



UFRRJ

UNIVERSIDADE FEDERAL RURAL DO RIO DE JANEIRO
DEPARTAMENTO DE COMPUTAÇÃO
IC596 – Linguagem de Programação II

Seropédica, 2021

Prof. Claver Pari Soto

1. IC596 – LP II

IC596 é o código da disciplina Linguagem de Programação II que o Departamento da Computação da UFRRJ oferece aos diferentes cursos de ciências exatas. O objetivo geral da disciplina é que o aluno seja capaz de elaborar programas computacionais complexos, utilizando algoritmos avançados de computação.

2. Linguagens de Programação

Para escrever programas independentes do processador no qual serão executados, podemos usar linguagens de alto nível que combinam expressões algébricas e símbolos tirados do inglês. Por exemplo, $a = a + b$;

Esta declaração significa "adicionar os valores das variáveis a e b , e armazenar o resultado na variável a (substituindo o valor anterior de a)."

Antes que um programa de linguagem de alto nível possa ser executado, ele deve primeiro ser traduzido para a linguagem de máquina do processador de destino. O programa que faz essa tradução é chamado de compilador. A figura abaixo ilustra a função do compilador no processo de desenvolvimento e teste de um programa de linguagem de alto nível. Tanto a entrada (Código Fonte) quanto a saída (Código Objeto) do compilador (quando é bem-sucedido) são programas. A entrada para o compilador é um arquivo fonte (*source file*) que contém o texto de um programa de linguagem de alto nível. O desenvolvedor do software cria esse arquivo usando um **processador de texto** ou editor. O formato do arquivo de origem é texto, o que significa que é uma coleção de códigos de caracteres. Por exemplo, pode-se digitar um programa em um arquivo chamado **meuProg.c**.

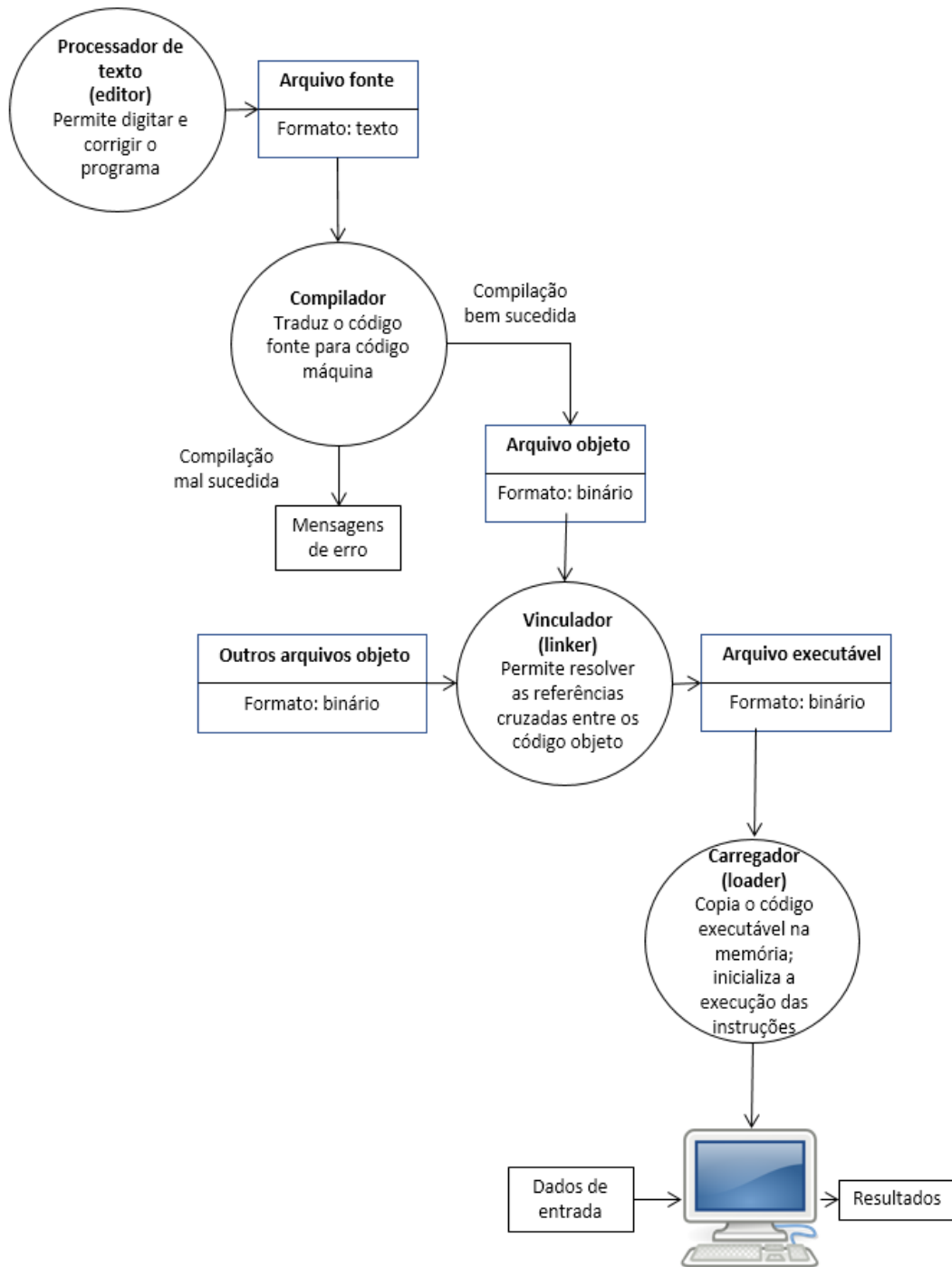
O **compilador** irá verificar este arquivo de origem, para ver se ele segue as regras de sintaxe (gramática) da linguagem de alto nível. Se o programa estiver sintaticamente correto, o compilador salva em um arquivo de objeto (*object file*) as instruções em linguagem de máquina que realizam o propósito do programa. Para o programa **meuprog.c**, o arquivo objeto criado pode ser denominado **meuProg.o**. Observe que o formato deste arquivo é binário. Isso significa que você não deve enviá-lo para uma impressora, exibi-lo em sua tela ou tentar trabalhar com ele em um processador de texto, pois aparecerão símbolos sem sentido para um processador de texto, impressora ou tela. Se o programa de origem contiver erros de sintaxe, o compilador listará esses erros, mas não criará um arquivo-objeto. O desenvolvedor deve retornar ao processador de texto, corrigir os erros e recompilar o programa.

Embora um arquivo de objeto contenha instruções de máquina, nem todas as instruções estão completas. Linguagens de alto nível fornecem ao desenvolvedor de software muitos blocos de código em forma de funções formando "bibliotecas" com nomes específicos para operações de que o desenvolvedor provavelmente precisará. Quase todos os programas de linguagem de alto nível usam pelo menos uma dessas bibliotecas de código que residem em outros arquivos de objeto disponíveis para o sistema. O programa **vinculador** (*linker*) combina essas funções pré-fabricadas com o arquivo-objeto, criando um programa de linguagem de máquina completo que está pronto para ser executado. Para nosso exemplo, o vinculador pode nomear o arquivo executável ("aplicativo") como **meuProg.exe**.

Enquanto o arquivo executável estiver apenas armazenado no HD, ele não fará nada. Para executá-lo, o **carregador** (*loader*) deve copiar todas as instruções do arquivo executável na memória e direcionar a CPU para iniciar a execução com a primeira

instrução. Conforme o programa é executado, ele obtém dados de entrada de uma ou mais fontes e envia os resultados para dispositivos de armazenamento de saída.

Alguns sistemas de computador exigem que o usuário peça ao sistemas operacional para executar separadamente cada etapa ilustrada na Figura. No entanto, a maioria dos compiladores de linguagem de alto nível é vendida como parte de um ambiente de desenvolvimento integrado (*IDE- integrated development environment*), um pacote que combina um processador de texto simples com um compilador, vinculador e carregador. Esses ambientes fornecem ao desenvolvedor menus para selecionar a próxima etapa e, se o desenvolvedor tentar uma etapa fora da sequência, o ambiente simplesmente preencherá as etapas ausentes automaticamente. Um IDE frequentemente usará o termo *build* para significar "compilar e vincular".



3. História do C

Em 1970, um programador, Dennis Ritchie, criou uma nova linguagem chamada C. (o nome surgiu porque substituiu a antiga linguagem de programação que ele estava usando: B.) C foi projetado com um objetivo em mente: escrever sistemas operacionais. A linguagem era extremamente simples e flexível, e logo foi usada para muitos tipos diferentes de programas. Rapidamente se tornou uma das linguagens de programação mais populares do mundo. A popularidade de C foi devido a dois fatores principais:

- A primeira era que a linguagem não atrapalhava o programador. Ele poderia fazer quase qualquer coisa usando a construção C adequada (essa flexibilidade também é uma desvantagem, pois permite que o programa faça coisas que o programador nunca pretendeu)
- A segunda razão pela qual C é popular é que um compilador C portátil foi amplamente disponibilizado. Consequentemente, as pessoas poderiam anexar um compilador C para sua máquina facilmente e com poucas despesas.

Em 1980, Bjarne Stroustrup começou a trabalhar em uma nova linguagem, chamada “C com classes”. Esta linguagem melhorou C adicionando uma série de novos recursos. Essa nova linguagem foi aprimorada e aumentada e, finalmente, tornou-se C ++.

4. Primeiro programa em C

meuProg.c

```
1  #include <stdio.h>
2  #define LITROS_POR_GALAO 3.7854
3
4  int main(void){
5
6      double galoos;
7      double litros;
8
9      printf("Ingresse o numero de galoos: ");
10     scanf("%lf", &galoos);
11
12     litros = LITROS_POR_GALAO * galoos;
13     printf("%lf galoos equivalem a %f litros.\n", galoos, litros);
14     return 0;
15 }
```

```
Ingresse o numero de galoos: 23
23.000000 galoos equivalem a 87.064200 litros.
```

Linha 1: a diretiva de pré-processador **#include** fornece ao programa acesso a uma biblioteca. Uma biblioteca é um conjunto de código pré-programado feito para realizar tarefas relacionadas a um determinado assunto. O arquivo **stdio.h** é o denominado *arquivo de cabeçalho padrão* (*standard header file*) da biblioteca que contém por exemplo as funções **scanf** e **printf**.

Linha 2: a diretiva de pré-processador **#define** associa a macro constante **LITROS_POR_GALAO** com o valor **3.7854**

Linha 4: marca o início da função **main**, onde o programa inicia sua execução. Todo programa em C tem uma função **main**. Como toda função, ela pode ter argumentos entre parênteses (neste caso, **void**, ou seja sem argumentos) e algum valor de retorno (neste caso, ao terminar a função **main**, espera-se um valor tipo **int** como saída da função). O corpo da função **main** inicia-se na chave de abertura {

Linha 6: declaração da variável **galoos**, como de tipo **double**.

Linha 7: declaração da variável **litros**, como de tipo **double**.

Linha 9 e 13: a chamada à função **printf** (pronuncia-se “print-efe”) permite ver na tela do computador os resultados da execução de um programa.

Linha 10: a chamada à função **scanf** (pronuncia-se “escan-efe”) permite copiar um valor (fornecido pelo usuário, desde o teclado) na célula de memória definida pelo endereço de uma variável, neste caso da variável **galoos**.

Linha 12: Declaração de atribuição. Nesta linha acontece a atribuição do resultado computacional da operação aritmética (**3.78540 * <valor atual da variável galoos>**) para a variável **litros**.

