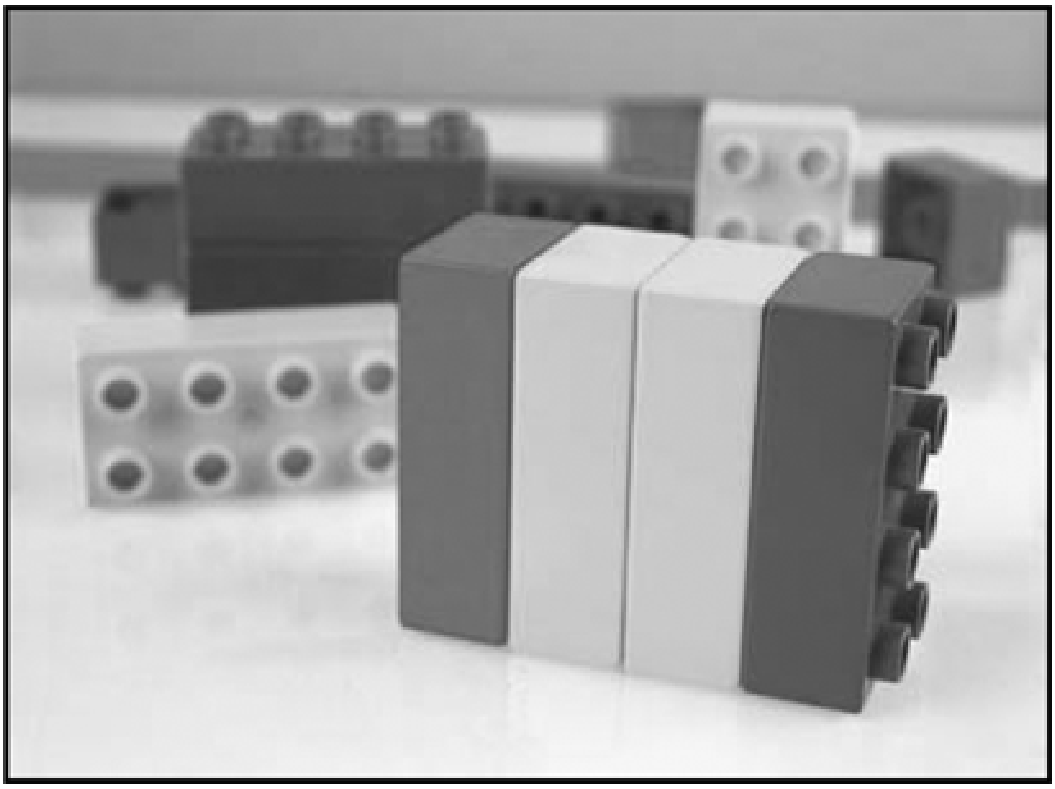
Construção de Algoritmos



Licenciatura em Computação

Alessandreia Marta de Oliveira
Rodrigo Luis de Souza da Silva

Índice

1. Introdução

O uso de algoritmos surgiu como uma forma de indicar o caminho para a solução dos mais variados problemas. Dado um problema, os principais cuidados daquele que for resolvê-lo utilizando algoritmos é:

- Entender perfeitamente o problema;
- Escolher métodos para sua solução;
- Desenvolver um algoritmo baseado nos métodos;
- Codificar o algoritmo na linguagem de programação disponível.

A parte mais importante da tarefa de programar é a construção de algoritmos. Segundo Niklaus Wirth: “Programação é a arte de construir e formular algoritmos de uma forma sistemática”.

O aprendizado de algoritmos não se consegue a não ser através de muitos exercícios. Algoritmos não se aprendem copiando ou estudando códigos. **Algoritmos só se aprende construindo e testando códigos**.

Conceitos

Ação

Ação é um evento que ocorre num período de tempo finito, estabelecendo um efeito intencionado e bem definido. Como exemplos, podem ser citados: “Colocar o livro em cima da mesa”; “Atribuir o valor 3,1416 a uma variável”.

Toda ação deve ser executada em um tempo finito (do instante t_0 até o instante t_1). O que realmente interessa é o efeito produzido na execução da ação. Pode-se descrever o efeito de uma ação comparando o estado no instante t_0 com o estado no instante t_1.

Estado

Estado de um dado sistema de objetos é o conjunto de propriedades desses objetos que são relevantes em uma situação considerada, como por exemplo: “Livro na estante ou sobre a mesa”; “Conjunto de valores das variáveis do programa num certo instante da execução”.

Processo

Processo é um evento considerado como uma seqüência temporal de (sub)ações, cujo efeito total é igual ao efeito acumulado dessas (sub)ações.

Pode-se, geralmente, considerar um mesmo evento como uma ação ou como um processo, dependendo se o interesse está simplesmente no efeito total (da ação) ou se interessa um ou mais estados intermediários (do processo). Em outras palavras, se há interesse, uma ação pode ser geralmente detalhada em um processo.

Padrão de Comportamento

Em todo o evento pode-se reconhecer um padrão de comportamento, isto é, cada vez que o padrão de comportamento é seguido, o evento ocorre.

Como exemplo, pode-se considerar a seguinte descrição: “Uma dona-de-casa descasca as batatas para o jantar”; “traz a cesta com batatas da dispensa”; “traz a panela do armário”; “descasca as batatas”; “devolve a cesta a dispensa”.

Essa descrição pode ser usada para descrever eventos distintos (dias diferentes, batatas diferentes etc.). Isso somente é possível porque os eventos possuem o mesmo padrão de comportamento.

O efeito de um evento fica totalmente determinado pelo padrão de comportamento e eventualmente pelo estado inicial.

Algoritmo

Algoritmo é a descrição de um padrão de comportamento, expresso em termos de um repertório bem definido e finito de ações primitivas que, com certeza, podem ser executadas. Um algoritmo possui caráter imperativo, razão pela qual uma ação em um algoritmo é chamada de comando.

O exemplo a seguir apresenta um algoritmo para descascar batatas para o jantar:

“traga a cesta com batatas da dispensa”;

“traga a panela do armário”;

“descasque as batatas”;

“devolva a cesta a dispensa”;

Um algoritmo (ou programa) apresenta dois aspectos complementares: aspecto dinâmico, que é a execução do algoritmo no tempo e aspecto estático, que é a representação concreta do algoritmo através de um texto contendo comandos que devem ser executados numa ordem prescrita (atemporal).

O problema central da computação consiste em relacionar esses dois aspectos, isto é, consiste no entendimento (visualização) das estruturas dinâmicas das possíveis execuções do algoritmo a partir da estrutura estática do seu texto.

A restrição a um número limitado de estruturas de controle (de execução dos comandos do algoritmo) permite reduzir o abismo existente entre o aspecto estático e o dinâmico do algoritmo. São usadas somente três estruturas de controle: Seqüência Simples, Alternativa e Repetição.

A generalização do algoritmo para descascar batatas para o jantar pode ser dada por:

“traga a cesta com batatas da dispensa”;

“traga a panela do armário”;

se “saia é clara” // Alternativa

    então “coloque avental”;

enquanto “número de batatas é insuficiente” faça //Repetição

    “descasque uma batata”;

“devolva a cesta a dispensa”;

Estas estruturas podem ser organizadas em uma pseudolinguagem ou pseudocódigo.

PseudoLinguagem

Uma pseudolinguagem é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples intermediária entre a linguagem natural e uma linguagem de programação. É uma linguagem que não pode ser executada num sistema real (computador).

Um algoritmo deve ser determinístico, isto é, dadas as mesmas condições iniciais, deve produzir em sua execução os mesmos resultados.

E somente interessam os algoritmos executáveis em tempo finito.

Lógica de programação

O desenvolvimento de um programa requer a utilização de um raciocínio ímpar em relação aos raciocínios utilizados na solução de problemas de outros campos do saber. Para resolver um determinado problema é necessário encontrar uma sequência de instruções cuja execução resulte na solução da questão.

Pode-se dizer que programa é um algoritmo que pode ser executado em um computador e lógica de programação é um conjunto de raciocínios utilizados para o desenvolvimento de algoritmos (e, portanto, de programas).

Linguagem de Programação

É um conjunto de regras sintáticas e semânticas usadas para definir um programa de computador. Uma linguagem permite que um programador especifique precisamente sobre quais dados um computador vai atuar, como estes dados serão armazenados ou transmitidos e quais ações devem ser tomadas sob várias circunstâncias.

Linguagem C

A linguagem de programação que será utilizada durante a disciplina é a linguagem C. As bases da linguagem C foram desenvolvidas entre 1969-1973, em paralelo com o desenvolvimento do sistema operacional Unix.

A linguagem C é amplamente utilizada, principalmente no meio acadêmico. O sucesso do sistema operacional Unix auxiliou na popularização do C.

IDE

IDE significa *Integrated Development Environment* ou Ambiente Integrado de Desenvolvimento. É um programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar este processo.

O CodeBlocks é um IDE disponível para Linux e Windows e o *download* gratuito pode ser obtido em **http://www.codeblocks.org/downloads** e poderá ser utilizado no curso para o desenvolvimento de nossos algoritmos.

Algumas dicas de como configurar seu primeiro projeto no Code::Blocks podem ser vistas em:

https://sites.google.com/site/algoritmosufjf/Arquivos/Compilandoseu.primeiro.codigo.no.CodeBlocks.pdf

Exercícios

1. Qual o padrão de comportamento utilizado para gerar esta sequência?

   1, 5, 9, 13, 17, 21, 25

2. Qual o padrão de comportamento utilizado para gerar esta sequência?

   1, 1, 2, 3, 5, 8, 13, 21, 34

3. Qual o padrão de comportamento utilizado para gerar esta sequência?

   1, 2, 4, 9, 16, 25, 36, 49, 64, 81

4. Escrever um algoritmo para descrever como você faz para ir da sua casa até o supermercado mais perto.

5. Escrever um algoritmo para descrever como você faz para trocar o pneu do seu carro. Considere que você estava dirigindo quando percebeu que o pneu havia furado.

6. Cite 2 exemplos de IDE que permitem o desenvolvimento de programas na linguagem C.

7. Cite 5 exemplos de linguagem de programação utilizadas atualmente.

8. Que características um bom programa precisa ter?

9. Escreva um algoritmo para calcular a soma de um conjunto de números. Use como base o algoritmo para descascar batatas para o jantar visto anteriormente.

10. Escreva um algoritmo para calcular a média das duas notas de um aluno de algoritmos. Se o aluno obtiver média maior ou igual a 70 ele será aprovado. Use como base o algoritmo para descascar batatas para o jantar visto anteriormente.

2. Tipos de Dados, Variáveis e Comandos de Entrada e Saída

João possui vários objetos e precisa colocá-los dentro da caixa com buracos de vários formatos. De acordo com as características de cada um dos objetos, eles somente se encaixam em um determinado buraco. Se João tentar colocar um objeto triangular em um buraco circular, por exemplo, não vai encaixar...

Da mesma forma, quando criarmos os nossos algoritmos, trabalharemos com o conceito de tipo de dados. Veremos neste capítulo que determinadas informações somente poderão ser armazenadas em determinados "recipientes" que chamaremos de variáveis. Além disso, vamos aprender outras formas de utilização destas variáveis.

Introdução

Para cada comando serão apresentadas a sua sintaxe, que é o formato geral do comando e que deve ser aceita e respeitada como padrão e a sua semântica, que corresponde ao significado da ação realizada pelo comando, em tempo de execução.

No texto (estático) de um programa (ou algoritmo), um valor pode ser representado na forma de constante ou de variável. A constante é representada em um programa diretamente pelo seu valor (que não se altera durante a execução do programa).

Variáveis

A variável é representada no texto de um programa por um nome que corresponde a uma posição da memória que contém o seu valor. Em tempo de execução, o nome da variável permanece sempre o mesmo e seu valor pode ser modificado.

O nome de uma variável ou identificador é criado pelo programador e deve ser iniciado por uma letra que pode ser seguida por tantas letras, algarismos ou sublinha quanto se desejar e é aconselhável que seja significativo do valor que ela representa.

Exemplos de nomes de variáveis:

Certo: nome, telefone, salario_func, x1, nota1, Media, SOMA

Errado: 1ano, sal/hora, _nome

Vale lembrar que a linguagem C é sensível ao caso (*case sensitive*), ou seja, as letras maiúsculas diferem das minúsculas.

Tipos de Dados

Todo valor (constante ou variável) de um programa tem um tipo de dados associado. Um tipo de dados é constituído de duas partes: um conjunto de objetos (domínio de dados) e um conjunto de operações aplicáveis aos objetos do domínio.

Toda linguagem de programação tem embutido um conjunto de tipos de dados, também chamados de implícitos, primitivos ou básicos.

Os tipos de dados básicos são: inteiro, real, caracter e lógico. Em C, os tipos são, respectivamente, int, float/double e char. Não há o tipo lógico em C.

A ocupação em memória e a faixa de valores possíveis para esses tipos da dados em C pode ser observada na tabela a seguir.

Tipo	***Espaço que ocupa na memória***	***Faixa***
Char	**1 byte**	*-128 a 127* (incluindo letras e símbolos)
Int	**4 bytes**	-2147483648 a 2147483647
Float	**4 bytes**	*3.4E-38* a 3.4E+38 (6 casas de precisão)
double	**8 bytes**	*1.7E-308* a *1.7E+308* (15 casas de precisão)

A seguir são apresentados os tipos de dados básicos, bem como o domínio e operações associados a cada um deles.

