada campo. Por exemplo, pode se comparar a idade de dois cadastros.

5.1.1 INICIALIZAÇÃO DE ESTRUTURAS

Assim como nos arrays, uma estrutura também pode ser inicializada, independente do tipo das variáveis contidas nela. Para tanto, na declaração da variável do tipo da estrutura, basta definir uma lista de valores separados por vírgula e delimitado pelo operador de chaves {}.

```
struct cadastro c = {"Carlos",18,"Avenida Brasil",1082 };
```

Nesse caso, como nos arrays, a ordem é mantida. Isso significa que o primeiro valor da inicialização será atribuído a primeira variável membro (*nome*) da estrutura e assim por diante.

Elementos omitidos durante a inicialização são inicializados com 0. Se for uma string, a mesma será inicializada com uma string vazia ().

```
struct cadastro c = {"Carlos",18 };
```

No exemplo acima, o campo *rua* é inicializado com e *numero* com zero.

5.1.2 ARRAY DE ESTRUTURAS

Voltemos ao problema do cadastro de pessoas. Vimos que o uso de estruturas facilita muito a vida do programador na manipulação dos dados do programa. Imagine ter que declarar 4 cadastros, para 4 pessoas diferentes:

```
char nome1[50], nome2[50], nome3[50], nome4[50];  
int idade1, idade2, idade3, idade4;  
char rua1[50], rua2[50], rua3[50], rua4[50];  
int numero1, numero2, numero3, numero4;
```

Utilizando uma estrutura, o mesmo pode ser feito da seguinte maneira:

```
struct cadastro c1, c2, c3, c4;
```

A representação desses 4 cadastros pode ser ainda mais simplificada se utilizarmos o conceito de arrays:

```
struct cadastro c[4];
```

Desse modo, cria-se um array de estruturas, onde cada posição do array é uma estrutura do tipo cadastro.



A declaração de uma array de estruturas é similar a declaração de uma array de um tipo básico.

A combinação de arrays e estruturas permite que se manipule de modo muito mais prático várias variáveis de estrutura. Como vimos no uso de arrays, o uso de um índice permite que usemos comando de repetição para executar uma mesma tarefa para diferentes posições do array. Agora, os quatro cadastros anteriores podem ser lidos com o auxílio de um comando de repetição:

Exemplo: lendo um array de estruturas do teclado

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main () {
10    struct cadastro c[4];
11    int i;
12    for (i=0; i<4; i++){
13        gets(c[i].nome);
14        scanf("%d",&c[i].idade);
15        gets(c[i].rua);
16        scanf("%d",&c[i].numero);
17    }
18    system("pause");
19    return 0;
20 }
```



Em um array de estruturas, o operador de ponto (.) vem depois dos colchetes [] do índice do array.

Essa ordem deve ser respeitada pois o índice do array é quem indica qual posição do array queremos acessar, onde cada posição do array é uma estrutura. Somente depois de definida qual das estruturas contidas dentro do array nós queremos acessar é que podemos acessar os seus campos.

5.1.3 ATRIBUIÇÃO ENTRE ESTRUTURAS

As únicas operações possíveis em um estrutura são as de acesso aos membros da estrutura, por meio do operador ponto (.), e as de cópia ou atribuição (=). A atribuição entre duas variáveis de estrutura faz com que os conteúdos das variáveis contidas dentro de uma estrutura sejam copiado para outra estrutura.



Atribuições entre estruturas só podem ser feitas quando as estruturas são AS MESMAS, ou seja, possuem o mesmo nome!

Exemplo: atribuição entre estruturas

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct ponto {
5     int x;
6     int y;
7 };
8
9 struct novo_ponto {
10    int x;
11    int y;
12 };
13
14 int main () {
15     struct ponto p1, p2= {1,2};
16     struct novo_ponto p3= {3,4};
17
18     p1 = p2;
19     printf("p1 = %d e %d", p1.x, p1.y);
20
21     //ERRO! TIPOS DIFERENTES
22     p1 = p3;
23     printf("p1 = %d e %d", p1.x, p1.y);
24
25     system("pause");
26     return 0;
27 }
```

No exemplo acima, *p2* é atribuído a *p1*. Essa operação está correta pois ambas as variáveis são do tipo **ponto**. Sendo assim, o valor de *p2.x* é copiado para *p1.x* e o valor de *p2.y* é copiado para *p1.y*.

Já na segunda atribuição (*p1 = p3;*) ocorre um erro. Isso por que os tipos das estruturas das variáveis são diferentes: uma pertence ao tipo *struct ponto* enquanto a outra pertence ao tipo *struct novo_ponto*. Note que o mais importante é o nome do tipo da estrutura, e não as variáveis dentro dela.



No caso de estarmos trabalhando com arrays de estruturas, a atribuição entre diferentes elementos do array também é válida.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct cadastro{
4     char nome[50];
5     int idade;
6     char rua[50];
7     int numero;
8 };
9 int main(){
10     struct cadastro c[10];
11     ...
12     c[1] = c[2]; //CORRETO
13
14     system("pause");
15     return 0;
16 }
```

Um array ou "vetor" é um conjunto de variáveis do mesmo tipo utilizando apenas um nome. Como todos os elementos do array são do mesmo tipo, a atribuição entre elas é possível, mesmo que o tipo do array seja uma estrutura.

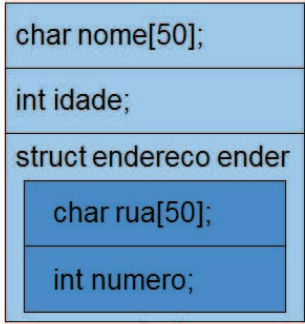
5.1.4 ESTRUTURAS ANINHADAS

Uma estrutura pode agrupar um número arbitrário de variáveis de tipos diferentes. Uma estrutura também é um tipo de dado, com a diferença de se trata de um tipo de dado criado pelo programador. Sendo assim, podemos declarar uma estrutura que possua uma variável do tipo de outra estrutura previamente definida. A uma estrutura que contenha outra estrutura

dentro dela damos o nome de **estruturas aninhadas**. O exemplo abaixo exemplifica bem isso:

Exemplo: struct aninhada.

