

Ministério da Educação - MEC  
Secretaria de Educação Profissional e Tecnológica (SETEC)  
Instituto Federal de Educação, Ciência e Tecnologia do Ceará.



## Técnico em Automação Industrial

Algoritmo e Linguagem de programação

[www.ifce.edu.br/pronatec](http://www.ifce.edu.br/pronatec)

Professor: José Ciro dos Santos



**Presidente**

Dilma Vana Rousseff

**Ministro da Educação**

Aloizio Mercadante Oliva

**Secretaria de Educação Profissional e Tecnológica**

Marco Antonio de Oliveira

**Reitor do IFCE**

Cláudio Ricardo Gomes de Lima

**Pró-Reitor de Extensão**

Gilmar Lopes Ribeiro

**Pró-Reitor de Ensino**

Gilmar Lopes Ribeiro

**Pró-Reitor de Administração**

Gilmar Lopes Ribeiro

**Coordenador Geral**

Jose Wally Mendonça Menezes

## Sumário

O QUE É O PRONATEC.....	3
<b>ALGORISMOS ESTRUTURADOS .....</b>	<b>8</b>
INTRODUÇÃO.....	4
CONCEITOS .....	5
ELABORAÇÃO DE ALGORÍTMO .....	8
DIAGRAMAS DE BLOCO.....	12
CONSTANTES, VARIÁVEIS E TIPOS DE DADOS.....	14
OPERADORES.....	16
ESTRUTURAS DE DECISÃO .....	20
ESTRUTURAS DE REPETIÇÃO .....	24
CHAMADA DE ALGORÍTMOS.....	28
MODULAÇÃO .....	29
CONCEITOS DE PROGRAMAÇÃO ORIENTADA A OBJETOS.....	31
ALGORÍTMOS DE ORDENAÇÃO.....	36
<b>LINGUAGEM DE PROGRAMAÇÃO C .....</b>	<b>42</b>
CONCEITO FUNDAMENTAIS .....	42
EXPRESSÕES .....	50
CONTROLE DE FLUXO.....	64
FUNÇÕES .....	75
VETORES E ALOCAÇÃO.....	90
CADEIA DE CARACTERES.....	99
MATRIZES .....	115
TIPOS ESTRUTURADOS.....	125
<b>CRÉDITOS.....</b>	<b>138</b>

## **O QUE É O PRONATEC?**

Criado no dia 26 de Outubro de 2011 com a sanção da Lei nº 12.513/2011 pela Presidenta Dilma Rousseff, o Programa Nacional de Acesso ao Ensino Técnico e Emprego (Pronatec) tem como objetivo principal expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional e Tecnológica (EPT) para a população brasileira. Para tanto, prevê uma série de subprogramas, projetos e ações de assistência técnica e financeira que juntos oferecerão oito milhões de vagas a brasileiros de diferentes perfis nos próximos quatro anos. Os destaques do Pronatec são:

- Criação da Bolsa-Formação;
- Criação do FIES Técnico;
- Consolidação da Rede e-Tec Brasil;
- Fomento às redes estaduais de EPT por intermédio do Brasil Profissionalizado;
- Expansão da Rede Federal de Educação Profissional Tecnológica (EPT).

A principal novidade do Pronatec é a criação da Bolsa-Formação, que permitirá a oferta de vagas em cursos técnicos e de Formação Inicial e Continuada (FIC), também conhecidos como cursos de qualificação. Oferecidos gratuitamente a trabalhadores, estudantes e pessoas em vulnerabilidade social, esses cursos presenciais serão realizados pela Rede Federal de Educação Profissional, Científica e Tecnológica, por escolas estaduais de EPT e por unidades de serviços nacionais de aprendizagem como o SENAC e o SENAI.

### **Objetivos**

- Expandir, interiorizar e democratizar a oferta de cursos de Educação Profissional Técnica de nível médio e de cursos e programas de formação inicial e continuada de trabalhadores;
- Fomentar e apoiar a expansão da rede física de atendimento da Educação Profissional e Tecnológica;
- Contribuir para a melhoria da qualidade do Ensino Médio Público, por meio da Educação Profissional;
- Ampliar as oportunidades educacionais dos trabalhadores por meio do incremento da formação profissional.

### **Ações**

- Ampliação de vagas e expansão da Rede Federal de Educação Profissional e Tecnológica;
- Fomento à ampliação de vagas e à expansão das redes estaduais de Educação Profissional;
- Incentivo à ampliação de vagas e à expansão da rede física de atendimento dos Serviços Nacionais de Aprendizagem;
- Oferta de Bolsa-Formação, nas modalidades:

- Bolsa-Formação Estudante;
- Bolsa-Formação Trabalhador.
- Atendimento a beneficiários do Seguro-Desemprego;

### *Algoritmo Estruturado*

## ***1. INTRODUÇÃO***

---

A automatização de tarefas é um aspecto marcante da sociedade moderna. O aperfeiçoamento tecnológico alcançado, com respeito a isto, teve como elementos fundamentais a análise e a obtenção de descrições da execução de tarefas em termos de ações simples o suficiente, tal que pudessem ser automatizadas por uma máquina especialmente desenvolvida para este fim, O COMPUTADOR.

Em ciência da computação houve um processo de desenvolvimento simultâneo e interativo de máquinas (hardware) e dos elementos que gerenciam a execução automática (software) de uma dada tarefa. E essa descrição da execução de uma tarefa, como considerada acima, é chamada **algoritmo**.

O objetivo desse curso é a Lógica de Programação dando uma base teórica e prática, suficientemente boa, para que, o aluno domine os algoritmos e esteja habilitado a aprender uma linguagem de programação. Será mostrado também um grupo de algoritmos clássicos para tarefas cotidianas, tais como : ordenação e pesquisa.

## **2. CONCEITOS**

---

### **2.1. Lógica**

A lógica de programação é necessária para pessoas que desejam trabalhar com desenvolvimento de sistemas e programas, ela permite definir a sequência lógica para a resolução de um problema.

**Lógica de programação é a técnica de encadear pensamentos para atingir determinado objetivo.**

### **2.2. Sequência Lógica**

Estes pensamentos podem ser descritos como uma sequência de instruções, que devem ser seguidas para se cumprir uma determinada tarefa.

Sequência Lógica são passos executados até atingir um objetivo ou solução de um problema.

### **2.3. Instruções**

Na linguagem comum, entende-se por instruções **um conjunto de regras ou normas definidas para a realização ou emprego de algo**.

Em informática, porém, instrução é a informação que indica a um computador uma ação elementar a executar.

Convém ressaltar que uma ordem isolada não permite realizar o processo completo, para isso é necessário um conjunto de instruções colocadas em ordem sequencial lógica.

Por exemplo, se quisermos fazer uma omelete de batatas, precisaremos colocar em prática uma série de instruções: descascar as batatas, bater os ovos, fritar as batatas, etc...

É evidente que essas instruções tem que ser executadas em uma ordem adequada ó não se pode descascar as batatas depois de fritá-las.

Dessa maneira, uma instrução tomada em separado não tem muito sentido; para obtermos o resultado, precisamos colocar em prática o conjunto de todas as instruções, na ordem correta.

## 2.4. Algoritmo

- "O conceito central da programação e da Ciência da Computação é o conceito de algoritmos, isto é, programar é basicamente construir algoritmos".
- É a descrição, de forma lógica, dos passos a serem executados no cumprimento de determinada tarefa.
- "O algoritmo pode ser usado como uma ferramenta genérica para representar a solução de tarefas independente do desejo de automatizá-las, mas em geral está associado ao processamento eletrônico de dados, onde representa o rascunho para programas (Software)".
- "Serve como modelo para programas, pois sua linguagem é intermediária à linguagem humana e às linguagens de programação, sendo então, uma boa ferramenta na validação da lógica de tarefas a serem automatizadas".
- "Um algoritmo é uma receita para um processo computacional e consiste de uma série de operações primitivas, interconectadas devidamente, sobre um conjunto de objetos. Os objetos manipulados por essas receitas são as variáveis".
- O algoritmo pode ter vários níveis de abstrações de acordo com a necessidade de representar ou encapsular detalhes inerentes às linguagens de programação. Ex: Certamente um algoritmo feito com o objetivo de servir como modelo para uma linguagem de III geração é diferente daquele para uma linguagem de IV geração. Mas isso não impede que a ferramenta em si possa ser usada em ambos o caso.
- Como qualquer modelo, um algoritmo é uma abstração da realidade. A abstração é o processo de identificar as propriedades relevantes do fenômeno que está sendo modelado. Usando o modelo abstrato, podemos nos centrar unicamente nas propriedades relevantes para nós, dependendo da finalidade da abstração, e ignorar as irrelevantes.

É a forma pela qual descrevemos soluções de problemas do **nosso mundo**, a fim de serem implementadas utilizando os recursos do **mundo computacional**. Como este possui severas limitações em relação ao nosso



mundo, exige que, sejam impostas algumas regras básicas na forma de solucionar os problemas, para que, possamos utilizar os recursos de **hardware** e **software** disponíveis. Pois, os algoritmos, apesar de servirem para representar a solução de qualquer problema, no caso do Processamento de Dados, eles devem seguir as regras básicas de programação para que sejam compatíveis com as **linguagens de programação**.

Exemplos de algoritmos:

*õEscovar os dentesö.*

Passo1: Pegar a escova

Passo2: Colocar o creme dental

Passo3: Levar a escova aos dentes e movimenta sobre eles

Passo4: lavar os dentes e a escova

Passo5: Guardar a escova

*õSomar dois números quaisquerö.*

Passo1: Escreva o primeiro número

Passo2: Escreva o segundo número no retângulo B

Passo3: Some o número A com número B e escreva resultado

## **2.5. Programas**

Os programas de computadores nada mais são do que algoritmos escritos numa linguagem de computador (Pascal, C, Cobol, Fortran, Visual Basic entre outras) e que são interpretados e executados por uma máquina, no caso um computador. Notem que dada esta interpretação rigorosa, um programa é por natureza muito específico e rígido em relação aos algoritmos da vida real.



### ***3. Elaboração de Algoritmos***

---

Para escrevermos algoritmos é preciso uma linguagem clara e que não deixe margem a ambiguidades, para isto, devemos definir uma sintaxe e uma semântica, de forma a permitir uma única interpretação das instruções num algoritmo.

#### ***3.1. Como elaborar algoritmos?***

Para elaborar algoritmos podemos adotar algumas das formas sugeridas abaixo:

- Adotar soluções clássicas, copiando e adaptando de livros, experiências, idéias novas, etc.
- Emulação de processos manuais, observando como solucionamos o problema, executamos a tarefa manualmente.
- Descrição de passos para solução do problema. Precisamos fazer uma receita de como resolvemos o problema, de forma a instruir o computador, um ser ignorante.

#### ***3.2. Pseudocódigo***

Os algoritmos são descritos em uma linguagem chamada **pseudocódigo**. Este nome é uma alusão à posterior implementação em uma linguagem de programação, ou seja, quando formos programar em uma linguagem. Por isso os algoritmos são independentes das linguagens de programação. Ao contrário de uma linguagem de programação não existe um formalismo rígido de como deve ser escrito o algoritmo.

O algoritmo deve ser fácil de interpretar e fácil de codificar. Ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação.

### ***3.3. Regras para construção do Algoritmo***

Para escrever um algoritmo precisamos descrever a sequência de instruções, de maneira simples e objetiva. Para isso utilizaremos algumas técnicas:

- Usar somente um verbo por frase
- Imaginar que você está desenvolvendo um algoritmo para pessoas que não trabalham com informática
- Usar frases curtas e simples
- Ser objetivo
- Procurar usar palavras que não tenham sentido dúbios.

### ***3.4. Estrutura um Algoritmo***

**Algoritmo** Nome\_Do\_Algoritmo

**variáveis**

Declaração das variáveis

**Procedimentos**

Declaração dos procedimentos

**Funções**

Declaração das funções

**Início**

Corpo do Algoritmo

**Fim**

### ***3.6. Exemplo de algoritmo***

*Exemplo de Algoritmo ó Trocar Pneu furado*

**Algoritmo** Troca de um pneu furado

**Troca\_Pneu\_Furado**

**Início**

Se (o estepe estiver vazio)

Chamar o borracheiro;

**Senão**

Afrouxar todos os parafusos da roda;

Levantar o carro com o macaco;  
Retirar todos os parafusos da roda;  
Retirar o pneu furado;  
Colocar o estepe;  
Recolocar e apertar os parafusos;  
Baixar o carro com macaco;  
Apertar fortemente os parafusos;

**Fim se**

**Fim**

### Exemplo de Algoritmo ó Calcular Média

Imagine o seguinte problema: Calcular a média final dos alunos curso técnico. Os alunos realizarão duas provas: T(Teórica) e P(Prática).

Onde: Média Final =  $(T+P) / 2$

Para montar o algoritmo proposto, faremos três perguntas:

**a) Quais são os dados de entrada?**

R: Os dados de entrada são T e P

**b) Qual será o processamento a ser utilizado?**

R: O procedimento será somar todos os dados de entrada e dividi-los por 2 (dois)

**c) Quais serão os dados de saída?**

R: O dado de saída será a média final

#### Algoritmo Calcular Média

Passo1: Receba a nota da prova teórica(T)

Passo2: Receba a nota de prova prática(P)

Passo3: Some todas as notas e divida o resultado por 2(dois)

Passo4: Mostre o resultado da divisão

## **3.7. Propriedades de Bons Algoritmos**

- **Propriedade:** o tempo de execução deve ser finito para qualquer entrada ou permitir interrupção por determinadas entradas.
- **Propriedade:** os comandos do algoritmo devem ser precisos.
- **Propriedade:** o algoritmo deve ter pelo menos uma saída de resultados, mas pode ter zero ou mais entradas de dados.
- **Propriedade:** os comandos devem ser executáveis.
- **Propriedade:** o algoritmo deve ser suficientemente detalhado (o tipo de detalhamento depende da linguagem escolhida)
- **Propriedade:** o algoritmo deve ser bem estruturado, legível e de fácil correção.

Existem metodologias para se tentar garantir essa propriedade:

- Programação top down
- Programação orientada a objetos
- Programação modular
- Programação estruturada

### ***3.7. Programação Top Down***

Esta é a metodologia de refinamento passo a passo, de forma hierárquica, do início para o final do algoritmo.

Inicia-se com comandos de alto nível (ideia geral) e parte-se para sucessivos detalhamentos dos passos.

Podemos citar as seguintes características deste modelo de resolução de problemas:

- Um problema pode ser decomposto em subproblemas menores.
- Cada subproblema também pode ser decomposto da mesma maneira.
- Os subproblemas são subdivididos até que se obtenha uma coletânea de problemas triviais.
- O algoritmo do problema inicial é uma coletânea de comandos, um comando para resolver cada um desses problemas triviais.

Exemplo:

**Algoritmo** *Média entre dois números*

**Início**

LerNúmeros(A,B)

C=Soma (A, B)/2;

Imprimir(C)  
**Fim**

## ***4. Diagramas de Bloco***

---

### ***4.1. O que é um diagrama de bloco?***


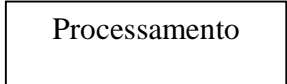
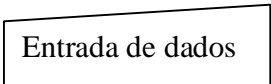
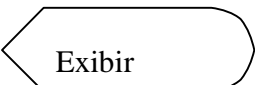


O diagrama de blocos é uma forma padronizada e eficaz para representar os passos lógicos de um determinado processamento.

Com o diagrama podemos definir uma seqüência de símbolos, com significado bem definido, portanto, sua principal função é a de facilitar a visualização dos passos de um processamento.

### ***4.2. Simbologia***

Existem diversos símbolos em um diagrama de bloco. No decorrer do curso apresentaremos os mais utilizados.

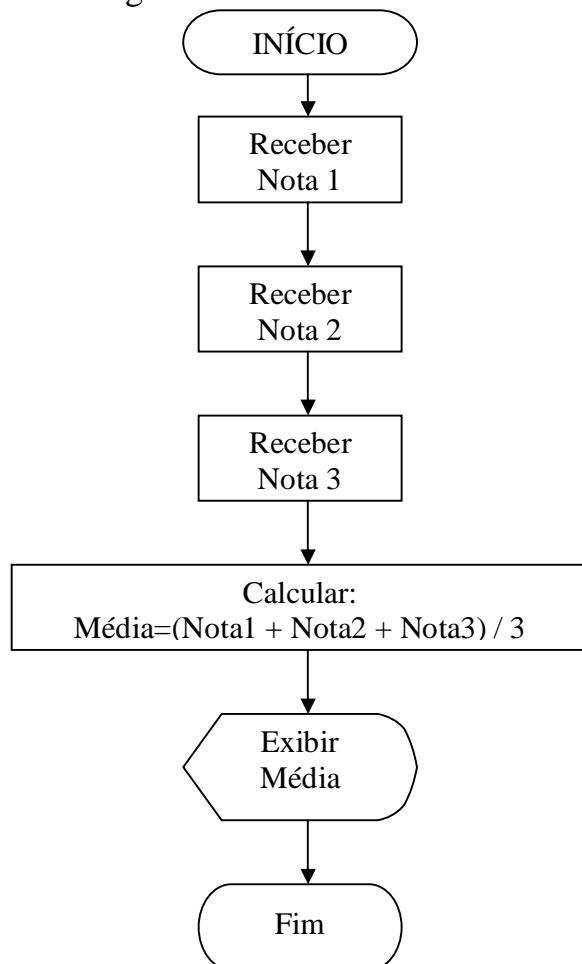
Veja no quadro abaixo alguns dos símbolos que iremos utilizar:

	Indica o INICIO ou FIM de um processamento Exemplo: Início de um Algoritmo
	Alteração de valores, processamentos em geral Exemplo: Cálculo da média de 2 números
	Indica a entrada de dados através do teclado Exemplo: Digite a nota da prova 1
	Mostra informações ou resultados Exemplo: Mostre o resultado do cálculo
	Avalia uma expressão e modifica o rumo do algoritmo. Exemplo: Se valor=1...
	Estabelece uma conexão após um desvio. Exemplo: após um desvio do tipo (Se)

**Ilustração 1 - Símbolos de Fluxogramas**

Dentro do símbolo sempre terá algo escrito, pois somente os símbolos não nos dizem nada.

Observe o exemplo de fluxograma abaixo:



**Fluxograma 1- Exemplo de Fluxograma**



## ***5. Constantes, Variáveis e Tipos de Dados***

---

Variáveis e constantes são os elementos básicos que um programa manipula. Uma variável é um espaço reservado na memória do computador para armazenar um tipo de dado determinado.

Variáveis devem receber nomes para poderem ser referenciadas e modificadas quando necessário. Um programa deve conter declarações que especificam de que tipo são as variáveis que ele utilizará e às vezes um valor inicial. Tipos podem ser, por exemplo: inteiros, reais, caracteres, etc. As expressões combinam variáveis e constantes para calcular novos valores.

### ***5.1. Identificadores***

Representam os nomes escolhidos para rotular as variáveis, procedimentos e funções, normalmente, obedecem as seguintes regras:

- O primeiro caractere deve ser uma letra
- Os nomes devem ser formados por caracteres pertencentes ao seguinte conjunto: {a,b,c,...z,A,B,C,...Z,0,1,2,...,9,\_}
- Os nomes escolhidos devem explicitar seu conteúdo.

### ***5.2. Constantes***

Constante é um determinado valor fixo que não se modifica ao longo do tempo, durante a execução de um programa. Conforme o seu tipo, a constante é classificada como sendo numérica, lógica e literal.

No exemplo da média das provas: Média =  $(P+T) / 2$ , o número (2) é uma constante que nunca varia.

### ***5.3. Variáveis***

Variável é a representação simbólica dos elementos de um certo conjunto. Cada variável corresponde a uma posição de memória, cujo conteúdo pode se alterado ao longo do tempo durante a execução de um programa. Embora uma variável possa assumir diferentes valores, ela só pode armazenar um valor a cada instante.

Exemplos de variáveis: Média; T e P.

### *Tipos de variáveis primitivas*

As variáveis e as constantes podem ser basicamente de três tipos: Numéricas, Alfanuméricas ou Lógicas e Vetores.

**Variáveis Numéricas:** Específicas para armazenamento de números, que posteriormente poderão ser utilizados para cálculos. Podem ser ainda classificadas basicamente como Inteiras ou Reais. As variáveis do tipo inteiro são para armazenamento de números inteiros e as Reais são para o armazenamento de números que possuam casas decimais.

**Variáveis Alfanuméricas:** Específicas para dados que contenham letras e/ou números. Pode em determinados momentos conter somente dados numéricos ou somente literais. Se usado somente para armazenamento de números, não poderá ser utilizada para operações matemáticas.

**Variáveis Lógicas:** Armazenam somente dados lógicos que podem ser Verdadeiro ou Falso.

### *Tipos de estruturas de variáveis*

Estruturas são formadas por um conjunto de variáveis permitindo modelar de forma mais natural os dados.

As estruturas mais comuns são:

**Vetores:** Estrutura formada por um conjunto unidimensional de dados de mesmo tipo (homogêneo) e possuindo número fixo de elementos (Estático). Na declaração dos vetores devemos informar o seu nome, seu tipo (inteiro, real, caractere,...), e seu tamanho (número de elementos). Cada elemento do vetor é identificado por um índice (unidimensional), o qual indica a sua posição no vetor.

**Matriz:** estrutura semelhante ao vetor, sendo que, pode possuir **duas** dimensões. Desta forma para fazer referência aos elementos de uma matriz precisaremos de tantos índices quanto for suas dimensões.

**Registro:** estrutura formada por um conjunto de variáveis, que podem possuir tipos diferentes (Heterogêneo), agrupadas em uma só unidade.

**Obs:** Podemos ainda definir um vetor formado por registros.

## ***6. Operadores***

---

Os operadores são meios pelo qual incrementamos, decrementamos, comparamos e avaliamos dados dentro do computador. Temos os seguintes tipos de operadores:

- Operadores de Atribuição
- Operadores Aritméticos
- Operadores Relacionais
- Operadores Lógicos

### ***6.1. Operadores de Atribuição***

Os operadores de atribuição são utilizados para atribuir um valor a uma variável, função ou constante no seu algoritmo.

Sua representação é uma seta para esquerda, porém cada linguagem de programação tem seu atribuidor.

Exemplo:

nome ← "Einstein"

valor = 12,3

### ***6.2. Operadores Aritméticos***

Os operadores aritméticos são os utilizados para obter resultados numéricos. Além da adição, subtração, multiplicação e divisão, podem utilizar também o operador para exponenciação e resto.

Os símbolos para os operadores aritméticos são:

<b>Operação</b>	<b>Símbolo</b>
Adição	+
Subtração	-
Multiplicação	*
Divisão	/

Exponenciação	** ou ^
Resto	%

Tabela 1 - Operadores Aritméticos

### 6.3. Operadores Relacionais

Os operadores relacionais são utilizados para comparar *String* de caracteres e números. Os valores a serem comparados podem ser caracteres ou variáveis. Estes operadores sempre retornam valores lógicos (verdadeiro ou falso/ True ou False). Para estabelecer prioridades, no que diz respeito a qual operação executar primeiro, utilize os parênteses. Os operadores relacionais são:

Descrição	Símbolo
Igual a	=
Menor que	<
Diferente de	<> ou !=
Maior ou igual a	>=
Maior que	>
Menor ou igual a	<=

Tabela 2 - Operadores Relacionais

Os resultados das expressões seriam:

Expressão	Resultado
A = B	Falso
A <> B	Verdadeiro
A > B	Verdadeiro
A < B	Falso
A >= B	Verdadeiro
A <= B	Falso

Tabela 3 -Resultado de Operações Relacionais

### 6.4. Operadores Lógicos

Os operadores lógicos servem para combinar resultados de expressões, retornando se o resultado final é verdadeiro ou falso.

Os operadores lógicos são:

E	AND
OU	OR

NÃO	NOT
-----	-----

**Tabela 4 - Tipo de Operadores Lógicos**

- **E / AND** - Uma expressão AND (E) é verdadeira se todas as condições forem verdadeiras
- **OR/OU** - Uma expressão OR (OU) é verdadeira se pelo menos uma condição for verdadeira
- **NOT** - Uma expressão NOT (NÃO) inverte o valor da expressão ou condição, se verdadeira inverte para falsa e vice-versa.

### 6.5. Prioridade de Operadores:

Durante a execução de uma expressão que envolve vários operadores, é necessário à existência de prioridades, caso contrário poderemos obter valores que não representam o resultado esperado.

A maioria das linguagens de programação utiliza as seguintes prioridades de operadores:

1º - Efetuar operações embutidas em parênteses "mais internos"

2º - Efetuar Funções

3º - Efetuar multiplicação e/ou divisão

4º - Efetuar adição e/ou subtração

5º - Operadores Relacionais

6º - Operadores Lógicos

**OBS:** O programador tem plena liberdade para incluir novas variáveis, operadores ou funções para adaptar o algoritmo as suas necessidades, lembrando sempre, de que, estes devem ser compatíveis com a linguagem de programação a ser utilizada.

Exemplo da prioridade dos operadores utilizado na Linguagem C++:

Ordem	Classe	Operadores
1	Unários	+ - ! ( ) ++ --
2	Multiplicativos	* / %
3	Aditivos	+ -
4	Relacionais	< <= > >=
5	De Igualdade	== !=
6	Lógico Multiplicativo	&&
7	Lógico Aditivo	
8	De Atribuição	= += -= *= /= %=

**Tabela 5 - Ordem de precedência de operadores**

## 6.6. OPERADORES LÓGICAS

Operações Lógicas são utilizadas quando se torna necessário tomar decisões em um diagrama de bloco.

Num diagrama de bloco, toda decisão terá sempre como resposta o resultado VERDADEIRO ou FALSO.

Imaginemos que algumas pessoas vão ao KUKUKAYA, caso seja universitária, entrará de graça, veja o exemplo do algoritmo ãENTRAR NO KUKUKAYAö:

**Se (é universitária)**

Entra sem pagar

**senão**

Paga R\$ 5,00

**fim se**

Neste caso a operação lógica avaliada é (é universitária) e quando esta for verdadeira será executado: ãEntra sem pagarö. Se esta operação for falsa será executado ãPaga R\$ 5,00ö.

## 7. Estruturas de Decisão

Como vimos anteriormente em *Operações Lógicas*, verificamos que na maioria das vezes necessitamos tomar decisões no andamento do algoritmo. Essas decisões interferem diretamente no andamento do programa.

Os comandos de decisão ou desvio fazem parte das técnicas de programação que conduzem a estruturas de programas que não são totalmente sequenciais. Com as instruções de SALTO ou DESVIO pode-se fazer com que o programa proceda de uma ou outra maneira, de acordo com as decisões lógicas tomadas em função dos dados ou resultados anteriores. As principais estruturas de decisão são: *Se Então*, *Se então Senão* e *Caso Selecione*.