Inteiro:

Domínio: conjunto dos inteiros.

Operações: usam dois argumentos inteiros e, de acordo com o resultado são:

+, -, *, div (7 div 2 =3, ou seja, o operador div efetua a divisão do primeiro pelo segundo número e tem como resultado a parte inteira desta divisão), e **mod** (7 mod 2 = 1, ou seja, o operador mod efetua a divisão do primeiro pelo segundo número e tem como resultado o resto desta divisão): resultado inteiro

/: resultado real

<, ≤, =, >, ≥ e ≠: resultado lógico

Real: Domínio: conjunto dos reais. Operações: usam dois argumentos reais e, de acordo com o resultado são: **+, -, * e /:** resultado real **<, ≤, =, >, ≥ e ≠:** resultado lógico
Caracter: Domínio: conjunto de caracteres alfanuméricos. Operações: usam dois argumentos do domínio e fornecem resultado lógico: **<, ≤, =, >, ≥ e ≠**
Lógico: Domínio: {verdadeiro, falso}. Operações: usam dois argumentos do domínio e fornecem resultado lógico: Conectivos lógicos: conjunção (**e**, ^), disjunção (**ou**, ∨), disjunção exclusiva (**xou**, ⊕), negação (**não**, ¬). A negação trabalha com somente um argumento. Conectivos relacionais: **= e ≠.**

Operadores

A tabela a seguir apresenta os operadores que são diferentes em C em relação à pseudolinguagem.

Pseudolinguagem	C
Mod	%
≤	<=
≥	>=
=	==
≠	!=

O símbolo "=" em C é utilizado para atribuição, correspondendo ao "←" da pseudolinguagem adotada nesta apostila. Vale mencionar também que não há correspondente direto em C para o operador "div".

Tipos de Constantes e de Variáveis

O tipo básico associado a uma constante fica determinado pela própria apresentação da constante. Como exemplos podem ser citados:

7 → constante inteira

7.0 ou 7. → constante real

'7' → constante caracter

falso → constante lógica

Declaração de Variáveis

Para as variáveis, devem ser feitas no início do programa (ou de um bloco) as declarações (de tipo) de variáveis.

Sintaxe: **<tipo> nome (s);**

Declarar uma variável implica em efeturar a alocação de um espaço na memória que possa conter um valor do seu tipo e associação do endereço dessa posição da memória ao nome da variável.

A seguir alguns exemplos de declaração de variáveis:

```
caracter f, n;
inteiro idade;
real a, b, X1;
lógico vendido;
```

Declarada uma variável, toda vez que ela for referenciada em qualquer comando do programa, o computador vai trabalhar com o conteúdo de seu endereço, que é o valor da variável.

Uma variável não pode ter o mesmo nome de uma palavra-chave de C, como por exemplo:
main, int, float, char, short, return, case, void etc.

As variáveis só podem armazenar informações ou dados sempre de um mesmo tipo (inteiro, real, caractere ou char).

Na linguagem C, a declaração de variáveis obedece a seguinte sintaxe:
<tipo> <nome_var>;
ou
<tipo> <nome_var1>, <nome_var2>, ... ,<nome_varn>;

É importante não esquecer que quando se declara uma variável, um espaço num determinado endereço na memória é alocado para armazenar um dado que obrigatoriamente tem que ser do mesmo tipo que o da variável. Toda e qualquer variável deve ser declarada e sua declaração deve ser feita antes de sua utilização no programa.

Exemplo de declaração de variáveis em C:

```
int a, b, c;
char letra, d, e;
float f1;
float f2,f3;
```

Comandos Básicos

A seguir são apresentados os comandos básicos que podem ser utilizados no desenvolvimento dos algoritmos.

O comando de atribuição (Sintaxe: variável ← expressão;) é responsável por atribuir à variável o resultado da expressão. O operador de atribuição em C é o sinal de igual = (sintaxe: <variavel> = <expressão>;). Para exemplificar, na linguagem C, tem-se:

```
int a,b,c,d;
a = 5;
c = 7;
b = a;
d = a + b + c;
```

ou

```
int a=5;
int c=7;
int b,d;
b = a;
d = a+b+c;
```

Sobre as expressões aritméticas, como já foi falado anteriormente, estas fornecem resultado numérico (inteiro ou real). As operações básicas são: +, -, *, /. Além disso tem-se a exponenciação: potencia(A + B , N) bem como as funções matemáticas comuns: sen (X), cos (X), abs (X), raiz (X), arctan (X), exp (X), log (X), ln (X) etc.

Sobre os operadores para inteiros, tem-se o *mod* e o *div*:

Existem também as expressões lógicas que fornecem um resultado lógico. Os conectivos relacionais são <, ≤, =, >, ≥ e ≠. Os conectivos lógicos estão representados por: Conjunção: e; Disjunção: ou; Disjunção Exclusiva: xou; Negação: não

A tabela a seguir é denominada "tabela verdade", sendo A e B duas expressões lógicas.

A	**B**	**A e B**	**A ou B**	**A xou B**	**Não A**
V	V	V	V	F	F
V	F	F	V	V	F
F	V	F	V	V	V
F	F	F	F	F	V

A tabela a seguir apresenta os Operadores Lógicos e Relacionais em C

Relacionais		**Lógicos**	
>	Maior que	**&&**	And (e)
>=	Maior ou igual	**||**	Or (ou)
<	Menor que	**!**	Not (não)
<=	Menor ou igual		
==	Igual		
!=	Diferente		

A tabela a seguir apresenta os Operadores Aritméticos em C

Aritmético	**Tipo**	**Opção**	**Prioridade**
+	Binário	Adição	5
-	Binário	Subtração	5
%	Binário	Resto da divisão	4
*	Binário	Multiplicação	3
/	Binário	Divisão	3
++	Unário	Incremento	2
--	Unário	Decremento	2
+	Unário	Manutenção do sinal	1
-	Unário	Inversão do sinal	1

Exemplos:

```
a = a + b;
a = a + 4;
a = b / 2;
a = 4 * 2 + 3; // qual o valor final de a?
b = 2 * 3 – 2 * 2; // qual o valor final de b?
a++; // similar a   a = a + 1;
b--; // similar a     b = b – 1;
```

Sobre a prioridade de execução das operações em uma expressão, algumas regras devem ser seguidas:

1º. Parênteses (dos mais internos para os mais externos)

2º. Expressões aritméticas, seguindo a ordem: funções, * e /, + e –

3º. Comparações: <, ≤, =, >, ≥ e ≠

4º. não

5º. e

6º. ou e xou

7º. Da esquerda para a direita quando houver indeterminações.

Comentários

Tanto na pseudolinguagem utilizada quanto em C, os comentários podem ser utilizados colocando-se duas barras "//" antes do texto. No exemplo a seguir tanto a primeira linha quanto a segunda possuem comentários.

```
inteiro maior; //maior valor lido
//ler os valores das variáveis A, B, C
```

Bloco

Blocos são comandos delimitados por chaves ({ }). Pode-se declarar variáveis em seu interior e delimitar seu escopo como visto a seguir.

```
{
   <declaração de variáveis>;
   <comandos>;
}
```

Comandos de impressão e leitura

Em pseudolinguagem, o comando que será utilizado para representar valores a serem informados para usuário é o comando *imprima*. O comando de leitura, ou seja, aquele que representará uma entrada de informação do usuário no sistema é o comando *leia*.

Exemplo:

```
leia (a, x); //serão lidos os valores das variáveis a, x nesta ordem
imprima ("Valor de N =", n, "Fatorial de N =", fat);
```

Serão vistos a seguir detalhes referentes a impressão e leitura na linguagem C. Estes detalhes servirão como base para auxiliá-los no desenvolvimento de seus programas no computador.

Comandos para impressão em C

A exibição dos resultados do processamento e de mensagens é feita através da função pré-definida *printf()*, cujo protótipo está contido no arquivo *stdio.h*. Sua sintaxe é a seguinte:

```
printf("Expressão" , Lista de argumentos );
```

Onde:

Expressão: contém mensagens a serem exibidas, códigos de formatação que indicam como o conteúdo de uma variável deve ser exibido e códigos especiais para a exibição de alguns caracteres especiais.

Lista de argumentos: pode conter identificadores de variáveis, expressões aritméticas ou lógicas e valores constantes.

Estrutura de um Programa que utiliza a função printf()

```
#include <stdio.h>
int main( )
{
   printf("Estou aprendendo a programar em C");
   return 0;
}
```

Impressão de Tipos de Dados

As tabelas a seguir apresentam alguns elementos extras que podem ser utilizados na impressão de tipos de dados e de códigos especiais.

Código	Tipo	Elemento armazenado
%c	char	um único caractere
%d ou %i	int	um inteiro
%f	float	um número em ponto flutuante
%lf	double	ponto flutuante com dupla precisão
%e	float ou double	um número na notação científica
%s	-	uma cadeia de caracteres

Impressão de Códigos Especiais

Código	Ação
\n	leva o cursor para a próxima linha
\t	executa uma tabulação
\b	executa um retrocesso
\f	leva o cursor para a próxima página
\a	emite um sinal sonoro (*beep*)
\"	exibe o caractere "
\\	exibe o caractere \
%%	exibe o caractere %

Alguns exemplos de utilização do comando *printf* são apresentados a seguir. Faça os testes no CodeBlocks e verifique o comportamento de cada um destes algoritmos.

```
#include <stdio.h>
int main( )
{
  printf("Valor recebido foi %d" ,10);
  return 0;
}
```

```
#include <stdio.h>
int main()
{
  printf("Caracter A: %c" , 'A');
  return 0;
}
```

```
#include <stdio.h>
int main()
{
  printf("Valor inteiro %d e um float %f" ,10 ,1.10);
  return 0;
}
```

```
#include <stdio.h>
int main()
{
  printf("Minha string: %s", "Entendi tudo!!!! ");
  return 0;
}
```

```
#include <stdio.h>
int main()
{
  printf("Deveria imprimir um float, ");
  printf("mas mandei imprimir um inteiro: %d" ,1.10);
  return 0;
}
```

```
#include <stdio.h>

int main()
{
  printf("\t\tx\n");
  printf("\tx\t\tx\n");
  printf("\t\tx\n");
  return 0;
}
```

Fixando as Casas Decimais

Por ***default***, a maioria dos compiladores C exibe os números em ponto flutuante com seis casas decimais. Para alterar este número pode-se acrescentar **.n** ao código de formatação da saída, sendo n o número de casas decimais pretendido, como mostra o exemplo a seguir.

```
#include <stdio.h>

int main()
{
  printf("Default: %f \n",3.1415169265);
  printf("Uma casa: %.1f \n",3.1415169265);
  printf("Duas casas: %.2f \n",3.1415169265);
  printf("Tres casas: %.3f \n",3.1415169265);
  printf("Notacao Cientifica: %e \n",3.1415169265);
  return 0;
}
```

Alinhamento de Saída

O programa pode fixar a coluna da tela a partir da qual o conteúdo de uma variável, ou o valor de uma constante será exibido. Isto é obtido se acrescentado um inteiro **m** ao código de formatação. Neste caso, *m* indicará o número de colunas que serão utilizadas para exibição do conteúdo.

```
#include <stdio.h>

int main()
{
   printf("Valor: %d \n",25);
   printf("Valor: %10d \n",25);
   return 0;
}
```

Leitura de dados em C

A leitura de dados em C é o complemento de *printf*() e permite ler dados formatados da entrada padrão (teclado). O principal comando de leitura é o *scanf* e sua sintaxe é a seguinte:

```
scanf("expressão de controle", argumentos);
```

Alguns exemplos de utilização do *scanf* são apresentados a seguir.

scanf() – Exemplo 1

```
#include <stdio.h>

int main()
{
  int num;
  printf("Digite um valor: ");
  scanf("%d",&num); // O operador "&" é utilizado para obter o
                    // endereço de memória da variável.

  printf("\nO valor digitado foi %d.",num);
  return 0;
}
```

scanf() – Exemplo 2

```
#include <stdio.h>

int main()
{
  int n1,n2,soma;
  printf("Digite dois valores: ");
  scanf("%d",&n1);
  scanf("%d",&n2);
  soma = n1 + n2;
  printf("\n%d + %d = %d.",n1,n2,soma);
  return 0;
}
```

Estruturas de Controle

Neste curso serão apresentadas três estruturas de controle sendo a primeira denominada sequência simples, que como o próprio nome diz, indica uma execução seqüencial dos comandos. Ou seja, o controle de fluxo de execução entra na estrutura, executa comando por comando, de cima para baixo e sai da estrutura, como mostra o exemplo a seguir.

Exemplo

```
Sintaxe
C1;
C2;
...
Cn;
```

```
leia (X, Y);
A ← X + Y;
B ← X – Y;
imprima (A, B);
```

```
scanf("%d %d", &X, &Y);
a = x + y;
b = x - y;
printf ("%d %d", a, b);
```

Métodos de Criação de Programas

Até agora, foram estudados vários recursos visando especificamente a construção de algoritmos: Conceito de Constante, Variável e Tipos de Dados, Tipos de Dados Básicos, Seqüência Simples e tantos outros. Já é possível então criar alguns algoritmos simples, bastando adotar um dos três métodos de desenvolvimento, apresentados a seguir.