```
1 struct endereco{
2   char rua[50]
3   int numero;
4 };
5 struct cadastro{
6   char nome[50];
7   int idade;
8   struct endereco
      ender;
9 };
```



cadastro

No exemplo acima, temos duas estruturas: uma chamada **endereco** e outra chamada de **cadastro**. Note que a estrutura **cadastro** possui uma variável *ender* do tipo **struct endereco**. Trata-se de uma estrutura aninhada dentro de outra.



No caso da estrutura **cadastro**, o acesso aos dados da variável do tipo **struct endereco** é feito utilizando-se novamente o operador "."(ponto).

Lembre-se, cada campo (variável) da estrutura pode ser acessada usando o operador "."(ponto). Assim, para acessar a variável *ender* é preciso usar o operador ponto (.). No entanto, a variável *ender* também é uma estrutura. Sendo assim, o operador ponto (.) é novamente utilizado para acessar as variáveis dentro dessa estrutura. Esse processo se repete sempre que houver uma nova estrutura aninhada. O exemplo abaixo mostra como a estrutura aninhada **cadastro** poderia ser facilmente lida do teclado:

Exemplo: lendo do teclado as variáveis da estrutura

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 struct endereco{
4     char rua[50]
5     int numero;
6 };
7 struct cadastro{
8     char nome[50];
9     int idade;
10    struct endereco ender;
11 };
12 int main (){
13     struct cadastro c;
14     //Lê do teclado uma string e armazena no campo nome
15     gets(c.nome);
16
17     //Lê do teclado um valor inteiro e armazena no campo idade
18     scanf("%d",&c.idade);
19
20     //Lê do teclado uma string
21     //e armazena no campo rua da variável ender
22     gets(c.ender.rua);
23
24     //Lê do teclado um valor inteiro
25     //e armazena no campo numero da variável ender
26     scanf("%d",&c.ender.numero);
27
28     system("pause");
29     return 0;
30 }
```

5.2 UNIÕES: UNIONS

Em breve

5.3 ENUMERAÇÕES: ENUMERATIONS

Em breve

5.4 COMANDO TYPEDEF

Em breve

6 FUNÇÕES

Uma função nada mais é do que um bloco de código (ou seja, declarações e outros comandos) que podem ser nomeados e chamados de dentro de um programa. Em outras palavras, uma função é uma sequência de comandos que recebe um nome e pode ser chamada de qualquer parte do programa, quantas vezes forem necessárias, durante a execução do programa.

A linguagem C possui muitas funções já implementadas e nós temos utilizadas elas constantemente. Um exemplo delas são as funções básicas de entrada e saída: **scanf()** e **printf()**. O programador não precisa saber qual o código contido dentro das funções de entrada e saída para utilizá-las. Basta saber seu nome e como utilizá-la.

A seguir, serão apresentados os conceitos e detalhes necessários para um programador criar suas próprias funções.

6.1 DEFINIÇÃO E ESTRUTURA BÁSICA

Duas são as principais razões para o uso de funções:

- **estruturação dos programas;**
- **reutilização de código.**

Por **estruturação dos programas** entende-se que agora o programa será construído a partir de pequenos blocos de código (isto é, funções) cada um deles com uma tarefa específica e bem definida. Isso facilita a compreensão do programa.



Programas grandes e complexos são construídos bloco a bloco com a ajuda de funções.

Já por **reutilização de código** entende-se que uma função é escrita para realizar uma determinada tarefa. Pode-se definir, por exemplo, uma função para calcular o fatorial de um determinado número. O código para essa função irá aparecer uma única vez em todo o programa, mas a função que calcula o fatorial poderá ser utilizadas diversas vezes e em pontos diferentes do programa.



O uso de funções evita a cópia desnecessária de trechos de código que realizam a mesma tarefa, diminuindo assim o tamanho do programa e a ocorrência de erros.

Em linguagem C, a declaração de uma função pelo programador segue a seguinte forma geral:

```
tipo_retornado nome_função (lista_de_parâmetros){  
    sequência de declarações e comandos  
}
```

O **nome_função** é como aquele trecho de código será conhecido dentro do programa. Para definir esse nome, valem, basicamente, as mesmas regras para se definir uma variável.

Com relação ao local de declaração de uma função, ela deve ser definida ou declarada antes de ser utilizada, ou seja, antes da cláusula **main**, como mostra o exemplo abaixo:



Exemplo: função declarada **antes** da cláusula **main**.

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int Square (int a){  
5     return (a*a);  
6 }  
7  
8 int main (){  
9     int num;  
10    printf ("Entre com um numero: ");  
11    scanf ("%d", &num);  
12    num = Square(num);  
13    printf ("O seu quadrado vale: %d\n", num);  
14    system("pause");  
15    return 0;  
16 }
```

Pode-se também declarar uma função depois da cláusula **main**. Nesse caso, é preciso declarar antes o protótipo da função:

tipo_retornado nome_função (lista_de_parâmetros);

O protótipo de uma função, é uma declaração de função que omite o corpo mas especifica o seu nome, tipo de retorno e lista de parâmetros, como mostra o exemplo abaixo:



Exemplo: função declarada **depois** da cláusula *main*.

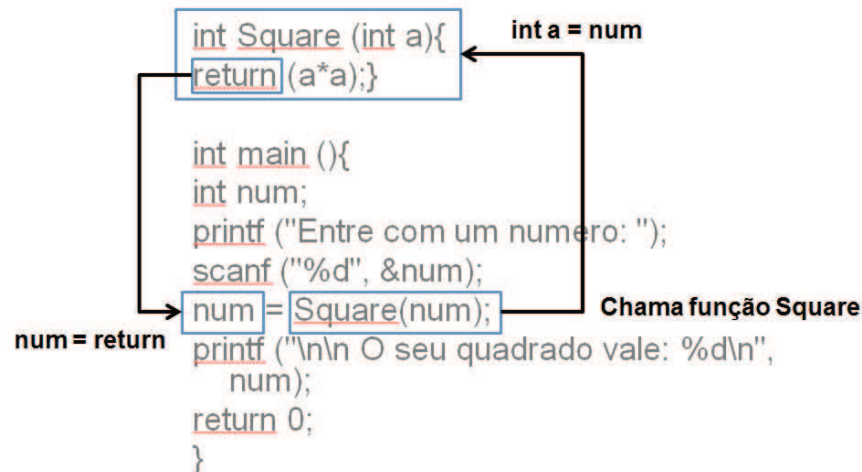
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 //protótipo da função
4 int Square (int a);
5
6 int main () {
7     int num;
8     printf ("Entre com um numero: ");
9     scanf ("%d", &num);
10    num = Square(num);
11    printf ("O seu quadrado vale: %d\n", num);
12    system("pause");
13    return 0;
14 }
15
16 int Square (int a){
17     return (a*a);
18 }
```