Linha 14: Declaração de controle. Linha que indica ao sistema operacional que o programa finalizou normalmente (*Status* = 0). Um status diferente de zero indica que aconteceu algum erro na execução do programa. Quanto mais severo o erro acontecido, maior o valor do **return**.

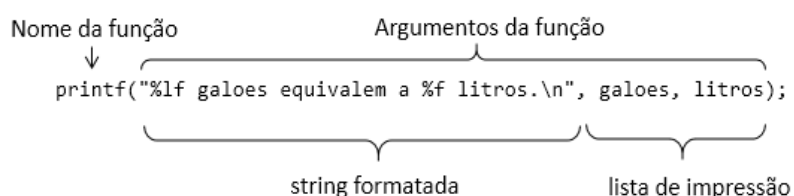
Linha 15: Chave de fechadura correspondente à chave de abertura do corpo da função **main** da linha 4.

Diretiva de preprocessador, #include

Sintaxe:	#include <arquivo de cabeçalho padrão>
Exemplos:	<pre>#include <stdio.h> #include <stdio.h> #include <math.h></pre>

Função main

Sintaxe:	<pre>int main(void){ <i>corpo da função</i> }</pre>
Exemplo:	<pre>int main(void){ printf("Oi turma!\n"); return 0; }</pre>

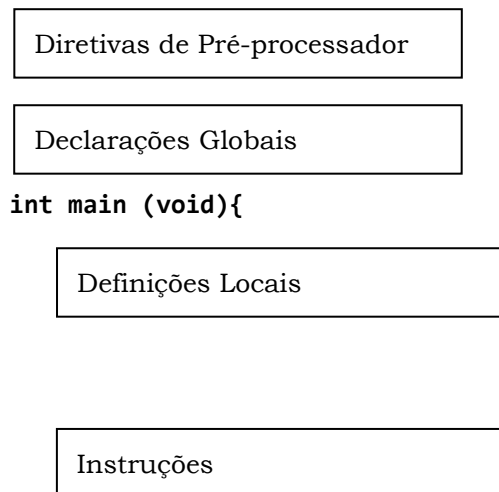


A função **printf** permite exibir uma mensagem na saída da tela. Uma chamada da função **printf** consiste de duas partes: o nome da função e os argumentos da função, entre parênteses. Os argumentos para **printf** consistem em uma string de formato (entre aspas) e uma lista de impressão (as variáveis **galoes** e **litros**, separadas por vírgulas). A chamada de função acima exibe a linha **“23.000000 galoes equivalem a 87.064200 litros”** que é o resultado da exibição da string de formato **“%lf galoes equivalem a %f litros.\n”**, após substituir o valor de **galoes** por seu marcador (**%lf**), e **litros** por seu marcador (**%f**) na string de formato. Um marcador de posição sempre começa com o símbolo **%**. Aqui, o espaço reservado **%f** marca a posição de exibição para um tipo de variável tipo **double**.

5. Estrutura de um programa na linguagem de programação C

A estrutura geral de um programa na linguagem C está representada no seguinte diagrama.

As diretivas do pré-processador dizem ao pré-processador olhar por bibliotecas de códigos de uso especial. As declarações globais estarão visíveis em todas as partes do programa.



Todas as funções, incluindo `main()`, podem ser divididas em duas seções: Definições Locais, e Instruções.

As definições locais devem ficar no início da codificação das funções, e descrevem os dados que serão usados na função. Os dados instanciados em definições locais, em oposição às declarações globais, são visíveis apenas para a função que os contém.

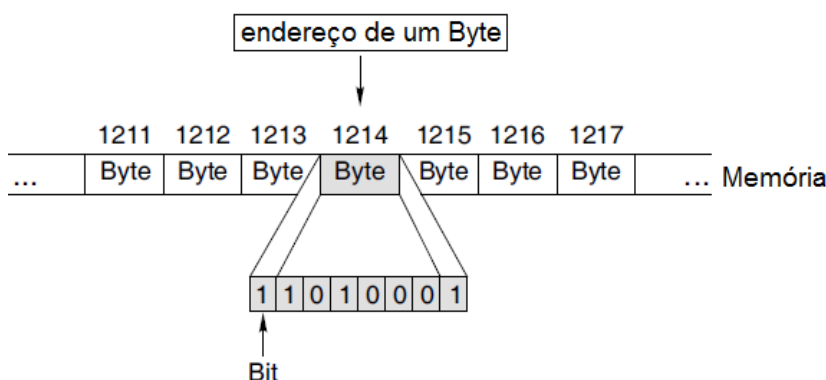
A seção de Instruções consiste nas instruções que fazem com que o computador faça algo.

```
    return 0;  
}
```

6. Variáveis e memória

Os programas operam com base em dados. As instruções que compõem o programa e os dados sobre os quais ele atua devem ser armazenados em algum lugar enquanto o computador está executando o programa. Uma linguagem de programação deve fornecer uma maneira de armazenar os dados a serem processados, caso contrário, ela se torna inútil. Neste contexto, pode ser mencionado que um computador fornece uma memória de acesso aleatório (RAM) para armazenar o código do programa executável e os dados que o programa manipula.

Uma memória de computador é composta por registros e células capazes de armazenar informações na forma de dígitos binários 0 e 1 (bits). Ele acessa os dados como uma coleção de bits, normalmente 8 bits, 16 bits, 32 bits ou 64 bits. Os dados são armazenados na memória em locais físicos da memória. Esses locais são conhecidos como endereço de memória. Portanto, cada byte pode ser identificado exclusivamente por seu endereço.

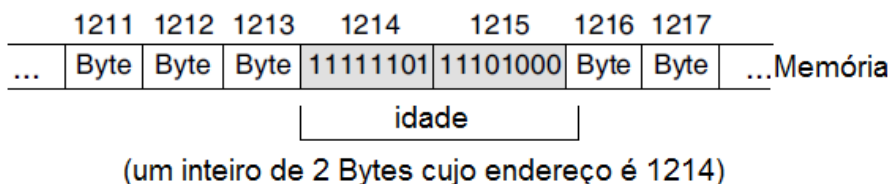


Uma variável é um identificador para um local de memória no qual os dados podem ser armazenados e subsequentemente recuperados.

Variáveis são usadas para armazenar valores de dados para que possam ser usados em vários cálculos em um programa. Cada variável é mapeada para um endereço de memória exclusivo. O compilador C gera um código executável que mapeia entidades de dados para locais de memória. Por exemplo, a definição de variável

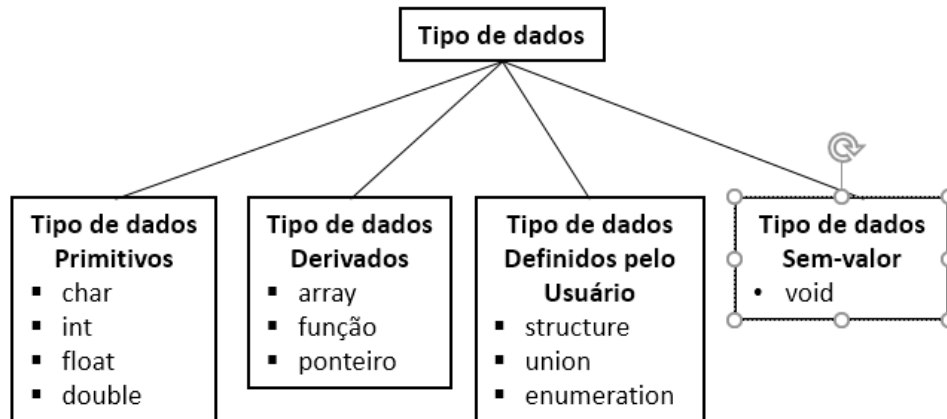
```
int idade = 20;
```

faz com que o compilador aloque alguns bytes para representar o valor da variável idade. O número exato de bytes alocados e o método usado para a representação binária do inteiro depende da implementação específica de C e do sistema operacional, mas assumindo que dois bytes contêm os dados codificados como um número binário 1111110111101000, o compilador usa o endereço do primeiro byte no qual a variável **idade** é alocada para se referir a ele. A atribuição acima faz com que o valor 20 seja armazenado como um número binário nos dois bytes alocados.



7. Tipos de dados

O tipo de dados de uma variável determina o conjunto de valores que uma variável pode assumir e o conjunto de operações que podem ser aplicadas a esses valores. Os tipos de dados podem ser amplamente classificados como mostrado na figura seguinte.



tiposDeDados.c

```

#include <stdio.h>
int main()
{
    int a;
    char b;
    float c;
    double d;
    printf("Tamanho no armazenamento de um dado tipo int: %d Bytes\n", sizeof(a));
    printf("Tamanho no armazenamento de um dado tipo char: %d Bytes\n", sizeof(b));
    printf("Tamanho no armazenamento de um dado tipo float: %d Bytes\n", sizeof(c));
    printf("Tamanho no armazenamento de um dado tipo double: %d Bytes\n", sizeof(d));
    return 0;
}
    
```

```

Tamanho no armazenamento de um dado tipo int: 4 Bytes
Tamanho no armazenamento de um dado tipo char: 1 Bytes
Tamanho no armazenamento de um dado tipo float: 4 Bytes
Tamanho no armazenamento de um dado tipo double: 8 Bytes
    
```

8. Instruções de Controle

Cada função incluindo `main()`, tem um corpo de função que consiste em um conjunto de uma ou mais instruções, ou seja, um bloco de instruções. Dentro de uma função, a execução prossegue de uma instrução para a próxima, de cima para baixo. No entanto, dependendo dos requisitos de um problema, pode ser necessário alterar a sequência normal de execução em um programa.

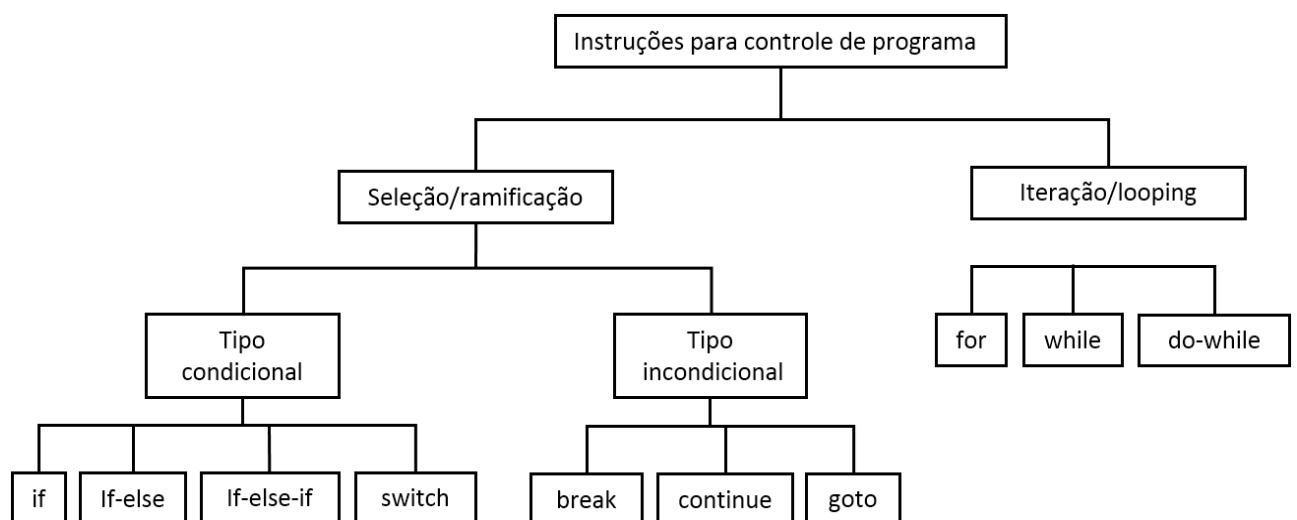
A ordem em que as instruções são executadas em um programa em execução é chamada de **fluxo de controle**. Controlar o fluxo de um programa é um aspecto muito importante da programação de computadores. O fluxo de controle está relacionado à ordem em que as operações de um programa são executadas.