O primeiro dos métodos é chamado de Método Direto e é formado pelos seguintes passos:

1. **Entender** o problema (**análise do enunciado**).

2. **Descrição de variáveis:** lista contendo para cada variável, o seu nome, uma descrição de seu objetivo e o seu tipo. Iniciar a lista pelos dados de entrada e pelos resultados a produzir.

3. **Estudo de métodos:** estabelecer (pelo menos) um método para obtenção de cada resultado pedido e para cada controle de processamento. Caso o método escolhido necessite utilizar novas variáveis, incluí-las na lista de descrição de variáveis.

4. **Elaborar um algoritmo** baseado no estudo de métodos, observando a seguinte seqüência:

```
principal
{
    definição de novos tipos;
    declaração de variáveis;
    inicialização de variáveis;
    corpo do algoritmo controlando:
        leitura;
        processamento;
        saída;
}
```

5. **Testar o algoritmo**, considerando, se possível, todas as alternativas de entrada.

6. **Programa**: traduzir o algoritmo para uma linguagem de programação e testá-lo exaustivamente no computador.

7. **Documentação**: o conjunto de documentos produzidos ao longo do método direto constitui a primeira documentação técnica do programa.

Uma segunda possibilidade é utilizar o Método dos Refinamentos Sucessivos, onde inicialmente, deve-se manter um nível mais alto de abstração (atenção em **o que fazer** e não em **como fazer**).

Dispensa-se o estudo de métodos e parte-se logo para a elaboração do algoritmo sendo que cada tarefa lógica distinta será representada por um pseudocomando.

A primeira versão do algoritmo é rapidamente obtida pela listagem de alguns poucos pseudocomandos que deverão ser detalhados em refinamentos posteriores (em função de outros pseudocomandos ou em função da própria pseudolinguagem).

Depois que todos os pseudocomandos já estiverem detalhados em termos da pseudolinguagem, pode-se juntar as partes montando a versão final e completa do algoritmo.

Uma terceira alternativa é usar o Método Misto onde os controles de caráter geral e as ações mais simples são desenvolvidas diretamente e a técnica de refinamentos sucessivos fica somente para as ações lógicas consideradas mais complexas.

Observação: Em qualquer dos métodos, pode-se desenvolver as ações lógicas bem definidas na forma de procedimentos (ou funções). A utilização destes módulos funcionais constitui uma técnica (programação modular) que geralmente diminui o esforço do programador e aumenta a qualidade de seus programas.

Exercícios

1. Sendo A=3, B=7 e C=4, informe se as expressões abaixo são verdadeiras ou falsas.

   a) (A+C) > B
   b) B >= (A + 2)
   c) C = (B –A)
   d) (B + A) <= C
   e) (C+A) > B

2. Sendo A=5, B=4 e C=3 e D=6, informe se as expressões abaixo são verdadeiras ou falsas.

   a) (A > C) **E** (C <= D)
   b) (A+B) > 10 **OU** (A + B) = (C + D)
   c) (A>=C) **E** (D >= C)

3. Determine os resultados obtidos na avaliação das expressões lógicas seguintes, sabendo que A, B, C, S1 e S2 contêm respectivamente 2, 7, 3.5, "noite", "frio" e que existe uma variável lógica L1 cujo valor é falso.

   a) B = A*C **E** L1
   b) "dia"= S1 **OU** "frio" ≠ "clima"
   c) A+ C < 5
   d) A*C/B > A*B*C
   e) **NÃO** FALSO

4. Determine o resultado lógico das expressões mencionadas (Verdadeira ou Falsa). Considere para as respostas os seguintes valores: X=1, A=3, B=5, C=8 e D=7.

   a) **NÃO** (X > 3)
   b) (X < 1) **E** (**NÃO** (B > D))
   c) **NÃO** (D < 0) **E** (C > 5)
   d) **NÃO** ((X > 3) **OU** (C < 7))
   e) (A > B) **OU** (C > B)
   f) (X >= 2)
   g) (X < 1) **E** (B >= D)
   h) (D < 0) **OU** (C > 5)
   i) **NÃO** (D > 3) **OU** (**NÃO** (B < 7))
   j) (A > B) **OU** (**NÃO** (C > B))

5. Fazer um programa que imprima o seu nome.

6. Modificar o programa anterior para imprimir na primeira linha o seu nome, na segunda linha a sua idade e na terceira sua altura.

7. Imprimir o valor 2.346728 com 1, 2, 3 e 5 casas decimais.

8. Ler uma temperatura em graus Celsius e apresentá-la convertida em graus Fahrenheit. A fórmula de conversão: F ← (9*C+160)/5

9. Construir um algoritmo para ler 5 valores inteiros, calcular e imprimir a soma desses valores.

10. Construir um algoritmo para ler 6 valores reais, calcular e imprimir a média desses valores.

11. Fazer um algoritmo para gerar e imprimir o resultado de H = 1 + 1/2 + 1/3 + 1/4 + 1/5.

12. Calcular e apresentar o volume de uma lata de óleo, utilizando a fórmula: volume = 3.14159*raio*raio*altura.

13. Elaborar um programa que calcule e apresente o volume de uma caixa retangular, por meio da fórmula volume = comprimento*largura*altura.

3. Estruturas Condicionais

João comprou um carro novo e ao abastecer ficou na dúvida sobre qual combustível colocar. Havia uma placa indicando que o álcool custava 75% do valor da gasolina. João sabe que o álcool é um combustível mais interessante se estiver no máximo a 70% do valor da gasolina. Caso contrário, não é interessante abastecer com álcool. Resolvido o dilema, João encheu o seu tanque com gasolina sem ter maiores dúvidas.

Da mesma forma que o caso ilustrado, muitas vezes em nossos algoritmos, nós precisamos tomar uma decisão baseado em uma condição. Esta aula trata exatamente desta questão. Veremos como criar **comandos condicionais** para que nossos algoritmos tenham a capacidade de resolver problemas de decisão como o ilustrado aqui.

Alternativa

A alternativa é utilizada quando a execução de uma ação depender de uma inspeção ou teste de uma condição (expressão lógica).

```
se (<expressão>)
{
  <sequência de comandos>
}
```

A expressão sempre será avaliada logicamente (verdadeiro ou falso). Em C, a expressão será considerada VERDADEIRA quando o seu resultado for diferente de zero e FALSO quando a expressão for igual a zero. Desta forma, sempre que um teste lógico é executado, ele retornará um valor diferente de zero quando for verdadeiro.

Existem 3 tipos de alternativas que são descritas a seguir: **Simples**, **Dupla** e **Múltipla Escolha**.

Alternativa Simples

O comando ***se*** pode decidir se uma sequência de comandos será ou não executada.

Sintaxe

```
se ( <expressão> )
{
   <sequência de comandos>;
}
```

```
if ( <expressão> )
{
   <sequência de comandos>;
}
```

Obs.: Se houver só um comando (uma única linha), não é necessário o uso de chaves ({bloco}). Esta regra serve para a maioria das estruturas de controle.

Exemplo 1

```
principal
{
   inteiro a, b, maior;
   a ← 9;
   b ← 2;
   maior ← b;
   se (a > maior)
   {
         maior ← a;
   }
   imprima(maior);
}
```

```
#include <stdio.h>
int main()
{
   int a, b, maior;
   a = 9;
   b = 2;
   maior = b;
   if (a > maior)
      maior = a;

   printf ("\nMAIOR = %d", maior);
   return 0;
}
```

Alternativa Dupla

O comando ***se..senão*** pode decidir entre duas sequências de comandos. O comando tem a seguinte sintaxe:

Pseudolinguagem

```
se (< condição >)
{
   C1;
}
senão
{
   C2;
}
```

Linguagem C

```
if (condição)
{
   C1;
}
else
{
   C2;
}
```

Como visto, o fluxo de execução do algoritmo seguirá para um dos dois possíveis caminhos C1 e C2, dependendo se a condição for verdadeira ou falsa.

Exemplo 1 - imprimir o MAIOR entre A e B

```
principal
{
   inteiro A, B, MAIOR;
   A ← 9;
   B ← 2;
   se (A > B)
   {
      MAIOR ← A;
   }
   senão
   {
      MAIOR ← B;
   }
   imprima(MAIOR);
}
```

```
#include <stdio.h>
int main()
{
   int a, b, maior;
   a = 9;
   b = 2;
   if (a > b)
   {
      maior = a;
   }
   else
   {
      maior = b;
   }
   printf ("\nMAIOR = %d", maior);
   return 0;
}
```

Exemplo 2: Ler duas variáveis (x e y) e imprimi-las em ordem crescente.

Pseudolinguagem

```
principal
{
  real x, y, aux;
  imprima("Digite os dois numeros");
  leia(x, y);
  imprima("Conteudos originais de x e de y: ", x, y);
  se (y < x)
  {
    aux ← x;
    x ← y;
    y ← aux;
  }
  imprima("Conteudos de x e de y ordenados: ", x, y);
}
```

Linguagem C

```
#include <stdio.h>
int main()
{
  float x, y, aux;
  printf("Digite os dois numeros");
  scanf("%f %f", &x, &y);
  printf("Conteudos originais de x e de y: %f , %f \n: ", x, y);
  if (y < x)
  {
    aux = x;
    x = y;
    y = aux;
  }
  printf("Conteudos de x e de y ordenados: %f , %f: \n", x, y);
  return 0;
}
```

Exemplo 3 - Verificar a paridade de um número.

```
principal
{
  inteiro x;
  imprima("Digite o numero: ");
  leia(x);
  se (x mod 2 = 0)
     imprima ("Número é par");
  senão
     imprima ("Número é ímpar");
}
```

```
#include <stdio.h>
int main()
{
   int x;
   printf("Digite o numero: ");
   scanf("%d", &x);
   if (x % 2 == 0)
      printf("Número é par.");
   else
      printf("Número é ímpar.");
   return 0;
}
```

Alternativa Múltipla Escolha

A alternativa múltipla escolha é utilizada quando uma variável pode ser igual a diferentes valores que se deseja avaliar. Estes valores podem ser do tipo inteiro ou do tipo caractere.

Pseudolinguagem

```
escolha (<expressão>)
{
   caso V1 :
      <sequência de comandos>;

   caso V2 :
      <sequência de comandos>;

   . . .
   caso Vn :
      <sequência de comandos>;

   senão   :
      <sequência de comandos>;
}
```

Linguagem C

```
switch(<expressão>)
{
    case V1:
      <sequência de comandos>;
    break;
    case V2:
      <sequência de comandos>;
    break;
    . . . . . .
    case Vn:
      <sequência de comandos>;
    break;
    default:
      <sequência de comandos>;
}
```

Exemplo

```
principal
{
  inteiro EPOCA;
  leia(EPOCA);
  escolha (EPOCA)
  {
     caso 1: imprima("verão");

     caso 2: imprima("outono");

     caso 3: imprima("inverno");

     caso 4: imprima("primavera");

     senão: imprima("Inválido");
  }
}
```

```
#include <stdio.h>
int main()
{
   int epoca;
   scanf("%d", &epoca);
   switch (epoca)
   {
      case 1: printf("verão");
      break;
      case 2: printf ("outono");
      break;
      case 3: printf ("inverno");
      break;
      case 4: printf ("primavera");
      break;
      default: printf("Inválido");
   }
   return 0;
}
```

Problema Exemplo

Agora que as três alternativas existentes foram descritas, que tal a descrição de um problema e sua respectiva solução passo-a-passo, utilizando o método direto, visto anteriormente? Observe atentamente a forma de raciocínio que será utilizada para atingir a solução do problema e tente usá-la nos exercícios no final deste capítulo.

Enunciado do problema:

Construir um algoritmo para ler os coeficientes A, B e C de uma equação do segundo grau levando-se em consideração os seguintes critérios:

- Se delta for negativo: imprimir a mensagem "Não há solução real".
- Se delta for maior ou igual a zero: calcular e imprimir as raízes da equação.

1) Enunciado está entendido?

Para resolver o problema é necessário saber a equação de 2º grau que tem a seguinte forma:

(-B ± raiz(DELTA)) / (2*A) onde DELTA = (B**2 – 4*A*C)

2) Quais variáveis serão necessárias?

A, B, C e DELTA serão as variáveis inteiras que comporão a fórmula da equação acima. Serão necessárias mais duas variáveis inteiras X1 e X2 que serão as raízes da equação.

3) Quais métodos serão utilizados?

Por se tratar de uma condição, será utilizada a estrutura de controle de alternativa.

4) Elaborar o Algoritmo para resolver o problema

```
principal
{
   inteiro A, B, C, DELTA, X1, X2;
   imprima("Leia os coeficientes A, B e C da equação de 2° grau");
   leia(A,B,C);
   DELTA ← (B*B – 4*A*C);
   se (DELTA ≥ 0)
   {
     X1 ← (-B + raiz(DELTA)) / (2*A);
     X2 ← (-B - raiz(DELTA)) / (2*A);
     imprima("Raizes da equação: ", X1, X2);
   }
   senão
   {
     imprima ("Não existem raízes reais");
   }
}
```

5) Testar o algoritmo

Para as entradas 1, 5, 4 as saídas esperadas serão -1 e -4. Nesta etapa, é interessante fazer um **teste de mesa**, isto é, simular no papel o que aconteceria se o conjunto de entrada proposto fosse utilizado. Se as saídas esperadas forem encontradas, o algoritmo está correto.

6) Traduzir o algoritmo para um programa

```
#include <stdio.h>
#include <math.h>
int main()
{
    int a, b, c, delta, x1, x2;
    printf("Leia os coeficientes A, B e C da equação de 2° grau: ");
    scanf("%d %d %d", &a, &b, &c);
    delta = pow(b,2) - 4*a*c;
    if(delta >= 0)
    {
      x1 = (-b + sqrt(delta)) / (2*a);
      x2 = (-b - sqrt(delta)) / (2*a);
      printf("Raizes da equação: %d %d", x1, x2);
    }
    else
    {
       printf("Não existem raízes reais");
    }
   return 0;
}
```

Exercícios

1. Ler dois números inteiros e informar se o primeiro é maior, menor ou igual ao segundo.

2. Faça um programa para ler dois números inteiros, faça a divisão do primeiro pelo segundo (somente se o segundo for diferente de zero).

3. Ler um número inteiro e informar se ele é divisível por 2.

4. Altere o algoritmo anterior para que seja informado se o número é divisível por 2 e por 3 simultaneamente.

5. Altere o algoritmo anterior para que seja informado se o número é divisível por 2 e por 3, mas que não seja divisível por 5.

6. Faça um programa para ler dois números reais e verificar se ambos são maiores que zero. Caso positivo, informar “Valores são válidos”. Caso contrário, informar “Valores inválidos”.

7. Tendo como dados de entrada a altura e o sexo de uma pessoa, construa um programa que calcule seu peso ideal, utilizando as seguintes fórmulas:

   para homens: (72.7*h)-58

   para mulheres: (62.1*h)-44.7

8. Faça um programa que leia 3 comprimentos (x, y e z) e responda se eles formam um triângulo, ou seja, se
   x < y + z e y < x + z e z < x + y.

9. Desenvolver um algoritmo para ler o número do dia da semana e imprimir o seu respectivo nome por extenso. Considerar o número 1 como domingo, 2 para segunda etc. Caso o dia não exista (menor que 1 ou maior que 7), exibir a mensagem “Dia da semana inválido".

10. Fazer um algoritmo para ler dois números e um dos símbolos das operações: +, -, * e /. Imprimir o resultado da operação efetuada sobre os números lidos.

11. Os funcionários de uma empresa receberam um aumento de salário: técnicos (código = 1), 50%; gerentes (código = 2), 30%; demais funcionários (código = 3), 20%. Escrever um algoritmo para ler o código do cargo de um funcionário e o valor do seu salário atual, calcular e imprimir o novo salário após o aumento.

12. Desenvolver um algoritmo para ler o valor inteiro da idade de uma pessoa e imprimir uma das mensagens: se idade < 13: Criança, se 13 ≤ idade < 20: Adolescente, se 20 ≤ idade < 60: Adulto e se idade ≥ 60: Idoso.

13. Faça um programa para ler um caracter e imprimir as seguintes mensagens, segundo o caso:

    “Sinal de Menor”

    “Sinal de maior”

    “Sinal de Igual”

    “Outro caracter”

14. Para auxiliar os vendedores de uma loja na orientação aos clientes sobre as diversas formas de pagamento, desenvolver um algoritmo para:

a) imprimir o seguinte menu:

Forma de pagamento:
1. A vista.
2 .Cheque para trinta dias.
3. Em duas vezes.
4. Em três vezes.
5. Em quatro vezes.
6. A partir de cinco vezes.

b) Ler o código da opção de pagamento.

c) Imprimir uma das mensagens de acordo com a opção lida:

Opção = 1: Desconto de 20%

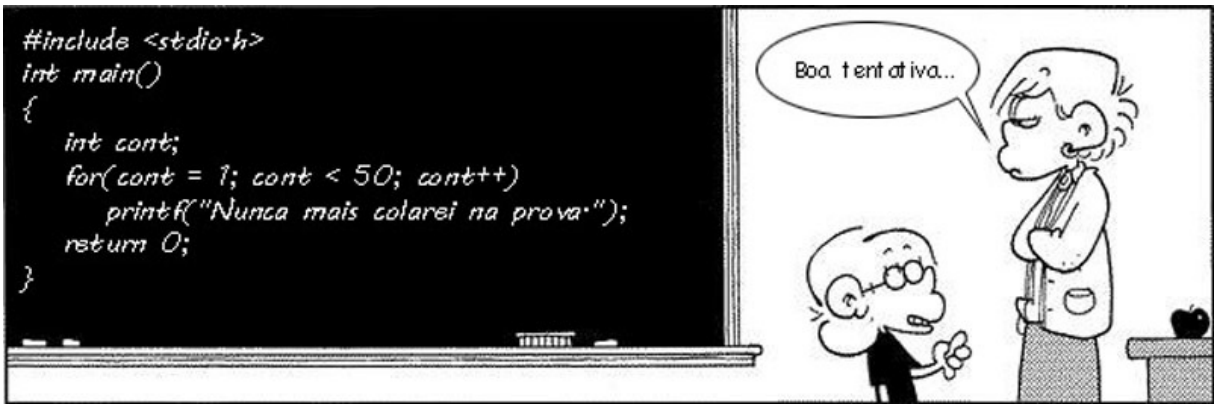
Opção = 2, 3 ou 4: Mesmo preço a vista

Opção = 5: Juros de 3% ao mês

Opção = 6: Juros de 5% ao mês

Se opção for menor do que 1 ou maior do que 6: Opção inválida

4. Comandos de Repetição



Maria chegou para fazer sua prova de cálculo mas infelizmente não havia estudado nada. Sua melhor amiga sabia bem a matéria e Maria optou por tentar colar dela as primeiras questões. Contudo, a professora viu a tentativa de Maria e a repreendeu. Como castigo, Maria teria que escrever 50 vezes no quadro a frase "Nunca mais colarei na prova". Maria não sabia cálculo, mas sabia algumas estruturas interessantes de Algoritmos. Uma destas estruturas é chamada de *repetição* que, como o nome sugere, auxiliaria Maria a repetir uma mesma ação por um número determinado de vezes. Maria então foi até o quadro e escreveu seu código para cumprir seu castigo. Infelizmente a professora não conhecia Algoritmos e não aceitou o algoritmo, mas a idéia de Maria foi no mínimo inteligente.

Nesta aula aprenderemos como criar estruturas como a utilizada por Maria neste exemplo. As estruturas de repetição são capazes de permitir que um conjunto de instruções sejam repetidos até que um critério de parada ocorra. Veremos neste capítulo como estas estruturas funcionam e vários exemplos de sua utilização.

Repetição

Chamadas de estruturas iterativas, iterações, laços ou loops estas estruturas permitem repetir a execução de uma ação várias vezes.

Podem ser:

- Repetição com Teste no Início
- Repetição com Teste no Fim
- Repetição com Variável de Controle

Repetição com Teste no Início

Sintaxe

Pseudolinguagem

```
enquanto (condição) faça
{
   C ;
}
```

Linguagem C

```
while (condição)
{
   C;
}
```

Semântica:

Enquanto a condição for verdadeira, a seqüência será repetida. Quando a condição fornecer resultado falso, o controle sai da estrutura passando para o comando seguinte ao final do bloco.

Repetição com uso de FLAGs

FLAG é um valor específico fornecido após o último dado de entrada, que serve para indicar o fim dos dados de entrada.

FLAG é somente uma marca de fim dos dados de entrada (não é um dado de entrada) e não pode ser processado.

A leitura do FLAG informa ao programa que os dados de entrada terminaram e que ele deve partir para a execução da finalização de seu processamento (cálculos finais, impressões finais etc.).

Controle de processamento genérico para uso de FLAG:

```
...
leia ("o 1° conjunto de dados");
enquanto ("não for FLAG")  faça
{
        "processar o conjunto de dados lido";
         leia ("o próximo conjunto de dados");
}
"finalizar processamento";
...
```

Exemplo

Desenvolver um algoritmo para ler uma seqüência de números inteiros, calcular e imprimir o quadrado de cada número lido. O último valor a ser lido é um FLAG = 0.

Algoritmo em Pseudolinguagem

```
principal
{
   inteiro NUM, QUADRADO;
   leia(NUM);   // lê o 1° conjunto de dados
   enquanto (NUM ≠ 0) faça     // testa se não é FLAG
   {
      QUADRADO ← NUM*NUM;
      imprima (QUADRADO);
      leia(NUM);     // lê o próximo conjunto de dados
   }
}
```

Teste do Algoritmo

Considerando que serão fornecidos os seguintes dados de entrada: 2 -3 1 4 0

O resultado do teste de mesa seria:

NUM	QUADRADO	IMPRIME
2	4	4
-3	9	9
1	1	1
4	16	16
0		

Código em C:

```
#include <stdio.h>

int main()
{
   int num, quadrado;
   scanf("%d", &num);
   while( num != 0)
   {
      quadrado = num * num;
      printf("\nO quadrado de %d é %d: ", num, quadrado);
      scanf("%d", &num);
   }
   return 0;
}
```

Repetição com uso de acumuladores

Repetição com uso de acumuladores pressupõe a utilização de uma variável que irá "acumular" um determinado valor durante a execução do algoritmo. Por exemplo, para se calcular a soma de um número indeterminado de valores, faríamos uso de uma variável que armazenaria a soma de todos os valores lidos. Esta variável é considerada uma variável acumuladora.

Exemplo

Desenvolver um algoritmo para ler uma seqüência de números inteiros com FLAG = 0, calcular e imprimir a soma desses números.

Como fazer?
Uma forma possível para resolver o problema de somatório de valores (acumulador) é imaginar uma variável que armazena as somas parciais. Essa variável deve iniciar com zero e somar número por número, até o final da seqüência.

Exemplo de acumulador em pseudolinguagem

```
principal
{
   inteiro NUM, SOMA;
   SOMA ← 0;
   leia(NUM);
   enquanto (NUM ≠ 0) faça
   {
      SOMA ← SOMA + NUM;
      leia(NUM);
   }
   imprima (SOMA);
}
```

Teste do Algoritmo

Considerando que serão fornecidos os seguintes dados de entrada: 8 -3 2 1 0

O resultado do teste seria:

NUM	SOMA	Imprime
	0	
8	8	
-3	5	
2	7	
1	8	
0		8

Código em C:

```
#include <stdio.h>

int main()
{
   int num, soma;
   scanf("%d", &num);
   soma = 0; // É obritório inicializar com 0
   while( num != 0)
   {
      soma = soma + num; // acumula os valores em SOMA
      scanf("%d", &num);
   }
   printf("A soma dos numeros digitados foi %d.", soma);
   return 0;
}
```

Repetição com uso de contadores

Outra função importante das estruturas de repetição é contar. Saber a quantidade de valores lidos dentro de uma repetição é importante em diferentes contextos. Para isso, utilizaremos um contador que nada mais é que um acumulador que é incrementado de 1 em 1.

Exemplo

Desenvolver um algoritmo para ler uma seqüência de números inteiros com FLAG = 0, calcular e imprimir a quantidade de números lidos.

Exemplo de acumulador em pseudolinguagem

```
principal
{
   inteiro NUM, CONT;
   CONT ← 0;
   leia(NUM);
   enquanto (NUM ≠ 0) faça
   {
      CONT ← CONT + 1;
      leia(NUM);
   }
   imprima (CONT);
}
```

Teste do Algoritmo

Considerando que serão fornecidos os seguintes dados de entrada: 8 -3 2 1 0

O resultado do teste seria:

NUM	SOMA	Imprime
	0	
8	1	
-3	2	
2	3	
1	4	
0		4

Código em C:

```
#include <stdio.h>

int main()
{
   int num, cont; // 'cont' representa o contador
   // imprime uma msg e lê o 1o. numero inteiro
   printf("Digite um número inteiro: ");
   scanf("%d", &num);
   cont = 0;
   while( num != 0)
   {
      cont = cont + 1;
      printf("\nDigite um número inteiro: ");
      scanf("%d", &num);
   }
   printf("Foram lidos %d numeros.", cont);
   return 0;
}
```

Exemplos de Resolução de Problemas

Veremos a seguir alguns exemplos de resolução de problemas.

Problema 1

Desenvolver um algoritmo para ler uma seqüência de números inteiros com FLAG = 0, calcular e imprimir a média aritmética dos números lidos.

Obs.: considerando NUM como a variável de leitura:

$$MEDIA = \frac{\sum_{1}^{N} NUM}{N} = \frac{SOMA}{N} \qquad MEDIA = \frac{SOMA}{CONT}$$

Solução em Pseudolinguagem

```
principal
{
   inteiro NUM, SOMA, CONT;
   real MEDIA;
   SOMA ← 0;
   CONT← 0;
   leia(NUM);
   enquanto (NUM ≠ 0) faça
   {
      SOMA ← SOMA + NUM;
      CONT ← CONT + 1;
      leia(NUM);
   }
   MEDIA ← SOMA / CONT;
   imprima (MEDIA);
}
```

Teste do Algoritmo

Considerando que serão fornecidos os seguintes dados de entrada: 2 9 4 22 1 7 0

O resultado do teste seria:

NUM	SOMA	CONT	MEDIA	Imprime
	0	0		
2	2	1		
9	11	2		
4	15	3		
22	37	4		
1	38	5		
7	45	6		
0			7.5	7.5

Código em C:

```
#include <stdio.h>

int main()
{
   int num;
   int soma; // 'soma' representa o acumulador
   int cont; // 'cont' representa o contador
   float media;
   scanf("%d", &num);
   cont = 0;
   soma = 0;
   while( num != 0)
   {
      cont = cont + 1;   // contador
      soma = soma + num; // acumulador
      scanf("%d", &num);
   }
   media = soma / (float) cont;
   printf("A media dos numeros lidos foi %.2f.", media);
   return 0;
}
```

Problema 2

Desenvolver um algoritmo para ler um número inteiro positivo, calcular e imprimir a soma de seus algarismos.

Obs.:

- Por exemplo, para o número 123, a saída deve ser: 1+2+3 = 6.
- Como separar os dígitos?

123 mod 10 = **3**
123 div 10 = 12

12 mod 10 = **2**
12 div 10 = 1

1 mod 10 = **1**
1 div 10 = **0**

3 + 2 + 1 = 6

Solução em pseudolinguagem

```
principal
{
   inteiro NUM, DIGITO, SOMA;
   SOMA ← 0;
   leia(NUM);
   enquanto (NUM ≠ 0) faça
   {
      DIGITO ← NUM mod 10; \\obtém ultimo digito
      NUM ← NUM div 10; \\elimina o ultimo digito
      SOMA ← SOMA + DIGITO;
   }
   imprima (SOMA);
}
```

Teste do Algoritmo

Considerando que serão fornecidos os seguintes dados de entrada: 374

O resultado do teste seria:

NUM	DIGITO	SOMA	Imprime
		0	
374	4	4	
37	7	11	
3	3	14	
0			14

Código em C:

```
#include <stdio.h>

int main()
{
   // 'digito' representa o digito mais a direita de 'num'
   int digito, num, soma;
   printf("Digite um número inteiro: ");
   scanf("%d", &num);
   soma = 0;
   while( num != 0)
   {
      digito = num % 10; // obtém o ultimo digito
      num = num / 10;    // elimina o ultimo digito
      soma = soma + digito;
   }
   printf("A soma dos digitos foi %d.", soma);
   return 0;
}
```

Repetição com Teste no Final

Sintaxe

Pseudolinguagem

```
faça
{
   C ;
}enquanto(condição) ;
```

Linguagem C

```
do
{
   C;
} while (condição);
```

Semântica

Enquanto a condição for verdadeira, a seqüência será repetida. Quando a condição fornecer resultado falso, o controle sai da estrutura passando para o comando seguinte.

Exemplo 1

Desenvolver um algoritmo para ler 100 números inteiros, calcular e imprimir o quadrado de cada número lido.

Solução em Pseudolinguagem:

```
principal
{
   inteiro NUM, CONT;
   CONT ← 0;
   faça
   {
      leia(NUM);
      imprima(potencia(NUM,2));
      CONT ← CONT + 1;
   } enquanto (CONT < 100);
}
```

Solução em C:

```
#include <stdio.h>
int main()
{
   int num, cont;
   cont = 0;
   do
   {
      scanf("%d", &num);
      printf("\n%d ao quadrado = %d", num, num*num);
      cont = cont + 1;
   }while (cont < 100);
   return 0;
}
```

Exemplo 2

O algoritmo a seguir utiliza o *faça-enquanto* para imprimir os valores inteiros que se encontram entre um valor inicial e um valor final.

Solução em Pseudolinguagem:

```
principal
{
   inteiro NUM_INICIAL, NUM_FINAL, I;
   leia(NUM_INICIAL);
   leia(NUM_FINAL);
   I ← NUM_INICIAL;
   faça
   {
      imprima ("Valor de I: ",I);
      I ← I + 1;
   }enquanto (I ≤ NUM_FINAL);
}
```

Solução em C:

```
#include <stdio.h>

int main()
{
   // 'digito' representa o digito mais a direita de 'num'
   int num_inicial, num_final, i;
   printf("Digite o número inicial: ");
   scanf("%d", &num_inicial);
   printf("Digite o número final: ");
   scanf("%d", &num_final);
   i = num_inicial;
   do
   {
      printf("valor de i: %d\n", i);
      i++; // ou i = i + 1;
   } while(i <= num_final);
   return 0;
}
```

Repetição com Variável de Controle

Sintaxe

Pseudolinguagem

```
para (V ← I;  V ≤ L;  V ← I+P) faça
{
   C;
}
```

Linguagem C

```
for (V = I; V <= L; V = V + P)
{
   C;
}
```

Sendo que V é a variável de controle, I é o valor inicial de V, L é o valor limite de V e P é o incremento sofrido por V após cada execução da ação C (bloco ou domínio da estrutura).

Semântica

Semântica: o controle de fluxo de execução entra na estrutura e faz a etapa de inicialização uma única vez (V ← I) iniciando a estrutura de repetição na seguinte seqüência:
(1) executa o teste (V ≤ L). Se for válido, vai para o passo seguinte fora do bloco. Senão, executa o primeiro comando após a estrutura de repetição;
(2) executa a ação C;
(3) executa o incremento/decremento (V ← V + P);
(4) retorna ao passo 1.

Exemplo 1:

O algoritmo a seguir utiliza o para-faça para imprimir os valores inteiros que se encontram entre um valor inicial e um valor final.

Solução em Pseudolinguagem:

```
principal
{
   inteiro NUM_INICIAL, NUM_FINAL, I;
   leia(NUM_INICIAL);
   leia(NUM_FINAL);
   para (I ← NUM_INICIAL; I ≤ NUM_FINAL; I ← I+1) faça
   {
      imprima ("Valor de I: ",I);
   }
}
```

Solução em C:

```
#include <stdio.h>

int main()
{
   int num_inicial, num_final, i;
   scanf("%d", &num_inicial);
   scanf("%d", &num_final);
   for( i = num_inicial; i <= num_final; i++)
   {
      printf("valor de i: %d\n", i);
   }
   return 0;
}
```

Exemplo 2:

Fazer um algoritmo para calcular e imprimir a tabuada de multiplicação do número TAB entre 1 e NUM.

Solução em Pseudolinguagem:

```
principal
{
   inteiro I, TAB, NUM;
   leia(TAB);
   leia(NUM);
   para (I ← 1;  I <= NUM; I ← I + 1) faça
   {
      imprima (I, " x ", TAB, " = ", I*TAB);
   }
}
```

Solução em C:

```
#include <stdio.h>

int main()
{
   int i, tab, num;
   printf("Tabuada de: ");
   scanf("%d", &tab);
   printf("Ate que numero: ");
   scanf("%d", &num);
   for( i = 1; i <= num; i++)
   {
      printf("%d x %d = %d\n", i, tab, i*tab);
   }
   return 0;
}
```

Estruturas de Controle (Resumo)

Uma estrutura de controle controla o fluxo de execução dos comandos que constituem o seu domínio (ou bloco).

Podem ser:

1. Seqüência Simples.
2. Alternativa:
   - 2.1. Simples (se).
   - 2.2. Dupla (se-senão).
   - 2.3. Múltipla Escolha (escolha).
3. Repetição:
   - 3.1. Com Teste no Início (enquanto-faça).
   - 3.2. Com Teste no Final (faça-enquanto).
   - 3.3. Com Variável de Controle (para-faça).

Em qualquer estrutura de controle, só existe um ponto de entrada e um ponto de saída do fluxo de execução.

Formato geral de um algoritmo (programa):

```
principal
{
   declaração de variáveis;
   inicialização de variáveis;
   corpo do algoritmo controlando:
   leitura;
   processamento;
   saída;
}
```

O corpo do algoritmo é constituído exclusivamente de estruturas de controle.

Só existem duas maneiras de se ligar duas estruturas de controle do corpo de um algoritmo:

1) Em seqüência:

...
início da estrutura 1
fim da estrutura 1
início da estrutura 2
fim da estrutura 2
...

2) Encadeadas (concatenadas):

...
início da estrutura 1
  início da estrutura 2
  fim da estrutura 2
fim da estrutura 1
...

Nos próximos capítulos utilizaremos várias estruturas em sequência e encadeadas para resolver nossos problemas. Opta-se por utilizar uma estratégia ou outra dependendo das necessidades do algoritmo.

Exercícios

1. Fazer um algoritmo que imprima todos os números pares no intervalo 1-100.

2. Fazer um algoritmo que imprima todos os números de 100 até 1.

3. Ler um número inteiro e positivo e verificar se este é ou não um número primo.

4. Dada uma dívida de 10000 reais que cresce a juros de 2,5% ao mês e uma aplicação de 1500 reais com rendimento de 4% ao mês, escrever um algoritmo que determina o número de meses necessários para pagar a dívida.

5. Calcular o valor de S:

   $S = 2 / 50 + 2^2/48 + 2^3/46 + \ ... + 2^{25}/2$

6. Escrever um algoritmo que lê um valor N inteiro e positivo e que calcula e escreve o valor de E.

   $E = 1 + 1/2 + 1/3 + ... + 1/N$

7. Escreva um algoritmo que leia 10 valores (usando a mesma variável) e encontre o maior e o menor deles. Mostre o resultado.

8. Chico tem 1,50 metro e cresce 2 centímetros por ano, enquanto Zé tem 1,40 metro e cresce 3 centímetros por ano. Construa um algoritmo que calcule e imprima quantos anos serão necessários para que Zé seja maior que Chico.

9. Escreva um algoritmo que leia a matrícula de um aluno e suas três notas. Calcule a média ponderada do aluno, considerando que o peso para a maior nota seja 4 e para as duas restantes, 3. Mostre ao final a média calculada e uma mensagem "APROVADO" se a média for maior ou igual a 5 e "REPROVADO" se a média for menor que 5. Repita a operação até que o código lido seja negativo.

10. Faça um programa que, dado um conjunto de valores inteiros e positivos (fornecidos um a um pelo usuário), determine qual o menor valor do conjunto. O final do conjunto de valores é conhecido através do valor zero, que não deve ser considerado.

11. A conversão de graus Fahrenheit para Centígrados é obtida pela fórmula
    $C = 9/5(F – 32)$. Escreva um programa que calcule e escreva uma tabela de graus centígrados em função de graus Fahrenheit que variem de 50 a 150 de 1 em 1.

12. Elabore um programa que calcule N! (fatorial de N), sendo que o valor inteiro de N é fornecido pelo usuário. Sabendo que:

    N! = N x (N-1) x (N-2) x ... x 3 x 2 x 1;

    0! = 1, por definição.

13. Fazer um programa para calcular e mostrar os N primeiros termos da série de Fibonacci. O número N é fornecido pelo usuário. A série de Fibonacci é gerada da seguinte forma:

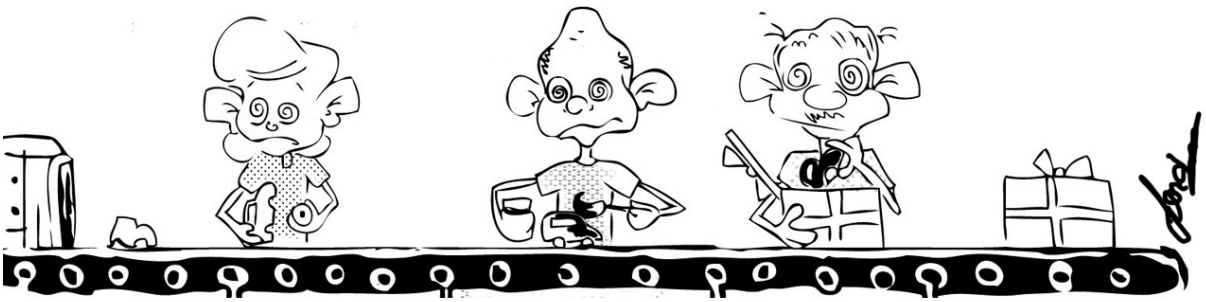
    f1 = f2 = 1;

    f3 = f1 + f2 = 2;

    f4 = f2 + f3 = 3;....

    O primeiro e segundo termos valem 1 e os seguintes são calculados somando os dois termos anteriores.

5. Subrotinas (Procedimentos e Funções)



João possui uma fábrica de carrinhos de brinquedos e montava seus carrinhos sempre sozinho. Para otimizar a produção de sua fábrica, João contratou 3 funcionários para realizarem operações específicas. O primeiro funcionário tem a função de montar o carrinho, o segundo tem a função de pintar e o terceiro de embalar. Desta forma, João conseguiu aumentar a produção da sua empresa pois cada funcionário tinha uma função bem específica na linha de produção e exercia esta função quantas vezes fosse preciso. João poderá solicitar o trabalho de montagem, pintura e empacotamento dos brinquedos sempre que for necessário.

A idéia de criar funções que são utilizadas para determinadas tarefas é amplamente utilizada no desenvolvimento de algoritmos através de *Subrotinas*, sendo este o tema deste capítulo. Vamos aprender aqui como criar e utilizar subrotinas em nossos algoritmos de forma a permitir uma melhor "modularização" e reaproveitando dos nossos códigos.

Definição

Podemos definir subrotina como sendo uma parcela de código computacional que executa uma tarefa bem definida, sendo que essa tarefa pode ser executada (chamada) diversas vezes num mesmo programa.

Uma das razões que motivou a utilização de subrotinas na construção de algoritmos é a necessidade de dividir um problema computacional em pequenas partes. Essa divisão é interessante, pois muitas destas pequenas partes tendem a se repetir durante a execução de um determinado código. Alguns exemplos de tarefas que comumente se repetem são: Impressão de mensagens, zerar um vetor, fazer uma operação matricial, etc.

Podemos sintetizar a necessidade de uso de subrotinas quando precisarmos de:

- Utilizar uma parte do código em várias partes do algoritmo;
- Disponibilizar os mesmos códigos para vários algoritmos;
- Abstrair a complexidade e facilitar o entendimento do problema.

A utilização de subrotinas facilita a programação estruturada. Dada as fases previstas nos refinamentos sucessivos decompõe-se o algoritmo em módulos funcionais. Tais módulos podem ser organizados/codificados como subrotinas, ou seja, o uso de subrotinas viabiliza a modularização dos códigos.

Características das subrotinas

As subrotinas devem executar tarefas bem definidas. Subrotinas não funcionam sozinhas, sendo sempre chamadas por um programa principal ou por outra subrotina.

Elas permitem a criação de variáveis próprias e a manipulação de variáveis externas (devidamente parametrizadas).

Além disso, subrotinas facilitam a legibilidade do código através da estruturação (subrotinas são agrupadas fora do programa principal) e do enxugamento (através de diversas chamadas da mesma subrotina).

Tipos de subrotinas

Existem dois tipos de subrotinas: **Procedimentos e Funções**.

Procedimentos são subrotinas que não retornam nenhum valor. São usados para realizar alguma operação que não gera dados (por exemplo, impressão de uma mensagem na tela).

As funções, por sua vez, sempre retornam algum valor. São utilizadas para realizar uma operação e retornam alguma resposta relativa à operação realizada (por exemplo, retornam o fatorial de número).

Veremos a seguir em detalhes como funcionam os procedimentos e as funções.

Procedimento - forma geral

A sintaxe de declaração de um procedimento é a seguinte:

```
<nome do procedimento> (<lista de parâmetros>)
{
   <declaração de variáveis locais>
   <comandos>
}
```

Exemplo

```
imprimeMaior(inteiro X, inteiro Y)
{
  se (X > Y)
  {
     imprima(X);
  }
  senão
  {
     imprima(Y);
  }
}
```

Veja agora um exemplo completo utilizando procedimentos. O exemplo é bem simples e ilustra um algoritmo que, através de um procedimento, imprime o maior valor entre dois valores inteiros passados como parâmetro.

```
imprimeMaior(inteiro X, inteiro Y)
{
  se (X > Y)
  {
     imprima(X);
  }
  senão
  {
     imprima(Y);
  }
}

principal
{
   inteiro A, B;
   leia(A,B);
    imprimeMaior(A,B);
}
```

Variáveis locais

Toda variável pertencente a um procedimento é chamada de local, pois ela só pode ser utilizada dentro do escopo do procedimento. Fazem parte das variáveis locais de um procedimento as variáveis declaradas dentro do procedimento e os parâmetros recebidos por ele.

Procedimento – teste de mesa

Vamos ver agora um teste de mesa para entendermos em que ordem um algoritmo completo seria executado se implementado. O índice que aparece do lado esquerdo do código indica um ponto de parada, sendo P1 o ponto de partida, P2 a segunda linha a ser executada e assim por diante. Logo abaixo você encontrará uma tabela representando o conteúdo de cada um das variáveis utilizadas neste exemplo. Esta forma de representação será utilizada várias vezes neste capítulo.

```
P6  imprimeMaior(inteiro X, inteiro Y,ref inteiro Z)
    {
P7     se (X > Y)
       {
P8          imprima(X);
P9        Z ← X;
       }
       senão
       {
P8'       imprima(Y);
P9'       Z ← Y;
       }
    }

P1  principal
    {
P2     inteiro A, B, C;
P3     A ← 22;
```

```
P4      B ← 11;
P5      imprimeMaior(A,B,C);
P10     imprima(C); //imprima(22)
    }
```

Linha	principal			imprimeMaior		
	A	B	C	X	Y	Z
P1	-	-	-	-	-	-
P2	“lixo”	“lixo”	“lixo”	-	-	-
P3	22	“lixo”	“lixo”	-	-	-
P4	22	11	“lixo”	-	-	-
P5	22	11	“lixo”	-	-	-
P6	22	11	“lixo”	22	11	“lixo”
P7	22	11	“lixo”	22	11	“lixo”
P8	22	11	“lixo”	22	11	“lixo”
P9	22	11	22	22	11	22
P8’						
P9’						
P9	22	11	22	-	-	-

Parâmetros de uma subrotina

É importante ressaltar que os parâmetros passados para uma subrotina (seja ela um procedimento ou uma função) podem ser feito por cópia ou por referência.

Nas passagens por valor, é passado uma cópia da variável para a subrotina, ou seja, é feito uma cópia do argumento para o parâmetro. Qualquer alteração feita no parâmetro não reflete em alteração no argumento.

Em passagens por referência, todas as alterações realizadas no parâmetro refletem em alterações no argumento, ou seja, ambas as variáveis apontam para o mesmo endereço de memória. Para isso, é necessário utilizar o modificar ***ref*** (em pseudolinguagem), indicando que será uma referência a uma variável.

Voltaremos a tratar este assunto mais a frente neste capítulo.

Procedimentos em C

A declaração de procedimentos em C é semelhante à forma como é feita em pseudolinguagem, com a diferença que colocamos a palavra *void* (que significa "vazio") ao lado do nome do procedimento.

```
void imprime (int val)
{
   printf(“%d”,val);
}
```

A passagem de parâmetro também é feita de forma análoga à vista em pseudolinguagem.

Observe a seguir um exemplo de utilização de procedimentos em C. Neste código o procedimento imprime na tela se o número passado como parâmetro é par ou ímpar. Do lado esquerdo da tabela é mostrado quais variáveis são "enxergadas" pela função principal e pelo procedimento. Observe que a variável *x* só existe dentro da função principal. Ela é passada para

a função *ParImpar* e seu conteúdo é copiado para a variável *a.* Dentro do procedimento também foi criado uma variada inteira chamada *resto* que só existe dentro do seu escopo.

```
#include <stdio.h>
void ParImpar(int a)
{
  int resto;
  resto=a%2;
  if (resto==0)
     printf("par");
  else
     printf( "ímpar");
}
```

Variáveis locais:

a (cópia da variável x)

resto

```
int main()
{
  int x;
  scanf("%d",&x);
  ParImpar(x);
  return 0;
}
```

Variáveis do main:

x

Funções – forma geral

Funções são subrotinas que possuem um valor de retorno. A sintaxe de declaração de uma função é a seguinte:

```
<tipo básico> <nome da função> (<lista de parâmetros formais>)
{
   <declaração de variáveis locais>
   <comandos>
   retorne <valor de retorno>;
}
```

Funções – teste de mesa

A função abaixo calcula o valor absoluto de uma variável *X* passada como parâmetro. A função retorna este valor para quem a chamou, neste caso, a função principal. O valor retornado é copiado para a variável *D* e posteriormente impresso na tela.

```
P5  real valorAbsoluto ( real X )
    {
P6    real Y;
P7    se (X ≥ 0)
      {
P8      Y ← X;
      }
      senão
      {
P8'     Y ← - X;
      }
P9    retorne Y;
    }
P1  principal
    {
P2     real A,D;
P3     A ← -40;
```

```
P4      D ← valorAbsoluto(A);
P10     imprima (D);
    }
```

Linha	principal		valorAbsoluto	
	A	D	X	Y
P1	-	-	-	-
P2	“lixo”	“lixo”	-	-
P3	-40	“lixo”	-	-
P4	-40	40	-	-
P5	-40	???	-40	-
P6	-40	???	-40	“lixo”
P7	-40	???	-40	“lixo”
P8				
P8’	-40	???	-40	40
P9	-40	40	-40	40
P10	-40	40		

Veja abaixo um exemplo de funções em C.

```
int soma (int a, int b)
{
   return (a+b);
}
```

Em C, chamadas por referência tem como característica alterar o conteúdo da variável que é passada como parâmetro. Veja no exemplo abaixo um código que faz a troca do conteúdo de duas variáveis. Observe que as variáveis são passadas com o símbolo "&". Desta forma, o endereço em memória da variável é passado ao invés do seu conteúdo. Na subrotina, para representar que a passagem é feita por referência, coloca-se o símbolo "*", simbolizando um ponteiro para uma determinada posição da memória.

```
#include <stdio.h>

void Troca(int *a1, int *b1)
{
  int aux=*a1;
  *a1=*b1;
  *b1=aux;
}

int main()
{
  int a=25, b=50, c=75;
  Troca(&a,&b);
  Troca(&b,&c);
  printf(“%d,%d,%d”,a,b,c);
  return 0;
}
```

Existem duas formas de utilizarmos subrotinas em C. Você pode definir a função por completo antes da função principal (coluna da esquerda da tabela a seguir) ou colocar apenas a declaração (protótipo) da subrotina antes da função principal e definí-la após (coluna da direita).

```
#includes.....

int Func1(int a)
{
   ........
}

main()
{
   ......
   x=Func1(10);
   .....
}
```

```
#includes.....

int Func1(int a); //protótipo

main()
{
  .....
  x=Func1(10);
  .....
}

int Func1(int a)
{
  .....
}
```

Exercícios

1. Escreva um procedimento que receba como parâmetro dois valores inteiros n1 e n2 e imprima o intervalo fechado entre eles, do menor para o maior.
   Por exemplo: se n1 = 2 e n2 = 5, o procedimento irá imprimir 2, 3, 4, 5.

2. Faça um procedimento que receba a idade de uma pessoa em dias e retorna essa idade expressa em anos, meses e dias.

3. Faça um procedimento que receba por parâmetro o tempo de duração de um experimento expresso em segundos e imprima na tela esse mesmo tempo em horas, minutos e segundos.

4. Faça uma função que receba por parâmetro o raio de uma esfera e calcula o seu volume: V = (4 * PI * R3) /3.

5. Faça uma função que receba a idade de uma pessoa em anos, meses e dias e retorna essa idade expressa em dias.

6. Faça uma função que recebe a média final de um aluno por parâmetro e retorna o seu conceito, conforme a tabela a seguir:

Nota	Conceito
De 0 a 49	D
De 50 a 69	C
De 70 a 89	B
De 90 a 100	A

7. Escrever uma função `int contaimpar (int n1, int n2)` que retorna o número de inteiros impares que existem entre *n1* e *n2* (inclusive ambos, se for o caso). A função deve funcionar inclusive se o valor de *n2* for menor que *n1*.

```
n=contaimpar(10,19); /* n recebe 5 (11,13,15,17,19) */
n=contaimpar(5,1);    /* n recebe 3 (1,3,5) */
```

8. Escrever uma função `int divisão (int dividendo, int divisor, int * resto)`, que retorna a divisão inteira (sem casas decimais) de dividendo por divisor e armazena no parâmetro resto, passado por referência, o resto da divisão.

```
int r, d;
d = divisao(5, 2, &r);
printf("Resultado:%d Resto:%d", d, r); /* Resultado:2 Resto:1 */
```

9. Escrever uma função `int somaintervalo(int n1, int n2)` que retorna a soma dos números inteiros que existem no intervalo fechado entre *n1* e *n2*. A função deve funcionar inclusive se o valor de *n2* for menor que *n1*.

```
n=somaintervalo(3, 6); /* n recebe 18 (3 + 4 + 5 + 6) */
n=somaintervalo(5,5);  /* n recebe 5 (5) */
n=somaintervalo(-2,3); /* n recebe 3 (-2 + -1 + 0 + 1 + 2 + 3) */
n=somaintervalo(4, 0); /* n recebe 10 (4 + 3 + 2 + 1 + 0) */
```

10. Escreva uma função que receba como parâmetro um valor n inteiro e positivo e que calcule a seguinte soma: S := 1 + 1/2 + 1/3 + 1/4 + ... + 1/n. A função deverá retornar o valor de S.

11. Faça um programa que apresente o seguinte “Menu” para o usuário:

    Escolha uma opção de cálculo para dois números:

    1) Soma

    2) Produto

    3) Quociente

    4) Sair

    Opção:__

    O “Menu” acima deve ser apresentado para o usuário enquanto ele não escolher a opção 4 (sair do programa). O usuário fornecerá 2 números se escolher as opções de cálculo 1, 2 ou 3. Para cada opção de cálculo deve existir (obrigatoriamente) uma função definida (soma, produto e quociente dos dois números fornecidos pelo usuário). O resultado do cálculo deve ser escrito na tela.

12. Idem ao exercício 5, agora com o seguinte “Menu”

    Escolha uma opção de cálculo para um número:

    a) Fatorial

    b) Exponencial

    c) Raiz quadrada

    d) Sair

    Opção: __

    Nas opções “b” e “c” pode-se utilizar funções predefinidas do C.