Independente de onde uma função seja declarada, seu funcionamento é basicamente o mesmo:

- o código do programa é executado até encontrar uma chamada de função;
- o programa é então interrompido temporariamente, e o fluxo do programa passa para a função chamada;
- se houver parâmetros na função, os valores da chamada da função são copiados para os parâmetros no código da função;
- os comandos da função são executados;
- quando a função termina (seus comandos acabaram ou o comando **return** foi encontrado), o programa volta ao ponto onde foi interrompido para continuar sua execução normal;

- se houver um comando **return**, o valor dele será copiado para a variável que foi escolhida para receber o retorno da função.

Na figura abaixo, é possível ter uma boa representação de como uma chamada de função ocorre:



Nas seções seguintes, cada um dos itens que definem uma função serão apresentados em detalhes.

6.1.1 PARÂMETROS DE UMA FUNÇÃO

Os parâmetros de uma função é o que o programador utiliza para passar a informação de um trecho de código para dentro da função. Basicamente, os parâmetros de uma função são uma lista de variáveis, separadas por vírgula, onde é especificado o tipo e o nome de cada parâmetro.



Por exemplo, a função **sqrt** possui a seguinte lista de parâmetros: `float sqrt(float x);`

Em linguagem C, a declaração dos parâmetros de uma função segue a seguinte forma geral:

```

tipo_retornado nome_função (tipo nome1, tipo nome2, ... ,
                             tipo nomeN){
    sequência de declarações e comandos
}

```



Diferente do que acontece na declaração de variáveis, onde muitas variáveis podem ser declaradas com o mesmo especificador de tipo, na declaração de parâmetros de uma função é necessário especificar o tipo de cada variável.

```
1 //Declaração CORRETA de parâmetros
2 int soma(int x, int y){
3     return x + y;
4 }
5
6 //Declaração ERRADA de parâmetros
7 int soma(int x, y){
8     return x + y;
9 }
```

Dependendo da função, ela pode possuir nenhum parâmetro. Nesse caso, pode-se optar por duas soluções:

- Deixar a lista de parâmetros vazia: `void imprime ();`
- Colocar void entre parênteses: `void imprime (void).`



Mesmo se não houver parâmetros na função, os parênteses ainda são necessários.

Apesar das duas declarações estarem corretas, existe uma diferença entre elas. Na primeira declaração, não é especificado nenhum parâmetro, portanto a função pode ser chamada passando-se valores para ela. O compilador não irá verificar se a função é realmente chamada sem argumentos e a função não conseguirá ter acesso a esses parâmetros. Já na segunda declaração, nenhum parâmetro é esperado. Nesse caso, o programa acusará um erro se o programador tentar passar um valor para essa função.



Colocar void na lista de parâmetros é diferente de se colocar nenhum parâmetro.

O exemplo abaixo ilustra bem essa situação:

Exemplo: função sem parâmetros	
Sem void	Com void
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 4 void imprime(){ 5 printf("Teste de funcao\n"); 6 } 7 8 int main (){ 9 imprime(); 10 imprime(5); 11 imprime(5, 'a'); 12 13 system("pause"); 14 return 0; 15 }</pre>	<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 4 void imprime(void){ 5 printf("Teste de funcao\n"); 6 } 7 8 int main (){ 9 imprime(); 10 imprime(5); //ERRO 11 imprime(5, 'a'); //ERRO 12 13 system("pause"); 14 return 0; 15 }</pre>

Os parâmetros das funções também estão sujeitos ao escopo das variáveis. O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa.



O parâmetro de uma função é uma variável local da função e portanto, só pode ser acessado dentro da função.

6.1.2 CORPO DA FUNÇÃO

Pode-se dizer que o corpo de uma função é a sua alma. É no corpo de uma função que se define qual a tarefa que a função irá realizar quando for chamada.

Basicamente, o corpo da função é formado por:

- sequência de declarações: variáveis, constantes, arrays, etc;
- sequência de comandos: comandos condicionais, de repetição, chamada de outras funções, etc.

Para melhor entender o corpo da função, considere que todo programa possui ao menos uma função: a função **main**. A função mais é a função

"principal"do programa, o "corpo"do programa. Note que nos exemplo usados até agora, a função main é sempre do tipo int, e sempre retorna o valor 0:

```
int main () {  
    sequência de declarações e comandos  
    return 0;  
}
```

Basicamente, é no corpo da função que as entradas (parâmetros) são processadas, as saídas são geradas ou outras ações são feitas. Além disso, a função main se encarrega de realizar a comunicação com o usuário, ou seja, é ela quem realiza as operações de entrada e saída de dados (comandos **scanf** e **printf**). Desse modo, tudo o que temos feito dentro de uma função main pode ser feito em uma função desenvolvida pelo programador.



Tudo o que temos feito dentro da função main pode ser feito em uma função desenvolvida pelo programador.

Uma função é construída com o intuito de realizar uma tarefa específica e bem definida. Por exemplo, uma função para calcular o fatorial deve ser construída de modo a receber um determinado número como parâmetro e retornar (usando o comando return) o valor calculado. As operações de entrada e saída de dados (comandos **scanf** e **printf**) devem ser feitas em quem chamou a função (por exemplo, na **main**). Isso garante que a função construída possa ser utilizada nas mais diversas aplicações, garantindo a sua generalidade.



De modo geral, evita-se fazer operações de leitura e escrita dentro de uma função.