### 7.1. SE ENTÃO / IF... THEN

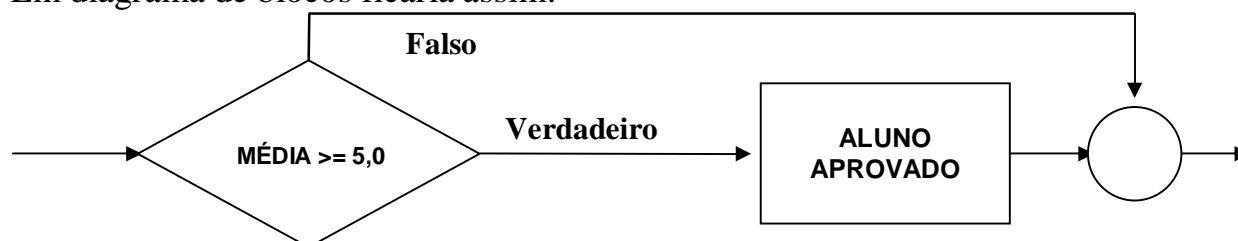
A estrutura de decisão *SE/IF* normalmente vem acompanhada de um comando, ou seja, se determinada condição for satisfeita pelo comando SE/IF então execute determinado comando.

Imagine um algoritmo que determinado aluno somente estará aprovado se sua média for maior ou igual a 5,0, veja no exemplo de algoritmo como ficaria.

**SE MÉDIA  $\geq$  5,0 ENTÃO**

**ALUNO APROVADO**

Em diagrama de blocos ficaria assim:



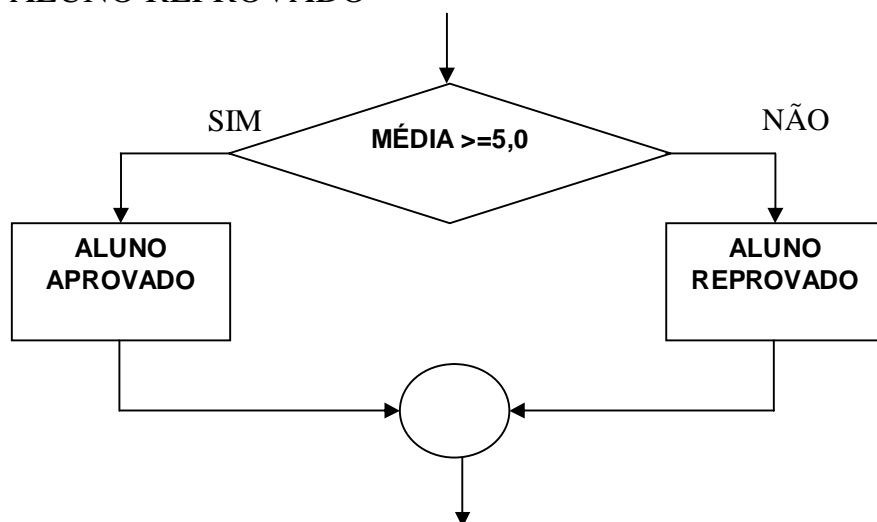
Fluxograma 2 - SE ENTÃO / IF ... THEN



## 7.2. SE ENTÃO SENÃO / IF ... THEN ... ELSE

A estrutura de decisão SE/ENTÃO/SENÃO, funciona exatamente como a estrutura SE, com apenas uma diferença, em SE somente podemos executar comandos caso a condição seja verdadeira, diferente de SE/ENTÃO pois sempre um comando será executado independente da condição, ou seja, caso a condição seja verdadeira o comando da condição será executado, caso contrário o comando da condição falsa será executado. Em algoritmo ficaria assim:

**SE MÉDIA  $\geq$  5,0 ENTÃO**  
**ALUNO APROVADO**  
**SENÃO**  
**ALUNO REPROVADO**



**Fluxograma 3 - SE ENTÃO SENÃO / IF ... THEN ... ELSE**

No exemplo acima está sendo executada uma condição que, se for verdadeira, executa o comando APROVADO, caso contrário executa o segundo comando REPROVADO.

Existe ainda uma forma mais complexa desta estrutura condicional onde podemos adicionar mais de uma condição de teste:

**Se ( Expressão 1)**  
**Comandos\_1**

**senão** (expressão 2)

Comandos\_2

**senão** (expressão 3)

Comandos\_3

**senão**

Comandos\_4

**Fim Se**

**Obs:** Apesar de ser possível acrescentar expressões condicionais intermediárias, a única que realmente é obrigatória é a expressão nº 1, a primeira.

No exemplo acima, os testes serão executados de cima para baixo até que uma das expressões seja verdadeira, quando será executado o comando interno da expressão verdadeira, e, após a conclusão destes comandos, o algoritmo retomará seu curso após o final da estrutura.

### **7.3. CASO SELECIONE / SELECT ... CASE**

A estrutura de decisão CASO/SELECIONE é utilizada para testar, na condição, uma única expressão, que produz um resultado, ou, então, o valor de uma variável, em que está armazenado um determinado conteúdo. Compara-se, então, o resultado obtido no teste com os valores fornecidos em cada cláusula *õCasoö*.

No exemplo do diagrama de blocos abaixo, é recebida uma variável *õOpö* e testado seu conteúdo, caso uma das condições seja satisfeita, é atribuído para a variável *õNOMEö* um valor de uma String, caso contrário é atribuído à string *õOpção Erradaö*.

Em algoritmo demonstraríamos assim:

#### **Algoritmo Caso**

Ler OP

**Caso** *OP*

Valor 1: Nome=*õPAULOö*

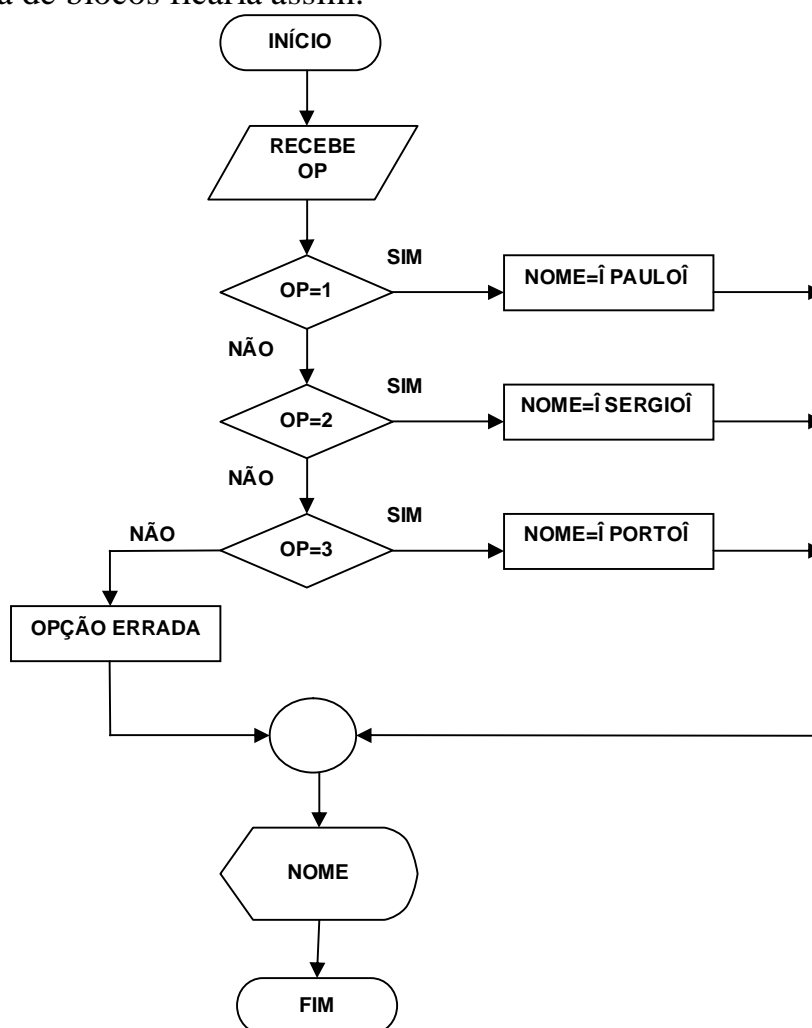
Valor 2: Nome=*õSERGIOö*

Valor 3: Nome=*õPORTOö*

Em outros casos: Nome=*õOPÇÃO ERRADAö*

**Fim caso**  
Mostre NOME  
**Fim**

Em diagrama de blocos ficaria assim:



**Fluxograma 4 - CASO SELECIONE / SELECT ... CASE**

## ***8. Estruturas de Repetição***

---

Utilizamos os comandos de repetição quando desejamos que: um determinado conjunto de instruções ou conjunto de comandos seja executado um número definido ou indefinido de vezes, ou enquanto um determinado estado de coisas prevalecer ou até que seja alcançado.

Trabalharemos com os seguintes modelos de comandos de repetição:

- Repetição determinada
- Repetição com pós-teste
- Repetição com pré-teste

### ***8.1. Repetição determinada ó ãParaö***

Quando uma seqüência de comandos deve ser executada repetidas vezes, tem-se uma estrutura de repetição.

A estrutura de repetição, assim como a de decisão, envolve sempre a avaliação de uma condição.

Na repetição determinada, o algoritmo apresenta previamente a quantidade de repetições a serem executadas.

A repetição por padrão determina o passo do valor inicial até o valor final como sendo 1. Determinadas linguagens possuem formas de alterar o valor do passo ou incremento e ainda até definir passo em ãólö isto é decremento.

Usaremos a estrutura ãParaö demonstrando essa repetição.

Algoritmo de uma repetição determinada:

k=0

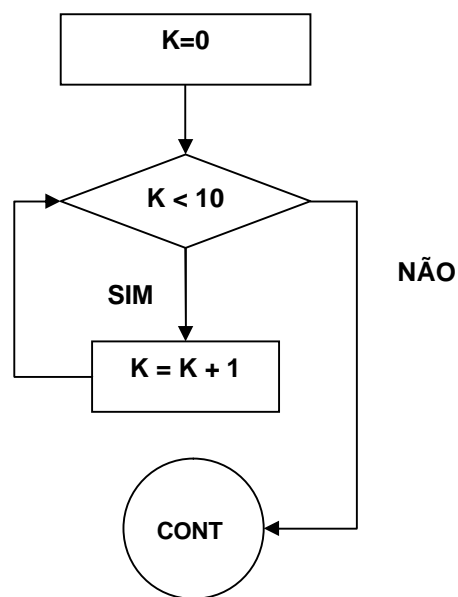
**Para** k=1 **até** 10

Ler(nota)

soma=soma+nota

**Fim Para**

Em diagrama de blocos ficaria assim:



Fluxograma 5 - Repetição determinada ó ãParaö

## 8.2. Repetição com pré-teste ó ãFaça enquantoö

Neste caso, o bloco de operações será executado enquanto a condição (x) for verdadeira. O teste da condição será sempre realizado antes de qualquer operação.

Enquanto a condição for verdadeira o processo se repete. Podemos utilizar essa estrutura para trabalharmos variáveis que tem seu valor modificado dentro da estrutura de repetição.

Teríamos o seguinte algoritmo:

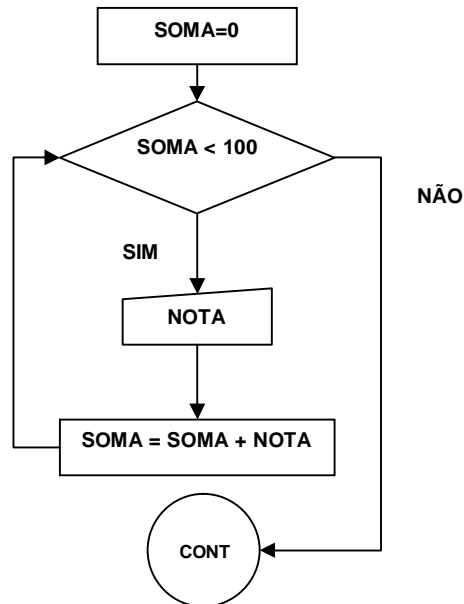
soma = 0;

**Enquanto** (soma < 100)

```

Ler (nota)
soma = soma + nota;
fim enquanto

```



Fluxograma 6 - Repetição com pré-teste ó ãFaça enquantoö

### 8.3. Repetição com pós-teste ó ãRepitaö

Neste caso primeiro são executados os comandos, e somente depois é realizado o teste da condição. Se a condição for verdadeira, os comandos são executados novamente, caso seja falso, o algoritmo encerra o comando ãRepitaö.

Esta estrutura é muito utilizada quando se tem a obrigatoriedade de se executar pelo menos uma vez um determinado bloco de instruções e somente continuar no öloopingö se validar uma pós-condição.

Teríamos o seguinte algoritmo:

Exemplo: // Calcular o fatorial  $n! = n*(n-1)*(n-2)... 5*4*3*2*1$

```
Fat←1;
```

```
Ler (n)
```

```
Repita
```

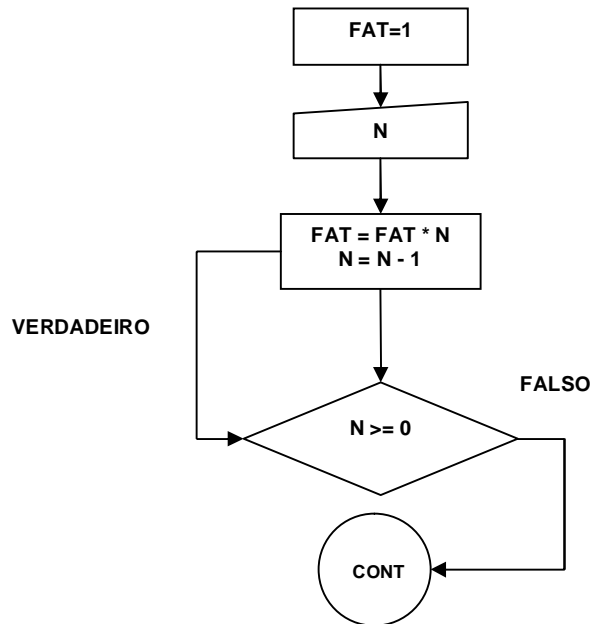
```
    Fat←Fat*n;
```



$n \leftarrow n-1$   
**Até**  $n \geq 0$



E o seguinte diagrama de blocos:



**Fluxograma 7 - Repetição com pós-teste**

## ***9. Chamada de ALGORITMOS***

---

Não é necessário copiar o detalhamento do algoritmo na estrutura principal, evitando elaborar algoritmos longos, difícil de entender por ser cheio de detalhes.

Para se repetir um algoritmo basta utilizar a seguinte notação:  
nome\_do\_algoritmo[(parâmetros)]

Exemplo:

$C \leftarrow \text{Soma}(A, B);$

A, B são valores para serem somados.

Os algoritmos devem ser vistos como sendo funções tal como a função matemática  $\text{seno}(x)$ ,  $\text{cosseno}(x)$ . Deve-se abstrair e estender o conceito de função para procedimentos ou processos que executam tarefas.

## ***10. Modularização***

---

A modularização consiste num método para facilitar a construção de grandes programas, através de sua divisão em pequenas etapas, que são: módulos, procedimentos, funções, rotinas, sub-rotinas ou sub-programas.

Esta modularização permite o reaproveitamento de códigos, já que podemos utilizar um módulo quantas vezes for necessário, eliminando assim a necessidade de escrever o mesmo código em situações repetitivas.

### ***10.1. Procedimentos***

Um procedimento é um bloco de código precedido de um cabeçalho que contém o Nome do procedimento e seus parâmetros. Com isto, podemos fazer referência ao bloco de código de qualquer ponto do algoritmo através do seu *nome* e passando os seus *parâmetros*.

#### **Declaração:**

**Procedimento** NomeDoProcedimento [(*parâmetros*)]

#### **Variáveis**

#### **Início**

Comandos;

#### **Fim;**

Onde, *parâmetros* representam as variáveis que devem ser passadas ao procedimento. Os parâmetros podem ser de: ENTRADA (passado por valor) ou de ENTRADA/SAÍDA (passado por referência). Os parâmetros de ENTRADA não podem ser alterados pelo procedimento, para que isso seja possível o parâmetro deve ser de ENTRADA/SAÍDA Para indicar que um parâmetro é de ENTRADA/SAÍDA devemos colocar a palavra *VAR* antes da sua declaração.

#### **Referência:**

NomeDoProcedimento(*variáveis*)

**OBS:** As variáveis passadas aos procedimentos são associadas aos parâmetros do procedimento de acordo com a ordem das variáveis e da lista de parâmetros.

Exemplo:

```
soma(3,4)
a = divide(6,3)
atualiza_a(&a, &b)
```

## ***10.2. Funções***

Uma função é semelhante a um procedimento, sendo que esta deve retornar, obrigatoriamente, um valor em seu nome, desta forma, é necessário declarar, no cabeçalho da função, qual o seu tipo.

**Declaração:**

Tipo\_da\_função **Função** NomeDaFunção [(parâmetros)]

**Variáveis**

**Início**

Comandos

**NomeDaFunção** => (expressão de retorno)

**Fim;**

**Referência:**

valor = NomeDaFunção(parâmetro)

**Exemplo:**

nome = AchaCliente(1);

## ***11. Conceitos de Programação Orientada a Objetos***

---

O termo orientação a objetos significa: organizar o mundo real como uma coleção de objetos que incorporam *estrutura de dados e comportamentos*.

### ***11.1. Por Que Orientação a Objetos?***

A construção de sistemas de software é um processo intrinsecamente complexo e tem se tornado ainda mais complexo devido aos novos requisitos impostos às aplicações modernas: alta confiabilidade, alto desempenho, desenvolvimento rápido e barato, e tamanho e complexidade grandes.

É possível, e muitas vezes até desejável, utilizar métodos e técnicas que nos permitem ignorar ou reduzir a complexidade de tal processo em algumas de suas etapas, mas isso não faz com que a complexidade desapareça, pois ela certamente aparecerá em uma outra etapa.

Os desenvolvedores de software já reconheceram há bastante tempo que a chave para o sucesso no desenvolvimento de softwares está no controle da sua complexidade.

O surgimento de linguagens de programação de alto nível e os seus compiladores é um exemplo de um esforço antigo que propiciou uma redução da complexidade de desenvolvimento de softwares, uma vez que ninguém mais necessitou programar em linguagem de máquina, que consiste em uma tarefa bem menos produtiva do que aquela de programar em uma linguagem de alto nível.

Os propósitos da Programação Orientada a Objetos são:

- Prover mecanismos para visualizar a complexidade do desenvolvimento de software da mesma forma que visualizamos a complexidade do mundo ao nosso redor;
- Acelerar o desenvolvimento de softwares com base na modularidade e acoplamento;
- Melhorar a qualidade do software desenvolvido.

## 11.2. Classes

Uma classe é um tipo definido pelo usuário que contém o molde, a especificação para os objetos, algo mais ou menos como o tipo inteiro contém o molde para as variáveis declaradas como inteiros. A classe envolve, associa, funções e dados, controlando o acesso a estes, defini-la implica em especificar os seus atributos (dados) e seus métodos (funções).

Um programa que utiliza uma interface controladora de um motor elétrico provavelmente definiria a **classe** motor. Os **atributos** desta classe seriam: temperatura, velocidade, tensão aplicada. Estes provavelmente seriam representados na classe por tipos como int ou float. Os **métodos** desta classe seriam funções para alterar a velocidade, ler a temperatura, etc.

### Representação gráfica

Nome	<b>MOTOR</b>
Atributos	<b>TEMPERATURA VELOCIDADE TENSÃO APLICADA</b>
Métodos	<b>ALTERAR VELOCIDADE LER A TEMPERATURA</b>

Tabela 6 - Representação gráfica de uma Classe

### Visão do mundo real

As classes são os moldes para criação de objetos estas especificam propriedades e ações em comum a todos seus objetos.

Por exemplo:

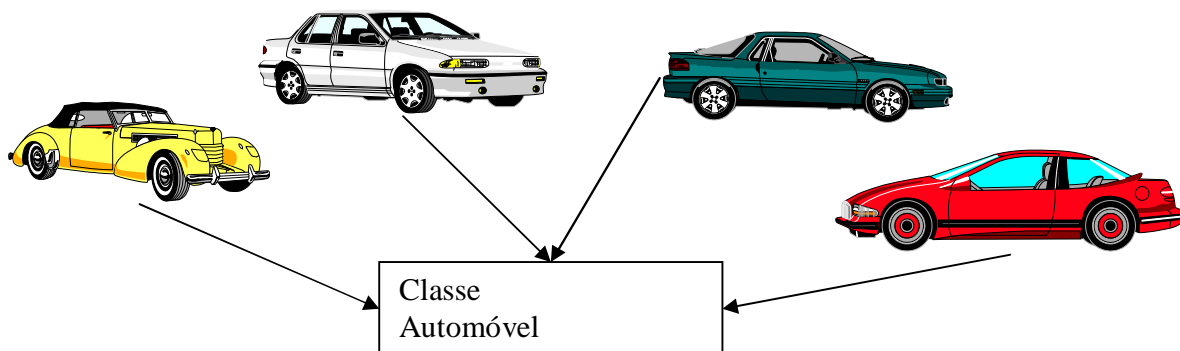


Ilustração 2 - Visão do mundo real

Exemplo de protótipo:

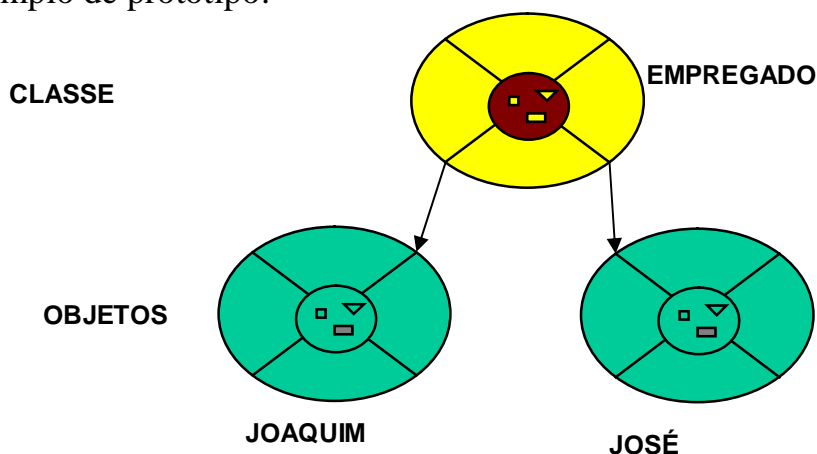


Ilustração 3 - Visão de Classes e Objetos

### 11.3. Objetos

Um objeto é uma representação abstrata de uma entidade do mundo real, que tem um identificador único, propriedades embutidas e a habilidade de interagir com outros objetos e consigo mesmo.

O estado do objeto é um conjunto de valores que os atributos do objeto podem ter em um determinado instante do tempo.

**OBJETO** é uma particular instância de uma classe.

*Fábrica de torradeiras não é uma torradeira, mas define o tipo de produto que sai dela.* As classes não ocupam espaço na memória, por serem abstrações, enquanto que, os objetos ocupam espaço de memória por serem concretizações dessas abstrações.



## Métodos

Os métodos definem as ações a serem tomadas e realizam as tarefas para as quais o programa foi escrito.

Por exemplo: produzir relatórios, realizar cálculos, criar gráficos, etc.

### 11.4. Criando Classes e Objetos

Considere o exemplo abaixo:

Classe:

```
public class Automovel
{
    String modelo,cor;
    int ano;
    boolean estadoMotor = false;
    public void ligaMotor()
    {
        estadoMotor = true;
    }
}
```

Objeto:

```
Automovel gol;
gol = new Automovel();
```

O primeiro comando(`Automovel gol;`), a declaração da classe, aloca apenas o espaço suficiente para a referência. O segundo comando (`gol = new Automovel();`) aloca o espaço para os atributos do objeto `gol`.

Somente após a criação do objeto é que seus membros (atributos e métodos) podem ser referenciados.

### 11.5. Atributos

Os atributos são as propriedades associadas a uma classe e seus objetos; Eles armazenam resultados do processamento feito pelos métodos da classe.

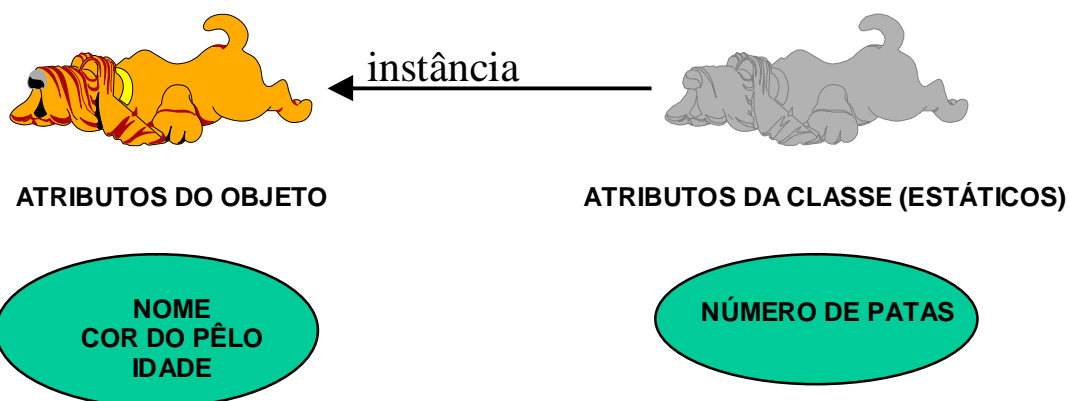


Ilustração 4 - Visão de atributos de classes

## Métodos

Os métodos são operações que manipulam o estado do objeto. Podem fazer parte da interface do objeto ou realizar uma função interna. Eles efetivamente realizam as tarefas de manipular os atributos (dados) do objeto.

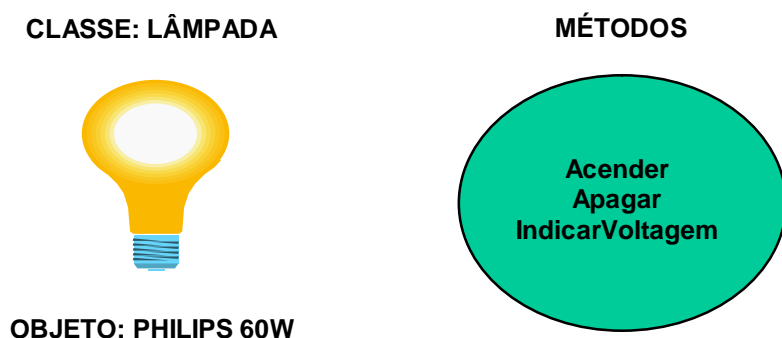


Ilustração 5 - Visão de métodos de classes

## 11.6. Referência a Membros de Objetos e Classes

Podemos fazer referências a atributos e aos métodos de um objeto. Para referenciarmos um atributo usamos:

Membros	Atributo	Método
Referência	<referência.variável>	<resultado = referência.método(parâmetros)>
Exemplo	fusca.ano = 69;	fusca.ligaMotor();

Tabela 7 - Referência a Membros de Objetos e Classes

## ***12. ALGORÍTMOS DE ORDENAÇÃO***

---

Os problemas de ordenação são comuns tanto em aplicações comerciais quanto científicas. Entretanto, raro são os problemas que se resumem à pura ordenação de seqüências de elementos. Normalmente, os problemas de ordenação são inseridos em problemas de pesquisa, intercalação e atualização. Isto torna ainda mais importante o projeto e a construção de algoritmos eficientes e confiáveis para tratar o problema.

O nosso objetivo é analisar os seguintes tipos de ordenação:

- Selection Sort
- Bubble Sort
- Insertion Sort

### ***12.1. Selection Sort***

Este método é um dos mais simples e intuitivos dentre os métodos existentes. Sua estratégia básica é selecionar o menor elemento da seqüência considerada e colocá-lo no início da seqüência. Assim, dada uma seqüência de tamanho  $n$ ,

várias iterações são efetuadas, sendo que a cada vez que esta estratégia é aplicada, uma nova sequência é gerada pela eliminação do menor elemento da sequência original.