As instruções de controle incorporam a lógica de decisão que informa ao programa em execução qual ação realizar a seguir, dependendo dos valores de certas variáveis ou instruções de expressão. As instruções de controle incluem instruções de **seleção**, **iteração** e **salto** que funcionam juntas para direcionar o fluxo do programa.

Uma instrução de seleção é uma instrução de controle que permite escolher entre dois ou mais caminhos de execução em um programa. As instruções de seleção em C são a instrução `if`, a instrução `if-else` e a instrução `switch`. Essas instruções nos permitem decidir qual instrução executar a seguir. Cada decisão é baseada em uma expressão booleana (também chamada de condição ou expressão de teste), que é uma expressão avaliada como verdadeira ou falsa. O resultado da expressão determina qual instrução é executada a seguir.

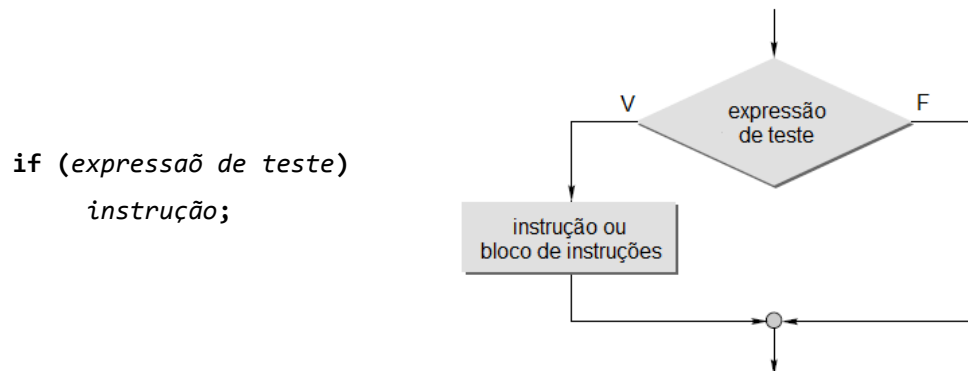
O mecanismo de programação que executa uma série de instruções repetidamente um determinado número de vezes, ou até que uma condição específica seja satisfeita, é chamado de **loop**. A construção usada para loop é conhecida como instrução de iteração. A linguagem C oferece três elementos de linguagem para formular declarações de iteração: `while`, `do-while` e `for`.

As instruções de salto transferem o controle para outro ponto do programa. As instruções de salto incluem `goto`, `break`, `continue` e `return`.

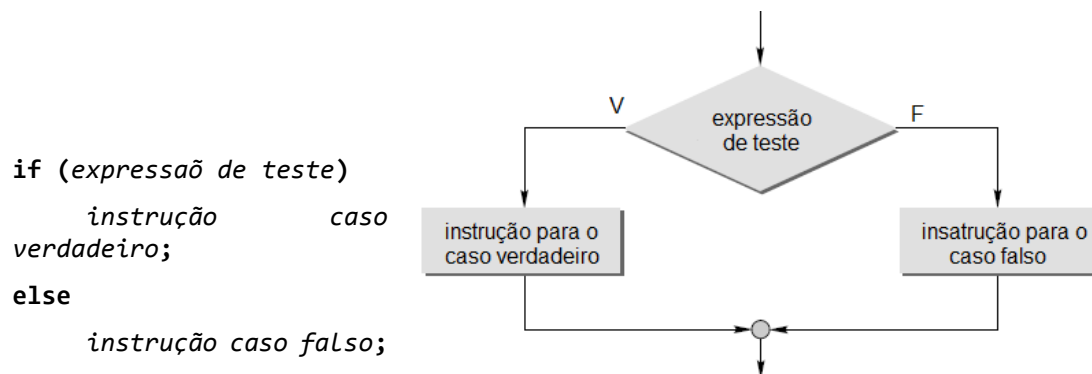


8.1. Estruturas de Seleção (ramificação)

▪ Decisão em uma-via usando a instrução `if`



▪ Decisão em duas-vias usando a instrução `if-else`



Exemplo 8.1_a: O programa obtém do usuário a idade. Logo o programa dá um “oi” e afirma se o usuário é menor, ou maior de idade.

```
Qual a sua idade?: 20
Oi, voce eh maior de idade
```

```

1  #include <stdio.h>
2  int main(){
3      int idade;
4
5      printf("Qual a sua idade?: ");
6      scanf("%d",&idade);
7
8      if(idade > 17)
9          printf("Oi, voce eh maior de idade\n");
10     else
    
```

```

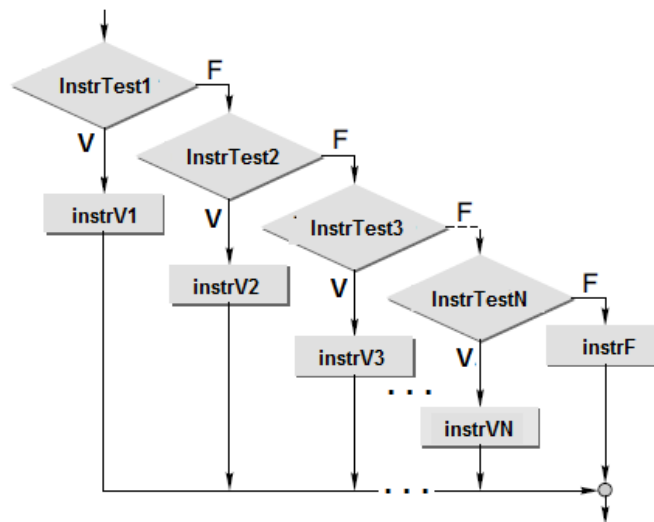
11         printf("Oi, voce eh menor de idade\n");
12
13     return 0;
14 }
    
```

Decisões multi-via

▪ Escada if-else-if

```

if(InstrTest1)
    instrV1;
else if(InstrTest2)
    instrV2;
else if(InstrTest3)
    instrV3;
.
.
.
else if(InstrTestN)
    instrVN;
else
    instrF;
    
```



Exemplo 8.1_b: O programa obtém do usuário a temperatura ambiente nesse momento na sua cidade. Logo o programa devolve uma mensagem dependendo se a temperatura é menor que 25°C, maior ou igual que 30°C, ou intermediária.

```

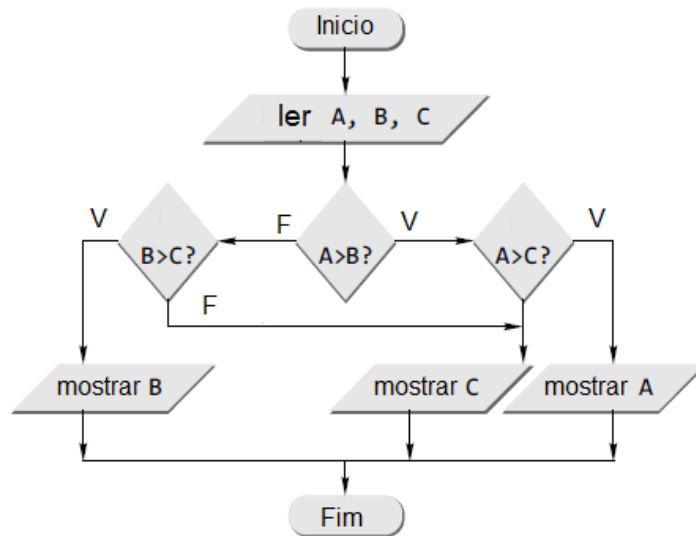
Qual a temperatura na sua cidade hoje?: 35
Nao esqueca de se hidratar!
    
```

```

1  #include <stdio.h>
2  int main(){
3      float temperatura;
4
5      printf("Qual a temperatura na sua cidade hoje?: ");
6      scanf("%f",&temperatura);
7
8      if(temperatura < 22)
9          printf("Eh melhor você se agasalhar!\n");
10     else if(temperatura >= 30)
11         printf("Nao esqueca de se hidratar!");
12     else
13         printf("Dia agradável! :)");
14     printf("\n\n");
15
16     return 0;
17 }
    
```

▪ if aninhado

Quando alguma instrução if é escrita sob outra instrução if, este if interno é chamado de if aninhado.



Exemplo 8.1_c: O programa obtém do usuário 3 números inteiros. Logo o programa informa qual desses números tem valor maior.

```

Ingresse 3 nums inteiros: 23 67 1
O numero maior eh: 67
    
```

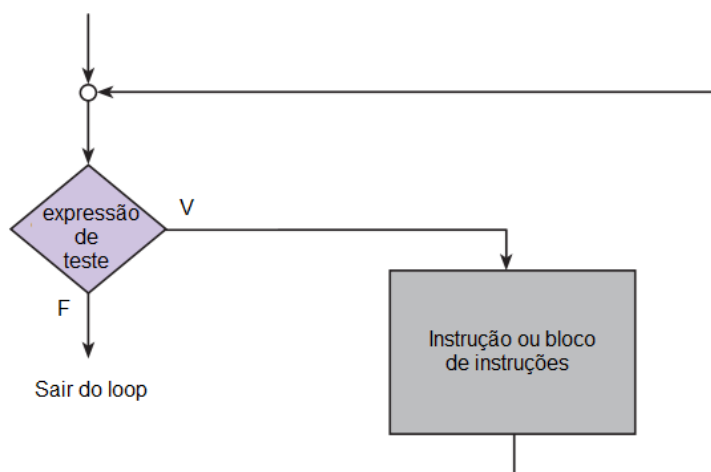
```

1  #include <stdio.h>
2  int main(){
3      int num1;
4      int num2;
5      int num3;
6      printf("Ingresse 3 nums inteiros: ");
7      scanf("%d %d %d", &num1, &num2, &num3);
8
9      if(num1 > num2)
10         if(num1 > num3)
11             printf("O numero maior eh: %d", num1);
12         else
13             printf("O numero maior eh: %d", num3);
14     else
15         if(num2 > num3)
16             printf("O numero maior eh: %d", num2);
17         else
18             printf("O numero maior eh: %d", num3);
19     printf("\n\n");
20     return 0;
21 }
    
```

Exercício 8.1_1 Adicione uma nova decisão no código do **Exemplo 8.1_b** de tal forma que o programa apresente a mensagem “Eh a temperatura ambiente ideal!” quando o usuário informe que sua cidade está em exatamente 24° C.

8.2. Estruturas de Iteração (loops)

Loop while (loop pré-teste)	
Sintaxe	while (condição para repetição do loop) instrução, ou grupo de instruções;
Exemplo	<pre> /* imprime N asteriscos */ contador = 0; while(contador < N) { printf("*"); contador = contador + 1; } </pre>
Interpretação	<p>A condição de repetição (condição de teste) do loop é testada; se for verdadeiro, a instrução (corpo do loop) é executada e a condição de repetição do loop é testada novamente. A instrução é repetida enquanto (while) a condição de repetição do loop for verdadeira. Quando esta condição é testada e resulte ser falsa, o loop while é encerrado e a próxima instrução do programa após a instrução while é executada.</p> <p>Nota: Se a condição de repetição do loop for avaliada como falsa na primeira vez que for testada, a instrução não será executada.</p>



Exercício 8.2_1 Escrever um programa que receba do usuário um número inteiro positivo, e que apresente a soma de todos os inteiros positivos pares menores ou iguais que o número fornecido pelo usuário.

Loop for

```

1  #include <stdio.h>
2  int main(){
3      int contador;
4      for(contador=0;contador<100;contador=contador+1)
5          printf("Nunca mais vou falar palavrão dentro da sala de aula\n");
6
7      return 0;
8  }
    
```

9. Ponteiros

Os ponteiros são um tipo especial de variável. Nas variáveis convencionais armazenamos algum valor de tipo **char**, de tipo **int**, de tipo **double**, etc. Essa variável convencional estará alocada automaticamente em algum endereço de memória, mas o que importa é o nome da variável e o valor armazenado nela. Nas variáveis ponteiro armazenamos algum endereço de memória. A própria variável ponteiro está alocada num endereço de memória, mas seu conteúdo é outro endereço de memória.

Então, os ponteiros podem ser usados para armazenar o endereço de uma célula de memória e acessar ou modificar o conteúdo dessa célula por meio de referência indireta.

9.1. Declaração de ponteiros e o operador de endereçamento

Na declaração

```
float *var_p;
```

estamos criando a variável com identificador **var_p** e ela está sendo criada como uma variável ponteiro de tipo “ponteiro para float”. Isto significa que podemos armazenar o endereço de memória de uma variável tipo float na variável ponteiro **var_p**.

É uma convenção entre os programadores de computadores nomear as variáveis ponteiro adicionando algum sufixo no identificador escolhido para a variável, para de tal forma enfatizar que essa variável é de tipo ponteiro. No nosso curso usaremos o sufixo **_p** como parte do nome das variáveis ponteiro. Também é costume usar o sufixo **Ptr** (do inglês, **Pointer**).

Declaração de variável tipo ponteiro

Sintaxe:	<i>tipo *variável;</i>
Exemplo:	float *var_p;
Interpretação:	<p>Neste exemplo está sendo declarada a variável var_p que pelo asterisco (*) ao seu lado, se declara como uma variável ponteiro, que aponta ao um endereço onde poderá ser armazenado um float.</p> <p>O * nesta declaração não é um operador. Só faz parte da sintaxe para declarar uma variável ponteiro.</p> <p>O valor da variável ponteiro var_p é um endereço de memória. Um item de dado cujo endereço é armazenado nesta variável deve ser do tipo especificado. No exemplo, o endereço armazenado na variável ponteiro var_p pode-se armazenar um dado de tipo float; Pode se falar “o ponteiro var_p aponta para um float”.</p>

Exemplo:

Consideremos as sentenças declarativas

```
int idade = 20;  
int *item_p;  
item_p = &idade;
```

A primeira declaração aloca memória para a variável **idade**, de tipo **int**.

A segunda linha aloca memória para o ponteiro **item_p** que aponta para um **int**.

A terceira linha atribui o endereço de memória da variável **idade** ao ponteiro **item_p**. Aqui é aplicado o **operador unário de endereçamento**, **&**, à variável **idade** para obter o endereço dela, que logo é armazenada em **item_p**.

Depois dessas três declarações podemos dizer que

“o ponteiro **item_p** aponta à variável **idade**”, ou

“**item_p** aponta a **idade**”

Operador de endereçamento & (“endereço-de”)	
Sintaxe:	<code>&<nome_da_variável></code>
Exemplo:	<code>&idade</code>
Interpretação:	O operador & fornece o endereço da variável idade . Este é o mesmo operador de endereçamento & aplicado nas variáveis na lista de entrada de instruções scanf .

9.2. Operador de des-endereçamento ou “indireção”

O operador de des-referenciamento ou indireção, ou operador asterisco, é outro operador unário aplicado a um ponteiro, que dá acesso ao valor armazenado em um endereço. Por exemplo considere as 4 instruções seguintes:

```
1  int num = 5;
2  int *inteiro_p = &num;
3  num = 10;
4  *inteiro_p = 15;
```

Linha 1: declara e inicializa a variável **num**, de tipo **int** e valor **5**.

Linha 2: declara e inicializa o ponteiro **inteiro_p**, que aponta a **int**. Esse ponteiro está inicializado com o endereço de **num**. O ***** usado nessa linha não é o operador des-endereçamento. Depois desta linha podemos dizer “**inteiro_p** aponta a **num**”

Linha 3: atualiza o valor de **num** para **10**

Linha 4: Aqui está se usando sim, o operador unário de des-endereçamento (*****). Este operador ***** dá acesso ao valor armazenado no endereço indicado pelo ponteiro **inteiro_p**. Já que **ponteiro_p** tem armazenado o endereço de **num**, ou seja, **ponteiro_p** aponta a **num**, logo, ***inteiro_p** dá acesso ao valor de **num**, que será atualizado para **15**.

Observação: as Linhas 3 e 4 fornecem duas maneiras de atualizar o valor de uma variável. A Linha 3 utiliza uma referência direta ao nome da variável, e a Linha 4 utiliza um ponteiro que aponta à variável, que pode ser considerada uma **referência indireta**.

Operador de des-endereçamento * (“valor-da-variável-apontada-por”)	
Sintaxe:	<code>*<nome_do_ponteiro></code>
Exemplo:	<code>*inteiro_p</code>
Interpretação:	O operador * dá acesso ao valor da variável apontada pelo ponteiro inteiro_p .

9.3. Aplicações de Ponteiros

A seguir, são apresentadas 3 aplicações do uso de ponteiros.

9.3.1. Ponteiros para arquivos

A linguagem C permite redirecionar os dispositivos padrão de entrada e saída (padrão de entrada: teclado; padrão de saída: tela/monitor) para arquivos.

Ponteiros de arquivo permitem que um programa acesse arquivos de entrada e saída.






Exemplo 9.3.1_a: O programa pega o arquivo de texto **entrada.txt** previamente salvo, que contém uma lista de números separados por espaço, e devolve um novo arquivo (se ele não existir) **saida.txt** com a lista de números arredondados para duas casas decimais. O resultado da execução do programa será visualizado somente ao abrir o arquivo de saída.

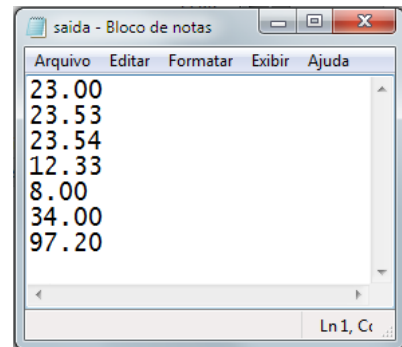
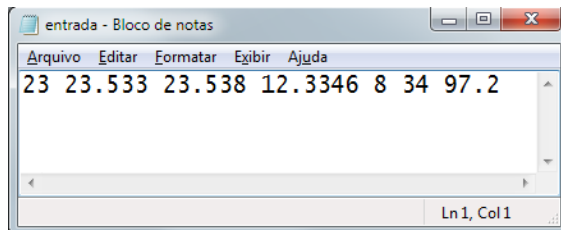
```
Salvando no arquivo de saída o item arredondado...
Salvando no arquivo de saída o item arredondado...
Salvando no arquivo de saída o item arredondado...
Salvando no arquivo de saída o item arredondado...
Salvando no arquivo de saída o item arredondado...
Salvando no arquivo de saída o item arredondado...
Salvando no arquivo de saída o item arredondado...
Fechando os arquivos...
```

ponteirosParaFile1.c

```
1  #include <stdio.h>
2
3  int main(void){
4      FILE *entrada_p;
5      FILE *saida_p;
6
7      double item;
8      int estado_da_entrada;
9
10     entrada_p = fopen("entrada.txt", "r");
11     saida_p = fopen("saida.txt", "w");
12
13     estado_da_entrada = fscanf(entrada_p, "%lf", &item);
14     while (estado_da_entrada == 1) {
15         printf("Salvando no arquivo de saída o item arredondado... \n");
16         fprintf(saida_p, "%.2f\n", item);
17         estado_da_entrada = fscanf(entrada_p, "%lf", &item);
18     }
19
20     printf("Fechando os arquivos...\n\n");
21     fclose(entrada_p);
22     fclose(saida_p);
23
24     return 0;
25 }
```

Antes de executar este exemplo, garanta que exista na mesma pasta onde está o programa compilado, o arquivo **entrada.txt**. O arquivo **saida.txt** será criado automaticamente pelo programa.

Nome	Tipo	Tamanho
 entrada	Documento de Texto	1 KB
 ponteirosParaFile	C source file	2 KB
 ponteirosParaFile	Aplicativo	44 KB
 ponteirosParaFile.o	Arquivo O	2 KB
 saida	Documento de Texto	1 KB



Neste exemplo, pode se observar no Bloco de Notas, o arquivo **entrada.txt** tem 7 números, 3 de tipo inteiro (**23**, **8** e **34**) e 4 com ponto decimal (**23.533**, **23.538**, **12.3346** e **97.2**). O programa salvará esses mesmos números arredondados para duas casas decimais, no arquivo **saida.txt**. Você pode mudar os valores e a quantidade de números salvos no arquivo de entrada.

Para que a linguagem C redirecione os dispositivos padrão de entrada e saída para arquivos é necessário declarar variáveis ponteiro para tipo FILE:

Linha 4: faz a declaração da variável ponteiro **entrada_p** para tipo **FILE**

Linha 5: faz a declaração da variável ponteiro **saida_p** para tipo **FILE**

Linha 7: faz a declaração da variável **item** de tipo **double**

Linha 8: faz a declaração da variável **estado_da_entrada** de tipo **int**

Linha 10: **fopen** prepara o acesso ao arquivo **entrada.txt** no modo leitura ("**r**", read/scan) e atribui o valor do acesso necessário na variável ponteiro **entrada_p**. Este ponteiro para arquivo será "de entrada"

Linha 11: nesta linha, **fopen** prepara o acesso ao arquivo **saida.txt** no modo de escrita ("**w**", write) e atribui o valor do acesso necessário na variável ponteiro **saida_p**. Este ponteiro para arquivo será "de saída"

Linha 13: a função **fscanf** é equivalente à função **scanf**, embora destinada ao uso com arquivos ("file scanf"). Como primeiro parâmetro a função **fscanf** deve receber um ponteiro para arquivo de entrada, neste caso está sendo o ponteiro **entrada_p**. A função **fscanf** retorna um inteiro, representando o número de itens de entrada assignados (neste exemplo, 1 item) e quando não tem mais itens para assignar é retornado **EOF** (End Of File), que equivale ao valor **-1**.

Linha 14: Inicia a estrutura de repetição **while** com a expressão de teste. Enquanto tenha um item a ser lido (scan) no arquivo de entrada, ou seja enquanto a expressão de teste (**estado_da_entrada == 1**) seja verdadeiro, o loop continuará.

Linha 15: envia uma mensagem na tela do computador informando que vai ser salvo um item no arquivo de saída.

Linha 16: A função **fprintf** é equivalente à função **printf**, embora destinada ao uso com arquivos ("file print"). Como primeiro parâmetro a função **fprintf** deve

receber um ponteiro para arquivo de saída, neste caso está sendo o ponteiro **saida_p**.

Linha 17: nesta linha é atualizado o valor da variável **estado_de_entrada** pela leitura do seguinte item do arquivo de entrada (com o **fscanf** e o ponteiro de entrada). Se ainda tiver item o valor será **1**; se não tiver mais item o valor será **-1**

Linha 18: parênteses de fechamento do bloco de instruções do loop **while**

Linha 20: Envia uma mensagem na tela indicando que o programa vai fechar os arquivos

Linha 21: Instrução para fechar o acesso ao arquivo de estrada referenciado pelo ponteiro **entrada_p**

Linha 22: Instrução para fechar o acesso ao arquivo de saída referenciado pelo ponteiro **saida_p**

Exemplo 9.3.1_b: O programa pega um arquivo de texto previamente salvo (ou cria ele, se não existir) e acrescenta linhas com o nome do aluno e notas de três matérias. Se no lugar de ingressar um novo nome o usuário apertar <enter>, o programa faz um relatório dos alunos cadastrados, suas 3 notas e a médias dessas notas.

```
Nome do(a) aluno(a): Ana Moraes
Notas de 3 materias: 7.8 2.5 9.2
Nome do(a) aluno(a):
Ana Moraes          7.80   2.50   9.20   6.50
Process returned 0 (0x0)   execution time : 26.110 s
Press any key to continue.
```

ponteirosParaFile2.c

```
1  #include <stdio.h>
2
3  int main(void){
4      FILE *arquivop;
5      char nome[20];
6      float nota1;
7      float nota2;
8      float nota3;
9      float media;
10
11     if ((arquivop = fopen("turma3notas.txt", "a+")) == NULL) {
12         perror("");
13         exit(2);
14     }
15
16     do {
17         fprintf(stderr, "Nome do(a) aluno(a): ");
18         fflush(stdin);
19
20         if (fscanf(stdin, "%[^\n]", nome) != 1)
21             break;
22         fprintf(stderr, "Notas de 3 materias separadas por espaço: ");
23         fflush(stdin);
24         fscanf(stdin, "%f%f%f", &nota1, &nota2, &nota3);
25
26         media = (nota1 + nota2 + nota3)/3;
27
28     }
```

```
29     fprintf(arquivop, "%s:%5.2f:%5.2f:%5.2f:%5.2f\n", nome, nota1, nota2,
30     nota3, media);
31     } while (1);
32
33     rewind(arquivop);
34
35     while (fscanf(arquivop, "%[^:]:%f:%f:%f:%f", nome, &nota1, &nota2, &nota3,
36     &media) == 5)
37         fprintf(stdout, "%-20s %6.2f %6.2f %6.2f %6.2f", nome, nota1, nota2,
        nota3, media);
        return(0);
    }
```

Exercício 9.3.1_1: A execução do programa de arquivo fonte **ponteirosParaFile.c** apresenta uma mensagem ao usuário por cada item arredondado (“Salvando no arquivo de saída o item arredondado...”). Para saber o número de itens considerados, seria necessário contar essas linhas de mensagem. Faça mudanças no código para que o programa informe automaticamente, especificamente o número de itens considerados.

Exercício 9.3.1_2: Fazer as alterações necessárias no arquivo fonte **ponteirosParaFile2.c** para que o novo programa aceite o nome e 4 notas de cada aluno.

Exercício 9.3.1_3: Fazer as alterações necessárias no arquivo fonte **ponteirosParaFile2.c** para que o novo programa aceite o nome e as 3 notas de cada aluno, mas no lugar da média, o novo programa deve apresentar no final da linha de reporte, a maior nota do aluno.

Exercício 9.3.1_4: Fazer as alterações necessárias no arquivo fonte **ponteirosParaFile2.c** para que o novo programa aceite o nome e as 3 notas de cada aluno, mas no reporte deve ser apresentado (e salvo) o nome e as 3 notas na ordem crescente.

9.3.2. Ponteiros que atuam como “retorno” de função

Em uma chamada de função passamos entradas para essa função e usamos a instrução **return** para enviar de volta ao bloco de onde foi feita chamada um valor de resultado da função.

A linguagem C é uma linguagem que só aceita chamada de função por valor. Quando uma chamada de função é executada, o computador aloca espaço de memória na área de dados da função para cada parâmetro formal. O valor de cada parâmetro real é armazenado na célula de memória alocada para seu parâmetro formal correspondente. Um artifício para poder simular uma chamada de função por referência, é usar o operador de endereçamento **&** para armazenar o endereço do parâmetro real, ou seja, um ponteiro para o parâmetro real em vez de seu valor. E logo, usa-se o operador de des-endereçamento ***** para retornar resultado(s) para a função que a chama.

Exemplo 9.3.2_a: O programa pega do usuário um número real, e usando ponteiros como parâmetros de função, devolve o mesmo número

desmembrado em três partes: sinal, parte inteira e parte fracionária.

A função **separar_partes** encontra o sinal, a magnitude da parte inteira e da parte fracionária do primeiro parâmetro formal da função. Tipicamente, todos os parâmetros formais de uma função representam as entradas. Na declaração da função **separar_partes** (Linha 4), entretanto, apenas o primeiro parâmetro formal, **num**, é uma entrada; os outros três parâmetros formais – **sinal_p**, **inteira_p** e **fracionaria_p** – serão parâmetros de saída, usados para transportar vários resultados da função **separar_partes** de volta para a função que a chama. Observe que os parâmetros de saída são declarados como ponteiros.

```
Ingrese um numero para desmembra-lo: -172.3402
Partes de -172.3402
Sinal: -
Parte inteira: 172
Parte_fracionaria: 0.3402
```

ponteirosParaParametrosDeSaida.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  void separar_partes(double num, char *sinal_p, int *inteira_p, double
5  *fracionaria_p);
6
7  int main(void){
8      double numero;
9      char sinal;
10     int parte_inteira;
11     double parte_fracionaria;
12
13     printf("Ingresa um numero para desmembra-lo: ");
14     scanf("%lf", &numero);
15
16     separar_partes(numero,          &sinal,          &parte_inteira,
17     &parte_fracionaria);
18
19     printf("Partes de %.4f\n", numero);
20     printf(" Sinal:          %c\n",  sinal);
21     printf(" Parte inteira:    %d\n",  parte_inteira);
22     printf(" Parte_fracionaria: %.4f\n", parte_fracionaria);
23
24     return 0;
25 }
26
27 void separar_partes(double num, char *sinal_p, int *inteira_p, double
28 *fracionaria_p) {
29
30     double magnitude;
31
32     if (num < 0)
33         *sinal_p = '-';
```

```
33     else if (num == 0)
34         *sinal_p = ' ';
35     else
36         *sinal_p = '+';
37
38     magnitude = fabs(num);
39     *inteira_p = floor(magnitude);
    *fracionaria_p = magnitude - *inteira_p;
}
```

Neste exemplo, tanto a declaração da função (Linha 4) como o cabeçalho da função (Linha 25) definem os argumentos formais **num**, **sinal_p**, **inteira_p** e **fracionaria_p**, da função **separar_partes**, reproduzidos a continuação

```
void separar_partes(double num, char *sinal_p, int *inteira_p, double
*fracionaria_p)
```

Por outro lado, a chamada a função (Linha 15) contém os argumentos reais **numero**, **&sinal**, **&parte_inteira** e **&parte_fracionaria** da função **separar_partes**

```
separar_partes(numero, &sinal, &parte_inteira, &parte_fracionaria);
```

O valor do argumento real **numero** passado para o parâmetro formal **num** é usado como parâmetro de entrada para que a função **separar_partes** possa determinar os valores a serem enviados de volta por meio dos parâmetros de saída **sinal_p**, **inteira_p** e **fracionaria_p**. As declarações dos parâmetros de saída no cabeçalho da função especificam que se tratam de ponteiros.

O tipo de retorno da função é **void**, pois é para funções que não retornam nenhum resultado, e o corpo da função não inclui uma instrução de retorno (**return**) para enviar de volta um único valor, como seria o típico.

A declaração **char *sinal_p** (linhas 4 e 25) informa ao compilador que o parâmetro de saída **sinal_p** conterá o endereço de uma variável de tipo **char**. Outra forma de expressar a ideia de que **sinal_p** é o endereço de uma variável de tipo **char** é dizer que o parâmetro **sinal_p** é um ponteiro para uma variável de tipo **char**. Da mesma forma, os parâmetros de saída **parte_inteira_p** e **parte_fracionaria_p** são ponteiros para variáveis dos tipos **int** e **double** respectivamente.

A função chamadora deve declarar variáveis nas quais a função chamada **separar_partes** possa receber os múltiplos resultados calculados. No exemplo, a função chamadora é a função **main**, que declara três variáveis para receber esses resultados: a variável **sinal**, de tipo **char**, a variável **parte_inteira** de tipo **int**, e a variável **parte_fracionaria** de tipo **double**.

A chamada à função

```
separar_partes(numero, &sinal, &parte_inteira, &parte_fracionaria);
```

Faz com que o valor do número no argumento real **numero** seja copiado no parâmetro formal de entrada **num** e os endereços dos argumentos **sinal**, **parte_inteira** e **parte_fracionaria** sejam colocados nos correspondentes parâmetros de saída **sinal_p**, **inteira_p** e **fracionaria_p**.

E, de que forma a função **separar_partes** manipula os ponteiros para poder enviar de retorno múltiplos resultados? As instruções que permitem isso (linhas 30, 32, 34, 37 e 38):

```
*sinal_p = '-';
*sinal_p = ' ';
```

```
*sinal_p = '+';  
*inteira_p = floor(magnitude);  
*fracionaria_p = magnitude - *inteira_p;
```

Em cada caso, o nome do parâmetro formal é precedido pelo operador de des-
endereçamento *. Lembremos que quando o operador unário * é aplicado a uma
referência que é de algum tipo ponteiro, tem o efeito de considerar o conteúdo da célula
no endereço referenciado pelo ponteiro, assim, por exemplo a linha 34

```
*sinal_p = '+';
```

atribui o caractere '+' como conteúdo da célula de memória apontada pelo ponteiro
sinal_p. Fazendo o seguimento, este parâmetro **char *sinal_p** recebeu o endereço da
variável **sinal** na função **main**. Logo, a variável **sinal** acabará recebendo o caractere '+'.
A instrução

```
*inteira_p = floor(magnitude);
```

Segue o ponteiro **inteira_p** para a célula denominada **parte_inteira** na função **main** e
armazena nela o valor **172**. Similarmente, a instrução

```
*fracionaria_p = magnitude - *inteira_p;
```

Usa dois referências indiretas: uma acessa o valor da variável local **parte_inteira** da
função **main** através do ponteiro **inteira_p**, e a outra outra acessa **parte_fracionária**
da função **main** através do ponteiro **fracionaria_p** para fornecer o valor final do
argumento final **0.3402**

9.3.3. Ponteiros para realizar múltiplas chamadas à função com parâmetros entrada/saída

Na aplicação anterior, passamos informações para uma função por meio de um
parâmetro de entrada e retornamos os resultados de uma função por meio de três
parâmetros de saída.

O próximo exemplo demonstra que pode se usar um parâmetro de função para enviar
um valor de dados para dentro da função e o mesmo parâmetro para trazer um valor de
resultado para fora da função. Também demonstra como uma função pode ser chamada
mais de uma vez em um determinado programa e processar dados diferentes em cada
chamada

Exemplo 9.3.3_a: O programa pega do usuário três números reais (com ponto
decimal) e usando ponteiros como parâmetros de entrada/saída
de função, devolve os três número ordenados crescentemente.

```
Ingresse 3 numeros separados por espaco: 7.5 9.6 5.5  
Os numeros em ordem ascendente sao: 5.50 7.50 9.60
```

ponteirosParaMultiplasChamadasFuncao.c

```
1  #include <stdio.h>  
2  
3  void ordenar(double *menor_p, double *maior_p);  
4  
5  int main(void){  
6      double num1, num2, num3;  
7  
8      printf("Ingresse 3 numeros separados por espaco: ");  
9      scanf("%lf%lf%lf", &num1, &num2, &num3);  
10
```

```

11     ordenar(&num1, &num2);
12     ordenar(&num1, &num3);
13     ordenar(&num2, &num3);
14
15     printf("Os numeros em ordem ascendente sao: %.2f %.2f %.2f\n", num1,
16 num2, num3);
17
18     return 0;
19 }
20
21 void ordenar(double *menor_p, double *maior_p){
22     double temp;
23
24     if (*menor_p > *maior_p) {
25         temp = *menor_p;
26         *menor_p = *maior_p;
27         *maior_p = temp;
28     }
29 }

```

A função **main** obtém três valores de dados: **num1**, **num2** e **num3**, e rearranja eles de forma crescente de tal forma que o menor valor esteja em **num1**. As três chamadas à função **ordenar** fazem a operação de ordenamento. A modo de exemplo, considere a seguinte execução do programa onde foram introduzidos os números com ponto decimal **7.5**, **9.6** e **5.5** e o programa os devolverá ordenados de forma crescente:

Cada vez que a função **ordenar** é executada, o menor de seus dois argumentos é colocado no seu primeiro argumento real e o maior valor é colocado no seu segundo argumento real,

Assim, a chamada à função

ordenar(&num1, &num2);

coloca o menor valor de **num1** e **num2** em **num1** e o maior valor em **num2**. No exemplo de execução, **num1** é **7.5** e **num2** é **9.6**, logo, esses valores não são trocados pela execução da função **ordenar**, já a chamada à função

ordenar(&num1, &num3);

troca os valores de **num1** (de valor inicial **7.5**) e **num3** (de valor inicial **5.5**). A seguinte tabela faz um seguimento da execução da função **main**

Instrução	Num1	Num2	Num3	Efeito
scanf("...", &num1, &num2, &num3);	7.5	9.6	5.5	Ingressa os dados
ordenar(&num1, &num2);				Nenhum cambio
ordenar(&num1, &num3);	5.5	9.6	7.5	Troca num1 e num3
ordenar(&num2, &num3);	5.5	7.5	9.6	Troca num2 e num3
printf("...", num1, num2, num3);				Imprime 5.5 7.5 9.6

O corpo da função **ordenar** é baseado na instrução **if**. O cabeçalho da função

void ordenar(double *menor_p, double *maior_p)

identifica **menor_p** e **maior_p** como parâmetros entrada/saída porque a função usa os valores dos argumentos reais atuais como entradas e podem retornar valores atualizados.

Durante a execução da segunda chamada à função

```
ordenar(&num1, &num3);
```

o parâmetro formal **menor_p** contém o endereço do argumento real **num1**, e o parâmetro formal **maior_p** contém o endereço do argumento real **num3**. O teste da condição

```
(*menor_p > *maior_p)
```

faz com que ambos os ponteiros sejam seguidos, resultando na condição

```
(7.5 > 5.5)
```

Cuja avaliação é verdadeira. A execução da primeira instrução de atribuição do bloco **if**

```
temp = *menor_p;
```

causa que **7.5** seja copiado na variável local **temp**.

A execução da seguinte instrução de atribuição,

```
*menor_p = *maior_p;
```

Fará que o valor **7.5** na variável apontada por **menor_p** seja trocado pelo **5.5** que é o valor da variável apontada por **maior_p**. A última instrução de atribuição,

```
*maior_p = temp;
```

Copia o conteúdo da variável temporária (**7.5**) na variável apontada por **maior_p**. Isto completa a troca de valores.

Exercício 9.3.3_1: Refazer o **Exercício 9.3.1_4** utilizando a função **ordenar** do **Exemplo 9.3.3_a**

10. Arrays

10.1. Arrays unidimensionais (1D)

Exemplo 10.1_a: O programa pega do usuário três números reais (com ponto decimal) que são armazenados em um array unidimensional. Usando este array, o programa calcula a média e o desvio padrão dos dados. O programa reporta o valor da média e do desvio padrão, assim como uma tabela com cada dado e sua diferença com a média.

```
Ingresse 3 numeros separados por espaco ou <enter>s
> 34.2 45.2 56.8
A media eh 45.40.
O desvio padrao eh 9.23.
```

Tabela de diferencas entre os dados e a media

Indice	Dado	Diferenca
0	34.20	-11.20
1	45.20	-0.20
2	56.80	11.40

desvioPadraoComArray1D.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define NUM_ITEMS 3
5
6  int main(void){
7      double x[NUM_ITEMS];
8      double soma;
9      double media;
10     double dif_dado_media;
11     double soma_qua_das_dif;
12     double desvio_padrao;
13
14     int i;
15
16     printf("Ingresse %d numeros separados por espaco ou <enter>s\n",
17 NUM_ITEMS);
18     for (i = 0; i < NUM_ITEMS; ++ i)
19         scanf("%lf", &x [i]);
20
21     soma = 0;
22     for (i = 0; i < NUM_ITEMS; ++i)
23         soma += x[i];
24     media = soma / NUM_ITEMS;
25
26     soma_qua_das_dif = 0;
27     for (i = 0; i < NUM_ITEMS; ++i){
28         dif_dado_media = x[i] - media;
29         soma_qua_das_dif += pow(dif_dado_media, 2);
30     }
31     desvio_padrao = sqrt(soma_qua_das_dif / NUM_ITEMS);
32
33     printf("A media eh %.2f.\n", media);
34     printf("O desvio padrao eh %.2f.\n", desvio_padrao);
35
36     printf("\nTabela de diferencas entre os dados e a media\n");
37     printf("Indice      Dado      Diferenca\n");
38     for (i = 0; i < NUM_ITEMS; ++i)
39         printf("%3d%4c%9.2f%5c%9.2f\n", i, ' ', x[i], ' ', x[i] - media);
40
41     return 0;
```

```
}
```

Exemplo 10.1_b: O programa calcula a média e o desvio padrão de 3 times (grupos) A, B e C. Para cada time é considerado 5 quesitos (5 pontuações). O programa depois faz um relatório apresentando para cada time sua média e seu desvio padrão.

```
Ingresse as 5 pontuacoes do Time A:
98.2 99.3 97.7 96.5 99.2
Ingresse as 5 pontuacoes do Time B:
99.3 96.3 99.2 97.7 98.2
Ingresse as 5 pontuacoes do Time C:
98.2 99.7 97.2 96.2 95.9
Time A: Media = 98.18 Desvio padrao = 1.03
Time B: Media = 98.14 Desvio padrao = 1.10
Time C: Media = 97.44 Desvio padrao = 1.39
```

desvioPadraoDe3Times5quesitosComArray1D.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define NUM_TIMES 3
5  #define NUM_QUESITOS 5
6
7  void preencher_array(double lista[], char);
8  double calcular_media(double lista[]);
9  double calcular_desvio_padrao(double lista[], double *);
10
11 int main(void){
12     double timeA[NUM_QUESITOS];
13     double timeB[NUM_QUESITOS];
14     double timeC[NUM_QUESITOS];
15
16     char nomes [NUM_TIMES] = {'A','B','C'};
17     double medias [NUM_TIMES] = {0.0, 0.0, 0.0};
18     double desvios[NUM_TIMES] = {0.0, 0.0, 0.0};
19
20     preencher_array(timeA,nomes[0]);
21     preencher_array(timeB,nomes[1]);
22     preencher_array(timeC,nomes[2]);
23     desvios[0] = calcular_desvio_padrao(timeA, &medias[0]);
24     desvios[1] = calcular_desvio_padrao(timeB, &medias[1]);
25     desvios[2] = calcular_desvio_padrao(timeC, &medias[2]);
26
27     int i;
28     for (i = 0; i < NUM_TIMES; ++ i)
29         printf("Time %c:\t Media = %.2f\t Desvio padrao = %.2f\n",
30             nomes[i],
31                                     medias[i],
32             desvios[i]);
33
34     return 0;
35 }
36
37 void preencher_array (double lista[], char nome) {
38     int i;
39     printf("Ingresse as %d pontuacoes do Time %c: \n",
40         NUM_QUESITOS,nome);
41     for (i = 0; i < NUM_QUESITOS; ++ i)
42         scanf("%lf", &lista[i]);
```

```
43 }
44
45 double calcular_desvio_padrao(double lista[], double *media){
46     *media = calcular_media(lista);
47     double dif_dado_media = 0.0;
48     double soma_qua_das_dif = 0.0;
49
50     int i;
51     for (i = 0; i < NUM_QUESITOS; ++i){
52         dif_dado_media = lista[i] - *media;
53         soma_qua_das_dif += pow(dif_dado_media, 2);
54     }
55     return sqrt(soma_qua_das_dif / NUM_QUESITOS);
56 }
57
58 double calcular_media(double lista[]){
59     double soma = 0;
60
61     int i;
62     for (i = 0; i < NUM_QUESITOS; ++i)
63         soma += lista[i];
64     return (soma / NUM_QUESITOS);
65 }
```

10.2. Matrices, arrays bidimensionais (2D)

Exemplo 10.2_a: O programa pega do usuário duas matrizes bidimensionais e realiza a soma delas (se as dimensões das matrizes forem as mesmas).

```
Quantas linhas tem a matriz 1? : 2
Quantas colunas tem a matriz 1?: 3
Preenchendo a matriz 1:
    Ingresse o valor para o elemento 1,1 : 2.3
    Ingresse o valor para o elemento 1,2 : 2.1
    Ingresse o valor para o elemento 1,3 : 3.2
    Ingresse o valor para o elemento 2,1 : 5.1
    Ingresse o valor para o elemento 2,2 : 2.1
    Ingresse o valor para o elemento 2,3 : 2.4

Quantas linhas tem a matriz 2? : 2
Quantas colunas tem a matriz 2?: 3
Preenchendo a matriz 2
    Ingresse o valor para o elemento 1,1 : 1.2
    Ingresse o valor para o elemento 1,2 : 1.3
    Ingresse o valor para o elemento 1,3 : 1.4
    Ingresse o valor para o elemento 2,1 : 1.7
    Ingresse o valor para o elemento 2,2 : 1.1
    Ingresse o valor para o elemento 2,3 : 1.2

Matriz 1:
    2.30  2.10  3.20
    5.10  2.10  2.40

Matriz 2:
    1.20  1.30  1.40
    1.70  1.10  1.20

A soma das matrizes eh possivel. A matriz soma eh:
    3.50  3.40  4.60
    6.80  3.20  3.60
```

somaDeMatrizes.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define linhas 10
5  #define colunas 10
6
7  int main() {
8      int linhas1, colunas1;
9      int linhas2, colunas2;
10     int i,j;
11     float matriz1[linhas][colunas];
```

```
12     float matriz2[linhas][colunas];
13     float matriz_soma[linhas][colunas];
14     printf("Quantas linhas tem a matriz 1? : ");
15     scanf("%d", &linhas1);
16     printf("Quantas colunas tem a matriz 1?: ");
17     scanf("%d", &colunas1);
18     printf("Preenchendo a matriz 1:\n");
19     for(i = 0; i < linhas1; i++){
20         for(j = 0; j < colunas1; j++){
21             printf("\tIngresse o valor para o elemento %d,%d : ", i+1,
22 j+1);
23             scanf("%f", &matriz1[i][j]);
24         }
25     }
26     printf("\n");
27     printf("Quantas linhas tem a matriz 2? : ");
28     scanf("%d", &linhas2);
29     printf("Quantas colunas tem a matriz 2?: ");
30     scanf("%d", &colunas2);
31     printf("Preenchendo a matriz 2:\n");
32     for(i = 0; i < linhas2; i++){
33         for(j = 0; j < colunas2; j++){
34             printf("\tIngresse o valor para o elemento %d,%d : ", i+1,
35 j+1);
36             scanf("%f", &matriz2[i][j]);
37         }
38     }
39     printf("\nMatriz 1:\n");
40     for(i = 0; i < linhas1; i++){
41         printf("\t");
42         for(j = 0; j < colunas1; j++){
43             printf("%.2f  ", matriz1[i][j]);
44         }
45         printf("\n");
46     }
47     printf("\nMatriz 2:\n");
48     for(i = 0; i < linhas2; i++){
49         printf("\t");
50         for(j = 0; j < colunas2; j++)
51             printf("%.2f  ", matriz2[i][j]);
52         printf("\n");
53     }
54     if((linhas1 == linhas2) && (colunas1 == colunas2)){
55         printf("\nA soma das matrizes eh possivel. ");
56         printf("A matriz soma eh: \n");
57         for(i = 0; i < linhas1; i++)
58             for(j = 0; j < colunas1; j++)
59                 matriz_soma[i][j] = matriz1[i][j]+matriz2[i][j];
60         for(i = 0; i < linhas1; i++){
61             printf("\t");
62             for(j = 0; j < colunas1; j++)
63                 printf("%.2f  ", matriz_soma[i][j]);
64             printf("\n");
65         }
66     }
67     else
68         printf("\nA soma das matrizes nao eh possivel.\n\n");
        return 0;
```

}

Exemplo 10.2_b: O programa realiza a mesma tarefa que no **Exemplo 10.1.b**, mas desta vez, fazendo uso de um array bidimensional (matriz). Calcula a média e o desvio padrão de 3 times (grupos) A, B e C. Para cada time é considerado 5 quesitos (5 pontuações). O programa depois faz um relatório apresentando para cada time sua média e seu desvio padrão.

```
Ingresse as 5 pontuacoes do Time A:
98.2 99.3 97.7 96.5 99.2
Ingresse as 5 pontuacoes do Time B:
99.3 96.3 99.2 97.7 98.2
Ingresse as 5 pontuacoes do Time C:
98.2 99.7 97.2 96.2 95.9
Time A: Media = 98.18 Desvio padrao = 1.03
Time B: Media = 98.14 Desvio padrao = 1.10
Time C: Media = 97.44 Desvio padrao = 1.39
```

desvioPadraoDe3Times5quesitosComArray2D.c

```
1  #include <stdio.h>
2  #include <math.h>
3
4  #define NUM_TIMES 3
5  #define NUM_QUESITOS 5
6
7  void preencher_array(double lista[][NUM_QUESITOS], char nom []);
8  double calcular_media(double lista[]);
9  double calcular_desvio_padrao(double lista[], double *);
10
11 int main(void){
12     double notas_dos_times[NUM_TIMES][NUM_QUESITOS];
13     char nomes [NUM_TIMES] = {'A','B','C'};
14     double resultados [NUM_TIMES][2] = {{0.0, 0.0},{0.0, 0.0},{0.0,
15 0.0}};
16
17     preencher_array(notas_dos_times, nomes);
18
19     int i;
20     for (i = 0; i < NUM_TIMES; ++ i){
21         resultados[i][1] = calcular_desvio_padrao(notas_dos_times[i],
22
23 &resultados[i][0]);
24         printf("Time %c:\t Media = %.2f\t Desvio padrao = %.2f\n",
25 nomes[i],
26
27 resultados[i][1] );
28     }
29
30     return 0;
31 }
32
33 void preencher_array (double lista[][NUM_QUESITOS], char nome[]) {
34     int i;
35     int j;
36     for (i = 0; i < NUM_TIMES; ++ i){
37         printf("Ingresse as %d pontuacoes do Time %c: \n",
38 NUM_QUESITOS,nome[i]);
39         for (j = 0; j < NUM_QUESITOS; ++ j)
40             scanf("%lf", &lista[i][j]);
41     }
42 }
```

```
43
44 double calcular_desvio_padrao(double lista[], double *media){
45     *media = calcular_media(lista);
46     double dif_dado_media = 0.0;
47     double soma_qua_das_dif = 0.0;
48
49     int i;
50     for (i = 0; i < NUM_QUESITOS; ++i){
51         dif_dado_media = lista[i] - *media;
52         soma_qua_das_dif += pow(dif_dado_media, 2);
53     }
54     return sqrt(soma_qua_das_dif / NUM_QUESITOS);
55 }
56
57 double calcular_media(double lista[]){
58     double soma = 0;
59
60     int i;
61     for (i = 0; i < NUM_QUESITOS; ++i)
62         soma += lista[i];
63     return (soma / NUM_QUESITOS);
64 }
```


10.3. Revisão de arrays

Um array é uma estrutura de dados fundamental que permite o armazenamento e a manipulação de quantidades potencialmente enormes de dados. Um array armazena uma sequência ordenada de valores homogêneos. Homogêneo significa que todos os valores são do mesmo tipo de dados.

Na linguagem C todo array tem duas propriedades fundamentais: o tipo de dado dos elementos do array, e o comprimento do array. Os elementos individuais do array são identificados por um número inteiro chamado índice. Os índices começam em zero e sempre são escritos dentro de colchetes.

Seja um grupo de 7 alunos com idades 19, 20, 25, 20, 21, 20 e 19 anos. Podemos declarar o array da seguinte forma:

```
int idades[7];
```

e logo inicializar o array **idades** já declarado, da seguinte forma:

```
int idades[] = {19, 20, 25, 20, 21, 20, 19};
```

O primeiro elemento da lista idades é acessado com a seguinte sintaxe:

```
idades[0]
```

ou seja, com o nome da lista seguido pelo **índice do elemento** dentro de colchetes.

O índice do elemento de uma lista é um número inteiro que indica a posição do elemento na lista. O primeiro elemento tem índice 0, o segundo elemento tem índice 1, e assim sucessivamente. A seguinte ilustração apresenta esses elementos e identificadores:

idades[0]	idades[1]	idades[2]	idades[3]	idades[4]	idades[5]	idades[6]
19	20	25	20	21	20	19
1º elem	2º elem	3º elem	4º elem	5º elem	6º elem	7º elem

```
1  #include <stdio.h>
2  #define NUM_ALUNOS 3
3
4  void caso1();
5  void caso2();
6  void caso3();
7
8  int main(){
9      caso1();
10     caso2();
11     caso3();
12     return 0;
13 }
14
15 void caso1(){
16     int idades [NUM_ALUNOS] = {19, 20, 25};
17     int i;
18     for (i=0; i<NUM_ALUNOS; i++)
19         printf("Idade do aluno %d: %d\n", i, idades[i]);
```

```
20 }
21
22 void caso2(){
23     int idades[NUM_ALUNOS];
24     int i;
25     for (i=0; i<NUM_ALUNOS; i++){
26         printf("Escreva a idade da aluna %d: ", i);
27         scanf("%d", &idades[i]);
28     }
29
30     for (i=0; i<NUM_ALUNOS; i++)
31         printf("Idade da aluna %d: %d\n", i, idades[i]);
32
33 }
34
35 void caso3(){
36     int tamanho;
37     printf("Escreva o tamanho da turma: ");
38     scanf("%d", &tamanho);
39
40     int idades[tamanho];
41     int i;
42     for (i=0; i<tamanho; i++){
43         printf("Escreva a idade da pessoa %d: ", i);
44         scanf("%d", &idades[i]);
45     }
46
47     for (i=0; i<tamanho; i++)
48         printf("Idade da pessoa %d: %d\n", i, idades[i]);
49
50 }
```

11. Tipos definidos pelo programador

Além dos tipos de variáveis básicos (**char**, **int**, **float**, **double** e **void**) e dos tipos compostos homogêneos (array), a linguagem C permite criar novos tipos de dados que podem ser:

- Estruturas/Registros (com o comando **struct**)
- Uniões (com o comando **union**)
- Enumerações (com o comando **enum**)
- Renomear um tipo existente (com o comando **typedef**)

11.1. Structs (também conhecidas como Registros)

A *structs* definem tipos de dados que agrupam variáveis sob um mesmo tipo de dado.

A ideia de usar uma struct é permitir que, ao armazenar os dados de uma mesma entidade, isto possa ser feito com uma única variável.

Por exemplo, ao fazer uma pesquisa dos filmes que em um cinema estão passando, gostaríamos de anotar (registrar) o nome do filme, o número de lugares vazios que ainda há para esse filme, e o preço de cada ingresso.

Podemos criar uma tabela da seguinte forma:

	filme	num_lugares	valor_unit
registro1	"Raya -o ultimo dragao-"	4	14.50
registro2	"Mulher Maravilha 1984"	5	25.00
registro3	"Meu pai"		
registro4	"Pinoquio"	7	12.50

Reparemos que todos os registros tem 3 campos (também chamados de membros): **filme** (de tipo **string**), **num_lugares** (de tipo **int**) e **valor_unit** (de tipo **float**). Então podemos afirmar que cada registro tem a mesma estrutura, estrutura que podemos chamar de "**ingresso_cinema**"; ou seja, **registro1** é do tipo **ingresso_cinema**, **registro2** é do tipo **ingresso_cinema**, e da mesma forma **registro3** e **registro4**.

Na linguagem C, essa mesma tabela pode ser criada usando o tipo de dado chamado *structure*. Em uma estrutura, cada elemento (chamado de campo, ou membro) tem um nome e seu próprio tipo de dados:

Do exemplo:	Em geral:
<pre>struct ingresso_cinema { char filme[30]; short num_lugares; float valor_unit; };</pre>	<pre>struct nome_da_estrutura { tipo nome_do_campo; tipo nome_do_campo; };</pre>

No seguinte programa se apresenta a forma de como criar esses 4 registros do exemplo. Os registros 1, 2 e 4 são preenchidos no tempo da compilação (pelo programador), e o registro 3 será preenchido nos dois campos vazios, no tempo de execução (pelo usuário).