Os exemplos abaixo ilustram bem essa situação. No primeiro exemplo temos o cálculo do fatorial realizado dentro da função main:

Exemplo: cálculo do fatorial dentro da função main

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main () {
5     printf("Digite um numero inteiro positivo:");
6     int x;
7     scanf("%d",&x);
8     int i, f = 1;
9     for (i=1; i<=x; i++)
10         f = f * i;
11
12     printf("O fatorial de %d eh: %d\n",x,f);
13     system("pause");
14     return 0;
15 }
```

Perceba que no exemplo acima, não foi feito nada de diferente do que temos feito até o momento. Já no exemplo abaixo, uma função específica para o cálculo do fatorial foi construída:

Exemplo: cálculo do fatorial em uma função própria

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int fatorial (int n){
5     int i, f = 1;
6     for (i=1; i<=n; i++)
7         f = f * i;
8
9     return f;
10 }
11
12 int main () {
13     printf("Digite um numero inteiro positivo:");
14     int x;
15     scanf("%d",&x);
16     int fat = fatorial(x);
17     printf("O fatorial de %d eh: %d\n",x,fat);
18
19     system("pause");
20     return 0;
21 }
```

Note que dentro da função responsável pelo cálculo do fatorial, apenas o trecho do código responsável pelo cálculo do fatorial está presente. As operações de entrada e saída de dados (comandos **scanf** e **printf**) são feitos em quem chamou a função **fatorial**, ou seja, na função **main**.



Operações de leitura e escrita não são proibidas dentro de uma função. Apenas não devem ser usadas se esse não for o foco da função.

Uma função deve conter apenas o trecho de código responsável por fazer aquilo que é o objetivo da função. Isso não impede que operações de leitura e escrita sejam utilizadas dentro da função. Elas só não devem ser usadas quando os valores podem ser passados para a função por meio dos parâmetros.

Abaixo temos um exemplo de função que realiza operações de leitura e escrita:

Exemplo: função contendo operações de leitura e escrita.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int menu(){
4     int i;
5     do {
6         printf ("Escolha uma opção:\n");
7         printf ("(1) Opcao 1\n");
8         printf ("(2) Opcao 2\n");
9         printf ("(3) Opcao 3\n");
10        scanf("%d", &i);
11    } while ((i < 1) || (i > 3));
12
13    return i;
14 }
15
16 int main(){
17     int op = menu();
18     printf ("Vc escolheu a Opcao %d.\n",op);
19     system("pause");
20     return 0;
21 }
```

Na função acima, um menu de opções é apresentado ao usuário que tem de escolher dentre uma delas. A função se encarrega de verificar se a opção digitada é válida e, caso não seja, solicitar uma nova opção ao usuário.

6.1.3 RETORNO DA FUNÇÃO

O retorno da função é a maneira como uma função devolve o resultado (se ele existir) da sua execução para quem a chamou. Nas seções anteriores vimos que uma função segue a seguinte forma geral:

```
tipo_retornado nome_função (lista_de_parâmetros){  
    sequência de declarações e comandos  
}
```

A expressão **tipo_retornado** estabelece o tipo de valor que a função irá devolver para quem chamá-la. Uma função pode retornar qualquer tipo válido em na linguagem C:

- tipos básicos pré-definidos: int, char, float, double, void e ponteiros;
- tipos definidos pelo programador: struct, array (indiretamente), etc.



Uma função também pode NÃO retornar um valor. Para isso, basta colocar o tipo **void** como valor retornado.

O tipo **void** é conhecido como o tipo *vazio*. Uma função declarada com o tipo **void** irá apenas executar um conjunto de comando e não irá devolver nenhum valor para quem a chamar. Veja o exemplo abaixo:

Exemplo: função com tipo void

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 void imprime(int n){  
4     int i;  
5     for (i=1; i<=n; i++)  
6         printf("Linha %d \n", i);  
7 }  
8  
9 int main() {  
10     imprime(5);  
11  
12     system("pause");  
13     return 0;  
14 }
```

No exemplo acima, a função **imprime** irá apenas imprimir uma mensagem na tela **n** vezes. Não há o que devolver para a função **main**. Portanto, podemos declarar ela como **void**.



Para executar uma função do tipo **void**, basta colocar no código o nome da função e seus parâmetros.

Se a função não for do tipo **void**, então ela deverá retornar um valor. O comando **return** é utilizado para retornar esse valor para o programa:

return *expressão*;



A expressão da cláusula **return** tem que ser compatível com o tipo de retorno declarado para a função.

A *expressão* do comando **return** consiste em qualquer constante, variável ou expressão aritmética que o programador deseje retornar para o trecho do programa que chamou a função. Essa expressão pode até mesmo ser uma outra função, como a função **sqrt()**:

return sqrt(x);



Para executar uma função que tenha o comando **return**, basta atribuir a chamada da função (nome da função e seus parâmetros) a uma variável compatível com o tipo do retorno.

O exemplo abaixo mostra uma função que recebe dois parâmetros inteiros e retorna a sua soma para a função **main**:

Exemplo: função com retorno

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int soma(int x, int y){
4     return x + y;
5 }
6
7 int main() {
8     int a,b,c;
9     printf (" Digite a: ");
10    scanf("%d", &a);
11    printf (" Digite b: ");
12    scanf("%d", &b);
13    printf (" Soma = %d\n",soma(a,b) );
14    system("pause");
15    return 0;
16 }
```

Note, no exemplo acima, que a chamada da função foi feita dentro do comando **printf**. Isso é possível pois a função retorna um valor inteiro (**x+y**) e o comando **printf** espera imprimir um valor inteiro (**%d**).



Uma função pode ter mais de uma declaração **return**.

O uso de vários comandos **return** é útil quando o retorno da função está relacionado a uma determinada condição dentro da função. Veja o exemplo abaixo:

Exemplo: função com vários return

```
1 int maior(int x, int y){
2     if (x > y)
3         return x;
4     else
5         return y;
6 }
```

No exemplo acima, a função será executada e dependendo dos valores de *x* e *y*, uma das cláusulas **return** será executada. No entanto, é conveniente limitar as funções a usar somente um comando **return**. O uso de vários comandos **return**, especialmente em função grandes e complexas, aumenta a dificuldade de se compreender o que realmente está sendo feito pela função. Na maioria dos casos, pode-se reescrever uma função para que ela use somente um comando **return**, como é mostrado abaixo:

Exemplo: substituindo os vários return da função