6. Vetores Numéricos

Campeonato

Carro 1 - 57 pontos
Carro 2 - 30 pontos
Carro 3 - 52 pontos
Carro 4 - 29 pontos
Carro 5 - 38 pontos

Maria precisa armazenar os dados de uma corrida de carros que está sendo organizada em sua cidade. Para isso, ela precisa criar uma tabela contendo o número dos carros e a quantidade de pontos que cada um dos carros vai acumulando com as corridas. Maria teve a idéia de fazer um pequeno sistema para controlar esta tabela. Maria poderia criar uma variável para cada carro e ir somando a quantidade de pontos de cada carro por variável, mas o controle destas variáveis seria complicado. Maria então fez uso de uma nova estrutura chamada *vetor.* Nesta estrutura, Maria poderia guardar os pontos de todos os carros usando apenas uma única variável com vários índices.

Neste capítulo veremos como criar e em quais situações é interessante utilizarmos esta nova estrutura.

Introdução

Em diversas situações os tipos básicos de dados (inteiro, real, caracter, ...) não são suficientes para representar a informação que se deseja armazenar. Por exemplo, se desejarmos armazenar uma quantidade grande de números, fica inviável fazê-lo com uma variável por cada valor. Neste tipo de situação, é interessante usar outras estratégias para armazenar este tipo de valor.

Existe a possibilidade de construção de novos tipos de dados a partir da composição (ou abstração) de tipos de dados primitivos. Neste capítulo introduziremos o conceito de Vetores, mostrando em que situação este tipo de estrutura deve ser utilizado e quais as vantagens encontradas na utilização desta estrutura.

Motivação

Para entendermos a necessidade da utilização de vetores, observe atentamente os dois problemas abaixo:

Problema 1: Como poderia ser feito um algoritmo para ler 50 notas de uma turma e calcular sua média?

```
principal
{
    inteiro i;  //variável de controle
    real nota, media, soma = 0;
    para (i ← 1; i <= N; i←i+1) faça
    {
        imprima("Digite uma nota:");
        leia(nota);
        soma ← soma + nota;
    }
    media ← soma/N;
    imprima("Media = ", media);
}
```

Problema 2: Fazer um programa para ler 50 notas de uma turma e calcular a sua média. Imprimir as notas lidas juntamente com a média da turma como na tabela abaixo.

Nota	Media
8.0	7.75
4.6	7.75
2.3	7.75
3.7	7.75
7.8	7.75
9.0	7.75
....	...

Veja que com o que aprendemos até agora, teríamos que criar uma variável para cada nota, impossibilitando o uso de estruturas de repetição. Imagine se ao invés de 50 notas fossem 50.000? Ficaria impraticável construir um programa para tratar esta quantidade de dados. A seguir veremos formos de resolver este tipo de problema.

Variáveis Compostas Homogêneas

Quando uma determinada estrutura de dados for composta de variáveis com o mesmo tipo primitivo, tem-se um conjunto homogêneo de dados. Estas variáveis são chamadas de variáveis compostas homogêneas.

As variáveis compostas homogêneas unidimensionais são utilizadas para representar arranjos unidimensionais de elementos de um mesmo tipo, em outras palavras, são utilizadas para representar **vetores** (ou *arrays* em inglês).

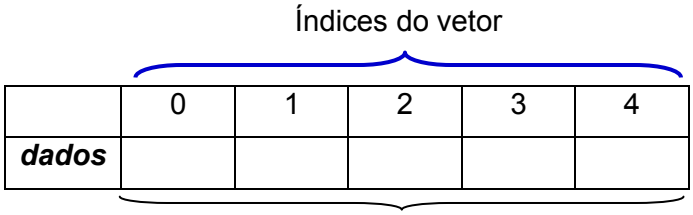
Vetores

A sintaxe para declaração de uma variável deste tipo, tanto na pseudolinguagem quanto em C é a seguinte:

```
tipo identificador [qtd de elementos];
```

Exemplo:

```
// vetor com 5 elementos do tipo inteiro
inteiro dados[5] ; // em C seria ficaria int dados[5];
```



*5 elementos quaisquer do tipo **int***

Cada um dos elementos de um vetor é referenciado individualmente por meio de um número inteiro entre colchetes após o nome do vetor.

Exemplos:

	0	1	2	3	4
dados	**3**	**2**	**4**	**7**	**1**

Considerando o vetor *dados*, veja algumas formas de fazer atribuições a outras variáveis a partir dos valores armazenados no vetor:

Pseudolinguagem

```
X ← dados[1];
Y ← dados[4];
```

Linguagem C

```
X = dados[1];
Y = dados[4];
```

A instrução a seguir atribui um valor ao elemento 0 (zero) do vetor dados:

Pseudolinguagem

```
dados[0] ← 6; ou
I ← 0;
dados[i] ←  6;
```

Linguagem C

```
dados[0] = 6; ou
I = 0;
dados[i] = 6;
```

Observações na implementação de vetores em C:

Os vetores NÃO são automaticamente inicializados!

Se houver menos valores do que o número de elementos do vetor, os elementos restantes são inicializados automaticamente com o valor zero.

```
int n[5] = {32, 64, 27};
```

A seguinte declaração causa um erro de sintaxe:

```
int n[5] = {32, 64, 27, 18, 95, 14};
```

O tamanho do vetores pode ser omitido:

```
int n[] = {1, 2, 3, 4, 5};
```

Exemplos

Exemplo 1: O programa a seguir, usa o comando *para* para inicializar com zeros os elementos de um array inteiro n de 10 elementos e o imprime sob a forma de uma tabela.

```
principal
{
  inteiro n[10], i;

  para (i←0; i <= 9; i←i+1) faça
  {
    n[i] ← 0;
  }

  imprima("Elemento   Valor");
  para (i←0; i <= 9; i←i+1) faça
  {
    imprima(i,"     ", n[i]);
  }
}
```

```
Elemento  Valor
      0     0
      1     0
      2     0
      3     0
      4     0
      5     0
      6     0
      7     0
      8     0
      9     0
```

Exemplo anterior em C:

```
//inicializando um vetor (array)
#include <stdio.h>

int main()
{
  int n[10], i;

  for (i=0; i<= 9; i++)
  {
     n[i] = 0;
  }
  printf("%s%13s\n","Elemento", "Valor");
  for (i=0; i<= 9; i++)
  {
     printf("%7d%13d\n",i,n[i]);
  }
  return 0;
}
```

Exemplo 2: O programa a seguir inicializa os dez elementos de um *array s* com os valores: 2, 4, 6, ..., 20 e imprime o *array* em um formato de tabela.

```
principal
{
   constante inteiro TAMANHO ← 10;
   inteiro s[TAMANHO], j;
   para (j←0; j<=TAMANHO - 1; j←j+1) faça
   {
      s[j] ←  2 + 2*j;
   }
   imprima("Elemento   Valor");
   para (j←0; j<= TAMANHO-1; j← j+1) faça
   {
      imprima (j, "    ",s[j]);
   }
}
```

```
Elemento  Valor
      0     2
      1     4
      2     6
      3     8
      4     10
      5     12
      6     14
      7     16
      8     18
      9     20
```

Exemplo anterior em C:

```
#include <stdio.h>

#define TAMANHO 10
int main()
{
    int s[TAMANHO], j;
    for (j=0; j<= TAMANHO - 1; j++)
    {
        s[j] = 2 + 2*j;
    }

    printf("%s%13s\n","Elemento", "Valor");
    for (j=0; j<= TAMANHO - 1; j++)
    {
        printf("%8d%13d\n",j,s[j]);
    }
    return 0;
}
```

Vetores e Subrotinas

Em pseudolinguagem, a passagem de parâmetros em procedimentos e funções é feita por cópia. Para realizar a passagem por referência, utilize a palavra ***ref*** antes do tipo de vetor. Em C, vetores são passados sempre por referência, ou seja, as modificações feitas na subrotina refletem nos dados do vetor passado como parâmetro pela função chamadora.

No exemplo a seguir é apresentado o procedimento *imprimeVetor* que imprime um vetor de tamanho *tam*.

```
imprimeVetor (inteiro vet[], inteiro tam)
{
   inteiro i;
   para (i ← 0; i <= tam - 1; i ← i+1) faça
   {
      imprima (vet[i]);
   }
}

principal
{
   constante inteiro TAMANHO ← 10;
   inteiro s[TAMANHO], j;
   para (j ← 0; j <= TAMANHO - 1; j ← j+1)
   {
      imprima ("Informe o valor do vetor na posição ", j);
      leia (s[j]);
   }
   imprimeVetor(s, TAMANHO);
}
```

Exemplo anterior em C:

```
#include <stdio.h>
#define TAMANHO 10
void imprimeVetor (int vet[], int tam)
{
   int i;
   for (i = 0; i <= tam - 1; i++)
   {
      printf("%d\n", vet[i]);
   }
}

int main()
{
   int s[TAMANHO], i;
   for (i = 0; i <= tam - 1; i++)
   {
      printf ("Informe o valor do vetor na posição %d: ", i);
      scanf ("%d",  &s[j]);
   }
   imprimeVetor(s, TAMANHO);
}
```

Exercício Resolvido 1

Criar uma função que receba um vetor de números reais e seu tamanho e retorne o índice do maior valor contido no vetor. Se houver mais de uma ocorrência do maior valor, retornar o índice do primeiro. Faça um programa *principal* para testar a função.

Solução Proposta:

```
inteiro encontraMaior (real vet[], inteiro tam)
{
   inteiro indice, i;
   real maior = vet[0];
   indice ← 0;
   para (i ← 1; i < tam; i ← i+1) faça
   {
      se( vet[i] > maior)
      {
         maior ← vet[i];
         indice ← i;
      }
   }
   retorne indice;
}

principal
{
   real vetor[6] = {3.0, 4.3, 5.6, 2.8, 7.9, 3.4};
   inteiro posicao;
   posicao ← encontraMaior( vetor, 6)
   imprima("Maior valor esta na posicao ", posicao);
}
```

Exercício Resolvido 2

Criar uma função em C que receba um vetor de números reais e um valor inteiro representando o seu tamanho. Essa função deverá ordenar o vetor em ordem crescente.

```
#include <stdio.h>

void ordena(float vet[], int tam)
{
   int i, j;
   float aux;
   for(i = 0; i <= (tam-2); i++)
   {
      for(j = tam-1; j > i; j--)
      {
         if ( vet[j] < vet[j-1] )
         {
            aux=vet[j];
            vet[j]= vet[j-1];
            vet[j-1]=aux;
         }
      }
   }
}

int main()
{
   int i;
   float vet[5]={11.0,22.0,3.0,44.0,5.0};
   ordena(vet, 5);
   for (i=0; i < 5; i++)
      printf("%.2f\n",vet[i]);
   return 0;
}
```

Exercícios

1. Quais são os elementos do vetor referenciados pelas expressões a seguir?

vet	1	2	4	7	4	2	8	9	0	6	5	4

a) vet[3] b) vet[0] c) vet[11]

2. Qual é a diferença entre os números “3” das duas instruções a seguir ?

```
inteiro vet[3];
vet[3] ← 5;
```

3. Dada um tabela contendo a idade de 10 alunos, faça um algoritmo que calcule o número de alunos com idade superior a média.

4. Faça um algoritmo para ler e somar dois vetores de 10 elementos inteiros. Imprima ao final os valores dessa soma.

5. Refaça o exercício anterior criando um procedimento para efetuar a leitura dos vetores e um segundo procedimento que imprimirá a soma dos vetores.

6. Refaça o exercício anterior criando uma função que receba o vetor com a idade dos alunos e retorne a quantidade de alunos com idade superior a média.

7. Faça um algoritmo que leia, via teclado, 20 valores do tipo inteiro e determine qual o menor valor existente no vetor e imprima valor e seu índice no vetor.

8. Refaça o exercício anterior criando um procedimento que receba como parâmetro o vetor e imprima o menor valor e seu índice no vetor.

9. Desenvolva um programa que leia um vetor de números reais, um escalar e imprima o resultado da multiplicação do vetor pelo escalar.

10. Faça um procedimento que faça a leitura um vetor de 10 elementos inteiros e imprima somente os valores armazenados nos índices pares.

11. Faça um programa que leia um vetor com 15 valores reais. A seguir, encontre o menor elemento do vetor e a sua posição dentro do vetor, mostrando: “O menor elemento do vetor esta na posição XXXX e tem o valor YYYYY.”

12. Faça um programa que leia um vetor de 15 posições (reais) e depois um valor a ser procurado no vetor. Imprima se o valor foi ou não encontrado e a quantidade de vezes que o valor está presente no vetor.

13. Faça uma função que receba um vetor de números inteiros e um valor inteiro. A função deverá procurar este segundo valor neste vetor e retornar seu índice se for encontrado. Se o elemento não for encontrado, retornar -1.

14. Dada uma tabela com as notas de uma turma de 20 alunos, faça funções que retornem:

    A média da turma.

    a quantidade de alunos aprovados (>=60)

    a quantidade de alunos reprovados (< 60)

15. Faça um programa que leia um conjunto de 20 valores e armazene-os num vetor V. Particione-o em dois outros vetores, P e I, conforme os valores de V forem pares ou ímpares. No final, imprima os valores dos 3 vetores.

16. Faça um programa que leia um vetor G[13] que é o gabarito de um teste da loteria esportiva, contendo os valores 1 quando for coluna 1, 0 quando for coluna do meio e 2 quando for coluna 2. Ler a seguir, para 5 apostadores, seu cartão de apostas (R[13]) e depois da leitura imprimir quantos acertos o apostador teve.

17. Com relação ao exercício anterior, calcule e mostre o percentual dos apostadores que fizeram de 10 a 13 pontos e o percentual dos apostadores que fizeram menos do que 10 pontos.

18. Faça um programa que leia um vetor de valores inteiros e imprima-o na ordem crescente. O vetor deve ter tamanho N (utilize a diretiva #define).

7. Vetores de caracteres

Maria ficou responsável por preparar o aniversário surpresa de sua irmã mais nova. Para tal, Maria comprou bolo, refrigerantes e fez umas bandeirinhas com a palavra PARABÉNS para sua irmã. Ao chegar em casa, sua irmã ficou tão feliz com a surpresa que nem viu a bandeirinha feita por Maria.

Supondo que quiséssemos armazenar em um programa a palavra que Maria escolheu para o aniversário de sua irmã, como poderíamos fazer? Com o que vimos até agora, poderíamos criar uma variável para cada letra. Funcionaria. Mas se o texto a ser armazenado for maior, fica complicado criar uma variável para cada letra. Como visto no capítulo anterior, aprendemos que é possível criar uma única estrutura para armazenar vários números. Esta mesmo estrutura pode ser utilizada para armazenar caracteres - são vetores de caracteres. Neste capítulo aprenderemos como utilizar estes vetores em seus algoritmos.

Cadeias de caracteres

Uma cadeia de caracteres é uma sequência de caracteres justapostos e são fundamentais no desenvolvimento de programas computacionais.

Exemplos de cadeias de caracteres (representadas internamente num programa):

Mensagem de e-mail;

Texto de um programa;

Nome e endereço em cadastro de clientes, alunos, etc...

- Sequencia genética. Um gene (ou o DNA de algum organismo) é composto de sequencias dos caracteres A, T, G e C (nucleotídeos)

Uma variável usada para armazenar um caractere é representada da seguinte maneira:

```
caracter letra; //a variável letra do tipo caracter
letra ← 'a'; // atribuída a letra "a" a variável letra
```

Se em uma variável do tipo *caracter* pode-se armazenar somente um caractere, então para armazenar vários caracteres (ex: “jose”, “carro”) é necessário utilizar as cadeias de caracteres, representadas por vetores do tipo caracter.

Sintaxe para declaração de cadeia de caracteres:

```
caracter identificador[qtde caracteres];
```

Observe a declaração a seguir:

```
caracter cidade[12] ;
```

- A variável **cidade** é um vetor de caracteres (cadeia de caracteres)
- A variável **cidade** pode armazenar qualquer cadeia de até 12 caracteres

cidade

0	1	2	3	4	5	6	7	8	9	10	11
J	u	i	Z		d	e		F	o	r	a

Vamos criar um vetor *nome* e atribuir a ele a palavra "Jose".

```
caracter nome[20] ;
nome ← “Jose”;
```

Observe a presença das aspas duplas na palavra *Jose*. Pode-se ainda obter um caracter qualquer da cadeia de caracteres da seguinte maneira:

```
caracter letra ← nome[1]; // letra receberá o caractere ‘o’.
```

Exemplo:

```
principal
{
    caracter s[20];
    imprima (“Digite uma string: “) ;
    leia (s);
    imprima (“String digitada: “,s);
}
```

Caracteres em C

Alguns caracteres visíveis (podem ser impressos):

	0	1	2	3	4	5	6	7	8	9
30			**s**	**!**	**"**	**#**	**$**	**%**	**&**	**'**
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**	**{**	**|**	**}**				

A diferença entre caracteres e inteiros está apenas na maneira como são tratados, por exemplo, o código:

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

Imprimirá:

```
int : 97 char : a
```

O conteúdo da variável c é impressa em dois formatos distintos: %d (int) e %c (char).

Por questão de portabilidade, não se deve usar diretamente, no programa, o código de caracteres. Ao invés, usa-se uma constante caractere entre apóstrofe. Reescrevendo o fragmento de código anterior:

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

Imprimirá:

```
int : 97 char : a
```

Pode-se trocar ***char*** c por ***int*** c: o resultado será o mesmo.

Entrada/Saída de caracteres em C:

```
ch = getchar();
```

armazena um caractere digitado em **ch** até que ENTER seja pressionado;

```
putchar(ch);
```

O exemplo abaixo imprime o caractere **ch** na tela do computador;

```
int main()
{
   char ch;
   ch = getchar();
   putchar(ch);
   return 0;
}
```

Cadeias de caracteres e Strings

Cadeias de caracteres são similares em C e em pseudolinguagem:

```
char graus[5] = {'A', 'B', 'C', 'A', 'D'};

char bin[8] = {'0','1','0', '1', '1', '1', '0', '1'};
```

Strings

Strings são tipos especiais de cadeias de caracteres em C e são terminadas, obrigatoriamente, pelo caractere nulo: '\0' (\zero). Portanto, deve-se reservar uma posição para este caractere de fim de cadeia.

Em pseudolinguagem, cadeias de caracteres e *strings* são semelhantes. Desta forma, para fins de implementação, o assunto será tratado apenas em C.

Exemplos de *strings* em C:

```
char cidade[4] = {'R', 'i', 'o', '\0'};
char disc[40] = {'A','l','g', 'o', 'r', 'i', 't', 'm', 'o', '\0'};
```

Outras formas de inicializar *strings* em C são:

```
char cidade[4] = "Rio";
char disc[40] = "Algoritmo";
```

Para ilustrar a declaração e a inicialização de *strings*, as seguintes declarações podem ser consideradas:

```
char s1[] = "" ; //2 aspas sem espaços entre elas
char s2[] = "Rio de Janeiro";
char s3[81];
char s4[81] = "Rio";
```

No exemplo acima:

- *s1* armazena uma string vazia. Tem um único elemento: '\0';
- *s2* representa um vetor com 15 elementos (caracteres);
- *s3* representa uma cadeia de caracteres com até 80 caracteres e não é inicializada;
- *s4* também é dimensionada para conter até 80 caracteres e é inicializada com a cadeia "Rio".

Veja abaixo um exemplo de leitura de *strings* (scanf).

```
int main()
{
   char s[20];
   printf("Digite uma string: ");
   scanf("%s",s);  //sem & antes de s
   printf("String digitada: %s",s);
   return 0;
}
```

Neste caso, a leitura será feita até encontrar um caractere branco: espaço (' '), tabulação ('\t') ou nova linha ('\n'). Assim, se for digitado "Rio de Janeiro", s conterá apenas "Rio";

Não é necessário o **&** antes da variável ***s*** ao lermos vetores com o comando *scanf*.

```
int main()
{
   char s[20];
   printf("Digite uma string: ");
   gets(s);
   printf("String digitada: ");
   puts(s);
   return 0;
}
```

Outros comandos para leitura e escrita de *strings*.

gets(s): lê a *string* s a partir do teclado;

puts(s): imprime uma *string* na tela seguida de nova linha.

Exemplo 1: o programa a seguir imprime uma cadeia de caracteres, caractere por caractere:

```
int main()
{
   char s[30];
   int i;
   printf("Digite uma string: ");
   gets(s);
   //imprime cada caractere da cadeia lida
   for(i=0; s[i]!='\0'; i++)
      printf("%c",s[i]);
   return 0;
}
```

O comando *for* acima equivale a printf("%s",s);

Exemplo 2: o programa a seguir calcula e imprime o comprimento (numero de caracteres) de uma cadeia:

```
int main()
{
   char s[30];
   int i, n = 0;
   printf("Digite uma string: ");
   gets(s);

   for(i=0; s[i]!='\0'; i++)
        n++;
   printf("\nO tamanho de \"%s\" eh: %d",s,n);
   return 0;
}
```

Exemplo 3: o programa a seguir faz uma copia de uma cadeia, fornecida pelo usuário, para outra:

```
int main()
{
   char dest[50], //string destino
        orig[50]; //string origem
   int i;
   printf("Digite uma string: ");
   gets(orig);
   //copia cada caractere de orig para dest
   for(i=0; orig[i]!='\0'; i++)
         dest[i] = orig[i];
   //coloca o caractere nulo para marcar o fim da string
   dest[i] = '\0';
   puts(dest);
   return 0;
}
```

Funções para manipulação de Strings

Existem várias funções em C para manipulação de *strings*. Essas funções estão declaradas no arquivo *string.h*. Entre elas pode-se destacar:

- `strcpy(char destino[ ], char origem[ ])`
  Copia a string origem na string destino
- `strlen(char str[ ])`
  Retorna o tamanho da string "str"
- `strcat(char destino[ ], char origem[ ])`
  Faz concatenação (junção) da string origem com a string destino. O resultado é armazenado na string destino

Exercício resolvido 1

Criar uma função que receba como parâmetro uma cadeia de caracteres (*cadeia*), seu tamanho (*tam*) e um segundo caractere (*procurado*). A função deverá retornar a quantidade que o caractere procurado foi encontrado na cadeia.

Solução:

É necessário "varrer" a cadeia de caracteres (estrutura de repetição) e contar quantos são iguais ao caractere procurado, caractere a caractere.

Solução proposta em pseudolinguagem

```
inteiro conta( caracter cadeia[], inteiro tam, caracter procurado )
{
   inteiro encontrados, i;
   i ← 0;
   encontrados ← 0;

   enquanto ( i < tam ) faça
   {
      se (cadeia[i] = procurado)
      {
         encontrados ← encontrados + 1;
      }
      i ← i + 1;
   }
   retorne encontrados;
}
```

Exercício resolvido 2:

Criar uma função para verificar se a *string* **s2** está contida na *string* **s1**. A função deverá retornar 1 se encontrar a *string* ou 0, caso contrário.

Ex.: Se *s1* fosse "Ana Maria Silva" e *s2* fosse "Maria", a função retornaria 1, pois *s2* está contido em *s1*.

Solução proposta em pseudolinguagem

```
int buscaString(char s1[], char s2[])
{
   int i,j,aux, tam1, tam2;
   tam1 = strlen(s1);
   tam2 = strlen(s2);
   for(i=0;i<tam1;i++)
   {
     aux=i;
     for(j=0;j<tam2 && aux<tam1;j++)
     {
        if (s2[j] != s1[aux])
           break;
        aux++;
     }
     if (j == tam2)
       return 1;
   }
   return 0;
}
```

Exercícios

1. Fazer um programa para contar o número de espaços em brancos de uma string.

2. Refaça o programa anterior criando uma função que receberá como parâmetro a string e retornará o número de espaços em branco que a string contem.

3. Fazer um programa para contar o número de vogais numa cadeia de caractere.

4. Refaça o programa anterior criando uma função que receberá como parâmetro a string e retornará o número de vogais que a string contem.

5. Escrever um programa para ler uma string (com mais de uma palavra) e faça com que a primeira letra de cada palavra fique em maiúscula. Para isso, basta subtrair 32 do elemento que deseja alterar para maiúsculo. chrNome[0] = chrNome[0] – 32;
   Exemplo:
   Entrada: lab. de linguagem de programacao
   Saída: Lab. De Linguagem De Programacao

6. Escreva uma função que receba uma string, conte quantos caracteres desta string são iguais a 'a' e substitua os que forem iguais a 'a' por 'b'. A função deverá retornar o número de caracteres modificados.

7. Crie um procedimento que receba uma frase e a exiba na tela de forma soletrada, ou seja, cada letra deve ser exibida na tela separada por hífen.

8. Crie um procedimento que receba uma **string** de no máximo 50 caracteres e inverta a ordem da **string** digitada;
   Exemplo:
   Entrada: Teste
   Saída: etseT

9. Fazer um programa para determinar e imprimir uma string que será a concatenação de duas outras strings lidas.

10. Fazer um procedimento para imprimir uma string recebida como parâmetro sem os espaços em branco.

11. Fazer um procedimento para receber uma string do usuário (máx. 50 caracteres) e imprimir uma estatística dos caracteres digitados. Isto é, imprimir o número de vogais, consoantes e outros caracteres.

12. Fazer um programa para ler uma string e transfira as consoantes para um vetor e as vogais para outro. Depois mostre cada um dos vetores.

13. Faça uma função que receba uma string do usuário (máx. 20 caracteres) e um caractere qualquer. A função deverá remover todas as ocorrências do caractere da string e retornar o número de remoções.

14. Escreva uma função que receba uma cadeia de caracteres de tamanho máximo 100, e retornar 1 se esta cadeia é uma *palíndrome* e zero caso contrário. Uma palavra é dita ser *palíndrome* se a seqüência de seus caracteres da esquerda para a direita é igual a seqüência de seus caracteres da direita para a esquerda. Ex.: **arara, asa**.

15. Um dos sistemas de encriptação mais antigos é atribuído a Júlio César: se uma letra a ser encriptada é a letra de número N do alfabeto, substitua-a com a letra (N+K), onde K é um número inteiro constante (César utilizava K = 3). Dessa forma, para K = 1 a mensagem “Ataque ao amanhecer” se torna “bubrfabpabnboifdfs”. Faça um programa que receba como entrada uma mensagem e um valor de K e retorne a mensagem criptografada pelo código de César.