- O registro1 é criado e preenchido junto à declaração da estrutura.
O registro2 é criado e completamente preenchido na linha 13.
O registro3 é criado e parcialmente preenchido na linha 14.
O registro4 é criado na linha 15, e preenchido nas linhas 17, 18 e 19.

```
O tamanho do registro1 eh 36

Os quatro filmes sao:
    Raya -o ultimo dragao-
    Mulher Maravilha 1984
    Meu pai, e
    Pinoquio

Ingresse o num de lugares e o valor para o registro3: 3 15.90
"Meu pai" tem 3 lugares que custam 15.90 cada.
```

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void){
5      struct ingresso_cinema {
6          char filme[30];
7          short num_lugares;
8          float valor_unit;
9      } registro1 = {"Raya -o ultimo dragao-", 4, 14.50};
10
11     printf("O tamanho do registro1 eh %d\n\n", sizeof registro1);
12
13     struct ingresso_cinema registro2 = {"Mulher Maravilha 1984", 5, 25.00};
14     struct ingresso_cinema registro3 = {"Meu pai"};
15     struct ingresso_cinema registro4;
16
17     strcpy(registro4.filme, "Pinoquio");
18     registro4.num_lugares = 7;
19     registro4.valor_unit = 12.50;
20     printf("Os quatro filmes sao:\n");
21     printf("\t%s\n\t%s\n\t%s, e\n\t%s\n\n", registro1.filme, registro2.filme,
22                                                registro3.filme, registro4.filme);
23
24     fputs("Ingresse o num de lugares e o valor para o registro3: ", stdout);
25     scanf("%hd%f", &registro3.num_lugares, &registro3.valor_unit);
26     printf("\t%s\n tem %hd lugares que custam %.2f cada.\n", registro3.filme,
27                                                registro3.num_lugares, registro3.valor_unit);
28     return 0;
29 }
```

strcpy	Função da biblioteca string.h Copia uma <i>string constante</i> dentro de uma variável. Na linha 17, copia a string constante “Pinoquio” dentro do campo filme , que é a string de 30 caracteres da variável registro4 de tipo struct ingresso_cinema .
fputs	Função da biblioteca stdio.h Escreve uma linha de caracteres para uma stream, e não adiciona uma nova linha ao final da string. A linha 24 escreve a string do primeiro parâmetro à stream da saída padrão (a tela)

11.2. Typedef e seu uso na declaração de estruturas

A palavra-chave `typedef` permite ao programador criar um novo nome de tipo de dados para um tipo de dados existente. Nenhum novo tipo de dados é produzido, mas um nome alternativo é fornecido para um tipo de dados conhecido. A forma geral da instrução de declaração usando a palavra-chave `typedef` é:

`typedef` <tipo de dados existente> <novo tipo de dados, ... >;

A instrução `typedef` não ocupa armazenamento; simplesmente define um novo tipo. As instruções `typedef` podem ser colocadas em qualquer lugar em um programa C, contanto que venham antes de seu primeiro uso no código. Os exemplos seguintes mostram o uso de `typedef`.

<code>typedef int id_numero;</code>	<code>id_numero</code> é o novo nome de tipo de dado atribuído para o tipo de dado <code>int</code>
<code>typedef float peso;</code>	<code>peso</code> é o novo nome de tipo de dado atribuído para o tipo de dado <code>float</code>
<code>typedef struct ilha{ char *nome; char *abre; char *fecha; struct ilha *proximo_p; } ilha_rio;</code>	<code>ilha_rio</code> é o novo nome de tipo de dado atribuído para o tipo de dado complexo <code>ilha</code> , que é uma estrutura. Neste caso, o nome da estrutura (<code>ilha</code>) não é obrigatório.

12. Recursividade

Exemplo (Cap. 9 do nosso livro texto): Para esvaziar um vaso contendo N flores, primeiro verificamos se o vaso está vazio. Caso esteja vazio, a tarefa está satisfatoriamente realizada. Se o vaso não está vazio, tiramos uma flor. Temos agora que esvaziar um vaso contendo N - 1 flores.

A ideia básica da recursão é “dividir e conquistar”:

- Divide-se um problema maior em um conjunto de problemas menores.
- Esses problemas menores são então resolvidos de forma independente.
- As soluções dos problemas menores são combinadas para gerar a solução final.

12.1. Fatorial de um número

O fatorial de um número inteiro não negativo é escrito como $n!$ (pronunciado “n fatorial”) e é definido assim:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1 \quad (\text{para } n \text{ maior ou igual a } 1)$$

e

$$n! = 1 \quad (\text{para } n = 0).$$

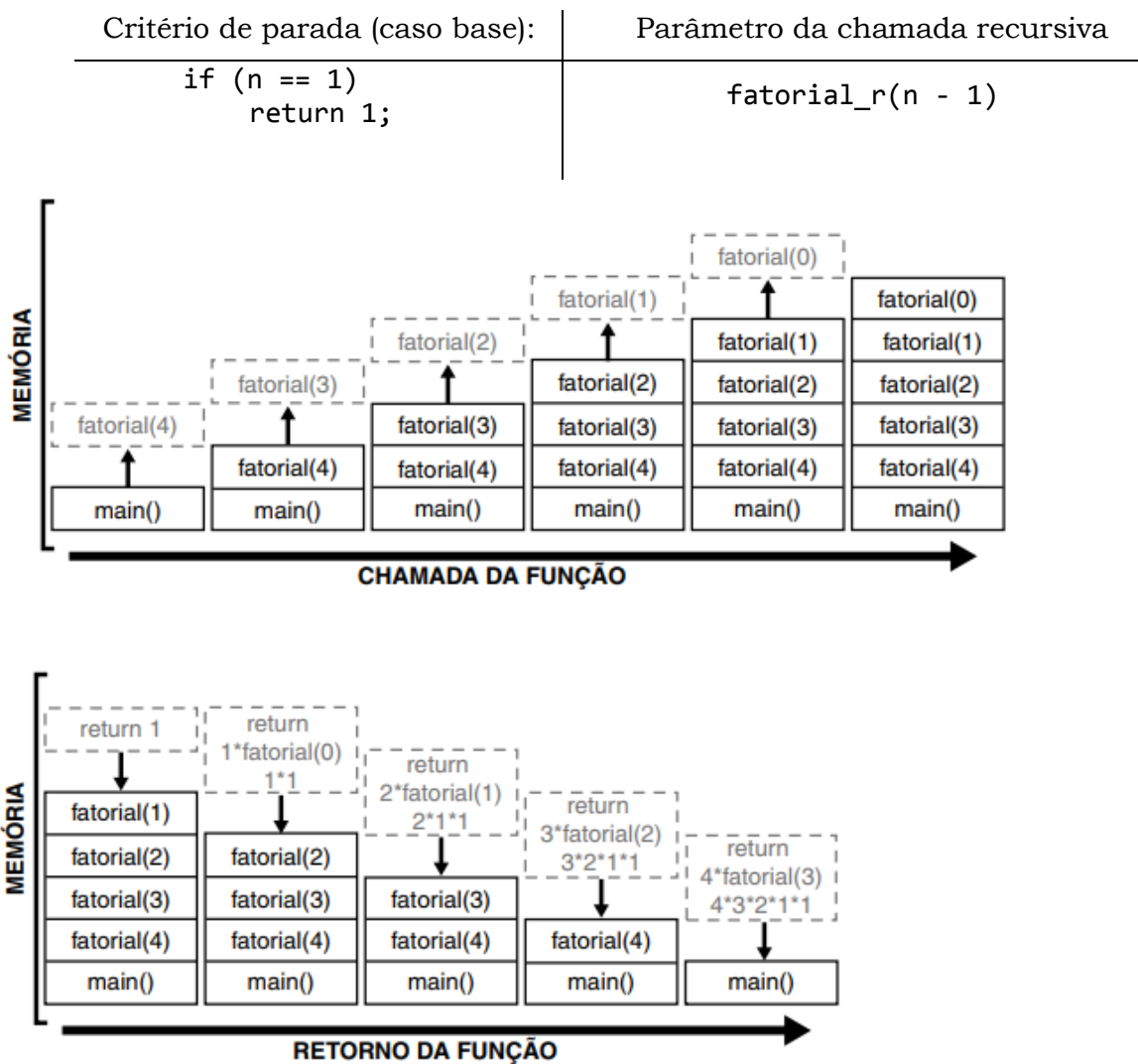
Por exemplo, $4! = 4 \cdot 3 \cdot 2 \cdot 1$, que é igual a 120.

Aplicando a ideia da recursão, temos que:

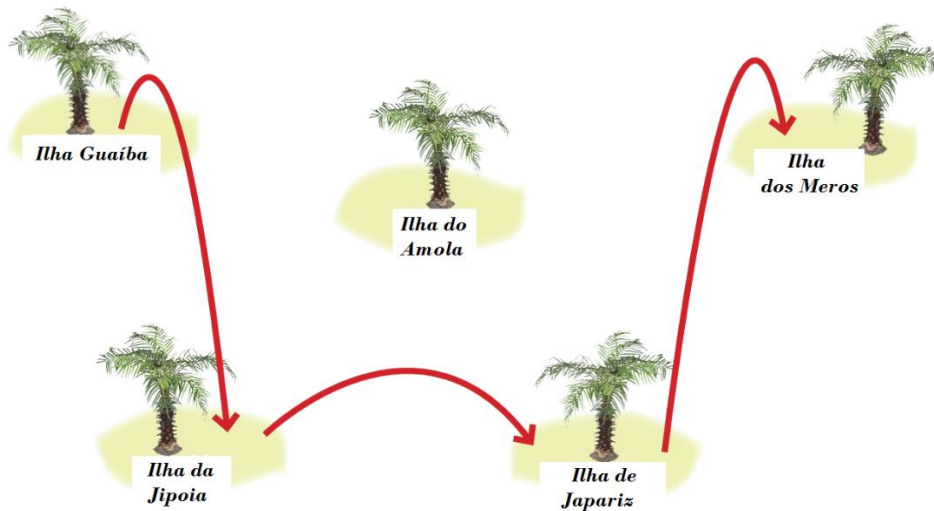
- O fatorial de 4 é definido em função do fatorial de 3. (**$4! = 4 \cdot 3!$**)
- O fatorial de 3 é definido em função do fatorial de 2. (**$3! = 3 \cdot 2!$**)
- O fatorial de 2 é definido em função do fatorial de 1. (**$2! = 2 \cdot 1!$**)
- O fatorial de 1 é definido em função do fatorial de 0. (**$1! = 1 \cdot 0!$**)
- O fatorial de 0 é definido como igual a 1. (**$0! = 1$**) **caso-base**

Perceba que o cálculo do fatorial prossegue até chegarmos ao fatorial de 0, nosso caso-base, que é igual a 1.

```
1  # include <stdio.h>
2
3  long int fatorial_d(int );
4  long int fatorial_e(int );
5  long int fatorial_r(int );
6
7  int main(){
8      int numero;
9      long int fatorial;
10     printf("\nIngresse um numero inteiro positivo: ");
11     scanf("%d",&numero);
12
13     //fatorial = fatorial_d(n);
14     //fatorial = fatorial_e(n);
15     fatorial = fatorial_r(numero);
16
17     printf("O fatorial de %d eh: %ld\n", numero, fatorial);
18
19 }
20
```



13. Estruturas recursivas e listas encadeadas



```

1  # include <stdio.h>
2
3  typedef struct ilha{
4      char *nome;
5      char *abre;
6      char *fecha;
7      struct ilha *proximo_p;
8  } ilha_rio;
9
10 void display(ilha_rio *);
11
12 int main(){
13
14     ilha_rio guaiba = {"Ilha Guaiba", "09:00", "17:00", NULL};
15     ilha_rio jipoia = {"Ilha da Jipoia", "09:00", "17:00", NULL};
16     ilha_rio japariz = {"Ilha de Japariz", "09:00", "17:00", NULL};
17     ilha_rio meros = {"Ilha dos Meros", "09:00", "17:00", NULL};
18
19     guaiba.proximo_p = &jipoia;
20     jipoia.proximo_p = &japariz;
21     japariz.proximo_p = &meros;
22
23     ilha_rio amola = {"Ilha do Amola", "09:00", "17:00", NULL};
24     japariz.proximo_p = &amola;
25     amola.proximo_p = &meros;
26
27     display(&guaiba);
28
29     return 0;
30 }
31
32 void display(ilha_rio *inicio){
33     ilha_rio *i = inicio;
34     for (; i!= NULL ; i = i->proximo_p)
35         printf("Nome: %20s Funciona: %s - %s\n",i->nome ,i->abre ,i->fecha );
36 }
    
```