```
1 int maior(int x, int y){
2     int z;
3     if (x > y)
4         z = x;
5     else
6         z = y;
7     return z;
8 }
```

No exemplo acima, os vários comando **return** foram substituídos por uma variável que será retornada no final da função.



Quando se chega a um comando **return**, a função é encerrada imediatamente.

O comando **return** é utilizado para retornar um valor para o programa. No entanto, esse comando também é usado para terminar a execução de uma função, similar ao comando **break** em um laço ou **switch**:

Exemplo: finalizando a função com return

```
1 int maior(int x, int y){
2     if (x > y)
3         return x;
4     else
5         return y;
6     printf("Fim da funcao\n");
7 }
```

No exemplo acima, a função irá terminar quando um dos comando **return** for executado. A mensagem "Fim da funcao"jamais será impressa na tela pois seu comando se encontra depois do comando **return**. Nesse caso, o comando **printf** será ignorado.



O comando **return** pode ser usado sem um valor associado a ele para terminar uma função do tipo **void**.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 void imprime_log(float x){
5     if (x <= 0)
6         return; //termina a função
7     printf("Log = %f\n", log(x));
8 }
9 int main(){
10     float x;
11     printf("Digite x: ");
12     scanf("%f", &f);
13     imprime_log(x);
14     system("pause");
15     return 0;
16 }
```

Na função contida no exemplo acima, se o valor de *x* é negativo ou zero, o comando **return** faz com que a função termine antes que o comando **printf** seja executado, mas nenhum valor é retornado.



O valor retornado por uma função não pode ser um array.

Lembre-se: a linguagem C não suporta a atribuição de um array para outro. Por esse motivo, não se pode ter como retorno de uma função um array.



É possível retornar um array indiretamente, desde que ela faça parte de uma estrutura.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 struct vetor{
5     int v[5];
6 };
7
8 struct vetor retorna_array(){
9     struct vetor v = {1,2,3,4,5};
10    return v;
11 }
12
13 int main (){
14     int i;
15     struct vetor vet = retorna_array();
16     for (i=0; i<5; i++)
17         printf("Valores: %d \n",vet.v[i]);
18     system("pause");
19     return 0;
20 }
```

A linguagem C não suporta a atribuição de um array para outro. Mas ela permite a atribuição entre estruturas. A atribuição entre duas variáveis de estrutura faz com que os conteúdos das variáveis contidas dentro de uma estrutura sejam copiados para outra estrutura. Desse modo, é possível retornar um array desde que o mesmo esteja dentro de uma estrutura.

6.2 TIPOS DE PASSAGEM DE PARÂMETROS

Já vimos que, na linguagem C, os parâmetros de uma função é o mecanismo que o programador utiliza para passar a informação de um trecho de código para dentro da função. Mas existem dois tipos de passagem de parâmetro: passagem **por valor** e **por referência**.

Nas seções seguintes, cada um dos tipos de passagem de parâmetros será explicado em detalhes.

6.2.1 PASSAGEM POR VALOR

Na linguagem C, os argumentos para uma função são sempre passados por valor (by value), ou seja, uma cópia do dado é feita e passada para a função. Esse tipo de passagem de parâmetro é o padrão para todos os tipos básicos pré-definidos (**int**, **char**, **float** e **double**) e estruturas definidas pelo programador (**struct**).



Mesmo que o valor de uma variável mude dentro da função, nada acontece com o valor de fora da função.

```
1 include <stdio.h>
2 include <stdlib.h>
3
4 void soma_mais_um( int n){
5     n = n + 1;
6     printf ( "Antes da funcao: x = %d\n",n);
7 }
8
9 int main (){
10     int x = 5;
11     printf ( "Antes da funcao: x = %d\n",x);
12     soma_mais_um(x);
13     printf ( "Antes da funcao: x = %d\n",x);
14     system("pause");
15     return 0;
16 }
```

Saída Antes da funcao: x = 5
 Dentro da funcao: x = 6
 Depois da funcao: x = 5

No exemplo acima, no momento em que a função **soma_mais_um** é chamada, o valor de **x** é **copiado** para o parâmetro **n** da função. O parâmetro **n** é uma variável local da função. Então, tudo o que acontecer com ele (**n**) não se reflete no valor **original** da variável **x**. Quando a função termina, a variável **n** é destruída e seu valor é descartado. O fluxo do programa é devolvido ao ponto onde a função foi inicialmente chamada, onde a variável **x** mantém o seu valor **original**.



Na passagem de parâmetros por valor, quaisquer modificações que a função fizer nos parâmetros existem apenas dentro da própria função.

6.2.2 PASSAGEM POR REFERÊNCIA

Na passagem de parâmetros por valor, as funções não podem modificar o valor original de uma variável passada para a função. Mas existem casos em que é necessário que toda modificação feita nos valores dos parâmetros dentro da função sejam repassados para quem chamou a função. Um exemplo bastante simples disso é a função **scanf**: sempre que desejamos ler algo do teclado, passamos para a função **scanf** o nome da variável onde o dado será armazenado. Essa variável tem seu valor modificado dentro da função **scanf** e seu valor pode ser acessado no programa principal.



A função **scanf** é um exemplo bastante simples de função que altera o valor de uma variável e essa mudança se reflete fora da função.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int x = 5;
5     printf ("Antes do scanf: x = %d\n", x);
6     printf ("Digite um numero: ");
7     scanf ("%d", &x);
8     printf ("Depois do scanf: x = %d\n", x);
9     system("pause");
10    return 0;
11 }
```

Quando se quer que o valor da variável mude dentro da função e essa mudança se reflita fora da função, usa-se passagem de parâmetros **por referência**.



Na passagem de parâmetros por referência não se passa para a função os valores das variáveis, mas sim os *endereços das variáveis na memória*.

Na passagem de parâmetros por referência o que é enviado para a função é o endereço de memória onde a variável está armazenada, e não uma simples cópia de seu valor. Assim, utilizando o endereço da variável na memória, qualquer alteração que a variável sofra dentro da função será também refletida fora da função.



Para passar um parâmetro por referência, usa-se o operador "*" na frente do nome do parâmetro durante a declaração da função.

Para passar para a função um parâmetro por referência, a função precisa usar ponteiros. Um ponteiro é um tipo especial de variável que armazena um endereço de memória, da mesma maneira como uma variável armazena um valor. Mais detalhes sobre o uso de ponteiros serão apresentados no capítulo seguinte.