Procedure SelectionSort ( var vet : vetor; n : integer);

```
{ordenado crescente}
var
i, j, pmin : integer;
begin
for i ← 1 to (n-1) do
begin
pmin ← i;
for j ← (i+1) to n do
if vet[j] < vet[pmin]
then pmin ← j;
trocar (vet[i], vet[pmin]) ;
end;
end;
```

## 12.2. Bubble Sort

A estratégia utilizada pelo BubbleSort consiste de comparações e trocas entre elementos consecutivos da sequência, a fim de "empurrar" o maior elemento para a última posição. Assim, várias iterações são efetuadas e, para cada sequência considerada, a aplicação da estratégia gera uma nova sequência pela eliminação do maior elemento da sequência original.

Além disto, uma variável de controle (lógica) é utilizada para registrar a ocorrência ou não de troca entre elementos da sequência. Quando nenhuma troca é efetuada, tem-se que a sequência considerada já estava ordenada. Esta particularidade determina, em alguns casos, um número menor de comparações que o método SelectionSort.

Procedure BubbleSort ( var vet : vetor ; n integer) ;

```
{ordem crescente}
var
i, limite : integer;
trocou : boolean;
begin
limite ← n;
repeat
trocou ← false;
for i ← 1 to (limite - 1) do
```

```
begin
if vet[i] > vet [i+1] then
begin
trocar(vet[i], vet[i+1]);
trocou ← true;
end;
end;
limite ← limite - 1
until not trocou
end;
```

### ***12.3. Insertion Sort***

Este método baseia-se no seguinte processo de inserção controlada:

Com o primeiro elemento da sequência forma-se uma sequência de tamanho 1, ordenada. Cada elemento restante da sequência original é inserido na sequência, de modo que esta permaneça ordenada. Isto é feito através de uma pesquisa na sequência ordenada que determina a posição que o novo elemento deverá ser inserido. Quando um elemento é inserido a frente de outro, estes deverão ser deslocados de uma posição.

### ***12.4. RECURSIVIDADE***

Recursão é um método geral para resolver problemas reduzindo-os a problemas mais simples do mesmo tipo. A estrutura geral de uma solução recursiva de um problema é assim :

Resolva de forma recursiva um problema.

- Se o problema é trivial, faça o obvio (resolva-o)
- Simplifique o problema
- Resolva de forma recursiva (um problema mais simples)
- Combine (na medida do possível) a solução do(os) problemas mais simples em uma solução do problema original

Um subprograma recursivo chama a si próprio constantemente, cada vez em uma situação mais simples, até chegar ao caso trivial, quando pára. Devemos lembrar que recursividade deve ser utilizada na solução de problemas que tenham a natureza recursiva.

### ***Exercícios:***

- 1) Implementar um algoritmo para resolver uma equação do segundo grau.

- 2) Implementar um algoritmo capaz de encontrar o maior dentre três números inteiros quaisquer. Suponha todos serem distintos.
- 3) Implementar um algoritmo capaz de dizer o tipo de triângulo dado seus lados.
- 4) Implementar um algoritmo que leia três números quaisquer e os imprima em ordem crescente.
- 5) Implementar um algoritmo que leia um conjunto de 50 elementos inteiros e os imprima em ordem contrária da que foi lida (DICA: use um vetor).
- 6) Dado um vetor com N elementos numéricos reais positivos obter a maior diferença entre dois elementos consecutivos neste vetor.
- 7) Implemente um algoritmo que escreva a série de N-ésimo termo da série de Finonacci.
- 8) Escreva um algoritmo para converter de graus CELSIUS para FAHRENHEIT.

## *Exercícios resolvidos*

- 1) Elabore um algoritmo para determinar imprimir em ordem crescente 3 números.

```
{  
    a,b,c,m1,m2,m3: Inteiros;  
    ler (a,b,c);  
    se (a >=b) e (a>=c)  
        m1=a  
        se (b>=c)  
            m2=b; m3=c  
        senao  
            m2=c; m3=b  
    // a-b-c ou a-c-b
```

```

se (b >=a) e (b<=c)
    m1=b
    se (a>=c)
        m2=a; m3=c
    senao
        m2=c; m3=a
// b-a-c ou b-c-a
se (c >=a) e (c<=b)
    m1=c
    se (a>=b)
        m2=a; m3=b
    senao
        m2=b; m3=a
// c-a-b ou c-b-a
imprima (m1, m2, m3)
}

```

2) Elabore um algoritmo para ler 3 lados de um triângulo e classificá-lo quanto aos lados (equilátero, isósceles e escaleno) e quanto ao ângulo( reto, obtuso e agudo).

```

{
    a,b,c: numeros;
    ler (a,c,b);
    // primeira parte
    se (a==b) e (a=c)
        mostre("equilatero")
    senao
        se (a==b) ou (b==c) ou (c==a)
            mostre("isoceles")
        senao
            mostre("escaleno")

    // segunda parte
    se (a ^ 2) == (b ^ 2) + (b ^ 2)
        mostre("angulo reto")
    senao
        se (a ^ 2) < (b ^ 2) + (b ^ 2)
            mostre("angulo agudo")
        senao
            mostre("angulo obtuso")
}

```

3) Elabore um algoritmo para fazer a média de um conjunto de valores.

```

{
    qtd, soma, valor: inteiro;
    ler (qtd);
    para i = 0 até qtd
        ler (valor);
        soma = soma + valor
    fim para
}

```

```

se (qtd <> 0)
    mostra ( "Média= " , soma/qtd)
senao
    mostra("0")
fim se
}

```

4) Elabore um algoritmo para calcular a nota final (média) de uma disciplina. As entradas são:

- nota de prova (30%);
- notas de 4 labs(5% cada um);
- exercícios(10% );
- exame final (40%).

Além da média, deve-se imprimir as entradas e se o aluno foi aprovado ou não.

```

calcula_media();
{
    qtd, np, lab1,lab2,lab3,lab4, ex, ef, media : numeros;
    ler(qtd);
    para i = 1 até qtd
        ler (np, lab1,lab2,lab3,lab4, ex, ef);
        se testa 0 < (np,lab1,lab2,lab3,lab4,ex,ef) < 100;
            mostra ("Prova=",np);
            mostra ("Labs=",lab1,lab2,lab3,lab4);
            mostra ("Exercicios=",ex);
            mostra ("Exame=",ef);
        media=(np*3+lab1*0.5+lab2*0.5+lab3*0.5+lab4*0.5+ex+f*4)/10);
        mostra ("Média= " , media);
    fim se
    se media > 7
        mostra("Aluno aprovado")
    senao
        mostra("Aluno reprovado")
    fim se
fim para
}

```

5) Elabore um algoritmo para calcular as notas de todos os alunos de um curso. Deve-se ler e imprimir o nome do aluno.

```

calcula_nota_turma();
{
    nome, curso: caracter;
    n_alunos      : numero;
    ler("Curso:",curso);
    para i = 1 a n_alunos
        ler("Nome:",nome);

```



```
        mostra("Aluno:",nome);  
        calcula_media();  
    fim para  
}
```

## *Linguagem de Programação C*

# ***1. CONCEITOS FUNDAMENTAIS***

---

### *Introdução*

A linguagem C, assim como as linguagens Fortran e Pascal, são ditas linguagens nãoconvencionais, projetadas a partir dos elementos fundamentais da arquitetura de Von Neuman, que serve como base para praticamente todos os computadores em uso. Para programar em uma linguagem convencional, precisamos de alguma maneira especificar as áreas de memória em que os dados com que queremos trabalhar estão armazenados e,

freqüentemente, considerar os endereços de memória em que os dados se situam, o que faz com que o processo de programação envolva detalhes adicionais, que podem ser ignorados quando se programa em uma linguagem como Scheme. Em compensação, temos um maior controle da máquina quando utilizamos uma linguagem convencional, e podemos fazer programas melhores, ou seja, menores e mais rápidos.

A linguagem C provê as construções fundamentais de fluxo de controle necessárias para programas bem estruturados: agrupamentos de comandos; tomadas de decisão (*if-else*); laços com testes de encerramento no início (*while*, *for*) ou no fim (*do-while*); e seleção de um dentre um conjunto de possíveis casos (*switch*). C oferece ainda acesso a apontadores e a habilidade de fazer aritmética com endereços. Por outro lado, a linguagem C não provê operações para manipular diretamente objetos compostos, tais como cadeias de caracteres, nem facilidades de entrada e saída: não há comandos READ e WRITE. Todos esses mecanismos devem ser fornecidos por funções explicitamente chamadas. Embora a falta de algumas dessas facilidades possa parecer uma deficiência grave (deve-se, por exemplo, chamar uma função para comparar duas cadeias de caracteres), a manutenção da linguagem em termos modestos tem trazido benefícios reais. C é uma linguagem relativamente pequena e, no entanto, tornou-se altamente poderosa e eficiente.

## 1.1. Modelo de um Computador

Existem diversos tipos de computadores. Embora não seja nosso objetivo estudar *hardware*, identificamos, nesta seção, os elementos essenciais de um computador. O conhecimento da existência destes elementos nos ajudará a compreender como um programa de computador funciona.

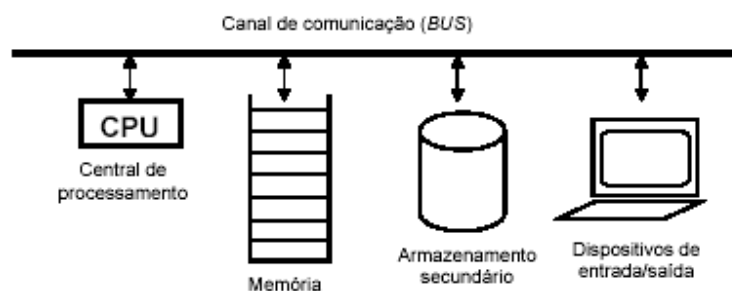


Figura 1.1: Elementos básicos de um computador típico.

O canal de comunicação (conhecido como *BUS*) representa o meio para a transferência de dados entre os diversos componentes. Na memória principal são armazenados os programas e os dados no computador. Ela tem acesso randômico, o que significa que podemos endereçar (isto é, acessar) diretamente qualquer posição da memória. Esta memória não é permanente e, para um programa, os dados são armazenados enquanto o programa está sendo executado. Em geral, após o término do programa, a área ocupada na memória fica

disponível para ser usada por outras aplicações. A área de armazenamento secundário é, em geral, representada por um disco (disco rígido, disquete, etc.). Esta memória secundária tem a vantagem de ser permanente. Os dados armazenados em disco permanecem válidos após o término dos programas. Esta memória tem um custo mais baixo do que a memória principal, porém o acesso aos dados é bem mais lento. Por fim, encontram-se os dispositivos de entrada e saída. Os dispositivos de entrada (por exemplo, teclado, *mouse*) permitem passarmos dados para um programa, enquanto os dispositivos de saída permitem que um programa exporte seus resultados, por exemplo em forma textual ou gráfica usando monitores ou impressoras.

## 1.2. Interpretação versus Compilação

Uma diferença importante entre as linguagens C e Scheme é que, via de regra, elas são implementadas de forma bastante diferente. Normalmente, Scheme é interpretada e C é compilada. Para entender a diferença entre essas duas formas de implementação, é necessário lembrar que os computadores só executam realmente programas em sua linguagem de máquina, que é específica para cada modelo (ou família de modelos) de computador. Ou seja, em qualquer computador, programas em C ou em Scheme não podem ser executados em sua forma original; apenas programas na linguagem de máquina (à qual vamos nos referir como M) podem ser efetivamente executados.

No caso da interpretação de Scheme, um programa interpretador (IM), escrito em M, lê o programa PS escrito em Scheme e simula cada uma de suas instruções, modificando os dados do programa da forma apropriada. No caso da compilação da linguagem C, um programa compilador (CM), escrito em M, lê o programa PC, escrito em C, e traduz cada uma de suas instruções para M, escrevendo um programa PM cujo efeito é o desejado. Como consequência deste processo, PM, por ser um programa escrito em M, pode ser executado em qualquer máquina com a mesma linguagem de máquina M, mesmo que esta máquina não possua um compilador.

Na prática, o programa fonte e o programa objeto são armazenados em arquivos em disco, aos quais nos referimos como arquivo fonte e arquivo objeto. As duas figuras a seguir esquematizam as duas formas básicas de implementação de linguagens de programação.

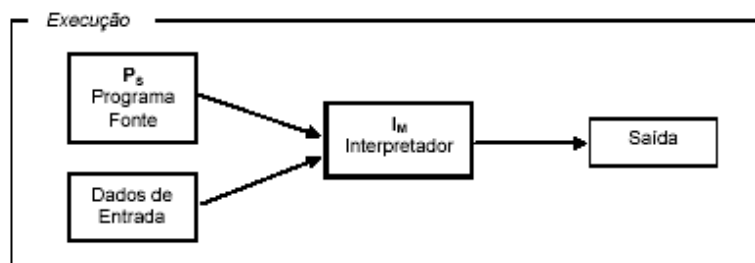


Figura 1.2: Execução de programas com linguagem interpretada.

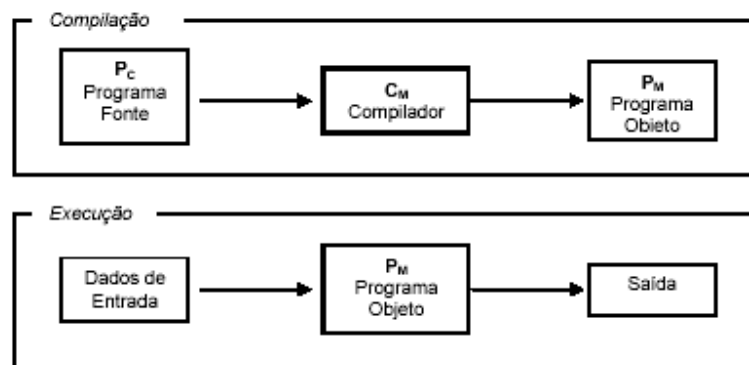


Figura 1.3: Execução de programas com linguagem compilada.

Devemos notar que, na Figura 1.2, o programa fonte é um dado de entrada a mais para o interpretador. No caso da compilação, Figura 1.3, identificamos duas fases: na primeira, o programa objeto é a saída do programa compilador e, na segunda, o programa objeto é executado, recebendo os dados de entrada e gerando a saída correspondente.

Observamos que, embora seja comum termos linguagens funcionais implementadas por interpretação e linguagens convencionais por compilação, há exceções, não existindo nenhum impedimento conceitual para implementar qualquer linguagem por qualquer dos dois métodos, ou até por uma combinação de ambos. O termo *ô máquina* usado acima é intencionalmente vago. Por exemplo, computadores idênticos com sistemas operacionais diferentes devem ser considerados *ô máquinas*, ou *ô plataformas*, diferentes. Assim, um programa em C, que foi compilado em um PC com Windows, não deverá ser executado em um PC com Linux, e vice-versa.

### *Exemplo de código em C*

Para exemplificar códigos escritos em C, consideremos um programa que tem a finalidade de converter valores de temperatura dados em Celsius para Fahrenheit. Este programa define uma função principal que captura um valor de temperatura em Celsius, fornecido via teclado pelo usuário, e exibe como saída a temperatura correspondente em Fahrenheit. Para fazer a conversão, é utilizada uma função auxiliar. O código C deste programa exemplo é mostrado abaixo.

```
/* Programa para conversão de temperatura */
```

```
#include <stdio.h>
float converte (float c)
{
    float f;
    f = 1.8*c + 32;
    return f;
}
```

```
int main (void)
{
float t1;
float t2;
/* mostra mensagem para usuario */
printf("Digite a temperatura em Celsius: ");
/* captura valor entrado via teclado */
scanf("%f",&t1);
/* faz a conversao */
t2 = converte(t1);
/* exhibe resultado */
printf("A temperatura em Fahrenheit é: %f\n", t2);
return 0;
}
```

Um programa em C, em geral, é constituído de diversas pequenas funções, que são independentes entre si ó não podemos, por exemplo, definir uma função dentro de outra. Dois tipos de ambientes são caracterizados em um código C. O ambiente global, externo às funções, e os ambientes locais, definidos pelas diversas funções (lembrando que os ambientes locais são independentes entre si). Podem-se inserir comentários no código fonte, iniciados com /\* e finalizados com \*/, conforme ilustrado acima. Devemos notar também que comandos e declarações em C são terminados pelo caractere ponto-e-vírgula (;). Um programa em C tem que, obrigatoriamente, conter a função principal (main). A execução de um programa começa pela função principal (a função main é automaticamente chamada quando o programa é carregado para a memória). As funções auxiliares são chamadas, direta ou indiretamente, a partir da função principal.

Em C, como nas demais linguagens õconvencionaisõ, devemos reservar área de memória para armazenar cada dado. Isto é feito através da declaração de variáveis, na qual informamos o tipo do dado que iremos armazenar naquela posição de memória. Assim, a declaração float t1;, do código mostrado, reserva um espaço de memória para armazenarmos um valor real (ponto flutuante ó float). Este espaço de memória é referenciado através do símbolo t1.

Uma característica fundamental da linguagem C diz respeito ao tempo de vida e à visibilidade das variáveis. Uma variável (local) declarada dentro de uma função "vive" enquanto esta função está sendo executada, e nenhuma outra função tem acesso direto a esta variável. Outra característica das variáveis locais é que devem sempre ser explicitamente inicializadas antes de seu uso, caso contrário conterão õlixoõ, isto é, valores indefinidos.

Como alternativa, é possível definir variáveis que sejam externas às funções, isto é, variáveis globais, que podem ser acessadas pelo nome por qualquer função subsequente

(são ôvisíveisô em todas as funções que se seguem à sua definição). Além do mais, devido às variáveis externas (ou globais) existirem permanentemente (pelo menos enquanto o programa estiver sendo executado), elas retêm seus valores mesmo quando as funções que as acessam deixam de existir. Embora seja possível definir variáveis globais em qualquer parte do ambiente global (entre quaisquer funções), é prática comum defini-las no início do arquivo-fonte.

Como regra geral, por razões de clareza e estruturação adequada do código, devemos evitar o uso indisciplinado de variáveis globais e resolver os problemas fazendo uso de variáveis locais sempre que possível. No próximo capítulo, discutiremos variáveis com mais detalhe.

### *Compilação de programas em C*

Para desenvolvermos programas em uma linguagem como C, precisamos de, no mínimo, um editor e um compilador. Estes programas têm finalidades bem definidas: com o editor de textos, escrevemos os programas fontes, que são salvos em arquivos<sup>1</sup>; com o compilador, transformamos os programas fontes em programas objetos, em linguagem de máquina, para poderem ser executados. Os programas fontes são, em geral, armazenados em arquivos cujo nome tem a extensão `.c`. Os programas executáveis possuem extensões que variam com o sistema operacional: no Windows, têm extensão `.exe`; no Unix (Linux), m geral, não têm extensão.

Para exemplificar o ciclo de desenvolvimento de um programa simples, consideremos que o código apresentado na seção anterior tenha sido salvo num arquivo com o nome `prog.c`. Devemos então compilar o programa para gerarmos um executável. Para ilustrar este processo, usaremos o compilador `gcc`. Na linha de comando do sistema operacional, fazemos:

```
> gcc -o prog prog.c
```

Se não houver erro de compilação no nosso código, este comando gera o executável com o nome `prog` (`prog.exe`, no Windows). Podemos então executar o programa:

```
> prog
```

*Digite a temperatura em Celsius: **10***

*A temperatura em Fahrenheit vale: 50.000000*

Em itálico, representamos as mensagens do programa e, em negrito, exemplificamos um dado fornecido pelo usuário via teclado.

## ***1.3. Programas com vários arquivos fontes***

Os programas reais são, naturalmente, maiores. Nestes casos, subdividimos o fonte do programa em vários arquivos. Para exemplificar a criação de um programa com dois arquivos, vamos considerar que o programa para conversão de unidades de temperatura apresentado anteriormente seja dividido em dois fontes: o arquivo `converte.c` e o arquivo `principal.c`. Teríamos que criar dois arquivos, como ilustrado abaixo:

Arquivo `converte.c`:

```
/* Implementação do módulo de conversão */  
float converte (float c)  
{  
    float f;  
    f = 1.8*c + 32;  
    return f;  
}
```

Arquivo `principal.c`:

```
/* Programa para conversão de temperatura */  
#include <stdio.h>  
float converte (float c);  
int main (void)  
{  
    float t1;  
    float t2;  
    /* mostra mensagem para usuario */  
    printf("Entre com temperatura em Celsius: ");  
    /* captura valor entrado via teclado */  
    scanf("%f",&t1);  
    /* faz a conversao */  
    t2 = converte(t1);  
    /* exhibe resultado */  
    printf("A temperatura em Fahrenheit vale: %f\n", t2);  
    return 0;  
}
```

Embora o entendimento completo desta organização de código não fique claro agora, interessa-nos apenas mostrar como geramos um executável de um programa com vários arquivos fontes. Uma alternativa é compilar tudo junto e gerar o executável como anteriormente:

```
> gcc -o prog converte.c principal.c
```

No entanto, esta não é a melhor estratégia, pois se alterarmos a implementação de um determinado módulo não precisaríamos re-compilar os outros. Uma forma mais eficiente é

compilarmos os módulos separadamente e depois ligar os diversos módulos objetos gerados para criar um executável.

```
> gcc -o obj1.o obj1.c
> gcc -o obj2.o obj2.c
> gcc -o prog obj1.o obj2.o
```

A opção `-o` do compilador gcc indica que não queremos criar um executável, apenas gerar o arquivo objeto (com extensão `.o` ou `.obj`). Depois, invocamos gcc para fazer a ligação dos objetos, gerando o executável.

## 1.4. Ciclo de desenvolvimento

Programas como editores, compiladores e ligadores são às vezes chamados de ferramentas, usadas na Engenharia de Software. Exceto no caso de programas muito pequenos (como é o caso de nosso exemplo), é raro que um programa seja composto de um único arquivo fonte. Normalmente, para facilitar o projeto, os programas são divididos em vários arquivos. Como vimos, cada um desses arquivos pode ser compilado em separado, mas para sua execução é necessário reunir os códigos de todos eles, sem esquecer das bibliotecas necessárias, e esta é a função do *ligador*.

A tarefa das bibliotecas é permitir que funções de interesse geral estejam disponíveis com facilidade. Nosso exemplo usa a biblioteca de entrada/saída padrão do C, `stdio`, que oferece funções que permitem a captura de dados a partir do teclado e a saída de dados para a tela. Além de bibliotecas preparadas pelo fornecedor do compilador, ou por outros fornecedores de *software*, podemos ter bibliotecas preparadas por um usuário qualquer, que pode empacotar funções com utilidades relacionadas em uma biblioteca e, dessa maneira, facilitar seu uso em outros programas.

Em alguns casos, a função do ligador é executada pelo próprio compilador. Por exemplo, quando compilamos o primeiro programa `prog.c`, o ligador foi chamado automaticamente para reunir o código do programa aos códigos de `scanf`, `printf` e de outras funções necessárias à execução independente do programa.

## 1.5. Verificação e Validação

Outro ponto que deve ser observado é que os programas podem conter (e, em geral, contêm) erros, que precisam ser identificados e corrigidos. Quase sempre a verificação é realizada por meio de testes, executando o programa a ser testado com diferentes valores de entrada. Identificado um ou mais erros, o código fonte é corrigido e deve ser novamente verificado. O processo de compilação, ligação e teste se repetem até que os resultados dos testes sejam satisfatórios e o programa seja considerado validado. Podemos descrever o ciclo através da Figura 1.4.



Este ciclo pode ser realizado usando programas (editor, compilador, ligador) separados ou empregando um ambiente integrado de desenvolvimento (*integrated development environment*, ou IDE). IDE é um programa que oferece janelas para a edição de programas e facilidades para abrir, fechar e salvar arquivos e para compilar, ligar e executar programas. Se um IDE estiver disponível, é possível criar e testar um programa, tudo em um mesmo ambiente, e todo o ciclo mencionado acima acontece de maneira mais confortável dentro de um mesmo ambiente, de preferência com uma interface amigável.

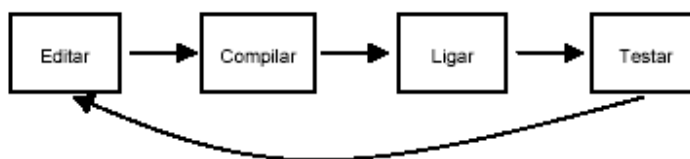


Figura 1.4: Ciclo de desenvolvimento.

## 2. EXPRESSÕES

---

Em C, uma expressão é uma combinação de variáveis, constantes e operadores que pode ser avaliada computacionalmente, resultando em um valor. O valor resultante é chamado de *valor da expressão*.

### 2.1. Variáveis

Podemos dizer que uma variável representa um espaço na memória do computador para armazenar determinado tipo de dado. Na linguagem C, todas as variáveis devem ser explicitamente declaradas. Na declaração de uma variável, obrigatoriamente, devem ser especificados seu *tipo* e seu *nome*: o nome da variável serve de referência ao dado armazenado no espaço de memória da variável e o tipo da variável determina a natureza do dado que será armazenado.

### Tipos básicos

A linguagem C oferece alguns tipos básicos. Para armazenar valores inteiros, existem três tipos básicos: char, short int, long int. Estes tipos diferem entre si pelo espaço de memória que ocupam e conseqüentemente pelo intervalo de valores que podem representar.

O tipo char, por exemplo, ocupa 1 byte de memória (8 bits), podendo representar 256 (=2<sup>8</sup>) valores distintos. Todos estes tipos podem ainda ser modificados para representarem apenas valores positivos, o que é feito precedendo o tipo com o modificador *unsigned*. A tabela abaixo compara os tipos para valores inteiros e suas representatividades.

Tipo	Tamanho	Representatividade
char	1 byte	-128 a 127
unsigned char	1 byte	0 a 255
short int	2 bytes	-32 768 a 32 767
unsigned short int	2 bytes	0 a 65 535
long int	4 bytes	-2 147 483 648 a 2 147 483 647

		647
unsigned long int	4 bytes	4 294 967 295

Os tipos short int e long int podem ser referenciados simplesmente com short e long, respectivamente. O tipo int puro é mapeado para o tipo inteiro natural da máquina, que pode ser short ou long. A maioria das máquinas que usamos hoje funcionam com processadores de 32 bits e o tipo int é mapeado para o inteiro de 4 bytes (long).<sup>1</sup> O tipo char é geralmente usado apenas para representar códigos de caracteres, como veremos nos capítulos subsequentes.

A linguagem oferece ainda dois tipos básicos para a representação de números reais (ponto flutuante): float e double. A tabela abaixo compara estes dois tipos.

Tipo	Tamanho	Representatividade
float	4 bytes	$\pm 10^{-38}$ a $10^{38}$
double	8 bytes	$\pm 10^{-308}$ a $10^{308}$

## Declaração de variáveis

Para armazenarmos um dado (valor) na memória do computador, devemos reservar o espaço correspondente ao tipo do dado a ser armazenado. A declaração de uma variável reserva um espaço na memória para armazenar um dado do tipo da variável e associa o nome da variável a este espaço de memória.

```
int a;      /* declara uma variável do tipo int */
int b;      /* declara outra variável do tipo int */
float c;    /* declara uma variável do tipo float */
a = 5;      /* armazena o valor 5 em a */
b = 10;     /* armazena o valor 10 em b */
c = 5.3;    /* armazena o valor 5.3 em c */
```

A linguagem permite que variáveis de mesmo tipo sejam declaradas juntas. Assim, as duas primeiras declarações acima poderiam ser substituídas por:

```
int a, b;    /* declara duas variáveis do tipo int */
```

Uma vez declarada a variável, podemos armazenar valores nos respectivos espaços de memória. Estes valores devem ter o mesmo tipo da variável,

conforme ilustrado acima. Não é possível, por exemplo, armazenar um número real numa variável do tipo int. Se fizermos:

```
int a;  
a = 4.3;      /* a variável armazenará o valor 4 */
```

será armazenada em a apenas a parte inteira do número real, isto é, 4. Alguns compiladores exibem uma advertência quando encontram este tipo de atribuição.

Em C, as variáveis podem ser inicializadas na declaração. Podemos, por exemplo, escrever:

```
int a = 5, b = 10;    /* declara e inicializa as variáveis */  
float c = 5.3;
```

## Valores constantes

Em nossos códigos, usamos também valores constantes. Quando escrevemos a atribuição:

```
a = b + 123;
```

sendo a e b variáveis supostamente já declaradas, reserva-se um espaço para armazenar a constante 123. No caso, a constante é do tipo inteiro, então um espaço de quatro bytes (em geral) é reservado e o valor 123 é armazenado nele. A diferença básica em relação às variáveis, como os nomes dizem (variáveis e constantes), é que o valor armazenado numa área de constante não pode ser alterado.

As constantes também podem ser do tipo real. Uma constante real deve ser escrita com um ponto decimal ou valor de expoente. Sem nenhum sufixo, uma constante real é do tipo double. Se quisermos uma constante real do tipo float, devemos, a rigor, acrescentar o sufixo F ou f. Alguns exemplos de constantes reais são:

12.45	constante real do tipo double
1245e-2	constante real do tipo double
12.45F	constante real do tipo float

Alguns compiladores exibem uma advertência quando encontram o código abaixo:

```
float x;  
...  
x = 12.45;
```

pois o código, a rigor, armazena um valor double (12.45) numa variável do tipo float. Desde que a constante seja representável dentro de um float, não precisamos nos preocupar com este tipo de advertência.

### Variáveis com valores indefinidos

Um dos erros comuns em programas de computador é o uso de variáveis cujos valores ainda estão indefinidos. Por exemplo, o trecho de código abaixo está errado, pois o valor armazenado na variável *b* está indefinido e tentamos usá-lo na atribuição a *c*. É comum dizermos que *b* tem *ólixoö*.

```
int a, b, c;  
a = 2;  
c = a + b    ; /* ERRO: b tem ólixoö */
```

Alguns destes erros são óbvios (como o ilustrado acima) e o compilador é capaz de nos reportar uma advertência; no entanto, muitas vezes o uso de uma variável não definida fica difícil de ser identificado no código. Repetimos que é um erro comum em programas e uma razão para alguns programas funcionarem na parte da manhã e não funcionarem na parte da tarde (ou funcionarem durante o desenvolvimento e não funcionarem quando entregamos para nosso cliente!). Todos os erros em computação têm lógica. A razão de o programa poder funcionar uma vez e não funcionar outra é que, apesar de indefinido, o valor da variável existe. No nosso caso acima, pode acontecer que o valor armazenado na memória ocupada por *b* seja 0, fazendo com que o programa funcione. Por outro lado, pode acontecer de o valor ser 0 293423 e o programa não funcionar.

## 2.2. Operadores

A linguagem C oferece uma gama variada de operadores, entre binários e unários. Os operadores básicos são apresentados a seguir.

## Operadores Aritméticos

Os operadores aritméticos binários são: +, -, \*, / e o operador módulo %. Há ainda o operador unário -. A operação é feita na precisão dos operandos. Assim, a expressão  $5/2$  resulta no valor 2, pois a operação de divisão é feita em precisão inteira, já que os dois operandos (5 e 2) são constantes inteiras. A divisão de inteiros trunca a parte fracionária, pois o valor resultante é sempre do mesmo tipo da expressão. Conseqüentemente, a expressão  $5.0/2.0$  resulta no valor real 2.5 pois a operação é feita na precisão real (double, no caso).

O operador módulo, %, não se aplica a valores reais, seus operandos devem ser do tipo inteiro. Este operador produz o resto da divisão do primeiro pelo segundo operando. Como exemplo de aplicação deste operador, podemos citar o caso em que desejamos saber se o valor armazenado numa determinada variável inteira  $x$  é par ou ímpar. Para tanto, basta analisar o resultado da aplicação do operador %, aplicado à variável e ao valor dois.

$x \% 2$                     *se resultado for zero (número é par)*  
 $x \% 2$                     *se resultado for um (número é ímpar)*

Os operadores \*, / e % têm precedência maior que os operadores + e -. O operador - unário tem precedência maior que \*, / e %. Operadores com mesma precedência são avaliados da esquerda para a direita. Assim, na expressão:

$a + b * c / d$

executa-se primeiro a multiplicação, seguida da divisão, seguida da soma. Podemos utilizar parênteses para alterar a ordem de avaliação de uma expressão. Assim, se quisermos avaliar a soma primeiro, podemos escrever:

$(a + b) * c / d$

Uma tabela de precedência dos operadores da linguagem C é apresentada no final desta seção.

## Operadores de atribuição

Na linguagem C, uma atribuição é uma expressão cujo valor resultante corresponde ao valor atribuído. Assim, da mesma forma que a expressão:

$$5 + 3$$

resulta no valor 8, a atribuição:

$$a = 5$$

resulta no valor 5 (além, é claro, de armazenar o valor 5 na variável a). Este tratamento das atribuições nos permite escrever comandos do tipo:

$$y = x = 5;$$

Neste caso, a ordem de avaliação é da direita para a esquerda. Assim, o computador avalia  $x = 5$ , armazenando 5 em x, e em seguida armazena em y o valor produzido por  $x = 5$ , que é 5. Portanto, ambos, x e y, recebem o valor 5.

A linguagem também permite utilizar os chamados operadores de atribuição compostos. Comandos do tipo:

$$i = i + 2;$$

em que a variável à esquerda do sinal de atribuição também aparece à direita, podem ser escritas de forma mais compacta:

$$i += 2;$$

usando o operador de atribuição composto  $+=$ . Analogamente, existem, entre outros, os operadores de atribuição:  $-=$ ,  $*=$ ,  $/=$ ,  $\%=$ . De forma geral, comandos do tipo:

*var op= expr;*  
*são equivalentes a:*  
*var = var op (expr);*

Salientamos a presença dos parênteses em torno de expr. Assim:

$x *= y + 1;$   
*equivale a*

$x = x * (y + 1)$   
*e não a*  
 $x = x * y + 1;$

## Operadores de incremento e decremento

A linguagem C apresenta ainda dois operadores não convencionais. São os operadores de incremento e decremento, que possuem precedência comparada ao - unário e servem para incrementar e decrementar uma unidade nos valores armazenados nas variáveis. Assim, se *n* é uma variável que armazena um valor, o comando:

$n++;$

incrementa de uma unidade o valor de *n* (análogo para o decremento em  $n--$ ). O aspecto não usual é que  $++$  e  $--$  podem ser usados tanto como operadores pré-fixados (antes da variável, como em  $++n$ ) ou pós-fixados (após a variável, como em  $n++$ ). Em ambos os casos, a variável *n* é incrementada. Porém, a expressão  $++n$  incrementa *n* *antes* de usar seu valor, enquanto  $n++$  incrementa *n* *após* seu valor ser usado. Isto significa que, num contexto onde o valor de *n* é usado,  $++n$  e  $n++$  são diferentes. Se *n* armazena o valor 5, então:

$x = n++;$

atribui 5 a *x*, mas:

$x = ++n;$

atribuiria 6 a *x*. Em ambos os casos, *n* passa a valer 6, pois seu valor foi incrementado em uma unidade. Os operadores de incremento e decremento podem ser aplicados somente em variáveis; uma expressão do tipo  $x = (i + 1)++$  é ilegal.

A linguagem C oferece diversas formas compactas para escrevermos um determinado comando. Neste curso, procuraremos evitar as formas compactas pois elas tendem a dificultar a compreensão do código. Mesmo para programadores experientes, o uso das formas compactas deve ser feito com critério. Por exemplo, os comandos:



```
a = a + 1;  
a += 1;  
a++;  
++a;
```

são todos equivalentes e o programador deve escolher o que achar mais adequado e simples. Em termos de desempenho, qualquer compilador razoável é capaz de otimizar todos estes comandos da mesma forma.

## Operadores relacionais e lógicos

Os operadores relacionais em C são:

```
< menor que  
> maior que  
<= menor ou igual que  
>= maior ou igual que  
== igual a  
!= diferente de
```

Estes operadores comparam dois valores. O resultado produzido por um operador relacional é zero ou um. Em C, não existe o tipo booleano (*true* ou *false*). O valor zero é interpretado como falso e qualquer valor diferente de zero é considerado verdadeiro. Assim, se o resultado de uma comparação for falso, produz-se o valor 0, caso contrário, produz-se o valor 1.

Os operadores lógicos combinam expressões booleanas. A linguagem oferece os seguintes operadores lógicos:

```
&& operador binário E (AND)  
|| operador binário OU (OR)  
! operador unário de NEGAÇÃO (NOT)
```

Expressões conectadas por && ou || são avaliadas da esquerda para a direita, e a avaliação pára assim que a veracidade ou falsidade do resultado for conhecida. Recomendamos o uso de parênteses em expressões que combinam operadores lógicos e relacionais.

Os operadores relacionais e lógicos são normalmente utilizados para tomada de decisões. No entanto, podemos utilizá-los para atribuir valores a variáveis. Por exemplo, o trecho de código abaixo é válido e armazena o valor 1 em a e 0 em b.

```
int a, b;  
int c = 23;  
int d = c + 4;  
a = (c < 20) || (d > c); /* verdadeiro */  
b = (c < 20) && (d > c); /* falso */
```

Devemos salientar que, na avaliação da expressão atribuída à variável b, a operação (d>c) não chega a ser avaliada, pois independente do seu resultado a expressão como um todo terá como resultado 0 (falso), uma vez que a operação (c<20) tem valor falso.

### **Operador *sizeof***

Outro operador fornecido por C, `sizeof`, resulta no número de bytes de um determinado tipo. Por exemplo:

```
int a = sizeof(float);
```

armazena o valor 4 na variável a, pois um float ocupa 4 bytes de memória. Este operador pode também ser aplicado a uma variável, retornando o número de bytes do tipo associado à variável.

### **Conversão de tipo**

Em C, como na maioria das linguagens, existem conversões automáticas de valores na avaliação de uma expressão. Assim, na expressão 3/1.5, o valor da constante 3 (tipo int) é promovido (convertido) para double antes de a expressão ser avaliada, pois o segundo operando é do tipo double (1.5) e a operação é feita na precisão do tipo mais representativo.

Quando, numa atribuição, o tipo do valor atribuído é diferente do tipo da variável, também há uma conversão automática de tipo. Por exemplo, se escrevermos:

```
int a = 3.5;
```

o valor 3.5 é convertido para inteiro (isto é, passa a valer 3) antes de a atribuição ser efetuada. Como resultado, como era de se esperar, o valor atribuído à variável é 3 (inteiro). Alguns compiladores exibem advertências quando a conversão de tipo pode significar uma perda de precisão (é o caso da conversão real para inteiro, por exemplo). O programador pode explicitamente requisitar uma conversão de tipo através do uso do operador de molde de tipo (operador *cast*). Por exemplo, são válidos (e isentos de qualquer advertência por parte dos compiladores) os comandos abaixo.

```
int a, b;
a = (int) 3.5;
b = (int) 3.5 % 2;
```

### Precedência e ordem de avaliação dos operadores

A tabela abaixo mostra a precedência, em ordem decrescente, dos principais operadores da linguagem C.

Operador	Associatividade
() [] ->.	esquerda para direita
! ~ ++ -- - (tipo) * & sizeof(tipo)	direita para esquerda
* / %	esquerda para direita
+ -	esquerda para direita
<< >>	esquerda para direita
< <= > >=	esquerda para direita
== !=	esquerda para direita
&	esquerda para direita
^	esquerda para direita
	esquerda para direita
&&	esquerda para direita
	esquerda para direita
?:	direita para esquerda
= += -= etc.	direita para esquerda
,	esquerda para direita

### 2.3.Entrada e saída básicas

A linguagem C não possui comandos de entrada e saída do tipo READ e WRITE encontrados na linguagem FORTRAN. Tudo em C é feito através de funções, inclusive as operações de entrada e saída. Por isso, já existe em C uma biblioteca padrão que possui as funções básicas normalmente necessárias. Na biblioteca padrão de C, podemos, por exemplo, encontrar funções matemáticas do tipo raiz quadrada, seno, cosseno, etc., funções para a manipulação de cadeias de caracteres e funções de entrada e saída. Nesta seção, serão apresentadas as duas funções básicas de entrada e saída disponibilizadas pela biblioteca padrão. Para utilizá-las, é necessário incluir o *protótipo* destas funções no código. Este assunto será tratado em detalhes na seção sobre funções. Por ora, basta saber que é preciso escrever:

```
#include <stdio.h>
```

no início do programa que utiliza as funções da biblioteca de entrada e saída.

## Função *printf*

A função *printf* possibilita a saída de valores (sejam eles constantes, variáveis ou resultado de expressões) segundo um determinado formato. Informalmente, podemos dizer que a forma da função é:

```
printf (formato, lista de constantes/variáveis/expressões...);
```

O primeiro parâmetro é uma cadeia de caracteres, em geral delimitada com aspas, que especifica o formato de saída das constantes, variáveis e expressões listadas em seguida. Para cada valor que se deseja imprimir, deve existir um especificador de formato correspondente na cadeia de caracteres *formato*. Os especificadores de formato variam com o tipo do valor e a precisão em que queremos que eles sejam impressos. Estes especificadores são precedidos pelo caractere % e podem ser, entre outros:

%c	<i>especifica um char</i>
%d	<i>especifica um int</i>
%u	<i>especifica um unsigned int</i>
%f	<i>especifica um double (ou float)</i>



*%e especifica um double (ou float) no formato científico*  
*%g especifica um double (ou float) no formato mais apropriado (%f ou %e)*  
*%s especifica uma cadeia de caracteres*

Alguns exemplos:

```
printf("%d %g\n", 33, 5.3);
```

tem como resultado a impressão da linha:

*33 5.3*

Ou:

```
printf("Inteiro = %d Real = %g\n", 33, 5.3);
```

com saída:

*Inteiro = 33 Real = 5.3*

Isto é, além dos especificadores de formato, podemos incluir textos no formato, que são mapeados diretamente para a saída. Assim, a saída é formada pela cadeia de caracteres do formato onde os especificadores são substituídos pelos valores correspondentes. Existem alguns caracteres de escape que são freqüentemente utilizados nos formatos de saída. São eles:

*\n caractere de nova linha*

*\t caractere de tabulação*

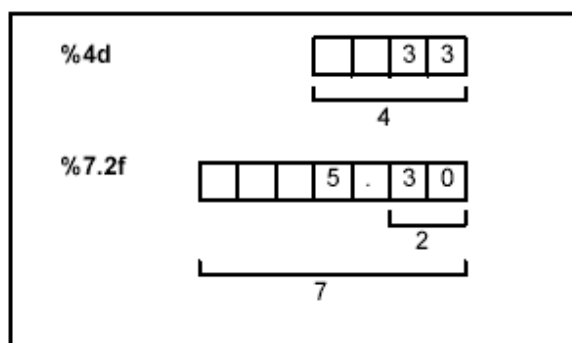
*\r caractere de retrocesso*

*\ " o caractere "*

*\\ o caractere \*

Ainda, se desejarmos ter como saída um caractere %, devemos, dentro do formato, escrever % %.

É possível também especificarmos o tamanho dos campos:



## Função *scanf*

A função *scanf* permite capturarmos valores fornecidos via teclado pelo usuário do programa. Informalmente, podemos dizer que sua forma geral é:

*scanf*(formato, lista de endereços das variáveis...);

O formato deve possuir especificadores de tipos similares aos mostrados para a função *printf*. Para a função *scanf*, no entanto, existem especificadores diferentes para o tipo float e o tipo double:

*%c especifica um char*

*%d especifica um int*

*%u especifica um unsigned int*

*%f,%e,%g especificam um float*

*%lf, %le, %lg especificam um double*

*%s especifica uma cadeia de caracteres*

A principal diferença é que o formato deve ser seguido por uma lista de endereços de variáveis (na função *printf* passamos os valores de constantes, variáveis e expressões). Na seção sobre ponteiros, este assunto será tratado em detalhes. Por ora, basta saber que, para ler um valor e atribuí-lo a uma variável, devemos passar o endereço da variável para a função *scanf*. O operador *&* retorna o endereço de uma variável. Assim, para ler um inteiro, devemos ter:

*int n;*

*scanf("%d", &n);*

Para a função `scanf`, os especificadores `%f`, `%e` e `%g` são equivalentes. Aqui, caracteres diferentes dos especificadores no formato servem para cercar a entrada. Por exemplo:

```
scanf ("%d:%d", &h, &m);
```

obriga que os valores (inteiros) fornecidos sejam separados pelo caractere dois pontos (:). Um espaço em branco dentro do formato faz com que sejam "pulados" eventuais brancos da entrada. Os especificadores `%d`, `%f`, `%e` e `%g` automaticamente pulam os brancos que precederem os valores numéricos a serem capturados. A função `scanf` retorna o número de campos lidos com sucesso.

### **Função *getchar()***

É a função original de entrada de caractere dos sistemas baseados em UNIX.

`getchar()` armazena a entrada até que ENTER seja pressionada.

Ex:

```
main()  
{  
  char ch;  
  ch=getchar();  
  printf ("%c\n",ch);  
}
```

### **Função *putchar()***

Escreve na tela o argumento de seu caractere na posição corrente.

Ex:

```
main()  
{  
  char ch;  
  printf ("digite uma letra minúscula : ");  
  ch=getchar();  
  putchar(toupper(ch));  
  putchar('\n');  
}
```



Há inúmeras outras funções de manipulação de char complementares às que foram vistas, como `isalpha()`, `isupper()`, `islower()`, `isdigit()`, `isspace()`, `toupper()`, `tolower()`.

### ***3. CONTROLE DE FLUXO***

---



A linguagem C provê as construções fundamentais de controle de fluxo necessárias para programas bem estruturados: agrupamentos de comandos, tomadas de decisão (if-else), laços com teste de encerramento no início (while, for) ou no fim (do-while), e seleção de um dentre um conjunto de possíveis casos (switch).

### ***3.1.Decisões com if***

if é o comando de decisão básico em C. Sua forma pode ser:

```
if (expr) {  
bloco de comandos 1  
...  
}
```

ou

```
if ( expr ) {  
bloco de comandos 1  
...  
}  
else {  
bloco de comandos 2  
...  
}
```

Se *expr* produzir um valor diferente de 0 (verdadeiro), o *bloco de comandos 1* será executado. A inclusão do else requisita a execução do *bloco de comandos 2* se a expressão produzir o valor 0 (falso). Cada bloco de comandos deve ser delimitado por uma chave aberta e uma chave fechada. Se dentro de um bloco tivermos apenas um comando a ser executado, as chaves podem ser omitidas (na verdade, deixamos de ter um bloco):

```
if ( expr )  
comando1;  
else  
comando2;
```

A indentação (recoo de linha) dos comandos é fundamental para uma maior clareza do código. O estilo de indentação varia a gosto do programador. Além

da forma ilustrada acima, outro estilo bastante utilizado por programadores C é:

```
if ( expr )
{
    bloco de comandos 1
...
}
else
{
    bloco de comandos 2
...
}
```

Podemos aninhar comandos if. Um exemplo simples é ilustrado a seguir:

```
#include <stdio.h>
int main (void)
{
    int a, b;
    printf("Insira dois numeros inteiros:");
    scanf("%d%d",&a,&b);
    if (a%2 == 0)
    if (b%2 == 0)
        printf("Voce inseriu dois numeros pares!\n");
    return 0;
}
```

Primeiramente, notamos que não foi necessário criar blocos ( { ... } ) porque a cada if está associado apenas um comando. Ao primeiro, associamos o segundo comando if, e ao segundo if associamos o comando que chama a função printf. Assim, o segundo if só será avaliado se o primeiro valor fornecido for par, e a mensagem só será impressa se o segundo valor fornecido também for par. Outra construção para este mesmo exemplo simples pode ser:

```
int main (void)
{
    int a, b;
    printf("Digite dois numeros inteiros:");
    scanf("%d%d",&a,&b);
    if ((a%2 == 0) && (b%2 == 0))
        printf( "Voce digitou dois numeros pares!\n");
    return 0;
}
```

produzindo resultados idênticos. Devemos, todavia, ter cuidado com o aninhamento de comandos if-else. Para ilustrar, consideremos o exemplo abaixo.

```
/* temperatura (versao 1 - incorreta) */  
#include <stdio.h>  
int main (void)  
{  
    int temp;  
    printf("Digite a temperatura: ");  
    scanf("%d", &temp);  
    if (temp < 30)  
    if (temp > 20)  
        printf(" Temperatura agradável \n");  
    else  
        printf(" Temperatura muito quente \n");  
    return 0;  
}
```

A idéia deste programa era imprimir a mensagem Temperatura agradável se fosse fornecido um valor entre 20 e 30, e imprimir a mensagem Temperatura muito quente se fosse fornecido um valor maior que 30. No entanto, vamos analisar o caso de ser fornecido o valor 5 para temp. Observando o código do programa, podemos pensar que nenhuma mensagem seria fornecida, pois o primeiro if daria resultado verdadeiro e então seria avaliado o segundo if. Neste caso, teríamos um resultado falso e como, *aparentemente*, não há um comando else associado, nada seria impresso. Puro engano. A indentação utilizada pode nos levar a erros de interpretação. O resultado para o valor 5 seria a mensagem Temperatura muito quente. Isto é, o programa está INCORRETO.

Em C, um else está associado ao último if que não tiver seu próprio else. Para os casos em que a associação entre if e else não está clara, recomendamos a criação explícita de blocos, mesmo contendo um único comando. Reescrevendo o programa, podemos obter o efeito desejado.

```
/* temperatura (versao 2) */  
#include <stdio.h>  
int main (void)  
{
```

```
int temp;
printf( "Digite a temperatura: " );
scanf( "%d", &temp );
if( temp < 30 )
{
    if( temp > 20 )
        printf( " Temperatura agradável \n" );
    }
    else
        printf( " Temperatura muito quente \n" );
return 0;
}
```

Esta regra de associação do else propicia a construção do tipo else-if, sem que se tenha o comando `elseif` explicitamente na gramática da linguagem. Na verdade, em C, construímos estruturas else-if com `if`s aninhados. O exemplo abaixo é válido e funciona como esperado.

```
/* temperatura (versao 3) */
#include <stdio.h>
int main (void)
{
    int temp;
    printf("Digite a temperatura: ");
    scanf("%d", &temp);
    if (temp < 10)
        printf("Temperatura muito fria \n");
    else if (temp < 20)
        printf(" Temperatura fria \n");
    else if (temp < 30)
        printf("Temperatura agradável \n");
    else
        printf("Temperatura muito quente \n");
    return 0;
}
```

### **3.2.Estruturas de bloco**

Observamos que uma função C é composta por estruturas de blocos. Cada chave aberta e fechada em C representa um bloco. As declarações de variáveis só podem ocorrer no início do corpo da função ou no início de um bloco, isto é, devem seguir uma chave aberta. Uma variável declarada dentro de um bloco

é válida apenas dentro do bloco. Após o término do bloco, a variável deixa de existir. Por exemplo:

```
...  
if ( n > 0 )  
{  
  int i;  
  ...  
}  
... /* a variável i não existe neste ponto do programa */
```

A variável *i*, definida dentro do bloco do *if*, só existe dentro deste bloco. É uma boa prática de programação declarar as variáveis o mais próximo possível dos seus usos.

### 3.3. Operador condicional

C possui também um chamado *operador condicional*. Trata-se de um operador que substitui construções do tipo:

```
...  
if ( a > b )  
  maximo = a;  
else  
  maximo = b;  
...
```

Sua forma geral é:

condição ? expressão1 : expressão2;

se a *condição* for verdadeira, a *expressão1* é avaliada; caso contrário, avalia-se a *expressão2*. O comando:

*maximo = a > b ? a : b ;*

substitui a construção com *if-else* mostrada acima.

### 3.4. Construções com laços

É muito comum, em programas computacionais, termos procedimentos iterativos, isto é, procedimentos que devem ser executados em vários passos. Como exemplo, vamos considerar o cálculo do valor do fatorial de um número inteiro não negativo. Por definição:

$$n! = n.(n-1)(n-2)...3.2.1, \text{ onde } 0! = 1.$$

Para calcular o fatorial de um número através de um programa de computador, utilizamos tipicamente um processo iterativo, em que o valor da variável varia de 1 a n. A linguagem C oferece diversas construções possíveis para a realização de laços iterativos. O primeiro a ser apresentado é o comando `while`. Sua forma geral é:

```
while (expr)
{
    bloco de comandos
...
}
```

Enquanto *expr* for avaliada em verdadeiro, o bloco de comandos é executado repetidamente. Se *expr* for avaliada em falso, o bloco de comando não é executado e a execução do programa prossegue. Uma possível implementação do cálculo do fatorial usando `while` é mostrada a seguir.

```
/* Fatorial */
#include <stdio.h>
int main (void)
{
    int i;
    int n;
    int f = 1;
    printf("Digite um número inteiro nao negativo:");
    scanf("%d", &n);
    /* calcula fatorial */
    i = 1;
    while (i <= n)
    {
        f *= i;
        i++;
    }
    printf(" Fatorial = %d \n", f);
    return 0;
}
```

}

Uma segunda forma de construção de laços em C, mais compacta e amplamente utilizada, é através de laços for. A forma geral do for é:

```
for (expr_inicial; expr_booleana; expr_de_incremento)
{
    bloco de comandos
    ...
}
```

A ordem de avaliação desta construção é ilustrada a seguir:

```
expr_inicial;
while (expr_booleana)
{
    bloco de comandos
    ...
    expr_de_incremento
}
```

A seguir, ilustramos a utilização do comando for no programa para cálculo do fatorial.

```
/* Fatorial (versao 2) */
#include <stdio.h>
int main (void)
{
    int i;
    int n;
    int f = 1;
    printf("Digite um número inteiro nao negativo:");
    scanf("%d", &n);
    /* calcula fatorial */
    for (i = 1; i <= n; i++)
    {
        f *= i;
    }
    printf(" Fatorial = %d \n", f);
    return 0;
}
```

Observamos que as chaves que seguem o comando `for`, neste caso, são desnecessárias, já que o corpo do bloco é composto por um único comando.

Tanto a construção com `while` como a construção com `for` avaliam a expressão booleana que caracteriza o teste de encerramento no início do laço. Assim, se esta expressão tiver valor igual a zero (falso), quando for avaliada pela primeira vez, os comandos do corpo do bloco não serão executados nem uma vez.

C provê outro comando para construção de laços cujo teste de encerramento é avaliado no final. Esta construção é o `do-while`, cuja forma geral é:

```
do
{
    bloco de comandos
} while (expr_booleana);
```

Um exemplo do uso desta construção é mostrado abaixo, onde validamos a inserção do usuário, isto é, o programa repetidamente requisita a inserção de um número enquanto o usuário inserir um inteiro negativo (cujo fatorial não está definido).

```
/* Fatorial (versao 3) */
#include <stdio.h>
int main (void)
{
    int i;
    int n;
    int f = 1;
    /* requisita valor do usuário */
    do
    {
        printf("Digite um valor inteiro não negativo:");
        scanf ("%d", &n);
    } while (n<0);
    /* calcula fatorial */
    for (i = 1; i <= n; i++)
        f *= i;
    printf(" Fatorial = %d\n", f);
    return 0;
```



}

### 3.5. Interrupções com break e continue

A linguagem C oferece ainda duas formas para a interrupção antecipada de um determinado laço. O comando `break`, quando utilizado dentro de um laço, interrompe e termina a execução do mesmo. A execução prossegue com os comandos subsequentes ao bloco. O código abaixo ilustra o efeito de sua utilização.

```
#include <stdio.h>
int main (void)
{
    int i;
    for (i = 0; i < 10; i++)
    {
        if (i == 5)
            break;
        printf("%d ", i);
    }
    printf("fim\n");
    return 0;
}
```

A saída deste programa, se executado, será:

0 1 2 3 4 fim

pois, quando `i` tiver o valor 5, o laço será interrompido e finalizado pelo comando `break`, passando o controle para o próximo comando após o laço, no caso uma chamada final de `printf`.

O comando `continue` também interrompe a execução dos comandos de um laço. A diferença básica em relação ao comando `break` é que o laço não é automaticamente finalizado. O comando `continue` interrompe a execução de um laço passando para a próxima iteração. Assim, o código:

```
#include <stdio.h>
int main (void)
{
    int i;
    for (i = 0; i < 10; i++ )
```

```
{  
if (i == 5) continue;  
printf("%d ", i);  
}  
printf("fim\n");  
return 0;  
}
```

gera a saída:

0 1 2 3 4 6 7 8 9 fim

Devemos ter cuidado com a utilização do comando `continue` nos laços `while`.  
O programa:

```
/* INCORRETO */  
#include <stdio.h>  
int main (void)  
{  
int i = 0;  
while (i < 10)  
{  
if (i == 5) continue;  
printf("%d ", i);  
i++;  
}  
printf("fim\n");  
return 0;  
}
```

é um programa **INCORRETO**, pois o laço criado não tem fim ó a execução do programa não termina. Isto porque a variável `i` nunca terá valor superior a 5, e o teste será sempre verdadeiro. O que ocorre é que o comando `continue` "pula" os demais comandos do laço quando `i` vale 5, inclusive o comando que incrementa a variável `i`.

### 3.6. Seleção

Além da construção `else-if`, C provê um comando (`switch`) para selecionar um dentre um conjunto de possíveis casos. Sua forma geral é:

```
switch ( expr )  
{
```

```
case op1:
... /* comandos executados se expr == op1 */
break;
case op2:
... /* comandos executados se expr == op2 */
break;
case op3:
... /* comandos executados se expr == op3 */
break;
default:
... /* executados se expr for diferente de todos */
break;
}
```

$op_i$  deve ser um número inteiro ou uma constante caractere. Se *expr* resultar no valor  $op_i$ , os comandos que se seguem ao caso  $op_i$  são executados, até que se encontre um `break`. Se o comando `break` for omitido, a execução do caso continua com os comandos do caso seguinte. O caso `default` (nenhum dos outros) pode aparecer em qualquer posição, mas normalmente é colocado por último. Para exemplificar, mostramos a seguir um programa que implementa uma calculadora convencional que efetua as quatro operações básicas. Este programa usa constantes caracteres, que serão discutidas em detalhe quando apresentarmos cadeias de caracteres em C. O importante aqui é entender conceitualmente a construção `switch`.

```
/* calculadora de quatro operações */
#include <stdio.h>
int main (void)
{
float num1, num2;
char op;
printf("Digite: numero op numero\n");
scanf ("%f%c%f", &num1, &op, &num2);
switch (op)
{
case '+':
printf(" = %f\n", num1+num2);
break;
case '-':
printf(" = %f\n", num1-num2);
break;
case '*':
printf(" = %f\n", num1*num2);
```

```
break;
case '/':
printf(" = %f\n", num1/num2);
break;
default:
printf("Operador invalido!\n");
break;
}
return 0;
}
```

## 4. FUNÇÕES

---

### 4.1. Definição de funções

As funções dividem grandes tarefas de computação em tarefas menores. Os programas em geral consistem de várias pequenas funções em vez de poucas de maior tamanho. A criação de funções evita a repetição de código, de modo que um procedimento que é repetido deve ser transformado numa função que, então, será chamada diversas vezes. Um programa bem estruturado deve ser pensado em termos de funções, e estas, por sua vez, podem (e devem, se possível) esconder do corpo principal do programa detalhes ou particularidades de implementação. Em C, tudo é feito através de funções. Os exemplos anteriores utilizam as funções da biblioteca padrão para realizar entrada e saída. Neste capítulo, discutiremos a codificação de nossas próprias funções.

A forma geral para definir uma função é:

```
tipo_retornado nome_da_função (lista de parâmetros...)  
{  
    corpo da função  
}
```

Para ilustrar a criação de funções, consideraremos o cálculo do fatorial de um número. Podemos escrever uma função que, dado um determinado número inteiro não negativo *n*, imprime o valor de seu fatorial. Um programa que utiliza esta função seria:

```
/* programa que le um numero e imprime seu fatorial */  
#include <stdio.h>  
void fat (int n);  
/* Função principal */  
int main (void)  
{  
    int n;  
    scanf("%d", &n);  
    fat(n);  
    return 0;  
}  
/* Função para imprimir o valor do fatorial */  
void fat ( int n )  
{  
    int i;  
    int f = 1;  
    for (i = 1; i <= n; i++)  
        f *= i;  
    printf("Fatorial = %d\n", f);  
}
```

Notamos, neste exemplo, que a função *fat* recebe como parâmetro o número cujo fatorial deve ser impresso. Os parâmetros de uma função devem ser listados, com seus respectivos tipos, entre os parênteses que seguem o nome da função. Quando uma função não tem parâmetros, colocamos a palavra reservada *void* entre os parênteses. Devemos notar que *main* também é uma função; sua única particularidade consiste em ser a função automaticamente executada após o programa ser carregado. Como as funções *main* que temos apresentado não recebem parâmetros, temos usado a palavra *void* na lista de parâmetros.

Além de receber parâmetros, uma função pode ter um valor de retorno associado. No exemplo do cálculo do fatorial, a função *fat* não tem nenhum valor de retorno, portanto colocamos a palavra *void* antes do nome da função, indicando a ausência de um valor de retorno.

```
void fat (int n)
{
...
}
```

A função *main* obrigatoriamente deve ter um valor inteiro como retorno. Esse valor pode ser usado pelo sistema operacional para testar a execução do programa. A convenção geralmente utilizada faz com que a função *main* retorne zero no caso da execução ser bem sucedida ou diferente de zero no caso de problemas durante a execução.

Por fim, salientamos que C exige que se coloque o *protótipo* da função antes desta ser chamada. O *protótipo* de uma função consiste na repetição da linha de sua definição seguida do caractere (;). Temos então:

```
void fat (int n); /* obs: existe ; no protótipo */
int main (void)
{
...
}
void fat (int n) /* obs: nao existe ; na definição */
{
...
}
```

A rigor, no protótipo não há necessidade de indicarmos os nomes dos parâmetros, apenas os seus tipos, portanto seria válido escrever: *void fat (int);*. Porém, geralmente mantemos os nomes dos parâmetros, pois servem como documentação do significado de cada parâmetro, desde que utilizemos nomes coerentes. O protótipo da função é necessário para que o compilador verifique os tipos dos parâmetros na chamada da função. Por exemplo, se tentássemos chamar a função com *fat(4.5);* o compilador provavelmente indicaria o erro, pois estaríamos passando um valor real enquanto a função espera um valor inteiro. É devido a esta necessidade que se exige a inclusão do arquivo *stdio.h* para a utilização das funções de entrada e saída da biblioteca padrão. Neste

arquivo, encontram-se, entre outras coisas, os protótipos das funções `printf` e `scanf`.

Uma função pode ter um valor de retorno associado. Para ilustrar a discussão, vamos reescrever o código acima, fazendo com que a função `fat` retorne o valor do fatorial. A função `main` fica então responsável pela impressão do valor.

```
/* programa que le um numero e imprime seu fatorial (versão 2) */
#include <stdio.h>
int fat (int n);
int main (void)
{
    int n, r;
    scanf("%d", &n);
    r = fat(n);
    printf("Fatorial = %d\n", r);
    return 0;
}
/* funcao para calcular o valor do fatorial */
int fat (int n)
{
    int i;
    int f = 1;
    for (i = 1; i <= n; i++)
        f *= i;
    return f;
}
```

## **4.2. Pilha de execução**

Apresentada a forma básica para a definição de funções, discutiremos agora, em detalhe, como funciona a comunicação entre a função que chama e a função que é chamada. Já mencionamos na introdução deste curso que as funções são independentes entre si. As variáveis locais definidas dentro do corpo de uma função (e isto inclui os parâmetros das funções) não existem fora da função. Cada vez que a função é executada, as variáveis locais são criadas, e, quando a execução da função termina, estas variáveis deixam de existir.

A transferência de dados entre funções é feita através dos parâmetros e do valor de retorno da função chamada. Conforme mencionado, uma função pode retornar um valor para a função que a chamou e isto é feito através do comando `return`. Quando uma função tem um valor de retorno, a chamada da função é uma expressão cujo valor resultante é o valor retornado pela função. Por isso, foi válido escrevermos na função `main` acima a expressão `r = fat(n)`; que chama a função `fat` armazenando seu valor de retorno na variável `r`.

A comunicação através dos parâmetros requer uma análise mais detalhada. Para ilustrar a discussão, vamos considerar o exemplo abaixo, no qual a implementação da função `fat` foi ligeiramente alterada:

```
/* programa que le um numero e imprime seu fatorial (versão 3) */
#include <stdio.h>
int fat (int n);
int main (void)
{
    int n = 5;
    int r;
    r = fat ( n );
    printf("Fatorial de %d = %d \n", n, r);
    return 0;
}
int fat (int n)
{
    int f = 1.0;
    while (n != 0)
    {
        f *= n;
        n--;
    }
    return f;
}
```

Neste exemplo, podemos verificar que, no final da função `fat`, o parâmetro `n` tem valor igual a zero (esta é a condição de encerramento do laço `while`). No entanto, a saída do programa será:

*Fatorial de 5 = 120*



pois o valor da variável *n* não mudou no programa principal. Isto porque a linguagem C trabalha com o conceito de **passagem por valor**. Na chamada de uma função, o valor passado é atribuído ao parâmetro da função chamada. Cada parâmetro funciona como uma variável local inicializada com o valor passado na chamada. Assim, a variável *n* (parâmetro da função *fat*) é local e não representa a variável *n* da função *main* (o fato de as duas variáveis terem o mesmo nome é indiferente; poderíamos chamar o parâmetro de *v*, por exemplo). Alterar o valor de *n* dentro de *fat* não afeta o valor da variável *n* de *main*.

A execução do programa funciona com o **modelo de pilha**. De forma simplificada, o modelo de pilha funciona da seguinte maneira: cada variável local de uma função é colocada na pilha de execução. Quando se faz uma chamada a uma função, os parâmetros são copiados para a pilha e são tratados como se fossem variáveis locais da função chamada. Quando a função termina, a parte da pilha correspondente àquela função é liberada, e por isso não podemos acessar as variáveis locais de fora da função em que elas foram definidas.

Para exemplificar, vamos considerar um esquema representativo da memória do computador. Salientamos que este esquema é apenas uma maneira didática de explicar o que ocorre na memória do computador. Suponhamos que as variáveis são armazenadas na memória como ilustrado abaixo. Os números à direita representam endereços (posições) fictícios de memória e os nomes à esquerda indicam os nomes das variáveis. A figura abaixo ilustra este esquema representativo da memória que adotaremos.

c	'x'	112	- variável c no endereço 112 com valor igual a 'x'
b	43.5	108	- variável b no endereço 108 com valor igual a 43.5
a	7	104	- variável a no endereço 104 com valor igual a 7

Figura 4.1: Esquema representativo da memória.

Podemos, então, analisar passo a passo a evolução do programa mostrado acima, ilustrando o funcionamento da pilha de execução.

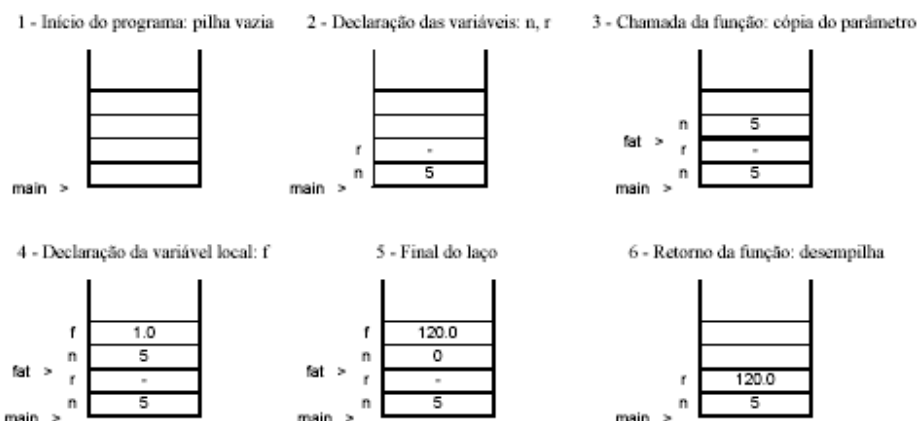


Figura 4.2: Execução do programa passo a passo.

Isto ilustra por que o valor da variável passada nunca será alterado dentro da função. A seguir, discutiremos uma forma para podermos alterar valores por passagem de parâmetros, o que será realizado passando o endereço de memória onde a variável está armazenada.

Vale salientar que existe outra forma de fazermos comunicação entre funções, que consiste no uso de variáveis globais. Se uma determinada variável global é visível em duas funções, ambas as funções podem acessar e/ou alterar o valor desta variável diretamente. No entanto, conforme já mencionamos, o uso de variáveis globais em um programa deve ser feito com critério, pois podemos criar códigos com uma alto grau de interdependência entre as funções, o que dificulta a manutenção e o reuso do código.

### 4.3. Ponteiro de variáveis

A linguagem C permite o armazenamento e a manipulação de valores de endereços de memória. Para cada tipo existente, há um tipo ponteiro que pode armazenar endereços de memória onde existem valores do tipo correspondente armazenados. Por exemplo, quando escrevemos:

```
int a;
```

declaramos uma variável com nome a que pode armazenar valores inteiros. Automaticamente, reserva-se um espaço na memória suficiente para armazenar valores inteiros (geralmente 4 bytes). Da mesma forma que declaramos variáveis para armazenar inteiros, podemos declarar variáveis que,

em vez de servirem para armazenar valores inteiros, servem para armazenar valores de endereços de memória onde há variáveis inteiras armazenadas. C não reserva uma palavra especial para a declaração de ponteiros; usamos a mesma palavra do tipo com os nomes das variáveis precedidas pelo caractere \*. Assim, podemos escrever:

```
int *p;
```

Neste caso, declaramos uma variável com nome p que pode armazenar endereços de memória onde existe um inteiro armazenado. Para atribuir e acessar endereços de memória, a linguagem oferece dois operadores unários ainda não discutidos. O operador unário & (o endereço de), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. O operador unário \* (o conteúdo de), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço de memória armazenado pela variável ponteiro. Para exemplificar, vamos ilustrar esquematicamente, através de um exemplo simples, o que ocorre na pilha de execução. Consideremos o trecho de código mostrado na figura abaixo.

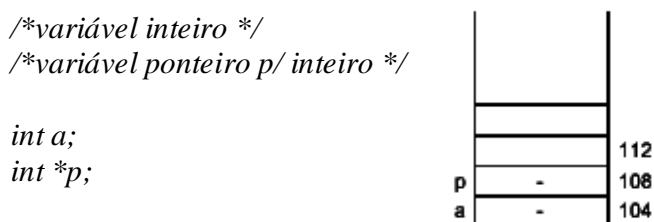
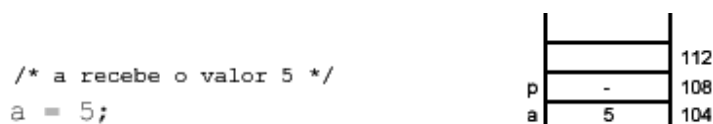


Figura 4.3: Efeito de declarações de variáveis na pilha de execução.

Após as declarações, ambas as variáveis, a e p, armazenam valores "lixo", pois não foram inicializadas. Podemos fazer atribuições como exemplificado nos fragmentos de código da figura a seguir:



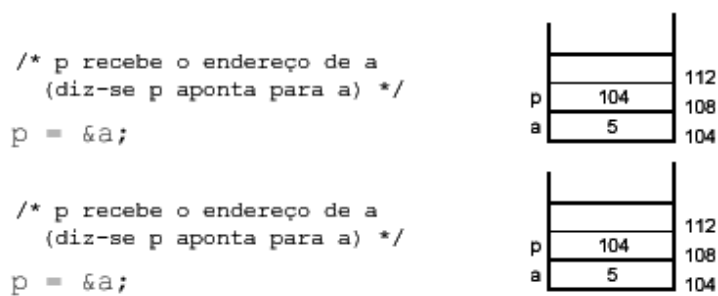


Figura 4.4: Efeito de atribuição de variáveis na pilha de execução.

Com as atribuições ilustradas na figura, a variável *a* recebe, indiretamente, o valor 6. Acessar *a* é equivalente a acessar *\*p*, pois *p* armazena o endereço de *a*. Dizemos que *p* *aponta* para *a*, daí o nome *ponteiro*. Em vez de criarmos valores fictícios para os endereços de memória no nosso esquema ilustrativo da memória, podemos desenhar setas graficamente, sinalizando que um ponteiro aponta para uma determinada variável.

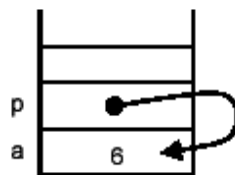


Figura 4.5: Representação gráfica do valor de um ponteiro.

A possibilidade de manipular ponteiros de variáveis é uma das maiores potencialidades de C. Por outro lado, o uso indevido desta manipulação é o maior causador de programas que "voam", isto é, não só não funcionam como, pior ainda, podem gerar efeitos colaterais não previstos.

A seguir, apresentamos outros exemplos de uso de ponteiros. O código abaixo:

```

int main ( void )
{
    int a;
    int *p;
    p = &a;
    *p = 2;
    printf(" %d ", a);
    return;
}

```

```
}  
imprime o valor 2.  
Agora, no exemplo abaixo:  
int main ( void )  
{  
  int a, b, *p;  
  a = 2;  
  *p = 3;  
  b = a + (*p);  
  printf(" %d ", b);  
  return 0;  
}
```

cometemos um ERRO típico de manipulação de ponteiros. O pior é que esse programa, mbora incorreto, às vezes pode funcionar. O erro está em usar a memória apontada por p para armazenar o valor 3. Ora, a variável p não tinha sido inicializada e, portanto, tinha armazenado um valor (no caso, endereço) "lixo". Assim, a atribuição `*p = 3;` armazena 3 num espaço de memória desconhecido, que tanto pode ser um espaço de memória não utilizado, e aí o programa aparentemente funciona bem, quanto um espaço que armazena outras informações fundamentais ó por exemplo, o espaço de memória utilizado por outras variáveis ou outros aplicativos. Neste caso, o erro pode ter efeitos colaterais indesejados.

Portanto, só podemos preencher o conteúdo de um ponteiro se este tiver sido devidamente inicializado, isto é, ele deve apontar para um espaço de memória onde já se prevê o armazenamento de valores do tipo em questão. De maneira análoga, podemos declarar ponteiros de outros tipos:

```
float *m;  
char *s;
```

#### ***4.4. Passando ponteiros para funções***

Os ponteiros oferecem meios de alterarmos valores de variáveis acessando-as indiretamente. Já discutimos que as funções não podem alterar diretamente valores de variáveis da função que fez a chamada. No entanto, se passarmos para uma função os valores dos endereços de memória onde suas variáveis estão armazenadas, a função pode alterar, indiretamente, os valores das variáveis da função que a chamou.

Vamos analisar o uso desta estratégia através de um exemplo. Consideremos uma função projetada para trocar os valores entre duas variáveis. O código abaixo:

```
/* funcao troca (versao ERRADA) */  
#include <stdio.h>  
void troca (int x, int y )  
{  
    int temp;  
    temp = x;  
    x = y;  
    y = temp;  
}  
int main ( void )  
{  
    int a = 5, b = 7;  
    troca(a, b);  
    printf("%d %d \n", a, b);  
    return 0;  
}
```

não funciona como esperado (serão impressos 5 e 7), pois os valores de a e b da função main não são alterados. Alterados são os valores de x e y dentro da função troca, mas eles não representam as variáveis da função main, apenas são inicializados com os valores de a e b. A alternativa é fazer com que a função receba os endereços das variáveis e, assim, alterar seus valores indiretamente. Reescrevendo:

```
/* funcao troca (versao CORRETA) */  
#include <stdio.h>  
void troca (int *px, int *py )  
{  
    int temp;  
    temp = *px;  
    *px = *py;  
    *py = temp;  
}  
int main ( void )  
{  
    int a = 5, b = 7;  
    troca(&a, &b); /* passamos os endereços das variáveis */  
    printf("%d %d \n", a, b);  
}
```

```
return 0;
}
```

A Figura 4.6 ilustra a execução deste programa mostrando o uso da memória. Assim, conseguimos o efeito desejado. Agora fica explicado por que passamos o endereço das variáveis para a função scanf, pois, caso contrário, a função não conseguiria devolver os valores lidos.

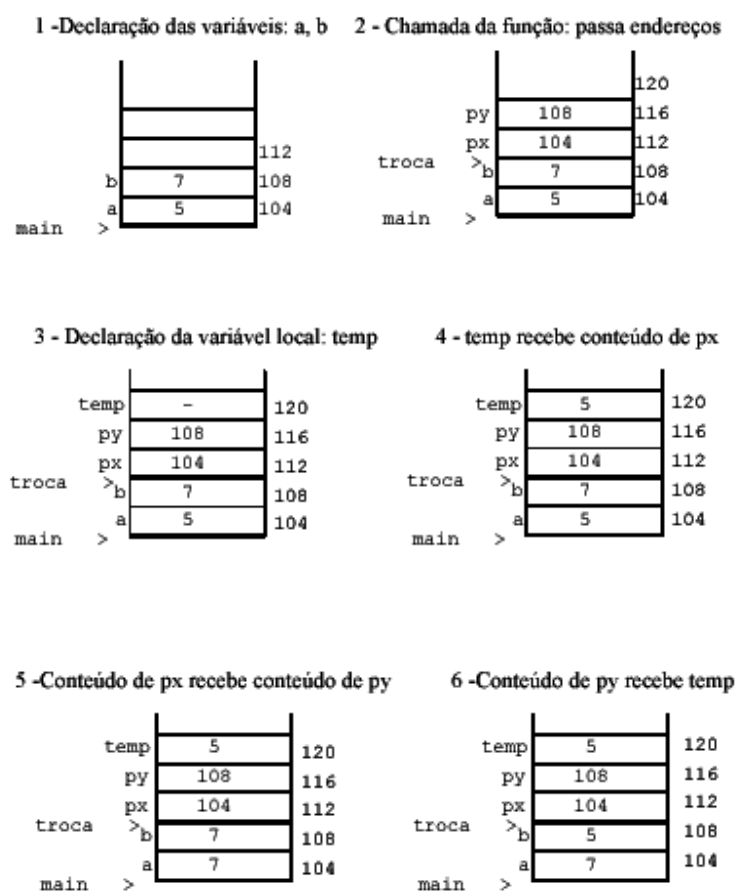


Figura 4.6: Passo a passo da função que troca dois valores.

## 4.5. Recursividade

As funções podem ser chamadas recursivamente, isto é, dentro do corpo de uma função podemos chamar novamente a própria função. Se uma função A chama a própria função A, dizemos que ocorre uma recursão direta. Se uma função A chama uma função B que, por sua vez, chama A, temos uma recursão indireta. Diversas implementações ficam muito mais fáceis usando

recursividade. Por outro lado, implementações não recursivas tendem a ser mais eficientes.

Para cada chamada de uma função, recursiva ou não, os parâmetros e as variáveis locais são empilhados na pilha de execução. Assim, mesmo quando uma função é chamada recursivamente, cria-se um ambiente local para cada chamada. As variáveis locais de chamadas recursivas são independentes entre si, como se estivessemos chamando funções diferentes.

As implementações recursivas devem ser pensadas considerando-se a definição recursiva do problema que desejamos resolver. Por exemplo, o valor do fatorial de um número pode ser definido de forma recursiva:

$$n = \begin{cases} 1, & \text{se } n = 0 \\ n(n-1)!, & \text{se } n \neq 0 \end{cases}$$

Considerando a definição acima, fica muito simples pensar na implementação recursiva de uma função que calcula e retorna o fatorial de um número.

```
/* Função recursiva para calculo do fatorial */  
int fat (int n)  
{  
    if (n==0)  
        return 1;  
    else  
        return n*fat(n-1);  
}
```

#### ***4.6. Variáveis estáticas dentro de funções***

Podemos declarar variáveis estáticas dentro de funções. Neste caso, as variáveis não são armazenadas na pilha, mas sim numa área de memória estática que existe enquanto o programa está sendo executado. Ao contrário das variáveis locais (ou automáticas), que existem apenas enquanto a função à qual elas pertencem estiver sendo executada, as estáticas, assim como as globais, continuam existindo mesmo antes ou depois de a função ser executada. No entanto, uma variável estática declarada dentro de uma função só é visível dentro dessa função. Uma utilização importante de variáveis estáticas dentro de funções é quando se necessita recuperar o valor de uma variável atribuída na última vez que



a função foi executada.

Para exemplificar a utilização de variáveis estáticas declaradas dentro de funções, consideremos uma função que serve para imprimir números reais. A característica desta função é que ela imprime um número por vez, separando-os por espaços em branco e colocando, no máximo, cinco números por linha. Com isto, do primeiro ao quinto número são impressos na primeira linha, do sexto ao décimo na segunda, e assim por diante.

```
void imprime ( float a )
{
    static int n = 1;
    printf(" %f ", a);
    if ((n % 5) == 0) printf(" \n ");
    n++;
}
```

Se uma variável estática não for explicitamente inicializada na declaração, ela é automaticamente inicializada com zero. (As variáveis globais também são, por *default*, inicializadas com zero.)

## 4.7. Pré-processador e macros

Um código C, **antes** de ser compilado, passa por um pré-processador. O pré-processador de C reconhece determinadas diretivas e altera o código para, então, enviá-lo ao compilador. Uma das diretivas reconhecidas pelo pré-processador, e já utilizada nos nossos exemplos, é `#include`. Ela é seguida por um nome de arquivo e o pré-processador a substitui pelo corpo do arquivo especificado. É como se o texto do arquivo incluído fizesse parte do código fonte. Uma observação: quando o nome do arquivo a ser incluído é envolto por aspas ("*arquivo*"), o pré-processador procura primeiro o arquivo no diretório atual e, caso não o encontre, o procura nos diretórios de *include* especificados para compilação. Se o arquivo é colocado entre os sinais de menor e maior (<*arquivo*>), o pré-processador não procura o arquivo no diretório atual.

Outra diretiva de pré-processamento que é muito utilizada e que será agora discutida é a diretiva de definição. Por exemplo, uma função para calcular a área de um círculo pode ser escrita da seguinte forma:

```
#define PI 3.14159
float area (float r)
{
float a = PI * r * r;
return a;
}
```

Neste caso, antes da compilação, toda ocorrência da palavra PI (desde que não envolvida por aspas) será trocada pelo número 3.14159. O uso de diretivas de definição para representarmos constantes simbólicas é fortemente recomendável, pois facilita a manutenção e acrescenta clareza ao código. C permite ainda a utilização da diretiva de definição com parâmetros. É válido escrever, por exemplo:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
assim, se após esta definição existir uma linha de código com o trecho:
v = 4.5;
c = MAX ( v, 3.0 );
o compilador verá:
v = 4.5;
c = ((v) > (4.5) ? (v) : (4.5));
```

Estas definições com parâmetros recebem o nome de **macros**. Devemos ter muito cuidado na definição de macros. Mesmo um erro de sintaxe pode ser difícil de ser detectado, pois o compilador indicará um erro na linha em que se utiliza a macro e não na linha de definição da macro (onde efetivamente encontra-se o erro). Outros efeitos colaterais de macros mal definidas podem ser ainda piores. Por exemplo, no código abaixo:

```
#include <stdio.h>
#define DIF(a,b) a - b
int main (void)
{
printf(" %d ", 4 * DIF(5,3));
return 0;
}
```

o resultado impresso é 17 e não 8, como poderia ser esperado. A razão é simples, pois para o compilador (fazendo a substituição da macro) está escrito:

```
printf(" %d ", 4 * 5 - 3);
```

e a multiplicação tem precedência sobre a subtração. Neste caso, parênteses envolvendo a macro resolveriam o problema. Porém, neste outro exemplo que envolve a macro com parênteses:

```
#include <stdio.h>
#define PROD(a,b) (a * b)
int main (void)
{
    printf(" %d ", PROD(3+4, 2));
    return 0;
}
```

o resultado é 11 e não 14. A macro corretamente definida seria:

```
#define PROD(a,b) ((a) * (b))
```

Concluimos, portanto, que, como regra básica para a definição de macros, devemos envolver cada parâmetro, e a macro como um todo, com parênteses.

## ***5. VETORES E ALOCAÇÃO DINÂMICA***

---

### ***5.1. Vetores***

A forma mais simples de estruturarmos um conjunto de dados é por meio de vetores. Como a maioria das linguagens de programação, C permite a definição de vetores. Definimos um vetor em C da seguinte forma:

```
int v[10];
```

A declaração acima diz que *v* é um vetor de inteiros dimensionado com 10 elementos, isto é, reservamos um espaço de memória **contínuo** para armazenar 10 valores inteiros. Assim, se cada *int* ocupa 4 bytes, a declaração acima reserva um espaço de memória de 40 bytes, como ilustra a figura abaixo.

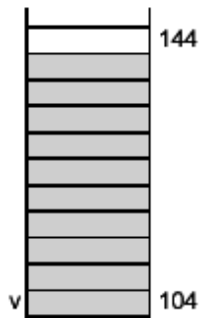


Figura 5.1: Espaço de memória de um vetor de 10 elementos inteiros.

O acesso a cada elemento do vetor é feito através de uma indexação da variável  $v$ . Observamos que, em C, a indexação de um vetor varia de zero a  $n-1$ , onde  $n$  representa a dimensão do vetor. Assim:

$v[0]$             para acessar o primeiro elemento de  $v$   
 $v[1]$             para acessar o segundo elemento de  $v$   
 ...  
 $v[9]$             para acessar o último elemento de  $v$

Mas:

$v[10]$             está ERRADO (invasão de memória)

Para exemplificar o uso de vetores, vamos considerar um programa que lê 10 números reais, fornecidos via teclado, e calcula a média e a variância destes números. A média e a variância são dadas por:

$$m = \frac{\sum x}{N}, \quad v = \frac{\sum (x - m)^2}{N}$$

Uma possível implementação é apresentada a seguir.

```
/* Cálculo da média e da variância de 10 números reais */
#include <stdio.h>
int main ( void )
{
    float v[10]; /* declara vetor com 10 elementos */
    float med, var; /* variáveis para armazenar a média e a variância */
    int i; /* variável usada como índice do vetor */
    /* leitura dos valores */
    for ( i = 0; i < 10; i++ ) /* faz índice variar de 0 a 9 */
        scanf("%f", &v[i]); /* lê cada elemento do vetor */
```

```
/* cálculo da média */
med = 0.0; /* inicializa média com zero */
for (i = 0; i < 10; i++)
    med = med + v[i]; /* acumula soma dos elementos */
med = med / 10; /* calcula a média */
/* cálculo da variância */
var = 0.0; /* inicializa variância com zero */
for ( i = 0; i < 10; i++ )
    var = var+(v[i]-med)*(v[i]-med); /* acumula quadrado da diferença */
var = var / 10; /* calcula a variância */
printf( "Media = %f Variância = %f\n", med, var );
return 0;
}
```

Devemos observar que passamos para a função scanf o endereço de cada elemento do vetor (&v[i]), pois desejamos que os valores capturados sejam armazenados nos elementos do vetor. Se v[i] representa o (i+1)-ésimo elemento do vetor, &v[i] representa o endereço de memória onde esse elemento está armazenado.

Na verdade, existe uma associação forte entre vetores e ponteiros, pois se existe a declaração:

```
int v[10];
```

a variável v, que representa o vetor, é uma constante que armazena o endereço inicial do vetor, isto é, v, sem indexação, aponta para o primeiro elemento do vetor.

A linguagem C também suporta aritmética de ponteiros. Podemos somar e subtrair ponteiros, desde que o valor do ponteiro resultante aponte para dentro da área reservada para o vetor. Se p representa um ponteiro para um inteiro, p+1 representa um ponteiro para o próximo inteiro armazenado na memória, isto é, o valor de p é incrementado de 4 (mais uma vez assumindo que um inteiro tem 4 bytes). Com isto, num vetor temos as seguintes equivalências:

v+0	aponta para o primeiro elemento do vetor
v+1	aponta para o segundo elemento do vetor
v+2	aponta para o terceiro elemento do vetor
...	
v+9	aponta para o último elemento do vetor

Portanto, escrever `&v[i]` é equivalente a escrever `(v+i)`. De maneira análoga, escrever `v[i]` é equivalente a escrever `*(v+i)` (é lógico que a forma indexada é mais clara e adequada). Devemos notar que o uso da aritmética de ponteiros aqui é perfeitamente válido, pois os elementos dos vetores são armazenados de forma contínua na memória. Os vetores também podem ser inicializados na declaração:

```
int v[5] = { 5, 10, 15, 20, 25 };
```

ou simplesmente:

```
int v[] = { 5, 10, 15, 20, 25 };
```

Neste último caso, a linguagem dimensiona o vetor pelo número de elementos inicializados.

## ***5.2. Passagem de vetores para funções***

Passar um vetor para uma função consiste em passar o endereço da primeira posição do vetor. Se passarmos um valor de endereço, a função chamada deve ter um parâmetro do tipo ponteiro para armazenar este valor. Assim, se passarmos para uma função um vetor de `int`, devemos ter um parâmetro do tipo `int*`, capaz de armazenar endereços de inteiros. Salientamos que a expressão `passar um vetor para uma função` deve ser interpretada como `passar o endereço inicial do vetor`. Os elementos do vetor não são copiados para a função, o argumento copiado é apenas o endereço do primeiro elemento.

Para exemplificar, vamos modificar o código do exemplo acima, usando funções separadas para o cálculo da média e da variância. (Aqui, usamos ainda os operadores de atribuição `+=` para acumular as somas.)

```
/* Cálculo da media e da variância de 10 reais (segunda versão) */
#include <stdio.h>
/* Função para cálculo da média */
float media (int n, float* v)
{
    int i;
    float s = 0.0;
```

```
for (i = 0; i < n; i++)
    s += v[i];
return s/n;
}
/* Função para cálculo da variância */
float variancia (int n, float* v, float m)
{
    int i;
    float s = 0.0;
    for (i = 0; i < n; i++)
        s += (v[i] - m) * (v[i] - m);
    return s/n;
}
int main ( void )
{
    float v[10];
    float med, var;
    int i;
    /* leitura dos valores */
    for ( i = 0; i < 10; i++ )
        scanf("%f", &v[i]);
    med = media(10,v);
    var = variancia(10,v,med);
    printf( "Media = %f Variância = %f\n", med, var);
    return 0;
}
```

Observamos ainda que, como é passado para a função o endereço do primeiro elemento do vetor (e não os elementos propriamente ditos), podemos alterar os valores dos elementos do vetor dentro da função. O exemplo abaixo ilustra:

```
/* Incrementa elementos de um vetor */
#include <stdio.h>
void incr_vetor ( int n, int *v )
{
    int i;
    for (i = 0; i < n; i++)
        v[i]++;
}
int main ( void )
{
    int a[ ] = {1, 3, 5};
    incr_vetor(3, a);
    printf("%d %d %d\n", a[0], a[1], a[2]);
}
```



```
return 0;  
}
```

A saída do programa é 2 4 6, pois os elementos do vetor serão incrementados dentro da função.

### ***5.3. Alocação dinâmica***

Até aqui, na declaração de um vetor, foi preciso dimensioná-lo. Isto nos obrigava a saber, de antemão, quanto espaço seria necessário, isto é, tínhamos que prever o número máximo de elementos no vetor durante a codificação. Este pré-dimensionamento do vetor é um fator limitante. Por exemplo, se desenvolvermos um programa para calcular a média e a variância das notas de uma prova, teremos que prever o número máximo de alunos. Uma solução é dimensionar o vetor com um número absurdamente alto para não termos limitações quando da utilização do programa. No entanto, isto levaria a um desperdício de memória que é inaceitável em diversas aplicações. Se, por outro lado, formos modestos no pré-dimensionamento do vetor, o uso do programa fica muito limitado, pois não conseguiríamos tratar turmas com o número de alunos maior que o previsto.

Felizmente, a linguagem C oferece meios de requisitarmos espaços de memória em tempo de execução. Dizemos que podemos alocar memória dinamicamente. Com este recurso, nosso programa para o cálculo da média e variância discutido acima pode, em tempo de execução, consultar o número de alunos da turma e então fazer a alocação do vetor dinamicamente, sem desperdício de memória.

### ***5.4. Uso da memória***

Informalmente, podemos dizer que existem três maneiras de reservarmos espaço de memória para o armazenamento de informações. A primeira delas é através do uso de variáveis globais (e estáticas). O espaço reservado para uma variável global existe enquanto o programa estiver sendo executado. A segunda maneira é através do uso de variáveis locais. Neste caso, como já discutimos, o espaço existe apenas enquanto a função que declarou a variável está sendo executada, sendo liberado para outros usos quando a execução da função termina. Por este motivo, a função que chama não pode fazer

referência ao espaço local da função chamada. As variáveis globais ou locais podem ser simples ou vetores. Para os vetores, precisamos informar o número máximo de elementos, caso contrário o compilador não saberia o tamanho do espaço a ser reservado.

A terceira maneira de reservarmos memória é requisitando ao sistema, em tempo de execução, um espaço de um determinado tamanho. Este espaço alocado dinamicamente permanece reservado até que explicitamente seja liberado pelo programa. Por isso, podemos alocar dinamicamente um espaço de memória numa função e acessá-lo em outra. A partir do momento que liberarmos o espaço, ele estará disponibilizado para outros usos e não podemos mais acessá-lo. Se o programa não liberar um espaço alocado, este será automaticamente liberado quando a execução do programa terminar.

Apresentamos abaixo um *esquema didático* que ilustra de maneira fictícia a distribuição do uso da memória pelo sistema operacional.

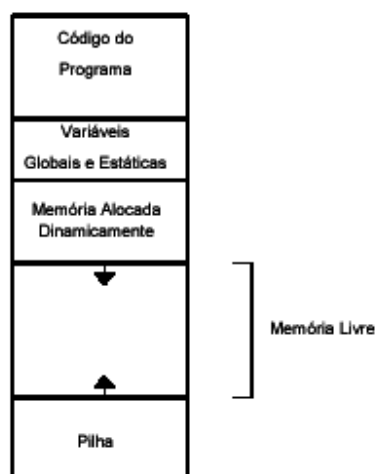


Figura 5.2: Alocação esquemática de memória.

Quando requisitamos ao sistema operacional para executar um determinado programa, o código em linguagem de máquina do programa deve ser carregado na memória, conforme discutido no primeiro capítulo. O sistema operacional reserva também os espaços necessários para armazenarmos as variáveis globais (e estáticas) existentes no programa. O restante da memória livre é utilizado pelas variáveis locais e pelas variáveis alocadas dinamicamente. Cada vez que uma determinada função é chamada, o sistema reserva o espaço necessário para as variáveis locais da função. Este espaço

pertence à pilha de execução e, quando a função termina, é desempilhado. A parte da memória não ocupada pela pilha de execução pode ser requisitada dinamicamente. Se a pilha tentar crescer mais do que o espaço disponível existente, dizemos que ela *estourou* e o programa é abortado com erro. Similarmente, se o espaço de memória livre for menor que o espaço requisitado dinamicamente, a alocação não é feita e o programa pode prever um tratamento de erro adequado (por exemplo, podemos imprimir a mensagem *Memória insuficiente* e interromper a execução do programa).

### ***5.5. Funções da biblioteca padrão***

Existem funções, presentes na biblioteca padrão *stdlib*, que permitem alocar e liberar memória dinamicamente. A função básica para alocar memória é *malloc*. Ela recebe como parâmetro o número de bytes que se deseja alocar e retorna o endereço inicial da área de memória alocada.

Para exemplificar, vamos considerar a alocação dinâmica de um vetor de inteiros com 10 elementos. Como a função *malloc* retorna o endereço da área alocada e, neste exemplo, desejamos armazenar valores inteiros nessa área, devemos declarar um ponteiro de inteiro para receber o endereço inicial do espaço alocado. O trecho de código então seria:

```
int *v;  
v = malloc(10*4);
```

Após este comando, se a alocação for bem sucedida, *v* armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros. Podemos, então, tratar *v* como tratamos um vetor declarado estaticamente, pois, se *v* aponta para o início da área alocada, podemos dizer que *v*[0] acessa o espaço para o primeiro elemento que armazenaremos, *v*[1] acessa o segundo, e assim por diante (até *v*[9]).

No exemplo acima, consideramos que um inteiro ocupa 4 bytes. Para ficarmos independentes de compiladores e máquinas, usamos o operador *sizeof*( ). *v* = *malloc*(10\**sizeof*(int)); Além disso, devemos lembrar que a função *malloc* é usada para alocar espaço para armazenarmos valores de qualquer tipo. Por este motivo, *malloc* retorna um ponteiro genérico, para um tipo qualquer, representado por *void\**, que pode ser convertido automaticamente pela

linguagem para o tipo apropriado na atribuição. No entanto, é comum fazermos a conversão explicitamente, utilizando o operador de molde de tipo (*cast*).

O comando para a alocação do vetor de inteiros fica então:

```
v = (int *) malloc(10*sizeof(int));
```

A figura abaixo ilustra de maneira esquemática o que ocorre na memória:

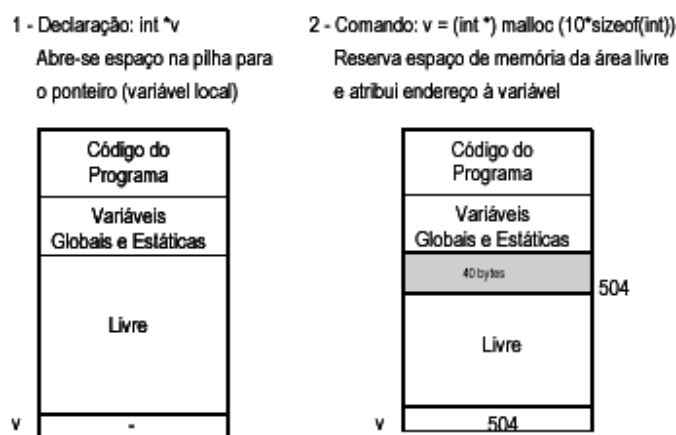


Figura 5.3: Alocação dinâmica de memória.

Se, porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo (representado pelo símbolo `NULL`, definido em *stdlib.h*). Podemos cercar o erro na alocação do programa verificando o valor de retorno da função `malloc`. Por exemplo, podemos imprimir uma mensagem e abortar o programa com a função `exit`, também definida na *stdlib*.

```
í
v = (int*) malloc(10*sizeof(int));
if (v==NULL)
{
printf("Memoria insuficiente.\n");
exit(1); /* aborta o programa e retorna 1 para o sist. operacional */
}
í
```

Para liberar um espaço de memória alocado dinamicamente, usamos a função `free`. Esta função recebe como parâmetro o ponteiro da memória a ser liberada. Assim, para liberar o vetor `v`, fazemos:

*free (v);*

Só podemos passar para a função *free* um endereço de memória que tenha sido alocado dinamicamente. Devemos lembrar ainda que não podemos acessar o espaço na memória depois que o liberamos.

Para exemplificar o uso da alocação dinâmica, alteraremos o programa para o cálculo da média e da variância mostrado anteriormente. Agora, o programa lê o número de valores que serão fornecidos, aloca um vetor dinamicamente e faz os cálculos. Somente a função principal precisa ser alterada, pois as funções para calcular a média e a variância anteriormente apresentadas independem do fato de o vetor ter sido alocado estática ou dinamicamente.

```
/* Cálculo da média e da variância de n reais */
#include <stdio.h>
#include <stdlib.h>
...
int main ( void )
{
    int i, n;
    float *v;
    float med, var;
    /* leitura do número de valores */
    scanf("%d", &n);
    /* alocação dinâmica */
    v = (float*) malloc(n*sizeof(float));
    if (v==NULL) {
        printf("Memoria insuficiente.\n");
        return 1;
    }
    /* leitura dos valores */
    for (i = 0; i < n; i++)
        scanf("%f", &v[i]);
    med = media(n,v);
    var = variancia(n,v,med);
    printf("Media = %f Variancia = %f\n", med, var);
    /* libera memória */
    free(v);
    return 0;
}
```

## 6. CADEIA DE CARACTERES

### 6.1. Caracteres

Efetivamente, a linguagem C não oferece um tipo caractere. Os caracteres são representados por códigos numéricos. A linguagem oferece o tipo `char`, que pode armazenar valores inteiros pequenos: um `char` tem tamanho de 1 byte, 8 bits, e sua versão com sinal pode representar valores que variam de -128 a 127. Como os códigos associados aos caracteres estão dentro desse intervalo, usamos o tipo `char` para representar caracteres<sup>1</sup>. A correspondência entre os caracteres e seus códigos numéricos é feita por uma tabela de códigos. Em geral, usa-se a tabela ASCII, mas diferentes máquinas podem usar diferentes códigos. Contudo, se desejamos escrever códigos portáteis, isto é, que possam ser compilados e executados em máquinas diferentes, devemos evitar o uso explícito dos códigos referentes a uma determinada tabela, como será discutido nos exemplos subseqüentes. Como ilustração, mostramos a seguir os códigos associados a alguns caracteres segundo a tabela ASCII.

Alguns caracteres que podem ser impressos (sp representa o branco, ou espaço):

	0	1	2	3	4	5	6	7	8	9
30			sp	!	"	#	\$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{		}	~			

Alguns caracteres de controle:

0	nul	<i>null</i> : nulo
7	bel	<i>bell</i> : campainha
8	bs	<i>backspace</i> : voltar e apagar um caractere
9	ht	<i>tab</i> ou tabulação horizontal
10	nl	<i>newline</i> ou <i>line feed</i> : mudança de linha
13	cr	<i>carriage return</i> : volta ao início da linha
127	del	<i>delete</i> : apagar um caractere

Em C, a diferença entre caracteres e inteiros é feita apenas através da maneira pela qual são tratados. Por exemplo, podemos imprimir o mesmo valor de duas formas diferentes usando formatos diferentes. Vamos analisar o fragmento de código abaixo:

```
char c = 97;
printf("%d %c\n", c, c);
```

Considerando a codificação de caracteres via tabela ASCII, a variável *c*, que foi inicializada com o valor 97, representa o caractere *a*. A função *printf* imprime o conteúdo da variável *c* usando dois formatos distintos: com o especificador de formato para inteiro, *%d*, será impresso o valor do código numérico, 97; com o formato de caractere, *%c*, será impresso o caractere associado ao código, a letra *a*.

Conforme mencionamos, devemos evitar o uso explícito de códigos de caracteres. Para tanto, a linguagem C permite a escrita de *constantes caracteres*. Uma constante caractere é escrita envolvendo o caractere com aspas simples. Assim, a expressão *'a'* representa uma constante caractere e resulta no valor numérico associado ao caractere *a*. Podemos, então, reescrever o fragmento de código acima sem particularizar a tabela ASCII.

```
char c = 'a';
printf("%d %c\n", c, c);
```

Além de agregar portabilidade e clareza ao código, o uso de constantes caracteres nos livra de conhecermos os códigos associados a cada caractere. Independente da tabela de códigos numéricos utilizada, garante-se que os dígitos são codificados em sequência. Deste modo, se o dígito *zero* tem código 48, o dígito *um* tem obrigatoriamente código 49, e assim por diante. As letras minúsculas e as letras maiúsculas também formam dois grupos de códigos

seqüenciais. O exemplo a seguir tira proveito desta seqüência dos códigos de caracteres.

Exemplo. Suponhamos que queremos escrever uma função para testar se um caractere *c* é um *dígito* (um dos caracteres entre '0' e '9'). Esta função pode ter o protótipo:

```
int digito(char c);
```

e ter como resultado 1 (verdadeiro) se *c* for um dígito, e 0 (falso) se não for. A implementação desta função pode ser dada por:

```
int digito(char c)
{
    if ((c>='0') && (c<='9'))
        return 1;
    else
        return 0;
}
```

## 6.2. Cadeia de caracteres (*strings*)

Cadeias de caracteres (*strings*), em C, são representadas por vetores do tipo `char` terminadas, *obrigatoriamente*, pelo caractere nulo (`'\0'`). Portanto, para armazenarmos uma cadeia de caracteres, devemos reservar uma posição adicional para o caractere de fim da cadeia. Todas as funções que manipulam cadeias de caracteres (e a biblioteca padrão de C oferece várias delas) recebem como parâmetro um vetor de `char`, isto é, um ponteiro para o primeiro elemento do vetor que representa a cadeia, e processam caractere por caractere, até encontrarem o caractere nulo, que sinaliza o final da cadeia.

Por exemplo, o especificador de formato `%s` da função `printf` permite imprimir uma cadeia de caracteres. A função `printf` então recebe um vetor de `char` e imprime elemento por elemento, até encontrar o caractere nulo.

O código abaixo ilustra a representação de uma cadeia de caracteres. Como queremos representar a palavra `Rio`, composta por 3 caracteres, declaramos um vetor com dimensão 4 (um elemento adicional para armazenarmos o



caractere nulo no final da cadeia). O código preenche os elementos do vetor, incluindo o caractere '\0', e imprime a palavra na tela.

```
int main ( void )
{
char cidade[4];
cidade[0] = 'R';
cidade[1] = 'i';
cidade[2] = 'o';
cidade[3] = '\0';
printf("%s\n", cidade);
return 0;
}
```

Se o caractere '\0' não fosse colocado, a função printf executaria de forma errada, pois não conseguiria identificar o final da cadeia. Como as cadeias de caracteres são vetores, podemos reescrever o código acima inicializando os valores dos elementos do vetor na declaração:

```
int main ( void )
{
char cidade[ ] = {'R', 'i', 'o', '\0'};
printf("%s\n", cidade);
return 0;
}
```

A inicialização de cadeias de caracteres é tão comum em códigos C que a linguagem permite que elas sejam inicializadas escrevendo-se os caracteres entre aspas duplas. Neste caso, o caractere nulo é representado implicitamente. O código acima pode ser reescrito da seguinte forma:

```
int main ( void )
{
char cidade[ ] = "Rio";
printf("%s\n", cidade);
return 0;
}
```

A variável cidade é automaticamente dimensionada e inicializada com 4 elementos. Para ilustrar a declaração e inicialização de cadeias de caracteres, consideremos as declarações abaixo:

```
char s1[] = "";
char s2[] = "Rio de Janeiro";
char s3[81];
```

```
char s4[81] = "Rio";
```

Nestas declarações, a variável `s1` armazena uma cadeia de caracteres vazia, representada por um vetor com um único elemento, o caractere `\0`. A variável `s2` representa um vetor com 15 elementos. A variável `s3` representa uma cadeia de caracteres capaz de representar cadeias com até 80 caracteres, já que foi dimensionada com 81 elementos. Esta variável, no entanto, não foi inicializada e seu conteúdo é desconhecido. A variável `s4` também foi dimensionada para armazenar cadeias até 80 caracteres, mas seus primeiros quatro elementos foram atribuídos na declaração.

### 6.3. Leituras de caracteres e cadeias de caracteres

Para capturarmos o valor de um caractere simples fornecido pelo usuário via teclado, usamos a função `scanf`, com o especificador de formato `%c`.

```
char a;  
...  
scanf("%c", &a);  
...
```

Desta forma, se o usuário digitar a letra `r`, por exemplo, o código associado à letra `r` será armazenado na variável `a`. Vale ressaltar que, diferente dos especificadores `%d` e `%f`, o especificador `%c` não pula os caracteres brancos<sup>2</sup>. Portanto, se o usuário teclar um espaço antes da letra `r`, o código do espaço será capturado e a letra `r` será capturada apenas numa próxima chamada da função `scanf`. Se desejarmos pular todas as ocorrências de caracteres brancos que porventura antecedam o caractere que queremos capturar, basta incluir um espaço em branco no formato, antes do especificador.

```
char a;  
...  
scanf(" %c", &a); /* o branco no formato pula brancos da entrada */  
...
```

Já mencionamos que o especificador `%s` pode ser usado na função `printf` para imprimir uma cadeia de caracteres. O mesmo especificador pode ser utilizado para capturar cadeias de caracteres na função `scanf`. No entanto, seu uso é muito limitado. O especificador `%s` na função `scanf` pula os eventuais

caracteres brancos e captura a sequência de caracteres não brancos. Consideremos o fragmento de código abaixo:

```
char cidade[81];  
...  
scanf("%s", cidade);  
...
```

Devemos notar que não usamos o caractere & na passagem da cadeia para a função, pois a cadeia é um vetor (o nome da variável representa o endereço do primeiro elemento do vetor e a função atribui os valores dos elementos a partir desse endereço). O uso do especificador de formato %s na leitura é limitado, pois o fragmento de código acima funciona apenas para capturar nomes simples. Se o usuário digitar Rio de Janeiro, apenas a palavra Rio será capturada, pois o %s lê somente uma sequência de caracteres não brancos.

Em geral, queremos ler nomes compostos (nome de pessoas, cidades, endereços para correspondência, etc.). Para capturarmos estes nomes, podemos usar o especificador de formato %[...], no qual listamos entre os colchetes todos os caracteres que aceitaremos na leitura. Assim, o formato "%[aeiou]" lê sequências de vogais, isto é, a leitura prossegue até que se encontre um caractere que não seja uma vogal. Se o primeiro caractere entre colchetes for o acento circunflexo (^), teremos o efeito inverso (negação). Assim, com o formato "%[^aeiou]" a leitura prossegue enquanto uma vogal não for encontrada.

Esta construção permite capturarmos nomes compostos. Consideremos o código abaixo:

```
char cidade[81];  
...  
scanf(" %[^\\n]", cidade);  
...
```

A função scanf agora lê uma sequência de caracteres até que seja encontrado o caractere de mudança de linha (\\n). Em termos práticos, captura-se a linha fornecida pelo usuário até que ele tecele *Enter*. A inclusão do espaço no formato (antes do sinal %) garante que eventuais caracteres brancos que precedam o nome serão pulados.

Para finalizar, devemos salientar que o trecho de código acima é perigoso, pois, se o usuário fornecer uma linha que tenha mais de 80 caracteres, estaremos invadindo um espaço de memória que não está reservado (o vetor foi dimensionado com 81 elementos).

Para evitar esta possível invasão, podemos limitar o número máximo de caracteres que serão capturados.

```
char cidade[81];  
...  
scanf("%80[^\n]", cidade); /* lê no máximo 80 caracteres */  
...
```

#### ***6.4. Exemplos de funções que manipulam cadeias de caracteres***

Nesta seção, discutiremos a implementação de algumas funções que manipulam cadeias de caracteres.

Exemplo. Impressão caractere por caractere. Vamos inicialmente considerar a implementação de uma função que imprime uma cadeia de caracteres, caractere por caractere. A implementação pode ser dada por:

```
void imprime (char* s)  
{  
    int i;  
    for (i=0; s[i] != '\0'; i++)  
        printf("%c",s[i]);  
    printf("\n");  
}  
que teria funcionalidade análoga à utilização do especificador de formato %s.  
void imprime (char* s)  
{  
    printf("%s\n",s);  
}
```

*Exemplo. Comprimento da cadeia de caracteres.*

*Consideremos a implementação de uma função que recebe como parâmetro de entrada uma*

*cadeia de caracteres e fornece como retorno o número de caracteres existentes na cadeia.*

*O*

*protótipo da função pode ser dado por:*

*int comprimento (char\* s);*

*Para contar o número de caracteres da cadeia, basta contarmos o número de caracteres até*

*que o caractere nulo (que indica o fim da cadeia) seja encontrado. O caractere nulo em si não deve ser contado. Uma possível implementação desta função é:*

```
int comprimento (char* s)  
{  
  int i;  
  int n = 0; /* contador */  
  for (i=0; s[i] != '\0'; i++)  
    n++;  
  return n;  
}
```

O trecho de código abaixo faz uso da função acima.

```
#include <stdio.h>  
int comprimento (char* s);  
int main (void)  
{  
  int tam;  
  char cidade[] = "Rio de Janeiro";  
  tam = comprimento(cidade);  
  printf("A string \"%s\" tem %d caracteres\n", cidade, tam);  
  return 0;  
}
```

A saída deste programa será: A string "Rio de Janeiro" tem 14 caracteres. Salientamos o uso do caractere de escape `\` para incluir as aspas na saída.

Exemplo. Cópia de cadeia de caracteres. Vamos agora considerar a implementação de uma função para copiar os elementos de uma cadeia de caracteres para outra. Assumimos que a cadeia que receberá a cópia tem espaço suficiente para que a operação seja realizada. O protótipo desta função pode ser dado por:

```
void copia (char* dest, char* orig);  
A função copia os elementos da cadeia original (orig) para a cadeia de destino (dest).  
Uma possível implementação desta função é mostrada abaixo:  
void copia (char* dest, char* orig)  
{  
  int i;  
  for (i=0; orig[i] != '\0'; i++)
```

```
dest[i] = orig[i];  
/* fecha a cadeia copiada */  
dest[i] = '\0';  
}
```

Salientamos a necessidade de fechar a cadeia copiada após a cópia dos caracteres não nulos. Quando o laço do for terminar, a variável *i* terá o índice de onde está armazenado o caractere nulo na cadeia original. A cópia também deve conter o '\0' nesta posição.

Exemplo. Concatenação de cadeias de caracteres. Vamos considerar uma extensão do exemplo anterior e discutir a implementação de uma função para concatenar uma cadeia de caracteres com outra já existente. Isto é, os caracteres de uma cadeia são copiados no final da outra cadeia. Assim, se uma cadeia representa inicialmente a cadeia PUC e concatenarmos a ela a cadeia Rio, teremos como resultado a cadeia PUCRio. Vamos mais uma vez considerar que existe espaço reservado que permite fazer a cópia dos caracteres. O protótipo da função pode ser dado por:

```
void concatena (char*dest, char* orig);
```

Uma possível implementação desta função é mostrada a seguir:

```
void concatena (char*dest, char* orig)  
{  
    int i = 0; /* indice usado na cadeia destino, inicializado com zero */  
    int j; /* indice usado na cadeia origem */  
    /* acha o final da cadeia destino */  
    i = 0;  
    while (s[i] != '\0')  
        i++;  
    /* copia elementos da origem para o final do destino */  
    for (j=0; orig[j] != '\0'; j++)  
    {  
        dest[i] = orig[j];  
        i++;  
    }  
    /* fecha cadeia destino */  
    dest[i] = '\0';  
}
```

Funções análogas às funções comprimento, copia e concatenação são disponibilizadas pela biblioteca padrão de C. As funções da biblioteca padrão são, respectivamente, `strlen`, `strcpy` e `strcat`, que fazem parte da biblioteca de cadeias de caracteres (*strings*), `string.h`. Existem diversas outras funções que manipulam cadeias de caracteres nessa biblioteca. A razão de mostrarmos possíveis implementações destas funções como exercício é ilustrar a codificação da manipulação de cadeias de caracteres.

Exemplo 5: Duplicação de cadeias de caracteres. Consideremos agora um exemplo com alocação dinâmica. O objetivo é implementar uma função que receba como parâmetro uma cadeia de caracteres e forneça uma cópia da cadeia, alocada dinamicamente. O protótipo desta função pode ser dado por:

```
char* duplica (char* s);  
Uma possível implementação, usando as funções da biblioteca padrão, é:  
#include <stdlib.h>  
#include <string.h>  
char* duplica (char* s)  
{  
    int n = strlen(s);  
    char* d = (char*) malloc ((n+1)*sizeof(char));  
    strcpy(d,s);  
    return d;  
}
```

A função que chama `duplica` fica responsável por liberar o espaço alocado.

## 6.5. Funções recursivas

Uma cadeia de caracteres pode ser definida de forma recursiva. Podemos dizer que uma cadeia de caracteres é representada por:

É uma cadeia de caracteres vazia; ou  
É um caractere seguido de uma (sub) cadeia de caracteres.

Isto é, podemos dizer que uma cadeia `s` não vazia pode ser representada pelo seu primeiro caractere `s[0]` seguido da cadeia que começa no endereço do então segundo caractere, `&s[1]`. Vamos reescrever algumas das funções mostradas acima, agora com a versão recursiva.

Exemplo. Impressão caractere por caractere.

Uma versão recursiva da função para imprimir a cadeia caractere por caractere é mostrada a seguir. Como já foi discutido, uma implementação recursiva deve ser projetada considerando-se a definição recursiva do objeto, no caso uma cadeia de caracteres. Assim, a função deve primeiro testar se a condição da cadeia é vazia. Se a cadeia for vazia, nada precisa ser impresso; se não for vazia, devemos imprimir o primeiro caractere e então chamar uma função para imprimir a sub-cadeia que se segue. Para imprimir a sub-cadeia podemos usar a própria função, recursivamente.

```
void imprime_rec (char* s)
{
    if (s[0] != '\0')
    {
        printf("%c", s[0]);
        imprime_rec(&s[1]);
    }
}
```

Algumas implementações ficam bem mais simples se feitas recursivamente. Por exemplo, é simples alterar a função acima e fazer com que os caracteres da cadeia sejam impressos em ordem inversa, de trás para a frente: basta imprimir a sub-cadeia antes de imprimir o primeiro caractere.

```
void imprime_inv (char* s)
{
    if (s[0] != '\0')
    {
        imprime_inv(&s[1]);
        printf("%c", s[0]);
    }
}
```

Como exercício, sugerimos implementar a impressão inversa sem usar recursividade.

Exemplo. Comprimento da cadeia de caracteres. Uma implementação recursiva da função que retorna o número de caracteres existentes na cadeia é mostrada a seguir:

```
int comprimento_rec (char* s)
```



```
{
if (s[0] == '\0')
return 0;
else
return 1 + comprimento_rec(&s[1]);
}
```

Exemplo. Cópia de cadeia de caracteres. Vamos mostrar agora uma possível implementação recursiva da função copia mostrada anteriormente.

```
void copia_rec (char* dest, char* orig)
{
if (orig[0] == '\0')
dest[0] = '\0';
else {
dest[0] = orig[0];
copia_rec(&dest[1], &orig[1]);
}
}
```

É fácil verificar que o código acima pode ser escrito de forma mais compacta:

```
void copia_rec_2 (char* dest, char* orig)
{
dest[0] = orig[0];
if (orig[0] != '\0')
copia_rec_2(&dest[1], &orig[1]);
}
```

## 6.6. Constante cadeia de caracteres

Em códigos C, uma sequência de caracteres delimitada por aspas representa uma constante cadeia de caracteres, ou seja, uma expressão constante, cuja avaliação resulta no ponteiro onde a cadeia de caracteres está armazenada. Para exemplificar, vamos considerar o trecho de código abaixo:

```
#include <string.h>
int main ( void )
{
char cidade[4];
strcpy (cidade, "Rio" );
printf ( "%s \n", cidade );
}
```

```
return 0;  
}
```

De forma ilustrativa, o que acontece é que, quando o compilador encontra a cadeia "Rio", automaticamente é alocada na área de constantes a seguinte sequência de caracteres: 'R', 'i', 'o', '\0' e é fornecido o ponteiro para o primeiro elemento desta sequência. Assim, a função strcpy recebe dois ponteiros de cadeias: o primeiro aponta para o espaço associado à variável cidade e o segundo aponta para a área de constantes onde está armazenada a cadeia Rio.

Desta forma, também é válido escrever:

```
int main (void)  
{  
    char *cidade; /* declara um ponteiro para char */  
    cidade = "Rio"; /* cidade recebe o endereço da cadeia "Rio" */  
    printf( "%s \n", cidade );  
    return 0;  
}
```

Existe uma diferença sutil entre as duas declarações abaixo:

```
char s1[] = "Rio de Janeiro";  
char* s2 = "Rio de Janeiro";
```

Na primeira, declaramos um vetor de char local que é inicializado com a cadeia de caracteres Rio de Janeiro, seguido do caractere nulo. A variável s1 ocupa, portanto, 15 bytes de memória. Na segunda, declaramos um ponteiro para char que é inicializado com o endereço de uma área de memória onde a constante cadeia de caracteres Rio de Janeiro está armazenada. A variável s2 ocupa 4 bytes (espaço de um ponteiro). Podemos verificar esta diferença imprimindo os valores sizeof(s1) e sizeof(s2). Como s1 é um vetor local, podemos alterar o valor de seus elementos. Por exemplo, é válido escrever s1[0]='X'; alterando o conteúdo da cadeia para Xio de Janeiro. No entanto, não é válido escrever s2[0]='X'; pois estaríamos tentando alterar o conteúdo de uma área de constante.

## **6.7. Vetor de cadeia de caracteres**

Em muitas aplicações, desejamos representar um vetor de cadeia de caracteres. Por exemplo, podemos considerar uma aplicação que armazene os nomes de todos os alunos de uma turma num vetor. Sabemos que uma cadeia de caracteres é representada por um vetor do tipo `char`. Para representarmos um vetor onde cada elemento é uma cadeia de caracteres, devemos ter um vetor cujos elementos são do tipo `char*`, isto é, um vetor de ponteiros para `char`. Assim, criamos um conjunto (vetor) bidimensional de `char`.

Assumindo que o nome de nenhum aluno terá mais do que 80 caracteres e que o número máximo de alunos numa turma é 50, podemos declarar um vetor bidimensional para armazenar os nomes dos alunos `char alunos[50][81]`; Com esta variável declarada, `alunos[i]` acessa a cadeia de caracteres com o nome do (i+1)-ésimo aluno da turma e, conseqüentemente, `alunos[i][j]` acessa a (j+1)-ésima letra do nome do (i+1)-ésimo aluno. Considerando que `alunos` é uma variável global, uma função para imprimir os nomes dos `n` alunos de uma turma poderia ser dada por:

```
void imprime (int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%s\n", alunos[i]);
}
```

No próximo capítulo, que trata de matrizes, discutiremos conjuntos bidimensionais com mais detalhes. Para a representação de vetores de cadeias de caracteres, optamos, em geral, por declarar um vetor de ponteiros e alocar dinamicamente cada elemento (no caso, uma cadeia de caracteres). Desta forma, otimizamos o uso do espaço de memória, pois não precisamos achar uma dimensão máxima para todas as cadeias do vetor nem desperdiçamos espaço excessivo quando temos poucos nomes de alunos a serem armazenados. Cada elemento do vetor é um ponteiro. Se precisarmos armazenar um nome na posição, alocamos o espaço de memória necessário para armazenar a cadeia de caracteres correspondente.

Assim, nosso vetor com os nomes dos alunos pode ser declarado da seguinte forma:

```
#define MAX 50
```

```
char* alunos[MAX];
```

Exemplo. Leitura e impressão dos nomes dos alunos. Vamos escrever uma função que captura os nomes dos alunos de uma turma. A função inicialmente lê o número de alunos da turma (que deve ser menor ou igual a MAX) e captura os nomes fornecidos por linha, fazendo a alocação correspondente. Para escrever esta função, podemos pensar numa função auxiliar que captura uma linha e fornece como retorno uma cadeia alocada dinamicamente com a linha inserida. Fazendo uso das funções que escrevemos acima, podemos ter:

```
char* lelinha (void)
```

```
{  
char linha[121]; /* variavel auxiliar para ler linha */  
scanf(" %120[^\n]", linha);  
return duplica(linha);  
}
```

A função para capturar os nomes dos alunos preenche o vetor de nomes e pode ter como valor de retorno o número de nomes lidos:

```
int lenomes (char** alunos)
```

```
{  
int i;  
int n;  
do {  
scanf("%d",&n);  
} while (n>MAX);  
for (i=0; i<n; i++)  
alunos[i] = lelinha();  
return n;  
}
```

A função para liberar os nomes alocados na tabela pode ser implementada por:

```
void liberanomes (int n, char** alunos)
```

```
{  
int i;  
for (i=0; i<n; i++)  
free(alunos[i]);  
}
```

Uma função para imprimir os nomes dos alunos pode ser dada por:

```
void imprimenomes (int n, char** alunos)
```

```
{  
int i;  
for (i=0; i<n; i++)  
printf("%s\n", alunos[i]);  
}
```

*Um programa que faz uso destas funções é mostrado a seguir:*

```
int main (void)
{
    char* alunos[MAX];
    int n = lenomes(alunos);
    imprimenomes(n,alunos);
    liberanomes(n,alunos);
    return 0;
}
```

## **6.8. Parâmetros da função main**

Em todos os exemplos mostrados, temos considerado que a função principal, main, não recebe parâmetros. Na verdade, a função main pode ser definida para receber zero ou dois parâmetros, geralmente chamados argc e argv. O parâmetro argc recebe o número de argumentos passados para o programa quando este é executado; por exemplo, de um comando de linha do sistema operacional. O parâmetro argv é um vetor de cadeias de caracteres, que armazena os nomes passados como argumentos. Por exemplo, se temos um programa executável com o nome mensagem e se ele for invocado através da linha de comando:

> mensagem estruturas de dados

a variável argc receberá o valor 4 e o vetor argv será inicializado com os seguintes elementos: argv[0]="mensagem", argv[1]="estruturas", argv[2]="de", e argv[3]="dados". Isto é, o primeiro elemento armazena o próprio nome do executável e os demais são preenchidos com os nomes passados na linha de comando. Esses parâmetros podem ser úteis para, por exemplo, passar o nome de um arquivo de onde serão capturados os dados de um programa. A manipulação de arquivos será discutida mais adiante no curso. Por ora, mostramos um exemplo simples que trata os dois parâmetros da função main.

```
#include <stdio.h>
int main (int argc, char** argv)
{
    int i;
    for (i=0; i<argc; i++)
        printf("%s\n", argv[i]);
}
```



```
return 0;  
}
```

Se este programa tiver seu executável chamado de mensagem e for invocado com a linha de comando mostrada acima, a saída será:

```
mensagem  
estruturas  
de  
dados
```

## ***7. MATRIZES***

---

Já discutimos em capítulos anteriores a construção de conjuntos unidimensionais através do uso de vetores. A linguagem C também permite a

construção de conjuntos bi ou multidimensionais. Neste capítulo, discutiremos em detalhe a manipulação de matrizes, representadas por conjuntos bidimensionais de valores numéricos. As construções apresentadas aqui podem ser estendidas para conjuntos de dimensões maiores.

## 7.1. Alocação estática versus dinâmica

Antes de tratarmos das construções de matrizes, vamos recapitular alguns conceitos apresentados com vetores. A forma mais simples de declararmos um vetor de inteiros em C é mostrada a seguir:

```
int v[10];
```

ou, se quisermos criar uma constante simbólica para a dimensão:

```
#define N 10  
int v[N];
```

Podemos dizer que, nestes casos, os vetores são declarados *estaticamente*. A variável que representa o vetor é uma *constante* que armazena o endereço ocupado pelo primeiro elemento do vetor. Esses vetores podem ser declarados como variáveis globais ou dentro do corpo de uma função. Se declarado dentro do corpo de uma função, o vetor existirá apenas enquanto a função estiver sendo executada, pois o espaço de memória para o vetor é reservado na pilha de execução. Portanto, não podemos fazer referência ao espaço de memória de um vetor local de uma função que já retornou.

O problema de declararmos um vetor estaticamente, seja como variável global ou local, é que precisamos saber de antemão a dimensão máxima do vetor. Usando alocação dinâmica, podemos determinar a dimensão do vetor em tempo de execução:

```
int* v;  
í  
v = (int*) malloc(n * sizeof(int));
```

Neste fragmento de código, *n* representa uma variável com a dimensão do vetor, determinada em tempo de execução (podemos, por exemplo, capturar o valor de *n* fornecido pelo usuário). Após a alocação dinâmica, acessamos os

elementos do vetor da mesma forma que os elementos de vetores criados estaticamente. Outra diferença importante: com alocação dinâmica, declaramos uma variável do tipo ponteiro que posteriormente recebe o valor do endereço do primeiro elemento do vetor, alocado dinamicamente. A área de memória ocupada pelo vetor permanece válida até que seja explicitamente liberada (através da função `free`). Portanto, mesmo que um vetor seja criado dinamicamente dentro da função, podemos acessá-lo depois da função ser finalizada, pois a área de memória ocupada por ele permanece válida, isto é, o vetor não está alocado na pilha de execução. Usamos esta propriedade quando escrevemos a função que duplica uma cadeia de caracteres (*string*): a função duplica aloca um vetor de char dinamicamente, preenche seus valores e retorna o ponteiro, para que a função que chama possa acessar a nova cadeia de caracteres.

A linguagem C oferece ainda um mecanismo para re-alocarmos um vetor dinamicamente. Em tempo de execução, podemos verificar que a dimensão inicialmente escolhida para um vetor tornou-se insuficiente (ou excessivamente grande), necessitando um re-dimensionamento. A função `realloc` da biblioteca padrão nos permite re-alocar um vetor, preservando o conteúdo dos elementos, que permanecem válidos após a re-alocação (no fragmento de código abaixo, `m` representa a nova dimensão do vetor).

```
v = (int*) realloc(v, m*sizeof(int));
```

Vale salientar que, sempre que possível, optamos por trabalhar com vetores criados estaticamente. Eles tendem a ser mais eficientes, já que os vetores alocados dinamicamente têm uma indireção a mais (primeiro acessa-se o valor do endereço armazenado na variável ponteiro para então acessar o elemento do vetor).

## **7.2. Vetores bidimensionais ó Matrizes**

A linguagem C permite a criação de vetores bidimensionais, declarados estaticamente. Por exemplo, para declararmos uma matriz de valores reais com 4 linhas e 3 colunas, fazemos:

```
float mat[4][3];
```



Esta declaração reserva um espaço de memória necessário para armazenar os 12 elementos da matriz, que são armazenados de maneira contínua, organizados linha a linha.

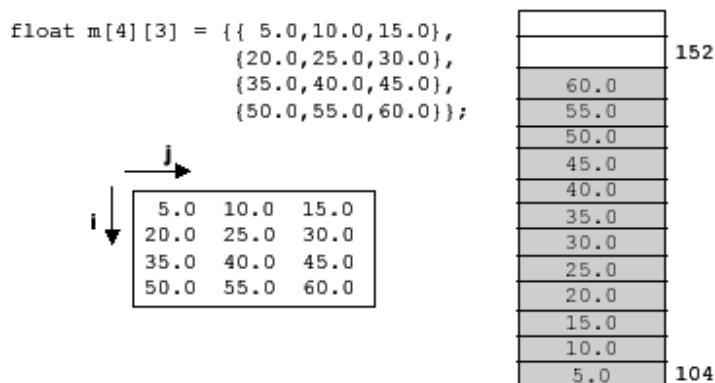


Figura 7.1: Alocação dos elementos de uma matriz.

Os elementos da matriz são acessados com indexação dupla: `mat[i][j]`. O primeiro índice, `i`, acessa a linha e o segundo, `j`, acessa a coluna. Como em C a indexação começa em zero, o elemento da primeira linha e primeira coluna é acessado por

`mat[0][0]`. Após a declaração estática de uma matriz, a variável que representa a matriz, `mat` no exemplo acima, representa um ponteiro para o primeiro vetor-linha, composto por 3 elementos. Com isto, `mat[1]` aponta para o primeiro elemento do segundo vetor-linha, e assim por diante.

As matrizes também podem ser inicializadas na declaração:

```
float mat[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
```

Ou podemos inicializar sequencialmente:

```
float mat[4][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

O número de elementos por linha pode ser omitido numa inicialização, mas o número de colunas deve, obrigatoriamente, ser fornecido:

```
float mat[][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

### 7.3. Passagem de matrizes para funções

Conforme dissemos acima, uma matriz criada estaticamente é representada por um ponteiro para um vetor-linha com o número de elementos da linha. Quando passamos uma matriz para uma função, o parâmetro da função deve ser deste tipo. Infelizmente, a sintaxe para representar este tipo é obscura. O protótipo de uma função que recebe a matriz declarada acima seria:

```
void f(..., float (*mat)[3], ...);
```

Uma segunda opção é declarar o parâmetro como matriz, podendo omitir o número de linhas:

```
void f(..., float mat[][3], ...);
```

De qualquer forma, o acesso aos elementos da matriz dentro da função é feito da forma usual, com indexação dupla.

Na próxima seção, examinaremos formas de trabalhar com matrizes alocadas dinamicamente. No entanto, vale salientar que recomendamos, sempre que possível, o uso de matrizes alocadas estaticamente. Em diversas aplicações, as matrizes têm dimensões fixas e não justificam a criação de estratégias para trabalhar com alocação dinâmica. Em aplicações da área de Computação Gráfica, por exemplo, é comum trabalharmos com matrizes de 4 por 4 para representar transformações geométricas e projeções. Nestes casos, é muito mais simples definirmos as matrizes estaticamente (`float mat[4][4];`), uma vez que sabemos de antemão as dimensões a serem usadas. Nestes casos, vale a pena definirmos um tipo próprio, pois nos livramos das construções sintáticas confusas explicitadas acima. Por exemplo, podemos definir o tipo `Matrix4`.

```
typedef float Matrix4[4][4];
```

Com esta definição podemos declarar variáveis e parâmetros deste tipo:

```
Matrix4 m; /* declaração de variável */
```

```
...
```

```
void f(..., Matrix4 m, ...); /* especificação de parâmetro */
```

## 7.4. Matrizes dinâmicas

As matrizes declaradas estaticamente sofrem das mesmas limitações dos vetores: precisamos saber de antemão suas dimensões. O problema que encontramos é que a linguagem C só permite alocarmos dinamicamente conjuntos unidimensionais. Para trabalharmos com matrizes alocadas dinamicamente, temos que criar abstrações conceituais com vetores para representar conjuntos bidimensionais. Nesta seção, discutiremos duas estratégias distintas para representar matrizes alocadas dinamicamente.

### Matriz representada por um vetor simples

Conceitualmente, podemos representar uma matriz num vetor simples. Reservamos as primeiras posições do vetor para armazenar os elementos da primeira linha, seguidos dos elementos da segunda linha, e assim por diante. Como, de fato, trabalharemos com um vetor unidimensional, temos que criar uma disciplina para acessar os elementos da matriz, representada conceitualmente. A estratégia de endereçamento para acessar os elementos é a seguinte: se quisermos acessar o que seria o elemento  $\text{mat}[i][j]$  de uma matriz, devemos acessar o elemento  $v[i*n+j]$ , onde  $n$  representa o número de colunas da matriz.

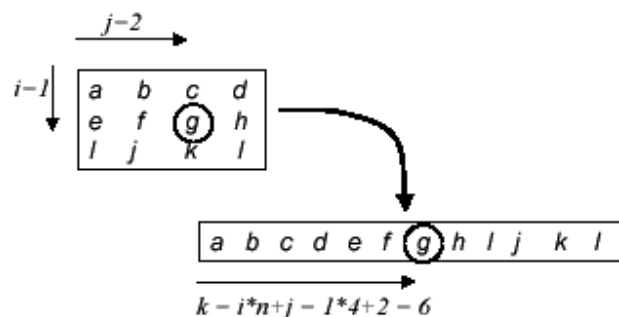


Figura 7.2: Matriz representada por vetor simples.

Esta conta de endereçamento é intuitiva: se quisermos acessar elementos da terceira ( $i=2$ ) linha da matriz, temos que pular duas linhas de elementos ( $i*n$ ) e depois indexar o elemento da linha com  $j$ . Com esta estratégia, a alocação da matriz recai numa alocação de vetor que tem  $m*n$  elementos, onde  $m$  e  $n$  representam as dimensões da matriz.

```
float *mat; /* matriz representada por um vetor */
...
mat = (float*) malloc(m*n*sizeof(float));
...
```

No entanto, somos obrigados a usar uma notação desconfortável,  $v[i*n+j]$ , para acessar os elementos, o que pode deixar o código pouco legível.

## Matriz representada por um vetor de ponteiros

Nesta segunda estratégia, faremos algo parecido com o que fizemos para tratar vetores de cadeias de caracteres, que em C são representados por conjuntos bidimensionais de caracteres. De acordo com esta estratégia, cada linha da matriz é representada por um vetor independente. A matriz é então representada por um vetor de vetores, ou vetor de ponteiros, no qual cada elemento armazena o endereço do primeiro elemento de cada linha. A figura abaixo ilustra o arranjo da memória utilizada nesta estratégia.

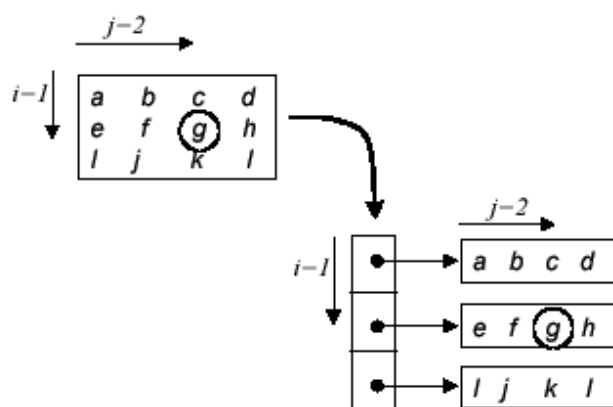


Figura 7.3: Matriz com vetor de ponteiros.

A alocação da matriz agora é mais elaborada. Primeiro, temos que alocar o vetor de ponteiros. Em seguida, alocamos cada uma das linhas da matriz, atribuindo seus endereços aos elementos do vetor de ponteiros criado. O fragmento de código abaixo ilustra esta codificação:

```
int i;
float **mat; /* matriz representada por um vetor de ponteiros */
...
mat = (float**) malloc(m*sizeof(float*));
for (i=0; i<m; i++)
    mat[i] = (float*) malloc(n*sizeof(float));
```

A grande vantagem desta estratégia é que o acesso aos elementos é feito da mesma forma que quando temos uma matriz criada estaticamente, pois, se mat

representa uma matriz alocada segundo esta estratégia, `mat[i]` representa o ponteiro para o primeiro elemento da linha `i`, e, conseqüentemente, `mat[i][j]` acessa o elemento da coluna `j` da linha `i`.

A liberação do espaço de memória ocupado pela matriz também exige a construção de um laço, pois temos que liberar cada linha antes de liberar o vetor de ponteiros:

```
...  
for (i=0; i<m; i++)  
    free(mat[i]);  
free(mat);
```

Devemos notar que, neste caso, a complexidade adicional na alocação da matriz nos permitiu acessar e atribuir os elementos usando a sintaxe convencional de acesso a conjuntos bidimensionais.

Exercício: Implemente duas versões, seguindo as diferentes estratégias de alocar matrizes discutidas, de uma função para determinar se uma matriz é ou não simétrica quadrada.

### 7.5. Representação de matrizes simétricas

Em uma matriz simétrica  $n$  por  $n$ , não há necessidade, no caso de `i!j`, de armazenar ambos os elementos `mat[i][j]` e `mat[j][i]`, porque os dois têm o mesmo valor. Portanto, basta guardar os valores dos elementos da diagonal e de metade dos elementos restantes ó por exemplo, os elementos abaixo da diagonal, para os quais  $i > j$ . Ou seja, podemos fazer uma economia de espaço usado para alocar a matriz. Em vez de  $n^2$  valores, podemos armazenar apenas  $s$  elementos, sendo  $s$  dado por:

$$s = n + \frac{(n^2 - n)}{2} = \frac{n(n+1)}{2}$$

Podemos também determinar  $s$  como sendo a soma de uma progressão aritmética, pois temos que armazenar um elemento da primeira linha, dois elementos da segunda, três da terceira, e assim por diante.

$$s = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

A representação de matrizes com essa economia de memória também pode ser feita com um vetor simples ou um vetor de ponteiros. A seguir, discutimos a

implementação de duas funções: uma para criar uma matriz quadrada simétrica e outra para, dada uma matriz já criada, acessar seus elementos.

## Matriz simétrica com vetor simples

A função para criar a matriz dinamicamente usando um vetor simples não apresenta nenhuma dificuldade, pois basta dimensionarmos o vetor com apenas  $s$  elementos. Uma função para realizar essa tarefa é mostrada a seguir. Note que a matriz é obrigatoriamente quadrada e, portanto, só precisamos passar uma dimensão.

```
float* cria (int n)
{
    int s = n*(n+1)/2;
    float* mat = (float*) malloc(s*sizeof(float));
    return mat;
}
```

O acesso aos elementos da matriz deve ser feito como se estivéssemos representando a matriz inteira. Se for um acesso a um elemento acima da diagonal ( $i < j$ ), o valor de retorno é o elemento simétrico da parte inferior, que está devidamente representado.

Dessa forma, isolamos dentro do código que manipula a matriz diretamente o fato da matriz não estar explicitamente toda armazenada. Assim, através dessa função, podemos escrever outras funções que operam sobre matrizes simétricas sem nos preocuparmos sobre a forma de representação interna dos elementos.

O endereçamento de um elemento da parte inferior da matriz é feito saltando-se os elementos das linhas superiores. Assim, se desejarmos acessar um elemento da quinta linha ( $i=4$ ), devemos saltar  $1+2+3+4$  elementos, isto é, devemos saltar  $1+2+\dots+i$  elementos, ou seja,  $i*(i+1)/2$  elementos. Depois, usamos o índice  $j$  para acessar a coluna.

Como estamos projetando uma função que acessa os elementos da matriz, podemos fazer uma teste adicional para evitar acessos inválidos: verificar se os índices realmente representam elementos da matriz. A função que acessa um elemento da matriz é dada a seguir.

```
float acessa (int n, float* mat, int i, int j)
{
    int k; /* índice do elemento no vetor */
    if (i<0 || i>=n || j<0 || j>=n) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    if (i>=j)
        k = i*(i+1)/2 + j;
    else
        k = j*(j+1)/2 + i;
    return mat[k];
}
```

### Matriz simétrica com vetor de ponteiros

A estratégia de trabalhar com vetores de ponteiros para matrizes alocadas dinamicamente é muito adequada para a representação matrizes simétricas. Numa matriz simétrica, para otimizar o uso da memória, armazenamos apenas a parte triangular inferior da matriz. Isto significa que a primeira linha será representada por um vetor de um único elemento, a segunda linha será representada por um vetor de dois elementos e assim por diante. Como o uso de um vetor de ponteiros trata as linhas como vetores independentes, a adaptação desta estratégia para matrizes simétricas fica simples.

Para criar a matriz, basta alocarmos um número variável de elementos para cada linha. O código abaixo ilustra uma possível implementação:

```
Float** cria (int n)
{
    int i;
    float* mat = (float**) malloc(n*sizeof(float*));
    for (i=0; i<n; i++)
        mat[i] = (float*) malloc((i+1)*sizeof(float));
    return mat;
}
```

O acesso aos elementos é natural, desde que tenhamos o cuidado de não acessar diretamente elementos que não estejam explicitamente alocados (isto é, elementos com  $i < j$ ).



```
float acessa (int n, float* mat, int i, int j)
{
    if (i<0 || i>=n || j<0 || j>=n) {
        printf("Acesso inválido!\n");
        exit(1);
    }
    if (i>=j)
        return mat[i][j];
    else
        return mat[j][i];
}
```

Finalmente, observamos que exatamente as mesmas técnicas poderiam ser usadas para representar uma matriz *õtriangularõ*, isto é, uma matriz cujos elementos acima (ou abaixo) da diagonal são todos nulos. Neste caso, a principal diferença seria na função *acessa*, que teria como resultado o valor zero em um dos lados da diagonal, em vez acessar o valor simétrico.

Exercício: Escreva códigos para representar uma matriz triangular inferior.

Exercício: Escreva códigos para representar uma matriz triangular superior.



## ***8. TIPOS ESTRUTURADOS***

---

Na linguagem C, existem os tipos básicos (char, int, float, etc.) e seus respectivos ponteiros que podem ser usados na declaração de variáveis. Para estruturar dados complexos, nos quais as informações são compostas por diversos campos, necessitamos de mecanismos que nos permitam agrupar tipos distintos. Neste capítulo, apresentaremos os mecanismos fundamentais da linguagem C para a estruturação de tipos.

### ***8.1. O tipo estrutura***

Em C, podemos definir um tipo de dado cujos campos são compostos de vários valores de tipos mais simples. Para ilustrar, vamos considerar o desenvolvimento de programas que manipulam pontos no plano cartesiano. Cada ponto pode ser representado por suas coordenadas  $x$  e  $y$ , ambas dadas por valores reais. Sem um mecanismo para agrupar as duas componentes, teríamos que representar cada ponto por duas variáveis independentes.

```
float x;  
float y;
```

No entanto, deste modo, os dois valores ficam dissociados e, no caso do programa manipular vários pontos, cabe ao programador não misturar a coordenada  $x$  de um ponto com a coordenada  $y$  de outro. Para facilitar este trabalho, a linguagem C oferece recursos para agruparmos dados. Uma estrutura, em C, serve basicamente para agrupar diversas variáveis dentro de um único contexto. No nosso exemplo, podemos definir uma estrutura ponto que contenha as duas variáveis. A sintaxe para a definição de uma estrutura é mostrada abaixo:

```
struct ponto {  
float x;  
float y;  
};
```

Desta forma, a estrutura ponto passa a ser um tipo e podemos então declarar variáveis deste tipo.

```
struct ponto p;
```

Esta linha de código declara  $p$  como sendo uma variável do tipo struct ponto. Os elementos de uma estrutura podem ser acessados através do operador de acesso a ponto ( $\cdot$ ). Assim, é válido escrever:

```
ponto.x = 10.0;  
ponto.y = 5.0;
```

Manipulamos os elementos de uma estrutura da mesma forma que variáveis simples. Podemos acessar seus valores, atribuir-lhes novos valores, acessar seus endereços, etc. Exemplo: Capturar e imprimir as coordenadas de um ponto.

Para exemplificar o uso de estruturas em programas, vamos considerar um exemplo simples em que capturamos e imprimimos as coordenadas de um ponto qualquer.

```
/* Captura e imprime as coordenadas de um ponto qualquer */  
#include <stdio.h>  
struct ponto {  
float x;  
float y;  
};
```

```
int main (void)
{
    struct ponto p;
    printf("Digite as coordenadas do ponto(x y): ");
    scanf("%f%f", &p.x, &p.y);
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
    return 0;
}
```

A variável `p`, definida dentro de `main`, é uma variável local como outra qualquer. Quando a declaração é encontrada, aloca-se, na pilha de execução, um espaço para seu armazenamento, isto é, um espaço suficiente para armazenar todos os campos da estrutura (no caso, dois números reais). Notamos que o acesso ao endereço de um campo da estrutura é feito da mesma forma que com variáveis simples: basta escrever `&(p.x)`, ou simplesmente `&p.x`, pois o operador de acesso ao campo da estrutura tem precedência sobre o operador de endereço.

## 8.2. Ponteiro para estruturas

Da mesma forma que podemos declarar variáveis do tipo estrutura:

```
struct ponto p;
```

podemos também declarar variáveis do tipo ponteiro para estrutura:

```
struct ponto *pp;
```

Se a variável `pp` armazenar o endereço de uma estrutura, podemos acessar os campos dessa estrutura indiretamente, através de seu ponteiro:

```
(*pp).x = 12.0;
```

Neste caso, os parênteses são indispensáveis, pois o operador de conteúdo de tem precedência menor que o operador de acesso. O acesso de campos de estruturas é tão comum em programas C que a linguagem oferece outro operador de acesso, que permite acessar campos a partir do ponteiro da estrutura. Este operador é composto por um traço seguido de um sinal de maior, formando uma seta (`->`). Portanto, podemos reescrever a atribuição anterior fazendo:

`pp->x = 12.0;`

Em resumo, se temos uma variável estrutura e queremos acessar seus campos, usamos o operador de acesso ponto (`p.x`); se temos uma variável ponteiro para estrutura, usamos o operador de acesso seta (`pp->x`). Seguindo o raciocínio, se temos o ponteiro e queremos acessar o endereço de um campo, fazemos `&pp->x!`

### ***8.3. Passagem de estruturas para funções***

Para exemplificar a passagem de variáveis do tipo estrutura para funções, podemos reescrever o programa simples, mostrado anteriormente, que captura e imprime as coordenadas de um ponto qualquer. Inicialmente, podemos pensar em escrever uma função que imprima as coordenadas do ponto. Esta função poderia ser dada por:

```
void imprime (struct ponto p)
{
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
}
```

A passagem de estruturas para funções se processa de forma análoga à passagem de variáveis simples, porém exige uma análise mais detalhada. Da forma como está escrita no código acima, a função recebe uma estrutura inteira como parâmetro. Portanto, faz-se uma cópia de toda a estrutura para a pilha e a função acessa os dados desta cópia.

Existem dois pontos a serem ressaltados. Primeiro, como em toda passagem por valor, a função não tem como alterar os valores dos elementos da estrutura original (na função `imprime` isso realmente não é necessário, mas seria numa função de leitura). O segundo ponto diz respeito à eficiência, visto que copiar uma estrutura inteira para a pilha pode ser uma operação custosa (principalmente se a estrutura for muito grande). É mais conveniente passar apenas o ponteiro da estrutura, mesmo que não seja necessário alterar os valores dos elementos dentro da função, pois copiar um ponteiro para a pilha é muito mais eficiente do que copiar uma estrutura inteira. Um ponteiro ocupa em geral 4 bytes, enquanto uma estrutura pode ser definida com um tamanho

muito grande. Desta forma, uma segunda (e mais adequada) alternativa para escrevermos a função imprime é:

```
void imprime (struct ponto* pp)
{
    printf("O ponto fornecido foi: (%.2f,%.2f)\n", pp->x, pp->y);
}
```

Podemos ainda pensar numa função para ler a hora do evento. Observamos que, neste caso, obrigatoriamente devemos passar o ponteiro da estrutura, caso contrário não seria possível passar ao programa principal os dados lidos:

```
void captura (struct ponto* pp)
{
    printf("Digite as coordenadas do ponto(x y): ");
    scanf("%f%f", &p->x, &p->y);
}
Com estas funções, nossa função main ficaria como mostrado abaixo.
int main (void)
{
    struct ponto p;
    captura(&p);
    imprime(&p);
    return 0;
}
```

Exercício: Função para determinar a distância entre dois pontos. Considere a implementação de uma função que tenha como valor de retorno a distância entre dois pontos. O protótipo da função pode ser dado por:

float distancia (struct ponto \*p, struct ponto \*q);

Nota: A distância entre dois pontos é dada por:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

## 8.4. Alocação dinâmica de estruturas

Da mesma forma que os vetores, as estruturas podem ser alocadas dinamicamente. Por exemplo, é válido escrever:

```
struct ponto* p;
```

```
p = (struct ponto*) malloc (sizeof(struct ponto));
```

Neste fragmento de código, o tamanho do espaço de memória alocado dinamicamente é dado pelo operador `sizeof` aplicado sobre o tipo estrutura (`sizeof(struct ponto)`). A função `malloc` retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura `ponto`.

Após uma alocação dinâmica, podemos acessar normalmente os campos da estrutura, através da variável ponteiro que armazena seu endereço:

```
...  
p->x = 12.0;  
...
```

## 8.5. Definição de "novos" tipos

A linguagem C permite criar nomes de tipos. Por exemplo, se escrevermos:

```
typedef float Real;
```

podemos usar o nome `Real` como um mnemônico para o tipo `float`. O uso de `typedef` é muito útil para abreviarmos nomes de tipos e para tratarmos tipos complexos. Alguns exemplos válidos de `typedef`:

```
typedef unsigned char UChar;  
typedef int* PInt;  
typedef float Vetor[4];
```

Neste fragmento de código, definimos `UChar` como sendo o tipo `char` sem sinal, `PInt` como um tipo ponteiro para `int`, e `Vetor` como um tipo que representa um vetor de quatro elementos. A partir dessas definições, podemos declarar variáveis usando estes mnemônicos:

```
Vetor v;  
...  
v[0] = 3;
```

Em geral, definimos nomes de tipos para as estruturas com as quais nossos programas trabalham. Por exemplo, podemos escrever:

```
struct ponto {
```

```
float x;  
float y;  
};  
typedef struct ponto Ponto;
```

Neste caso, Ponto passa a representar nossa estrutura de ponto. Também podemos definir um nome para o tipo ponteiro para a estrutura.

```
typedef struct ponto *PPonto;
```

Podemos ainda definir mais de um nome num mesmo typedef. Os dois typedef anteriores poderiam ser escritos por:

```
typedef struct ponto Ponto, *PPonto;
```

A sintaxe de um typedef pode parecer confusa, mas é equivalente à da declaração de variáveis. Por exemplo, na definição abaixo:

```
typedef float Vector[4];
```

se omitíssemos a palavra typedef, estaríamos declarando a variável Vector como sendo um vetor de 4 elementos do tipo float. Com typedef, estamos definindo um nome que representa o tipo vetor de 4 elementos float. De maneira análoga, na definição:

```
typedef struct ponto Ponto, *PPonto;
```

se omitíssemos a palavra typedef, estaríamos declarando a variável Ponto como sendo do tipo struct ponto e a variável PPonto como sendo do tipo ponteiro para struct ponto.

Por fim, vale salientar que podemos definir a estrutura e associar mnemônicos para elas em um mesmo comando:

```
typedef struct ponto {  
float x;  
float y;  
} Ponto, *PPonto;
```

É comum os programadores de C usarem nomes com as primeiras letras maiúsculas na definição de tipos. Isso não é uma obrigatoriedade, apenas um estilo de codificação.

## 8.6. Vetores de estruturas

Já discutimos o uso de vetores para agrupar elementos dos tipos básicos (vetores de inteiros, por exemplo). Nesta seção, vamos discutir o uso de vetores de estruturas, isto é, vetores cujos elementos são estruturas. Para ilustrar a discussão, vamos considerar o cálculo da área de um polígono plano qualquer delimitado por uma seqüência de  $n$  pontos. A área desse polígono pode ser calculada somando-se as áreas dos trapézios formados pelos lados do polígono e o eixo  $x$ , conforme ilustra a Figura 8.1.

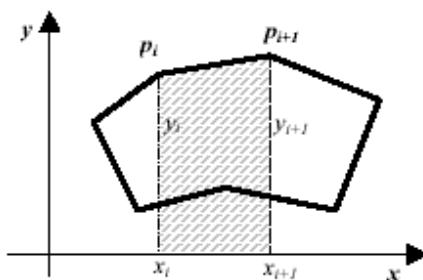


Figura 8.1: Cálculo da área de um polígono.

Na figura, ressaltamos a área do trapézio definido pela aresta que vai do ponto  $p_i$  ao ponto  $p_{i+1}$ . A área desse trapézio é dada por:  $a = (x_{i+1} - x_i)(y_{i+1} + y_i)/2$ . Somando-se as áreas (algumas delas negativas) dos trapézios definidos por todas as arestas chega-se a área do polígono (as áreas externas ao polígono são anuladas). Se a seqüência de pontos que define o polígono for dada em sentido anti-horário, chega-se a uma área de valor negativo. Neste caso, a área do polígono é o valor absoluto do resultado da soma.

Um vetor de estruturas pode ser usado para definir um polígono. O polígono passa a ser representado por uma seqüência de pontos. Podemos, então, escrever uma função para calcular a área de um polígono, dados o número de



pontos e o vetor de pontos que o representa. Uma implementação dessa função é mostrada abaixo.

```
float area (int n, Ponto* p)
{
    int i, j;
    float a = 0;
    for (i=0; i<n; i++) {
        j = (i+1) % n; /* próximo índice (incremento circular) */
        a += (p[j].x-p[i].x)*(p[i].y + p[j].y)/2;
    }
    if (a < 0)
        return -a;
    else
        return a;
}
```

*Um exemplo de uso dessa função é mostrado no código abaixo:*

```
int main (void)
{
    Ponto p[3] = {{1.0,1.0},{5.0,1.0},{4.0,3.0}};
    printf("area = %f\n",area (3,p));
    return 0;
}
```

Exercício: Altere o programa acima para capturar do teclado o número de pontos que delimitam o polígono. O programa deve, então, alocar dinamicamente o vetor de pontos, capturar as coordenadas dos pontos e, chamando a função área, exibir o valor da área.

## **8.7. Vetores de ponteiros para estruturas**

Da mesma forma que podemos declarar vetores de estruturas, podemos também declarar vetores de ponteiros para estruturas. O uso de vetores de ponteiros é útil quando temos que tratar um conjunto de elementos complexos. Para ilustrar o uso de estruturas complexas, consideremos um exemplo em que desejamos armazenar uma tabela com dados de alunos. Podemos organizar os dados dos alunos em um vetor. Para cada aluno, vamos supor que sejam necessárias as seguintes informações:

*Énome:* cadeia com até 80 caracteres



É*matricula*: número inteiro

É*endereço*: cadeia com até 120 caracteres

É*telefone*: cadeia com até 20 caracteres

Para estruturar esses dados, podemos definir um tipo que representa os dados de um aluno:

```
struct aluno {  
    char nome[81];  
    int mat;  
    char end[121];  
    char tel[21];  
};  
typedef struct aluno Aluno;
```

Vamos montar a tabela de alunos usando um vetor global com um número máximo de alunos. Uma primeira opção é declarar um vetor de estruturas:

```
#define MAX 100  
Aluno tab[MAX];
```

Desta forma, podemos armazenar nos elementos do vetor os dados dos alunos que queremos organizar. Seria válido, por exemplo, uma atribuição do tipo:

```
...  
tab[i].mat = 9912222;
```

No entanto, o uso de vetores de estruturas tem, neste caso, uma grande desvantagem. O tipo `Aluno` definido acima ocupa pelo menos 227 ( $=81+4+121+21$ ) bytes<sup>1</sup>. A declaração de um vetor desta estrutura representa um desperdício significativo de memória, pois provavelmente estaremos armazenando de fato um número de alunos bem inferior ao máximo estimado. Para contornar este problema, podemos trabalhar com um vetor de ponteiros.

```
typedef struct aluno *PAluno;  
#define MAX 100  
PAluno tab[MAX];
```

Assim, cada elemento do vetor ocupa apenas o espaço necessário para armazenar um ponteiro. Quando precisarmos alocar os dados de um aluno numa determinada posição do vetor, alocamos dinamicamente a estrutura `Aluno` e guardamos seu endereço no vetor de ponteiros.

Considerando o vetor de ponteiros declarado acima como uma variável global, podemos ilustrar a implementação de algumas funcionalidades para manipular nossa tabela de alunos. Inicialmente, vamos considerar uma função de inicialização. Uma posição do vetor estará vazia, isto é, disponível para armazenar informações de um novo aluno, se o valor do seu elemento for o ponteiro nulo. Portanto, numa função de inicialização, podemos atribuir NULL a todos os elementos da tabela, significando que temos, a princípio, uma tabela vazia.

```
void inicializa (void)
{
    int i;
    for (i=0; i<MAX; i++)
        tab[i] = NULL;
}
```

Uma segunda funcionalidade que podemos prever armazena os dados de um novo aluno numa posição do vetor. Vamos considerar que os dados serão fornecidos via teclado e que uma posição onde os dados serão armazenados será passada para a função. Se a posição da tabela estiver vazia, devemos alocar uma nova estrutura; caso contrário, atualizamos a estrutura já apontada pelo ponteiro.

```
void preenche (int i)
{
    if (tab[i]==NULL)
        tab[i] = (PAluno)malloc(sizeof(Aluno));
    printf("Entre com o nome:");
    scanf(" %80[^\n]", tab[i]->nome);
    printf("Entre com a matricula:");
    scanf("%d", &tab[i]->mat);
    printf("Entre com o endereco:");
    scanf(" %120[^\n]", tab[i]->end);
    printf("Entre com o telefone:");
    scanf(" %20[^\n]", tab[i]->tel);
}
```

Podemos também prever uma função para remover os dados de um aluno da tabela. Vamos considerar que a posição da tabela a ser liberada será passada para a função:

```
void remove (int i)
{
    if (tab[i] != NULL)
    {
        free(tab[i]);
        tab[i] = NULL;
    }
}
```

Para consultarmos os dados, vamos considerar uma função que imprime os dados armazenados numa determinada posição do vetor:

```
void imprime (int i)
{
    if (tab[i] != NULL)
    {
        printf("Nome: %s\n", tab[i]->nome);
        printf("Matrícula: %d\n", tab[i]->mat);
        printf("Endereço: %s\n", tab[i]->end);
        printf("Telefone: %s\n", tab[i]->tel);
    }
}
```

Por fim, podemos implementar uma função que imprima os dados de todos os alunos da tabela:

```
void imprime_tudo (void)
{
    int i;
    for (i=0; i<MAX; i++)
        imprime(i);
}
```

Exercício. Faça um programa que utilize as funções da tabela de alunos escritas acima.

Exercício. Re-escreva as funções acima sem usar uma variável global. Sugestão: Crie um tipo Tabela e faça as funções receberem este tipo como primeiro parâmetro.

## 8.8. Tipo *união*

Em C, uma *união* é uma localização de memória que é compartilhada por diferentes variáveis, que podem ser de tipos diferentes. As uniões são usadas quando queremos armazenar valores heterogêneos num mesmo espaço de memória. A definição de uma união é parecida com a de uma estrutura:

*union exemplo*

```
{  
int i;  
char c;  
}
```

Análogo à estrutura, este fragmento de código não declara nenhuma variável, apenas define o tipo união. Após uma definição, podemos declarar variáveis do tipo união:

*union exemplo v;*

Na variável *v*, os campos *i* e *c* compartilham o mesmo espaço de memória. A variável ocupa pelo menos o espaço necessário para armazenar o maior de seus campos (um inteiro, no caso). O acesso aos campos de uma união é análogo ao acesso a campos de uma estrutura. Usamos o operador ponto (.) para acessá-los diretamente e o operador seta (->) para acessá-los através de um ponteiro da união. Assim, dada a declaração acima, podemos escrever:

```
v.i = 10;  
ou  
v.c = 'x';
```

Salientamos, no entanto, que apenas um único elemento de uma união pode estar armazenado num determinado instante, pois a atribuição a um campo da união sobrescreve o valor anteriormente atribuído a qualquer outro campo.

## 8.9. Tipo *enumeração*

Uma enumeração é um conjunto de constantes inteiras com nomes que especifica os valores legais que uma variável daquele tipo pode ter. É uma forma mais elegante de organizar valores constantes. Como exemplo, consideremos a criação de um tipo booleano. Variáveis deste tipo podem receber os valores 0 (FALSE) ou 1 (TRUE).

Poderíamos definir duas constantes simbólicas dissociadas e usar um inteiro para representar o tipo booleano:

```
#define FALSE 0
#define TRUE 1
typedef int Bool;
```

Desta forma, as definições de FALSE e TRUE permitem a utilização destes símbolos no código, dando maior clareza, mas o tipo booleano criado, como é equivalente a um inteiro qualquer, pode armazenar qualquer valor inteiro, não apenas FALSE e TRUE, que seria mais adequado. Para validarmos os valores atribuídos, podemos enumerar os valores constantes que um determinado tipo pode assumir, usando enum:

```
enum bool {
    FALSE,
    TRUE
};
typedef enum bool Bool;
```

Com isto, definimos as constantes FALSE e TRUE. Por *default*, o primeiro símbolo representa o valor 0, o seguinte o valor 1, e assim por diante. Poderíamos explicitar os valores dos símbolos numa enumeração, como por exemplo:

```
enum bool {
    TRUE = 1,
    FALSE = 0,
};
```

No exemplo do tipo booleano, a numeração *default* coincide com a desejada (desde que o símbolo FALSE preceda o símbolo TRUE dentro da lista da enumeração).



A declaração de uma variável do tipo criado pode ser dada por:

*Bool resultado;*

onde resultado representa uma variável que pode receber apenas os valores FALSE (0) ou TRUE (1).

## CRÉDITOS

---