O exemplo abaixo mostra a mesma função declarada usando a passagem de parâmetro de valor e por referência:

Exemplo: passagem por valor e referência	
Por valor	Por referência
<pre>1 void soma_mais_um (int n) { 2 n = n + 1; 3 }</pre>	<pre>1 void soma_mais_um (int *n) { 2 *n = *n + 1; 3 }</pre>

Note, no exemplo acima, que a diferença entre os dois tipos de passagem de parâmetro é o uso do operador "*" na passagem por referência. Consequentemente, toda vez que a variável passada por referência for usada dentro da função, o operador "*" deverá ser usado na frente do nome da variável.



Na chamada da função é necessário utilizar o operador "&" na frente do nome da variável que será passada por referência.

Lembre-se do exemplo da função **scanf**. A função **scanf** é um exemplo de função que altera o valor de uma variável e essa mudança se reflete fora da função. Quando chamamos a função **scanf**, é necessário colocar o operador "&" na frente do nome da variável que será lida do teclado. O mesmo vale para outras funções que usam passagem de parâmetro por referência.



Na passagem de uma variável por referência é necessário usar o operador "*" sempre que se desejar acessar o conteúdo da variável dentro da função.

```
1 include <stdio.h>
2 include <stdlib.h>
3
4 void soma_mais_um(int *n){
5     *n = *n + 1;
6     printf ("Antes da funcao: x = %d\n",*n);
7 }
8
9 int main (){
10     int x = 5;
11     printf ("Antes da funcao: x = %d\n",x);
12     soma_mais_um(&x);
13     printf ("Antes da funcao: x = %d\n",x);
14     system("pause");
15     return 0;
16 }
```

Saída Antes da funcao: x = 5
 Dentro da funcao: x = 6
 Depois da funcao: x = 6

No exemplo acima, no momento em que a função **soma_mais_um** é chamada, o endereço de x ($\&x$) é **copiado** para o parâmetro n da função. O parâmetro n é um ponteiro dentro da função que guarda o endereço de onde o valor de x está guardado fora da função. Sempre que alteramos o valor de $*n$ (conteúdo da posição de memória guardada, ou seja, x), o valor de x fora da função também é modificado.

Abaixo temos outro exemplo que mostra a mesma função declarada usando a passagem de parâmetro de valor e por referência:

Exemplo: passagem por valor e referência	
Por valor	Por referência
<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 4 void Troca (int a,int b) 5 { 6 int temp; 7 temp = a; 8 a = b; 9 b = temp; 10 printf("Dentro: %d e %d\n",a,b); 11 } 12 int main () { 13 int x = 2; 14 int y = 3; 15 printf("Antes: %d e %d\n",x,y); 16 Troca(x,y); 17 printf("Depois: %d e %d\n",x,y); 18 system("pause"); 19 return 0; 20 }</pre>	<pre> 1 #include <stdio.h> 2 #include <stdlib.h> 3 4 void Troca (int*a,int*b) 5 { 6 int temp; 7 temp = *a; 8 *a = *b; 9 *b = temp; 10 printf("Dentro: %d e %d\n",*a,*b); 11 } 12 int main () { 13 int x = 2; 14 int y = 3; 15 printf("Antes: %d e %d\n",x,y); 16 Troca(&x,&y); 17 printf("Depois: %d e %d\n",x,y); 18 system("pause"); 19 return 0; 20 }</pre>
<p>Saída</p> <p>Antes: 2 e 3 Dentro: 3 e 2 Depois: 2 e 3</p>	<p>Saída</p> <p>Antes: 2 e 3 Dentro: 3 e 2 Depois: 3 e 2</p>

6.2.3 PASSAGEM DE ARRAYS COMO PARÂMETROS

Para utilizar arrays como parâmetros de funções alguns cuidados simples são necessários. Além do parâmetro do array que será utilizado na função, é necessário declarar um segundo parâmetro (em geral uma variável inteira) para passar para a função o tamanho do array **separadamente**.



Arrays são sempre passados *por referência* para uma função.

Quando passamos um array por parâmetro, independente do seu tipo, o que é de fato passado para a função é o endereço do primeiro elemento

do array.



A passagem de arrays por referência evita a cópia desnecessária de grandes quantidades de dados para outras áreas de memória durante a chamada da função, o que afetaria o desempenho do programa.

Na passagem de um array como parâmetro de uma função podemos declarar a função de diferentes maneiras, todas equivalentes:

```
void imprime (int *m, int n);  
void imprime (int m[], int n);  
void imprime (int m[5], int n);
```



Mesmo especificando o tamanho de um array no parâmetro da função a semântica é a mesma das outras declarações, pois não existe checagem dos limites do array em tempo de compilação.

O exemplo abaixo mostra como um array de uma única dimensão pode ser passado como parâmetro para uma função:

Exemplo: passagem de array como parâmetro

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 void imprime (int *n  
    , int m){  
5     int i;  
6     for (i=0; i<m; i++)  
7         printf ("%d \n",  
            n[i]);  
8 }  
9  
10 int main () {  
11     int v[5] =  
        {1,2,3,4,5};  
12     imprime(v,5);  
13     system("pause");  
14     return 0;  
15 }
```

Memória		
.		
.		
.		
#	var	conteúdo
123	int *n	#125
124		
125		1
126		2
127		3
128		4
129		5
.		
.		
.		

Note, no exemplo acima, que apenas o nome do array é passado para a função, sem colchetes. Isso significa que estamos passando o array inteiro. Se usássemos o colchete, estaríamos passando o valor de uma posição do array e não o seu endereço, o que resultaria em um erro.



Na chamada da função, passamos para ela somente o nome do array, sem os colchetes: o programa "já sabe" que um array será enviado, pois isso já foi definido no protótipo da função.

Vimos que, para arrays, não é necessário especificar o número de elementos para a função no parâmetro do array:

```
void imprime (int *m, int n);
```

```
void imprime (int m[], int n);
```



Arrays com mais de uma dimensão (por exemplo, matrizes), precisam da informação do tamanho das dimensões extras.

Para arrays com mais de uma dimensão é necessário o tamanho de todas as dimensões, exceto a primeira. Sendo assim, uma declaração possível para uma matriz de 4 linhas e 5 colunas seria a apresentada abaixo:

```
void imprime (int m[][5], int n);
```

A declaração de arrays com uma dimensão e com mais de uma dimensão é diferente porque na passagem de um array para uma função o compilador precisar saber o tamanho de cada elemento, não o número de elementos.



Um array bidimensional poder ser entendido como um array de arrays.

Para a linguagem C, um array bidimensional poder ser entendido como um array de arrays. Sendo assim, o seguinte array

```
int m[4][5];
```

pode ser entendido como um array de 4 elementos, onde cada elemento é um array de 5 posições inteiras. Logo, o compilador precisa saber o tamanho de um dos elementos (por exemplo, o número de colunas da matriz) no momento da declaração da função:

```
void imprime (int m[][5], int n);
```

Na notação acima, informamos ao compilador que estamos passando um array, onde cada elemento dele é outro array de 5 posições inteiras. Nesse caso, o array terá sempre 5 colunas, mas poderá ter quantas linhas quiser (parâmetro **n**).

Isso é necessário para que o programa saiba que o array possui mais de uma dimensão e mantenha a notação de um conjunto de colchetes por dimensão.

O exemplo abaixo mostra como um array de duas dimensões pode ser passado como parâmetro para uma função:

Exemplo: passagem de matriz como parâmetro

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void imprime_matriz(int m[][2], int n){
5     int i,j;
6     for (i=0; i<n; i++)
7         for (j=0; j< 2; j++)
8             printf ("%d \n", m[i][j]);
9 }
10
11 int main (){
12     int mat[3][2] = {{1,2},{3,4},{5,6}};
13     imprime_matriz(mat,3);
14     system("pause");
15     return 0;
16 }
```

As notações abaixo funcionam para arrays com mais de uma dimensão. Mas o array é tratado como se tivesse apenas uma dimensão dentro da função

```
void imprime (int *m, int n);
```

```
void imprime (int m[], int n);
```

O exemplo abaixo mostra como um array de duas dimensões pode ser passado como um array de uma única dimensão para uma função:

Exemplo: matriz como array de uma dimensão

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void imprime_matriz(int *m, int n){
5     int i;
6     for (i=0; i<n; i++)
7         printf ("%d \n", m[i]);
8 }
9
10 int main (){
11     int mat[3][2] = {{1,2},{3,4},{5,6}};
12     imprime_matriz(&mat[0][0], 6);
13     system("pause");
14     return 0;
15 }
```

Note que, nesse exemplo, ao invés de passarmos o nome do array nós passamos o endereço do primeiro elemento (&mat[0][0]). Isso faz com que percamos a notação de dois colchetes para a matriz, e ela seja tratada como se tivesse apenas uma dimensão.

6.2.4 OPERADOR SETA

De modo geral, uma estrutura é sempre passada por valor para uma função. Mas ela também pode ser passada por referência sempre que desejarmos alterar algum dos valores de seus campos.

Durante o estudo dos tipos definidos pelo programador, vimos que o operador "."(ponto) era utilizado para acessar os campos de uma estrutura. Se essa estrutura for passada por referência para uma função, será necessário usar ambos os operadores "*"e "."para acessar os valores originais dos campos da estrutura.

- operador "*": acessa o conteúdo da posição de memória (valor da variável fora da função) dentro da função;
- operador ".": acessa os campos de uma estrutura.



O operador **seta** "->" substitui o uso conjunto dos operadores "*"e "."no acesso ao campo de uma estrutura passada por referência para uma função.

O operador **seta** "->" é utilizado quando uma referência para uma estrutura (struct) é passada para uma função. Ele permite acessar o valor do campo

da estrutura fora da função sem utilizar o operador "*". O exemplo abaixo mostra como os campos de uma estrutura passada por referência podem ser acessado com ou sem o uso do operador seta "->":

Exemplo: passagem por valor e referência	
Sem operador seta	Com operador seta
<pre> 1 struct ponto { 2 int x, y; 3 }; 4 5 void func(struct ponto * p){ 6 (*p).x = 10; 7 (*p).y = 20; 8 } </pre>	<pre> 1 struct ponto { 2 int x, y; 3 }; 4 5 void func(struct ponto * p){ 6 p->x = 10; 7 p->y = 20; 8 } </pre>

6.3 RECURSÃO

Na linguagem C, uma função pode chamar outra função. Um exemplo disso é quando chamamos qualquer uma das nossas funções implementadas na função **main**. Uma função pode, inclusive, chamar a si própria. Uma função assim é chamada de *função recursiva*.



A recursão também é chamada de definição circular. Ela ocorre quando algo é definido em termos de si mesmo.

Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número. A função fatorial é definida como:

$$0! = 1$$

$$N! = N * (N - 1)!$$

A idéia básica da recursão é dividir um problema maior em um conjunto de problemas menores, que são então resolvidos de forma independente e depois combinados para gerar a solução final: dividir e conquistar.

Isso fica evidente no cálculo do fatorial. O fatorial de um número N é o produto de todos os números inteiros entre 1 e N . Por exemplo, o fatorial de 3 é igual a $1 * 2 * 3$, ou seja, **6**. No entanto, o fatorial desse mesmo

número 3 pode ser definido em termos do fatorial de 2, ou seja, $3! = 3 * 2!$. O exemplo abaixo apresenta as funções com e sem recursão para o cálculo do fatorial:

Exemplo: fatorial	
Com Recursão	Sem Recursão
<pre> 1 int fatorial (int n){ 2 if (n == 0) 3 return 1; 4 else 5 return n*fatorial(n 6 -1); </pre>	<pre> 1 int fatorial (int n){ 2 if (n == 0) 3 return 1; 4 else { 5 int i, f = 1; 6 for (i=2; i <= n; i 7 ++) 8 f = f * i; 9 return f; 10 } </pre>

Em geral, as formas recursivas dos algoritmos são consideradas "mais enxutas" e "mais elegantes" do que suas formas iterativas. Isso facilita a interpretação do código. Porém, esses algoritmos apresentam maior dificuldade na detecção de erros e podem ser ineficientes.



Todo cuidado é pouco ao se fazer funções recursivas, pois duas coisas devem ficar bem estabelecidas: o *critério de parada* e o *parâmetro da chamada recursiva*.

Durante a implementação de uma função recursiva temos que ter em mente duas coisas: o **critério de parada** e o **parâmetro da chamada recursiva**:

- **Critério de parada:** determina quando a função deverá parar de chamar a si mesma. Se ele não existir, a função irá executar infinitamente. No cálculo de fatorial, o critério de parada ocorre quando tentamos calcular o fatorial de zero: $0! = 1$.
- **Parâmetro da chamada recursiva:** quando chamamos a função dentro dela mesmo, devemos sempre mudar o valor do parâmetro passado, de forma que a recursão chegue a um término. Se o valor do parâmetro for sempre o mesmo a função irá executar infinitamente. No cálculo de fatorial, a mudança no parâmetro da chamada recursiva ocorre quando definimos o fatorial de N em termos no fatorial de $(N-1)$: $N! = N * (N - 1)!$.

O exemplo abaixo deixa bem claro o **critério de parada** e o **parâmetro da chamada recursiva** na função recursiva implementada em linguagem C:

Exemplo: fatorial	
1	<code>int fatorial (int n){</code>
2	<code> if (n == 0) //critério de parada</code>
3	<code> return 1;</code>
4	<code> else //parâmetro do fatorial sempre muda</code>
5	<code> return n*fatorial(n-1);</code>
6	<code>}</code>

Note que a implementação da função recursiva do fatorial em C segue exatamente o que foi definido matematicamente.



Algoritmos recursivos tendem a necessitar de mais tempo e/ou espaço do que algoritmos iterativos.

Sempre que chamamos uma função, é necessário um espaço de memória para armazenar os parâmetros, variáveis locais e endereço de retorno da função. Numa função recursiva, essas informações são armazenadas para cada chamada da recursão, sendo, portanto a memória necessária para armazená-las proporcional ao número de chamadas da recursão.

Além disso, todas essas tarefas de alocar e liberar memória, copiar informações, etc. envolvem tempo computacional, de modo que uma função recursiva gasta mais tempo que sua versão iterativa (sem recursão).

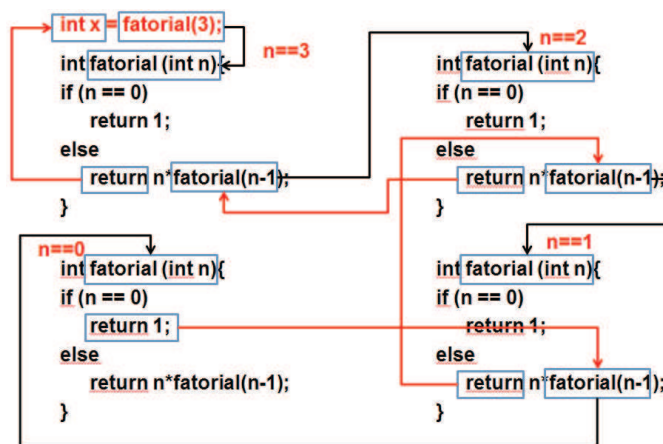


O que acontece quando chamamos a função fatorial com um valor como $N = 3$?

Nesse caso, a função será chamada tantas vezes quantas forem necessárias. A cada chamada, a função irá verificar se o valor de N é igual a zero. Se não for, uma nova chamada da função será realizada. Esse processo, identificado pelas setas pretas, continua até que o valor de N seja decrementado para ZERO. Ao chegar nesse ponto, a função começa o processo inverso (identificado pelas setas vermelhas): ela passa a devolver para quem a chamou o valor do comando **return**. A figura abaixo mostra esse processo para $N = 3$:

Outro exemplo clássico de recursão é a sequência de Fibonacci:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...



A sequência de Fibonacci é definida como uma função recursiva utilizando a fórmula abaixo:

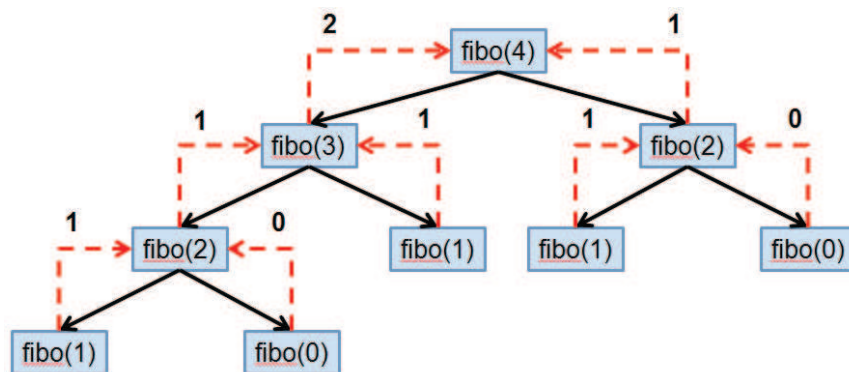
$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

O exemplo abaixo apresenta as funções com e sem recursão para o cálculo da sequência de Fibonacci:

Exemplo: sequência de Fibonacci	
Com Recursão	Sem Recursão
<pre> 1 int fibo(int n){ 2 if (n == 0 n == 1) 3 return n; 4 else 5 return fibo(n-1) + 6 fibo(n-2); 6 }</pre>	<pre> 1 int fibo(int n){ 2 int i,t,c,a=0, b=1; 3 for(i=0;i<n;i++){ 4 c = a + b; 5 a = b; 6 b = c; 7 } 8 return a; 9 }</pre>

Como se nota, a solução recursiva para a sequência de Fibonacci é muito elegante. Infelizmente, como se verifica na imagem abaixo, elegância não significa eficiência.

Na figura acima, as setas pretas indicam quando uma nova chamada da função é realizada, enquanto as setas vermelhas indicam o processo inverso, ou seja, quando a função passa a devolver para quem a chamou



o valor do comando **return**. O maior problema da solução recursiva está nos quadrados marcados com pontilhados verde. Neles, fica claro que o mesmo cálculo é realizado duas vezes, *um desperdício de tempo e espaço*!

Se, ao invés de calcularmos **fibo(4)** quisermos calcular **fibo(5)**, teremos um desperdício ainda maior de tempo e espaço, como mostra a figura abaixo:

