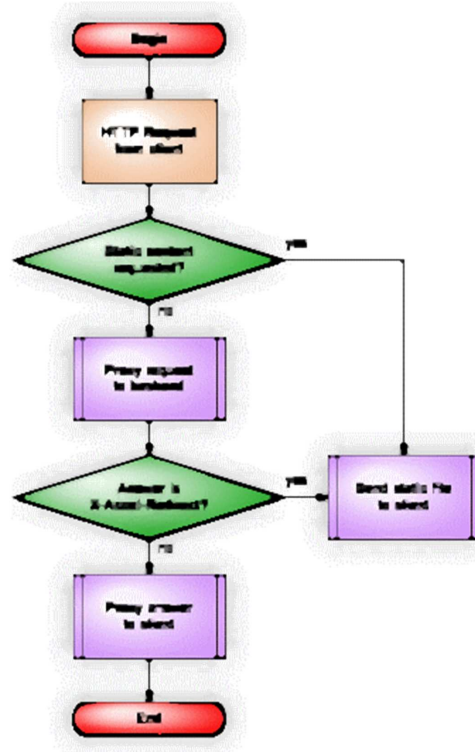


Prof. Dalton Vinicius Kozak



Introdução à Construção de Algoritmos

Versão 2018

CONTEÚDO

1. INTRODUÇÃO	1
1.1. O QUE É ALGORITMO?	1
1.2. POR QUE CONSTRUIR ALGORITMOS?	1
1.3. EXEMPLOS DE ALGORITMOS	2
1.4. RACIOCÍNIO LÓGICO - A ESSÊNCIA	5
1.5. PROCESSO DE SOLUÇÃO DE PROBLEMAS	6
1.5.1. O Esquema de Polya	6
1.5.2. Exemplo do Processo de Solução Conforme Polya	8
1.5.3. Juntando os Esquemas	10
1.6. TRANSFORMANDO UM PLANO DE SOLUÇÃO EM UM ALGORITMO	11
1.6.1. Sequência Simples	12
1.6.2. Alternativas	12
1.6.3. Repetição	15
1.7. EXERCÍCIOS FINAIS	18
1.8. O QUE NOS ESPERA?	23
2. TÓPICOS PRELIMINARES	24
2.1. COMPUTADORIZANDO AS SOLUÇÕES DE PROBLEMAS	24
2.2. QUE PROBLEMAS RESOLVER COM O AUXÍLIO DO COMPUTADOR	24
2.3. ORGANIZAÇÃO E ESTRUTURA BÁSICA DO COMPUTADOR	24
2.3.1. Entrada de Dados	25
2.3.2. Processamento de Dados	25
2.3.3. Saída de Dados	26
2.4. O ALGORITMO IMPLEMENTADO NO COMPUTADOR	26
2.4.1. Programas de Computador	26
2.4.2. Linguagens de Programação	26
2.5. TÉCNICAS A SEREM UTILIZADAS	28
2.5.1. Representação Textual - O Português Estruturado	28
Definição	29
Sintaxe e Semântica	29
Estrutura básica de um Algoritmo em Português Estruturado	30
2.5.2. Representação Gráfica	30
2.6. TIPOS E ESTRUTURAS DE DADOS	31
2.6.1. Conceito	31
2.6.2. Tipos Básicos de Dados	33
2.7. VARIÁVEIS	33
2.7.1. Definição	33
2.7.2. Declaração	34
2.8. OPERADORES E EXPRESSÕES	34
2.8.1. Operadores e Expressões Aritméticas	35
2.8.2. Operadores e Expressões Lógicas	37
2.8.3. Operadores Relacionais	38
2.8.4. Prioridade entre Operadores	39
2.8.5. Expressões Literais	41
2.9. COMANDOS BÁSICOS	41
2.9.1. Atribuição	42
2.9.2. Entrada e Saída de Dados	43

2.10. DOCUMENTANDO SEU ALGORITMO - COMENTÁRIOS.....	45
2.11. DO PORTUGUÊS ESTRUTURADO PARA AS LINGUAGENS DE PROGRAMAÇÃO.....	47
3. ESTRUTURAS DE CONTROLE	51
3.1. RECORDANDO	51
3.2. ESTRUTURA SEQUENCIAL E BLOCOS DE COMANDOS	51
3.3. ESTRUTURA DE SELEÇÃO	52
3.3.1. Alternativas Simples e Composta.....	52
3.3.2. Seleção Múltipla	56
3.4. ESTRUTURA DE REPETIÇÃO	60
3.4.1. Repetição Controlada por Condição no Início	60
3.4.2. Repetição Controlada por Condição no Final	63
3.4.3. Repetição Com Variável de Controle.....	65
3.5. ANINHAMENTO E ENDENTAÇÃO	67
3.6. COMANDOS DE DESVIO DE FLUXO.....	69
3.6.1. Comando Abandone	69
3.6.2. Desvio Incondicional	71
4. MODULARIZAÇÃO DE ALGORITMOS	76
4.1. DIVIDIR PARA CONQUISTAR.....	76
4.2. FUNÇÕES.....	77
4.3. PROCEDIMENTOS.....	78
4.4. PASSAGEM DE PARÂMETROS.....	79
4.5. EXEMPLOS DE MÓDULOS NAS LINGUAGENS DE PROGRAMAÇÃO	82
4.6. ESTRUTURA DE PROGRAMAS – MODULARIZAÇÃO.....	86
4.7. RECURSIVIDADE DE FUNÇÕES.....	86
5. INTRODUÇÃO ÀS ESTRUTURAS DE DADOS	91
5.1. ESTRUTURAS DE DADOS HOMOGÊNEAS	91
5.1.1. Definição	91
5.1.2. Vetores	91
Conceito	91
Sintaxe	92
Notação Simplificada	95
5.1.3. Matrizes	99
Conceito	99
Sintaxe	100
Notação Simplificada	101
5.2. ESTRUTURAS DE DADOS HETEROGÊNEAS	105
5.2.1. Definição	105
5.2.2. Registros.....	105
Conceito	105
Declaração de Registros.....	105
Trabalhando com os Campos do Registro	107
Um exemplo na Linguagem C	107
Matrizes de Registros.....	108
6. PONTEIROS E ALOCAÇÃO DINÂMICA DE MEMÓRIA	111
6.1. PONTEIROS	111
6.1.1. Conceito.....	111
Definição.....	111
Como Funcionam	111

6.1.2. Declaração.....	112
6.1.3. Ponteiros e Endereços	112
Operadores	112
Operador Unário & ("endereço de")	112
Operador Unário * ("referência a")	113
Inicialização de Ponteiros	113
Ponteiro para Ponteiro	113
Advertência	114
6.1.4. Operações	116
Igualdade.....	117
Incremento e Decremento.....	117
Adição e Subtração	117
Comparação	118
Operações Não Aplicáveis.....	118
6.1.5. Ponteiros e Argumentos de Funções.....	118
6.1.6. Ponteiros e Arranjos.....	120
Relação.....	120
Armazenamento na Memória	120
Vetores e Ponteiros	120
Ponteiros como vetores.....	123
Matrizes e Ponteiros	123
Vetores e Matrizes de ponteiros	125
Alguns Cuidados com a Notação.....	126
6.1.7. Ponteiros para Caracteres.....	126
6.1.8. Ponteiros para Funções	127
6.2. ALOCAÇÃO DINÂMICA DE MEMÓRIA	128
6.2.1. Conceito	128
6.2.2. Funções do C para Alocação Dinâmica	129
Função malloc()	129
Função calloc()	129
Função realloc()	130
Função free().....	131
6.2.3. Programa Exemplo	132
APÊNDICE A - EXEMPLOS DO PROCESSO DE SOLUÇÃO DE PROBLEMAS CONFORME POLYA.....	137
Exemplo de Solução de Problema 1	137
Exemplo de Solução de Problema 2.....	140

1. Introdução

1.1. O Que é Algoritmo?

A palavra *algoritmo* é um daqueles termos que, embora muito usado, poucos sabem dizer exatamente o que significa. É um termo de origem árabe¹, que traduzido para o português quer dizer "operação ou processo de cálculo", que por sua vez evoca a noção de um "processo para resolver um dado problema". Outras definições possíveis são:

"Algoritmo é uma sequência de passos que visam atingir um objetivo bem definido."^[1]
"Um algoritmo constitui-se numa série de procedimentos utilizados para resolução de um problema."^[2]

É importante notar que, como muitos pensam, não há nenhuma associação explícita entre algoritmo e computador. Na verdade, a construção de um algoritmo faz parte do processo de solução de um problema qualquer, que *eventualmente* pode ser resolvido através do uso do computador, como sugerido pela [Figura 1](#), onde é mostrado um possível esquema para resolver problemas (mais adiante esse esquema será refinado). Nessa figura, implementação significa a maneira como determinado algoritmo será realizado ou executado.

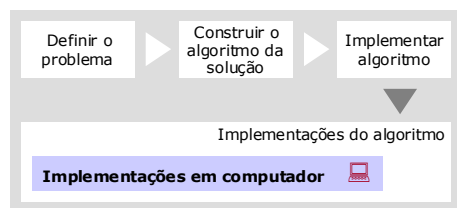


Figura 1: Um esquema para a solução de problemas: a implementação em computador pode ser uma das formas de se chegar à solução.

1.2. Por Que Construir Algoritmos?

Como hoje o computador está ao alcance de praticamente todos os profissionais, muitos tipos de problemas passaram a ser resolvidos utilizando esse recurso, aumentando a necessidade de uma melhor interação do profissional com o computador, não importando a sua formação: engenheiro (civil, de alimentos, mecânico, químico, eletrônico, naval, aeronáutico, etc.), economista, administrador, dentista, matemático, físico ou médico. Não necessariamente esses profissionais irão desenvolver programas de computador, mas com maior probabilidade poderão, em certo momento, estar em contato direto com profissionais de informática (que não entendem de engenharia, medicina, negócios, etc.), tendo de especificar para estes o quê deve ser resolvido pelo computador (o algoritmo da solução, que o especialista em informática muito provavelmente não conhece!). Para que essa comunicação seja a melhor possível, algum conhecimento sobre técnicas de construção de algoritmos (e, às vezes, de programação) é valioso.

A [Figura 2](#) mostra um exemplo da solução de um problema conforme o esquema proposto anteriormente ([Figura 1](#)). Note que se trata de um problema simples, com um algoritmo de solução também simples, o qual pode ser implementado de várias maneiras, inclusive com a utilização do computador.

¹ Sua origem remonta ao século IX, através de um estudioso e matemático persa chamado Mohammed Ibn Musa Abdu Djefar, conhecido por Al-Khwarismi, o qual escreveu importante livro sobre álgebra. Ao longo do tempo, e após repetido uso, o nome Al-Khwarismi foi sofrendo alterações, ocasionado mudança na pronúncia: Al-Karismi, Algarismi, chegando a Algoritmo, que é a representação numérica do sistema de cálculo utilizado em nossos dias. É desta mesma origem que vem o termo Algoritmo, muito utilizado em computação.

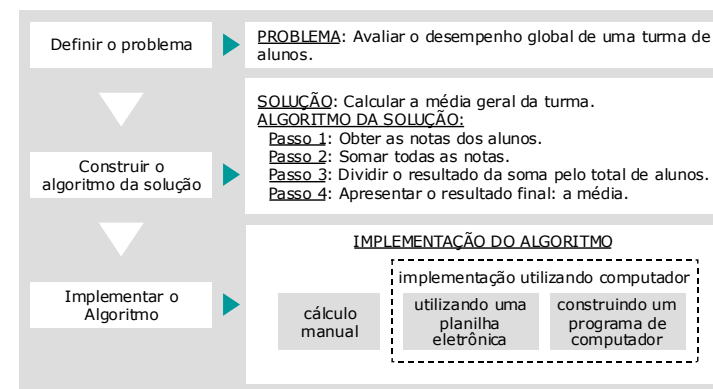


Figura 2: Exemplo da solução de um problema.

1.3. Exemplos de Algoritmos

O primeiro exemplo trata de um problema típico encontrado por professores, qual a solução identificada para resolvê-lo e o algoritmo utilizado para implementá-la. Esse exemplo segue as três etapas mostradas na [Figura 2](#), mas o algoritmo para avaliação é outro. O segundo exemplo mostra como uma representação gráfica - o fluxograma - pode ser utilizada para descrever um algoritmo.

Exemplo 1: algoritmo para avaliação bimestral.

Problema. Certo professor precisa definir um critério para avaliação bimestral de seus alunos.

Solução. A avaliação será feita considerando uma nota de prova, e uma outra nota obtida a partir da elaboração de trabalhos, práticas de laboratório e participação na aula teórica através da solução de exercícios e respostas a perguntas. Para isso, o professor resumiu na seguinte fórmula o cálculo da nota bimestral:

$$\text{Nota Bimestre} = 0,8 \times (\text{Nota Prova}) + \left(\frac{\text{NC}}{\text{NCO}} \right) \times 2,0$$

onde:

- NC (Número de Créditos)** representa a pontuação obtida pelo aluno a partir da elaboração de trabalhos, práticas de laboratório e participação na aula teórica através da solução de exercícios e respostas a perguntas. Exemplo: uma prática de laboratório pode valer 1 ou 2 créditos, um trabalho pode valer 3 ou 4 créditos, e assim por diante.
- NCO (Número de Créditos Obrigatórios)** representa o número de créditos obrigatórios no bimestre considerado (práticas de laboratório, listas de exercícios e trabalhos).

A [Tabela 1](#) abaixo exemplifica o resultado da aplicação de tal critério, tabela essa que é o resultado da implementação do algoritmo da solução, descrito a seguir.

Aluno	Nota Prova	NC	NCO	0,8 x (Nota Prova)	(NC/NCO) x 2,0	NOTA BIMESTRAL
1	7,0	15	19	5,6	1,58	7,2
2	3,0	8	19	2,4	0,84	3,2
3	10,0	19	19	8,0	2,00	10,0

Tabela 1: Exemplo do cálculo de notas bimestrais

Algoritmo para a Solução. Para chegar à solução desejada, mostrada acima, seguem-se os seguintes passos ou algoritmo:

"obter as notas das provas dos alunos";
 "somar os créditos obtidos por cada aluno (no laboratório, nos trabalhos, etc.)";
 "calcular o número de créditos obrigatórios do bimestre";
 "aplicar a fórmula para cálculo da nota do bimestre de cada aluno";
 "apresentar o resultado aos alunos".

(A 1)

Implementar o Algoritmo. Pode-se implementar esse algoritmo de algumas maneiras:

- manualmente, construindo-se com lápis e papel uma tabela (como a Tabela 1) e realizando-se os cálculos com uma calculadora;
- no computador, utilizando-se uma planilha eletrônica ou através de um programa escrito em uma linguagem de programação qualquer.

Exemplo 2: representação gráfica de um algoritmo – o fluxograma.

O algoritmo da [Figura 3](#) expressa de forma bem inteligível, através de uma representação gráfica, como um aluno é avaliado e acompanhado ao longo de uma disciplina de um determinado curso. Ele é resultado da solução do seguinte problema: como acompanhar e realizar a avaliação em uma disciplina com regime anual.

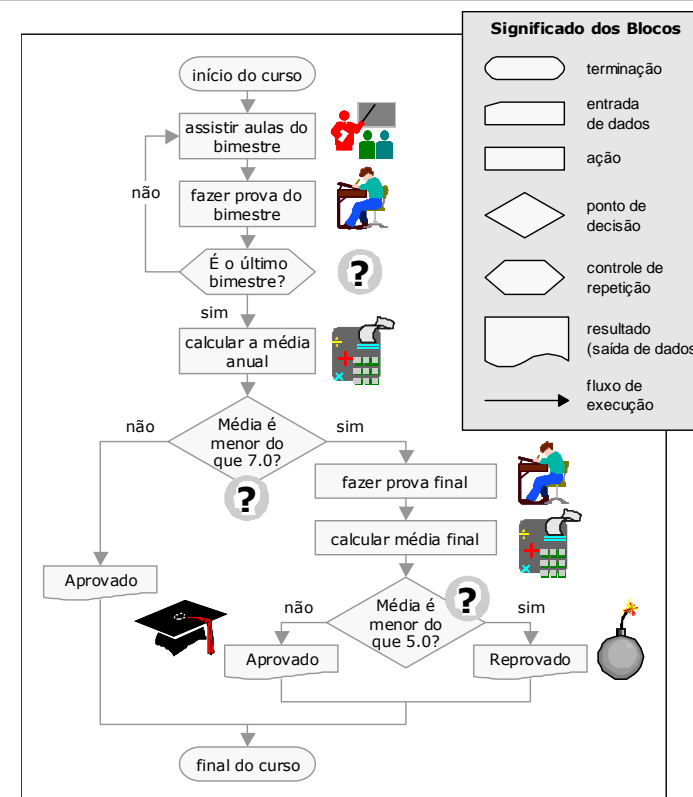


Figura 3: Algoritmo que descreve sucintamente o acompanhamento e a avaliação de um aluno em uma determinada disciplina de um curso em regime anual, com 4 bimestres.

O diagrama da figura acima é apenas um exemplo de uma das representações gráficas utilizadas para expressar algoritmos: o fluxograma. Os ícones adicionados nessa figura apenas servem para ilustrar um pouco mais o que acontece, mas não fazem parte do fluxograma propriamente dito.



Analizando o Algoritmo

Suponha que você seja aluno de uma escola com cursos em regime anual, com 4 bimestres. Suponha também que você frequentou três disciplinas, sendo que na primeira você passou por média (média > 7.0), na segunda você ficou para prova final e obteve média maior ou igual a 5.0, e na terceira você ficou para prova final e tirou uma nota menor do que 5.0.

Verifique se o algoritmo mostrado na [Figura 3](#) expressa realmente o seu resultado em cada uma dessas disciplinas. Analise cada uma das construções gráficas envolvidas. Responda: quantas vezes o fluxo de execução do algoritmo passa pelos blocos "assistir aulas do bimestre" e "fazer prova do bimestre"?

Resposta: _____.

1.4. Raciocínio Lógico - A Essência

Quando falamos em lógica, estamos nos referindo a raciocínio encadeado, ou idéias ligadas de maneira consistente e coerente. Ou seja, correção no pensamento. Também podemos nos referir à lógica como a arte de pensar corretamente, e considerando o raciocínio como a forma mais complexa do pensamento, podemos então dizer que a lógica tem como foco a "correção do raciocínio", ou o raciocínio lógico. O resultado disso é um pensamento ordenado e organizado. Exemplos²:

- João é irmão de José.
- João está no armário.
- José é irmão de Roberto.
- O armário está fechado.
- Portanto, João e Roberto são irmãos³.
- Portanto, para pegar o livro devo abrir o armário.

O uso do raciocínio lógico durante a elaboração de um algoritmo é imprescindível, especialmente se esse algoritmo for implementado em computador, que não tem a capacidade de pensar: ele realiza exatamente aquilo que lhe é instruído. As técnicas utilizadas para a construção de algoritmos visam facilitar a construção de estruturas lógicas, porém não substituem o pensamento humano.

Portanto, é necessário que o aluno esteja atento, pois ao contrário de outras disciplinas (como física ou cálculo), onde se está resolvendo problemas baseados em métodos (ou algoritmos) já conhecidos, nessa disciplina os algoritmos serão construídos, às vezes sem uma referência anterior de como poderia ser a solução para o problema que se está querendo resolver, ou de como escolher a melhor entre as possíveis soluções encontradas. Exatamente nesse ponto reside um dos maiores empecilhos no estudo de algoritmos: ter o conhecimento ou a habilidade em como resolver problemas.

A prática - a solução de diversos problemas - realizada de forma sistemática é a melhor maneira para se conseguir alguma proficiência nesse assunto, sendo essencial para o desenvolvimento do raciocínio lógico do aluno. Sem isso, a construção e a compreensão de algoritmos ficam seriamente prejudicados para alunos que têm dificuldades nesse fundamento. Assim sendo, antes de prosseguir, será estudado um pouco mais sobre o processo de solução de problemas.



Você estuda da maneira certa?

O estudo de algoritmos passa longe da "decoreba" e muito perto do raciocínio lógico, para não dizer em cima. Assim sendo, reflita sobre a seguinte frase:

"Não procure saber **como** resolver este ou aquele exercício ou problema sem saber os **porquês** da sua solução! Identificando sempre os 'porquês' de cada solução você estará mais apto, no futuro, a descobrir os 'comos' em novos problemas".

E acredite: você começara a deixar de ter problemas nesse estudo quando a frase acima fizer sentido e ficar bem clara para você.

Exercício 1: usando a lógica matemática - 1.

Um determinado conferencista, após uma palestra, perguntou a seu organizador: "Quantas pessoas estavam presentes no auditório?". O organizador respondeu: "Se estivessem 200 pessoas a mais, a metade seria exatamente 800". Quantas pessoas assistiram realmente à conferência?

Solução

$$(x + 200)/2 = 800 \Rightarrow x + 200 = 1600 \Rightarrow x = 1400 \text{ pessoas}$$

Exercício 2: usando a lógica matemática - 2.

Um burro e um jumento transportavam sacos de farinha dirigindo-se à feira. O burro gemia sob a massa de farinha que levava. "De que te queixas?", disse o jumento, ao que o burro respondeu:

² Um contra-exemplo: "Um trem passou por cima de um buraco; três pneus furaram; portanto, caíram três melancias". Você está enxergando alguma lógica nisso?

³ Para essa conclusão ser válida, deve-se ter como hipótese que nas duas primeiras assertivas estamos falando de irmãos por parte de pai e mãe.

"Se passasses para o meu lombo quatro dos sacos que estais transportando, eu ficaria com o dobro dos teus; se eu passasse para o teu lombo dois dos que estou transportando, só assim nossas cargas ficariam iguais". Após ler esse eloquente diálogo, responda: qual a carga que levava cada um dos animais?

Solução

B: sacos carregados pelo burro

J: sacos carregados pelo jumento

$$(B + 4) = 2(J - 4) \rightarrow B - 2J = -12 \quad (A)$$

$$B - 2 = J + 2 \rightarrow B - J = 4 \quad (B)$$

Subtraindo (A) de (B), obtém-se $J = 16$

Da equação (B), obtém-se $B = 20$

Exercício 3: usando a lógica matemática - 3.

Com o auxílio das operações de adição, subtração, multiplicação, divisão, potenciação, raiz quadrada e/ou fatorial, tornar verdadeira a seguinte igualdade:

$$0 \ 0 \ 0 \ 0 \ 0 = 120$$

Observe que do lado esquerdo há cinco operandos - os zeros. Entre eles existem operadores aritméticos, ou funções aplicadas sobre cada operando, ou conjunto de operandos (entre parênteses). Deve-se descobrir como tornar verdadeira essa igualdade com esses operandos.

Dica: $3 \ 3 \ 1 = 8 \rightarrow 3 \times 3 - 1 = 8$ (só foram incluídos operadores!)

Solução

$$(0! + 0! + 0! + 0!)! = (1 + 1 + 1 + 1 + 1)! = 5! = 120$$

$$(0! + 0! + 0!)! + 0! = (1 + 1 + 1)! + 1 = 6 + 1 = 7$$



A Diferença entre Problema e Exercício

Alguns autores enfatizam a seguinte distinção:

Exercício: resolver a equação $x^2 - 3x + 1 = 0$ (supondo que se conheça a fórmula de Bhaskara).

Problema: provar a fórmula de Bhaskara (supondo que se conheça apenas a fórmula, mas nunca se tenha visto essa demonstração).

Resumidamente, o exercício envolve a atividade de adestramento de alguma habilidade ou conhecimento já conhecido pelo resolutor, enquanto que a resolução de um problema envolve necessariamente criação e/ou invenção significativa.

Nos "exercícios" anteriores, a solução das equações obtidas é um exercício, enquanto que chegar a essas equações é resolver um problema: houve um processo de "criação" das equações a partir do raciocínio lógico.

Neste material, a legenda Exercício terá um significado mais amplo, englobando as duas definições acima, apenas por facilidade de notação. Porém, a distinção entre exercício e problema será feita toda vez que a situação o exigir, ou for conveniente.

1.5. Processo de Solução de Problemas

1.5.1. O Esquema de Polya

A discussão de como ensinar e desenvolver as técnicas de solução de problemas vem sendo abordada na literatura já há algum tempo. O livro clássico de George Polya^[3], com a 1ª edição datando de 1945, é uma das principais referências sobre o assunto, tendo sido fonte de inspiração de diversos outros trabalhos posteriores na mesma linha. Apesar de existirem algumas críticas e restrições no uso do esquema proposto por Polya, ainda é o procedimento mais utilizado nesse contexto. Este esquema é apresentado na Figura 4.

Figura 4: Como resolver um problema, segundo Polya.

Etapa	Descrição	Procedimentos
1	ENTENDIMENTO DO PROBLEMA É preciso <i>entender</i> o problema	<i>Qual é a incógnita? Quais são os dados? Qual é a condicionante?</i> É possível satisfazer a condicionante? A condicionante é suficiente para determinar a incógnita? Ou é insuficiente? Ou redundante? Ou contraditória? Trace uma (ou mais) figura(s). Adote uma notação inadequada. Separe as diversas partes da condicionante.
2	ESTABELECIMENTO DE UM PLANO Encontre a conexão entre os dados e a incógnita. É possível que seja obrigado a considerar problemas auxiliares ou particulares se não puder encontrar uma conexão imediata. É preciso chegar a um plano para a resolução.	Já o viu antes? Ou já viu o mesmo problema sob uma forma ligeiramente diferente? <i>Conhece um problema correlato/semelhante?</i> (Conhece um problema que lhe poderia ser útil? Você conhece teoremas ou fórmulas que possam ajudar?) <i>Considere a incógnita!</i> Procure pensar num problema conhecido que tenha a mesma incógnita ou outra semelhante. <i>Eis um problema correlato e já antes resolvido. É possível aproveitá-lo?</i> É possível utilizar o seu resultado? É possível utilizar o seu método? Deve-se introduzir algum elemento auxiliar para tornar possível a sua utilização? É possível reformular o problema? É possível enunciar-lo de outra maneira? Volte às definições. Se não puder resolver o problema proposto, procure antes algum problema correlato. É possível imaginar um problema correlato mais acessível? Um problema mais genérico? Um problema mais específico? Um problema análogo? É possível resolver uma parte do problema? Mantenha apenas uma parte da condicionante, deixe a outra de lado; até que ponto fica determinada a incógnita? Como ela pode variar? É possível obter dos dados alguma coisa de útil? É possível pensar em outros dados apropriados para determinar a incógnita? É possível variar a incógnita, ou os dados, ou todos eles, se necessário, de tal maneira que fiquem mais próximos entre si? Utilizou todos os dados? Utilizou toda a condicionante? Levou em conta todas as noções essenciais implicadas no problema?
3	EXECUÇÃO DO PLANO Execute seu plano.	Ao executar o seu plano de resolução, <i>verifique cada passo</i> . É possível verificar claramente que o passo está correto? É possível demonstrar que ele está correto?
4	RETROSPECTO (OU REVISÃO) Examine/revise a solução obtida	É possível <i>verificar o resultado</i> ? É possível verificar o argumento? É possível chegar ao resultado por um caminho diferente? É possível perceber isso num relance? É possível utilizar o resultado, ou o método, em algum outro problema?

O "roteiro" mostrado na Figura 4 não deve ser interpretado como sendo estritamente linear. Na verdade, o próprio Polya nunca pretendeu que: i) essas etapas fossem percorridas uma depois da outra sem que fosse conveniente, ou necessário, voltar atrás; ii) que essa divisão funcionasse como uma receita. Dessa forma, o retorno a um passo anterior sempre se fará necessário quando alguma coisa ainda não estiver bem esclarecida ou entendida, como sugerido na [Figura 5](#).

O processo de solução de problemas não é exatamente linear...

...mas envolve volta a passos anteriores para elucidar pontos não esclarecidos ou duvidosos.

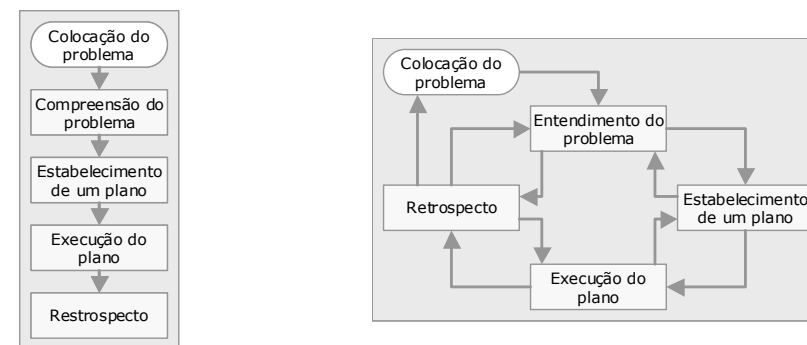


Figura 5: Interpretação correta das etapas para solução de problemas proposta por Polya - o processo é cíclico (retro-alimentação ou "feedback") e dinâmico.

A última etapa do processo de Polya pretende, além da verificação do resultado propriamente dito, duas coisas importantes: uma depuração e uma abstração da resolução^[4]. Na escola normalmente existem ao menos caricaturas das três primeiras etapas de Polya, mas praticamente nada no que se refere à etapa do retrospecto ou revisão, por diversos motivos alegados.



Afinal, por quê perder tempo compreendendo como é o processo de resolver problemas?

Porque, caro aluno, você não sabe (salvo algumas poucas exceções). Infelizmente, poucas escolas ensinam a resolver problemas: a maioria torna seus alunos meros solucionadores de exercícios (especialmente os cursinhos), que não requer muito pensamento e raciocínio. Na faculdade, você vai ter que resolver problemas, e é aí que o "bicho pega".

Um grande passo em direção ao sucesso na disciplina de algoritmos, e nas outras também, é saber como proceder na solução de problemas, criando para isso um modelo próprio para procura e abordagem da solução. Esse modelo certamente contemplará, de alguma forma, as quatro etapas previstas no processo de Polya.

A depuração da resolução tem como objetivo verificar a argumentação utilizada, procurando simplificá-la. Pode-se chegar ao extremo de buscar outras maneiras de resolver o problema, possivelmente mais simples, mas menos intuitivas, só agora acessíveis ao resolvidor, talvez como resultado do "amadurecimento" do problema em sua mente.

A abstração da resolução tem como objetivo a reflexão sobre o processo de resolução, procurando descobrir a essência do problema e do método de resolução empregado. Tendo-se sucesso nessa empreitada, poder-se-á resolver outros problemas mais gerais ou de aparência bastante diferente.

Estas duas atividades representam a possibilidade de aumento do "poder de fogo" do resolvidor, tornando-o apto a resolver problemas cada vez mais complexos. Por isso a importância de, tanto quanto for possível, procurar realizar essas atividades adicionais dentro da última etapa.

1.5.2. Exemplo do Processo de Solução Conforme Polya

Será visto um exemplo mostrando rapidamente como o esquema de Polya pode ser utilizado no processo de solução de um problema. A bem da verdade, esse esquema "é uma trilha, e não um trilho". Isto é, pode haver variações conforme o entendimento ou habilidade de cada um, porém

a mensagem básica é a seguinte: sempre deve existir o questionamento se aquilo que está sendo feito para chegar à solução é correto, e como pode ser verificado cada resultado intermediário obtido nas etapas da resolução. No *Apêndice A - Exemplos do Processo de Solução de Problemas Conforme Polya* o exemplo deste item (e mais outro) é apresentado com muito mais detalhes. Vale a pena dar uma olhada.

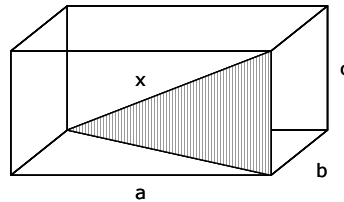
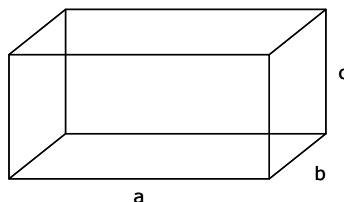
Vejamos, portanto, o exemplo.

⇒ Colocação do Problema

Calcular a diagonal de um paralelepípedo retângulo do qual são conhecidos o comprimento, a largura e a altura.

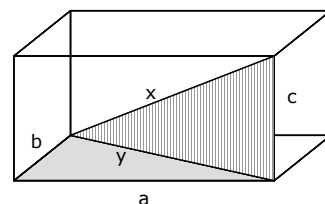
1) Compreensão do problema

Os dados são os lados do paralelepípedo, denominados de a , b e c . Uma vez conhecidos esses valores, o paralelepípedo fica determinado e, conseqüentemente, a diagonal também fica determinada.



2) Estabelecimento de um Plano

Observando a figura ao lado, é possível notar dois elementos geométricos conhecidos. O primeiro é um triângulo retângulo na base do paralelepípedo, onde os catetos a e b são conhecidos, permitindo calcular a diagonal y . O segundo elemento é outro triângulo retângulo cujos catetos y e c são conhecidos (y é calculado do primeiro triângulo), o que permite calcular a hipotenusa x . Ou seja, o plano consistente em resolver, na sequência, esses dois triângulos através do teorema de Pitágoras.



Pois bem. Agora se tem um plano para a solução do problema!

Conceber um plano, a idéia da resolução, não é fácil. Para conseguir isto é preciso, além de conhecimentos anteriores, de bons hábitos mentais e de concentração no objetivo, contando-se, às vezes, com uma "pitada de sorte"⁴. O objetivo desse item é tentar iniciar algo nesse sentido.



Antes de continuar, cabe a seguinte pergunta:

O que tem a ver a construção de algoritmos com o processo de resolução de problemas?

A resposta é: **TUDO!**

Observe que o plano de solução nada mais é do que um algoritmo. Cabem, portanto, as mesmas dificuldades de concepção descritas acima para o plano de solução. Por isso a importância do estudo do processo de solução de problemas.

Executar o plano (próximo passo) é muito mais fácil: paciência é o "ingrediente" mais importante.

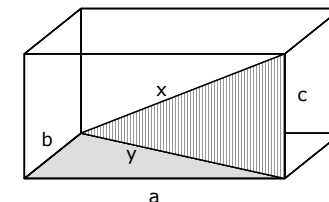
⁴ Se bem que alguns dizem que a sorte sorri para quem trabalha... Faz muito sentido em nosso contexto.

3) Execução do Plano

Incógnitas: x

Dados: a , b , c

Plano: Resolver o triângulo de lados a , b e y e, em seguida, resolver o triângulo de lados x , y e c aplicando duas vezes, sucessivamente, o teorema de Pitágoras.



Fórmulas:

1ª relação: $x^2 = y^2 + c^2$; 2ª relação: $y^2 = a^2 + b^2$

Relação Final: $x^2 = a^2 + b^2 + c^2$ ou $x = \sqrt{a^2 + b^2 + c^2}$

4) Retrospecto ou Revisão

Talvez esta seja a parte mais importante da solução do problema, pois irá consolidar o entendimento do problema e a validade da sua solução. Por incrível que possa parecer, para este problema simples que está sendo resolvido é possível realizar pelo menos cinco perguntas para validar o seu resultado. A primeira delas é:

Pergunta: É possível verificar o resultado?

Resposta: Sim. Através da medição das distâncias a , b , c e x em um paralelepípedo qualquer e sua substituição na fórmula encontrada. A relação $x = (a^2 + b^2 + c^2)^{1/2}$ deve se mostrar verdadeira para qualquer medição desse tipo

Você consegue imaginar quais poderiam ser as outras perguntas para garantir que a resposta encontrada é correta? Veja o primeiro exemplo do *Apêndice A* para saber quais são elas.



Nunca deixe de fazer o retrospecto!

A maior parte dos alunos, uma vez chegada à solução do problema e escrita a demonstração, fecham os livros e passam a outro assunto. Assim fazendo, eles perdem uma fase importante e instrutiva do trabalho da resolução. Se fosse feito um retrospecto (ou revisão) da resolução completa, reconsiderando-se e reexaminando-se o resultado final e o caminho que levou até ele, seria possível consolidar o conhecimento e aperfeiçoar a capacidade de resolver problemas. Nenhum problema fica completamente esgotado.

1.5.3. Juntando os Esquemas

Na Figura 1 foi sugerido um esquema para solucionar problemas dentro do contexto da construção de algoritmos. Em seguida, discutiu-se o esquema de solução de problemas conforme proposto pelo professor George Polya, mostrado na Figura 4.

Se analisarmos esses esquemas, veremos que é possível relacioná-los de alguma forma. Isso foi feito através da *Figura 6*, onde a Figura 1 e a Figura 4, com algumas alterações, foram combinadas - fica claro agora enxergar importância de se saber como resolver problemas no contexto da construção de algoritmos.

O aluno, a essas alturas, deve estar se perguntando qual esquema deve ser utilizado quando for resolver um problema. A resposta pode ser⁵ o seu esquema, se ele estiver funcionando bem para resolver a maior parte dos problemas encontrados até agora, e se ele, de alguma forma, contempla os pontos, ou a maior parte deles, discutidos nos esquemas apresentados até aqui. Caso contrário, se o aluno não tiver uma metodologia ou uma sequência de passos lógicos desenvolvidos em sua mente, deverá criá-la, certamente através da prática, e nesse ponto o

⁵ Essa é a opinião do professor que vos escreve esse material didático.

conteúdo da Figura 6 e toda a discussão desenvolvida até o momento são bastante úteis como ponto de partida. Mas atenção: os esquemas para solução de problemas não devem ser decorados, mas sim entendidos, pois a forma de aplicação pode variar em cada caso, e só o entendimento do processo permitirá a sua adaptação àquele problema em particular.

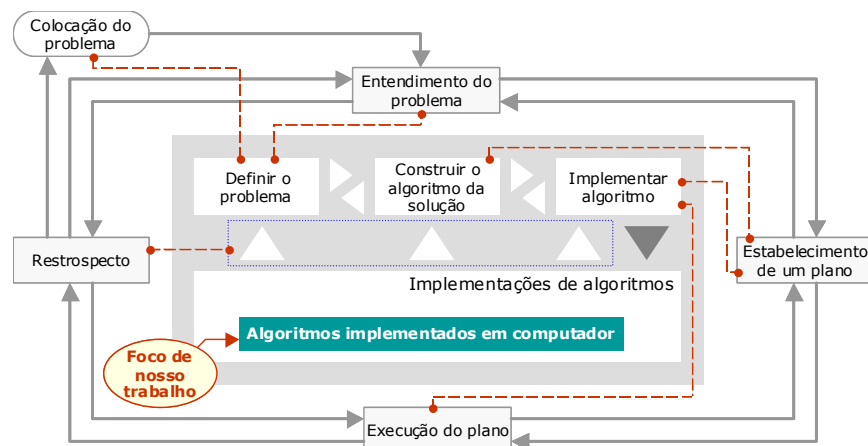


Figura 6: Relacionamento entre os esquemas vistos para a solução de problemas.

O foco do trabalho a ser desenvolvido será os algoritmos implementados em computador, e isso pode envolver tanto a transformação de um algoritmo já pronto em um programa de computador, como a construção do próprio algoritmo antes de implementá-lo, situação onde o conhecimento do processo de solução de problemas é valioso.

1.6. Transformando um Plano de Solução em um Algoritmo

Vamos recordar o algoritmo que foi mostrado na Figura 3. Lá foi utilizada uma representação gráfica que, de forma bastante sucinta e objetiva, descreveu como se procede no acompanhamento e avaliação de um aluno em uma disciplina de um curso numa determinada escola. Trata-se da representação em *Fluxograma*, cujos elementos básicos podem ser vistos na Figura 7. Através desse tipo de diagrama é possível descrever a solução de inúmeros tipos de problemas.

Na solução do primeiro exemplo do item 1.5.2 foram definidos como dados do problema o comprimento a , a largura b e a altura c do paralelepípedo retângulo; a incógnita do problema (resultado procurado) era o comprimento da diagonal do paralelepípedo, representada por x . No plano de solução obteve-se uma fórmula descrevendo a relação entre x e os dados a , b e c . A representação em fluxograma desse algoritmo pode ser vista na Figura 8. Note o uso dos terminadores com as palavras início e fim: simplesmente indicam onde começa e onde termina a execução do algoritmo.

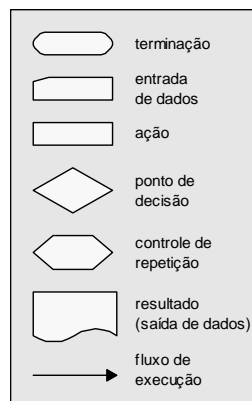


Figura 7: Elementos básicos de um fluxograma.

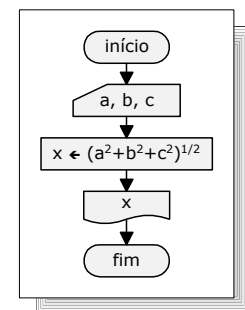


Figura 8: Fluxograma da solução do problema do item 1.5.2

Deve-se notar a objetividade, a simplicidade e a clareza com que foi apresentada a solução do problema do cálculo da diagonal do paralelepípedo retângulo através dessa representação. Essa característica é importante quando se trata da implementação em computador.

A partir dos elementos básicos do fluxograma pode-se construir as estruturas principais utilizadas na construção de algoritmos. Para começar a familiarização com algumas das técnicas de construção a partir dessas estruturas, serão apresentados alguns exemplos de fluxogramas da solução de problemas conhecidos. O entendimento conceitual dessas estruturas é fundamental para se prosseguir adiante, uma vez que a lógica nelas embutida é a mesma que será utilizada na construção dos algoritmos a serem implementados no computador. Portanto, atenção!

Deve ser notado que foi utilizado um símbolo diferente no bloco de ação da Figura 8: o símbolo de atribuição \leftarrow . Como veremos mais adiante, esse símbolo significa que a variável por ele apontado recebe o valor da expressão que se encontra do lado direito.

São três as estruturas básicas a serem vistas: seqüência simples, alternativa e a repetição.

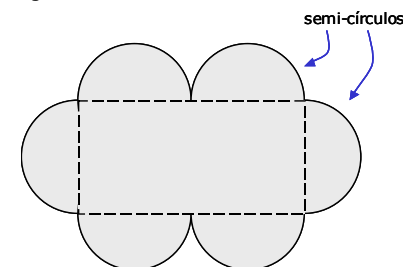
1.6.1. Seqüência Simples

A seqüência simples nada mais é do que uma série de ações que são realizadas em determinada ordem, uma após a outra, como no algoritmo da Figura 8.

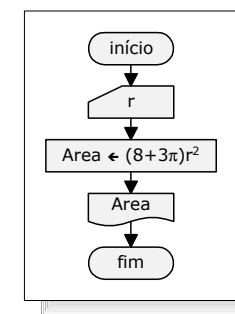
Exercício 4: seqüência simples.

Deseja-se calcular a área da seguinte figura:

Solução



Construa o fluxograma descrevendo o algoritmo da solução para esse problema. Considere os semicírculos iguais entre si. Defina quais os dados de entrada necessários, bem como a fórmula de cálculo.



1.6.2. Alternativas

Nem todas as soluções para problemas são apenas seqüenciais; na verdade, a maioria não é. Existem situações onde, dependendo de uma ou mais condições, os caminhos seguidos por uma solução podem variar; em outras palavras, podem existir *alternativas*.

Vamos analisar um problema de matemática bem conhecido: obter a solução de uma equação do segundo grau. Existem alguns métodos para obter essa solução, sendo bem conhecido o da

fórmula de Bhaskara. Esse método diz que, a partir dos coeficientes da equação é possível determinar a natureza da solução conforme o sinal de um parâmetro calculado denominado discriminante, e a partir deste define-se como calcular as raízes.

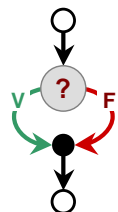
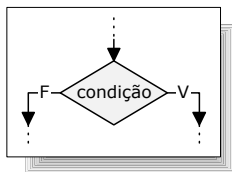
Se a equação for $ax^2 + bx + c = 0$ e o discriminante for Δ , a natureza da solução é dada conforme o seguinte:

- se Δ for positivo, existem duas raízes reais distintas;
- se Δ for nulo, existem duas raízes reais iguais;
- se Δ for negativo, existem duas raízes imaginárias conjugadas.

Vamos supor que o problema a ser resolvido seja o seguinte:

"Dada a equação de segundo grau $ax^2 + bx + c = 0$, onde os valores dos coeficientes são conhecidos e $a \neq 0$, determinar a natureza de sua solução".

A pergunta é: como representar no fluxograma as várias alternativas possíveis? Nos elementos básicos do fluxograma (Figura 7) existe um bloco que representa o ponto de decisão, já utilizado na Figura 3. O mecanismo é o seguinte: se determinada condição for verdade (V), um determinado caminho é seguido; se for falsa (F), outro caminho é escolhido; geralmente ambos os caminhos se encontram em um mesmo ponto mais adiante no fluxograma.



Note que há apenas duas possibilidades: V ou F. Ou seja, é uma bifurcação do fluxo de execução (figura à esquerda): ou segue-se um caminho, ou o outro, mas nunca os dois simultaneamente.

Usando tal estrutura, o fluxograma descrevendo o algoritmo da solução do problema proposto acima fica como mostrado na Figura 9. Note que apenas um caminho é seguido dentre os três possíveis (são três, não são?), cada caminho contendo a resposta conforme o sinal do valor calculado para o discriminante Δ .

Também é possível observar na Figura 9 que a estruturação do algoritmo no fluxograma é compatível com a forma com que os programas de computador tipicamente trabalham: entrada, processamento e saída de dados.

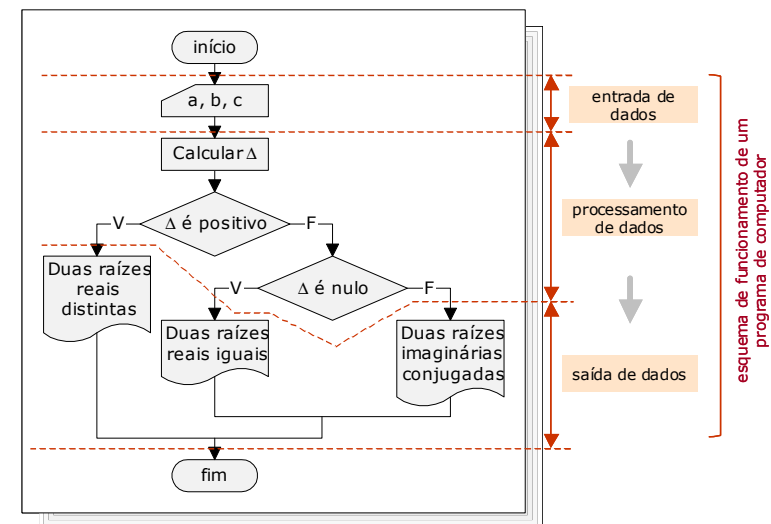


Figura 9: Fluxograma da análise da natureza da solução de uma equação do 2º grau, $ax^2+bx+c=0$, segundo o algoritmo de Bhaskara.

Exercício 5: algoritmo com alternativas.

Em um campeonato de determinado esporte deseja-se classificar os participantes em categorias conforme a idade:

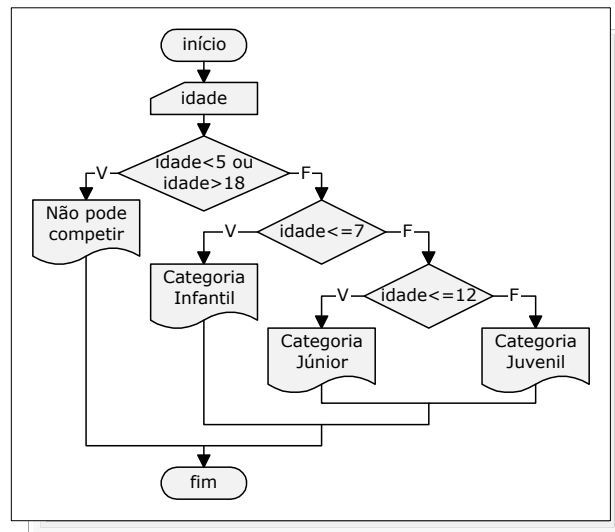
- se a idade é menor ou igual a 7 anos, a categoria é infantil;
- se a idade está entre 8 anos (inclusive) e 12 anos (inclusive), a categoria é júnior;
- para maiores de 13 anos (inclusive) a categoria é juvenil;

Nesse campeonato não é permitida a participação de pessoas com idade inferior a 5 anos ou superior a 18 anos.

Construa o fluxograma descrevendo esse algoritmo numa forma semelhante à feita na Figura 9. Se desejar, utilize os seguintes símbolos para simplificar a escrita:

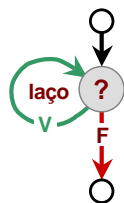
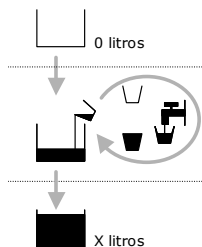
- $<$, $>$: menor, maior
- \leq , \geq : menor ou igual, maior ou igual

Solução



1.6.3. Repetição

Seja o seguinte problema: encher um tanque com capacidade para X litros d'água, sendo que esse tanque não pode sair do lugar e se localiza próximo a uma torneira, existindo à disposição apenas um recipiente menor, transportável: um balde. A solução imediata, e que salta aos olhos, é pegar o balde vazio, levá-lo até a torneira, enchê-lo e depois levá-lo até o tanque e derramar seu conteúdo. Para encher o tanque com os X litros desejados seria necessário *repetir* esse procedimento várias vezes até se constatar que o tanque esteja cheio. Imagine que esse processo precise ser documentado utilizando o fluxograma.



A representação de um processo repetitivo em um fluxograma é conseguida utilizando-se uma estrutura baseada no bloco de controle de repetição, que define em que momento a repetição deve se encerrar. Existem algumas variações possíveis para essa estrutura, conforme mostrado na Figura 10. Por exemplo, a estrutura do tipo 2 já foi utilizada no fluxograma visto na Figura 3. Todas os tipos realizam o que se chama laço de repetição ("loop" em inglês) conforme o controle estabelecido. A figura à esquerda mostra o esquema genérico do fluxo de execução de uma repetição: enquanto o controle permitir ("?"), o laço é realizado.

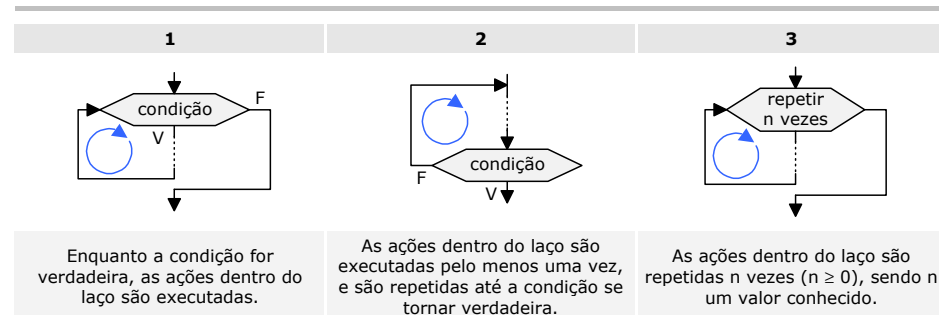


Figura 10: Estruturas de repetição utilizadas em fluxogramas.

Note que os tipos 1 e 2 são utilizados sempre que não se conhece antecipadamente o número de repetições a serem realizadas, que é o caso do exemplo apresentado acima para o tanque a ser enchido com água: quantos baldes serão necessários para enchê-lo? Com essa nova estrutura - repetição - os fluxogramas possíveis para resolver esse problema são dois, como pode ser visto na Figura 11. Nesses dois diagramas a hipótese é de que o tanque se encontra inicialmente vazio.

Existe uma diferença sutil nas duas soluções apresentadas, decorrentes do tipo de estrutura de repetição utilizada. No fluxograma com a repetição do tipo 2, sempre será derramado, no mínimo, um balde com água no tanque. Caso o tanque já estivesse inicialmente cheio, se fosse seguido o processo descrito no primeiro fluxograma da Figura 11, nada seria feito, o que é coerente: o tanque já está cheio. Porém, seguindo o segundo fluxograma, seria derramado um balde no tanque já cheio, o que seria uma operação desnecessária. Portanto, em situações onde há a possibilidade de, mesmo prevista a repetição, ocorrer um caso particular onde nada será feito, a repetição do tipo 2 deve ser deixada de lado.

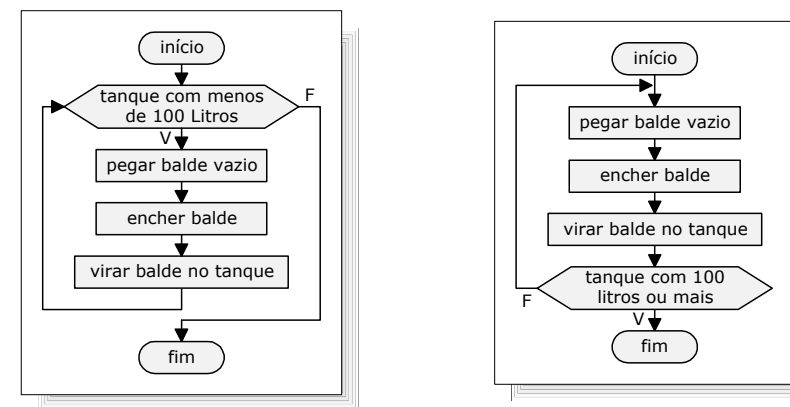


Figura 11: Fluxogramas possíveis para encher o tanque com 100 l de água.

Vamos supor agora que se conheça a capacidade do balde: 10 litros. Tendo esse valor, e sabendo também quanto deverá conter o tanque ($X=100$ litros), é fácil calcular quantas vezes será necessário encher o balde e virá-lo no tanque: 10 vezes. Nessa situação, a estrutura de repetição do tipo 3 pode também ser utilizada, já que o valor de "n" é conhecido: é 10.

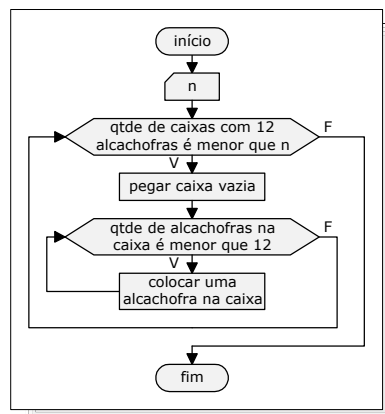
Na [Figura 12](#) temos a terceira variação possível para o fluxograma. Essa variação é a mais adequada sempre que se conhecer com antecedência o número de repetições (também chamadas de iterações) a serem realizadas.

Exercício 6: algoritmo com repetição.

Um certo quitandeiro vende alcachofras apenas em caixas de 12 unidades. Toda vez que um cliente solicita um número qualquer de caixas dessa hortaliça, ele toma um grande cuidado na preparação de cada caixa, visto que o seu produto se encontra acondicionado em excelentes condições em sua quitanda. Uma vez feito o pedido pelo cliente, esse quitandeiro pega uma caixa vazia por vez e, para cada caixa, coloca, uma a uma, cada alcachofra em seu interior até completar uma dúzia. Repete esse procedimento até ter o número de caixas pedidas pelo cliente. Monte um fluxograma descrevendo o processo de preparação das caixas para um pedido qualquer de um cliente. Faça três versões, uma para cada tipo de repetição vista.

Solução

Repetição do tipo 1



Repetição do tipo 2

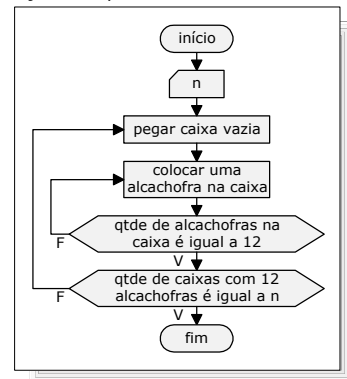
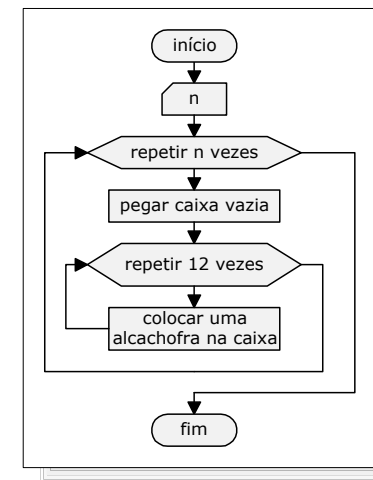


Figura 12: Fluxograma para encher o tanque utilizando a repetição do tipo 3.

Repetição do tipo 3



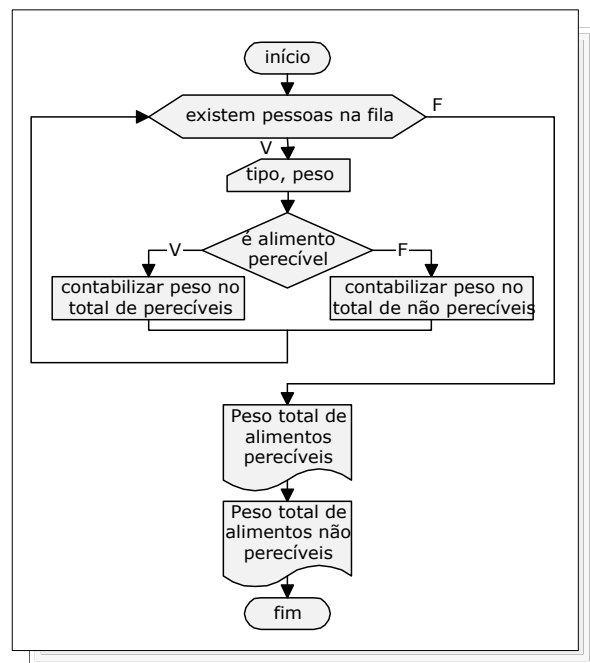
1.7. Exercícios Finais

Exercício 7: fluxograma com todas as estruturas básicas

Uma campanha para os mais necessitados está coletando doativos de gêneros alimentícios. A campanha recolhe os alimentos todos os dias enquanto existem pessoas na fila para doação. Esses doativos são separados em alimentos perecíveis (frutas, carnes, etc.) e não perecíveis (enlatados e conservas em geral), caracterizando o tipo do alimento, sendo o peso de cada doativo contabilizado no total acumulado do dia para cada categoria, para se poder avaliar o resultado diário da campanha. Hipótese: cada pessoa leva apenas elemento perecível ou não-perecível, nunca os dois.

Monte um fluxograma descrevendo sucintamente o procedimento de contabilização do peso dos donativos dessa campanha, conforme explicado acima. Considere como dados de entrada o tipo do alimento (perecível/não perecível) e o peso.

Solução



Pergunta: Seria possível fazer esse fluxograma utilizando os outros tipos de repetição?



Contabilização de Totais

Existe uma forma simplificada de representar em fluxograma o processo de contabilização do total de determinada quantidade. Imagine que, numa campanha semelhante à descrita no [Exercício 7](#), esteja-se anotando o total do peso de alimentos doados no dia num papel com título TPA - Total de Peso dos Alimentos.

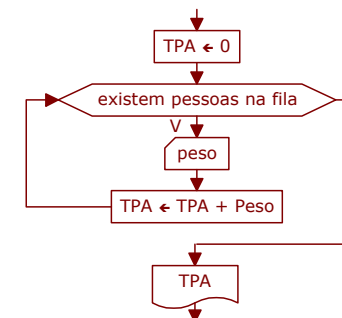
Pois bem. No início do dia, qual valor deve estar anotado nesse papel? Obviamente é zero (0). Toda vez que uma pessoa chega com alimentos, ao valor corrente anotado para o total deve ser adicionado o peso do novo donativo, e esse novo valor deve ser o novo total. Esse mecanismo é descrito na simulação da tabela abaixo:

Donativo	Peso (kg)	TPA (kg)
		0.0
1	2.0	2.0 \leftarrow 0.0 + 2.0
2	4.0	6.0 \leftarrow 2.0 + 4.0
3	1.5	7.5 \leftarrow 6.0 + 1.5
:	:	:
k-1	P _{k-1}	TPA _{novo} \leftarrow TPA _{anterior} + P _{k-1}
k	P _k	TPA _{novo} \leftarrow TPA _{anterior} + P _k
:	:	:
n	P _n	TPA _{novo} \leftarrow TPA _{anterior} + P _n

Note que, a cada novo donativo com peso P , o total acumulado é atualizado através da aplicação da relação $TPA_{novo} \leftarrow TPA_{anterior} + P$. Como só se tem um papel com o nome TPA, acontece o seguinte: pega-se o valor corrente de TPA, soma-se o valor de P , apaga-se o valor anterior de TPA e escreve-se (atribui-se: \leftarrow) no mesmo papel o novo valor calculado para TPA. A expressão que exprime isso é:

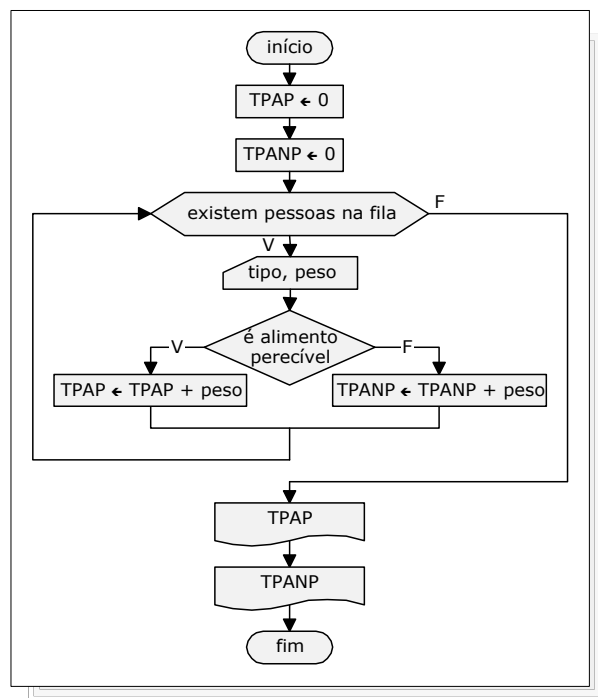
$$TPA \leftarrow TPA + P$$

TPA à direita do sinal de atribuição é o valor antigo; TPA à esquerda é o novo valor. A repetição, em fluxograma, para essa totalização fica da seguinte forma:



Exercício 8: fluxograma com contabilização de totais.

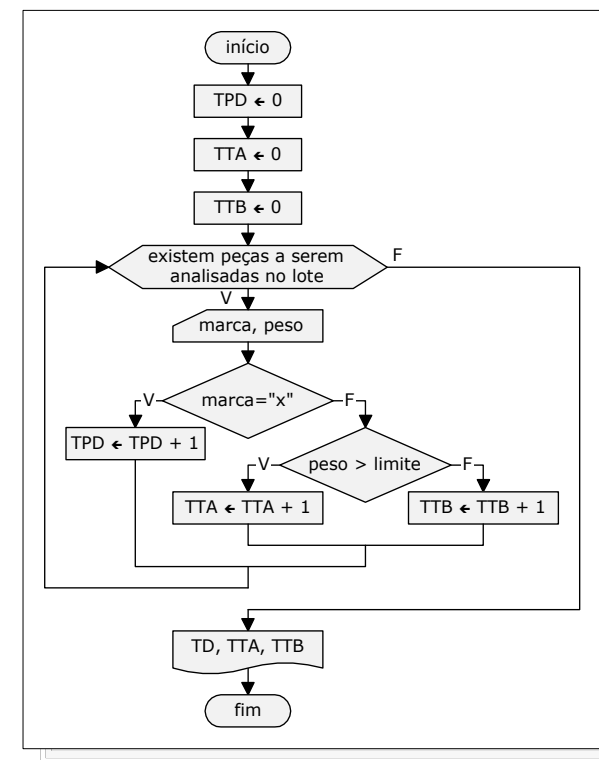
Modifique o fluxograma do Exercício 7 adotando a notação sugerida na observação acima.

Solução**Exercício 9:** fluxograma com contagens.

O controle de qualidade de uma empresa analisa lotes com números variáveis de peças. Em cada lote, as peças com defeito (tipo D) são retiradas e contadas, e as peças boas são classificadas

conforme um peso limite, sendo aquelas com peso maior que esse limite definidas como do tipo A, e as restantes como do tipo B. Cada tipo de peça boa (A ou B) também é contado.

Monte um fluxograma descrevendo esse processo de contagem para um lote. Considere como dado de entrada duas informações escritas em uma etiqueta colada na própria peça a ser analisada: uma marca (um "x": ao vê-lo, é possível saber que a peça é defeituosa) e o seu peso.

Solução

1.8. O Que nos Espera?

O objetivo desse capítulo é dar ao aluno uma primeira visão dos conceitos que serão desenvolvidos ao longo do curso. O que será visto de agora em diante é, conceitualmente falando, um refinamento e aprofundamento do que foi visto até esse ponto, colocado em termos mais específicos para a implementação em computador. Portanto, tenha certeza de que os conceitos vistos até aqui estejam claros antes de prosseguir.

Referências do Capítulo

- [1] Forbellone, A. L. V.; Eberspächer, H. F.: *"Lógica de Programação - A Construção de Algoritmos e Estruturas de Dados"*, Makron Books do Brasil Editora Ltda, São Paulo, Brasil, 1993.
- [2] Guimarães, A. M. e Lages, N. A. C.: *"Algoritmos e Estruturas de Dados"*, Livros Técnicos e Científicos Editora S.A, Rio de Janeiro, Brasil, 1994.
- [3] Polya, G.: *"A Arte de Resolver Problemas"*, 2ª reimpresssão da 2ª edição, Editora Interciência Ltda, Rio de Janeiro-RJ, Brasil, 1995.
- [4] Resolução de Problemas, encontrado no site: <http://athena.mat.ufrgs.br/~portosil/resu.html>.
- [5] Manzano, J. A. N. G. e Oliveira, J. F.: *"Algoritmos - Lógica para Desenvolvimento de Programação"*, Editora Érica Ltda., São Paulo, Brasil, 1996.

2. Tópicos Preliminares

2.1. Computadorizando as Soluções de Problemas

Foram vistos no Capítulo I esquemas de como solucionar problemas, e uma forma gráfica - o fluxograma - de como podemos descrever o algoritmo da solução. Essa representação gráfica permite objetividade, concisão e clareza na descrição dos passos a serem seguidos no processo de solução, o que é essencial para "ensinar" o computador a resolver problemas a partir de algoritmos. Entretanto, para conseguir isso é necessário ainda um nível maior de detalhamento do algoritmo, considerando já as características operacionais do equipamento. Para tal, serão apresentados nesse capítulo os primeiros conceitos de outro tipo de representação na forma textual, bem como outros procedimentos que permitem deixar a descrição da solução numa forma que seja "computadorizável".

2.2. Que Problemas Resolver com o Auxílio do Computador

Em princípio, qualquer problema relacionado com manipulação de informações pode ser resolvido através do uso do computador. Alguns exemplos:

- ☐ Aplicações científicas e de engenharia:
 - cálculo de estruturas (esforços, deformações, deslocamentos, etc.);
 - cálculo de trajetórias (de projéteis, foguetes, satélites, etc.);
 - determinação da dinâmica de vôo (de aeronaves);
 - simulação (de processos em indústria, de funcionamento de motores, etc.).
- ☐ Aplicações comerciais e administrativas:
 - folha de pagamento;
 - contas a pagar e a receber;
 - emissão de notas fiscais e faturamento;
 - controle de estoque;
 - cadastros de clientes, de fornecedores, de funcionários, etc.
- ☐ Aplicações domésticas:
 - controle de despesas e orçamento;
 - controle de contas bancárias;
 - cadastro de receitas de cozinha.

2.3. Organização e Estrutura Básica do Computador

O computador não é uma máquina com inteligência⁶. Na verdade, é uma máquina com uma grande capacidade para processamento de informações, tanto em volume de dados quanto na velocidade das operações que realiza sobre esses dados. Basicamente, podemos encarar o computador como estando organizado em três grandes funções ou áreas, que são: *entrada de dados, processamento de dados e saída de dados* - Figura 13.

⁶ Existe o conceito de inteligência artificial, que pode ser implementada no computador, porém ela está muito aquém da inteligência natural.

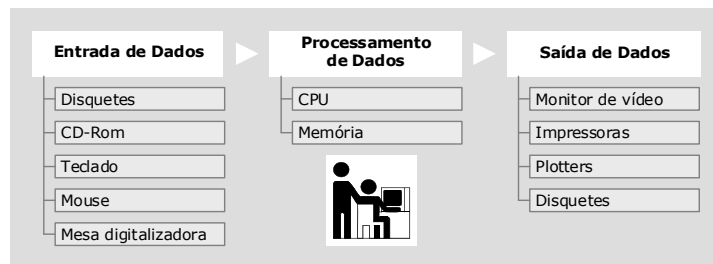


Figura 13: Organização e estrutura básica do computador (com os principais componentes).

A Figura 14 dá bem uma idéia da relação entre a estrutura funcional de um algoritmo e o esquema de funcionamento do computador.

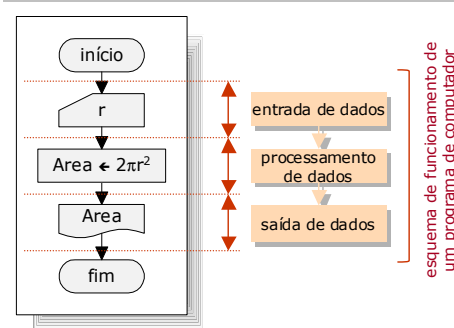


Figura 14: Relação entre o algoritmo e a estrutura funcional do computador

2.3.1. Entrada de Dados

Para o computador processar os dados, é preciso ter meios de fornecê-los a ele. Para isso, o computador dispõe de recursos como o teclado (para digitação, por exemplo, do texto que define um programa), o mouse (para selecionar opções e executar algumas operações em um software qualquer), disquetes e CDs (CD-ROMs) para entrada de dados (gerados provavelmente em algum outro computador), mesas digitalizadoras (muito utilizadas por programas CAD - Computer Aided Design, ou Projeto Auxiliado por Computador - e aplicativos gráficos em geral), e outros.

2.3.2. Processamento de Dados

Os dados fornecidos ao computador podem ser armazenados para processamento imediato ou posterior. Esse armazenamento de dados é feito na memória do computador, que pode ser volátil (isto é, desaparece quando o computador é desligado), referenciada como memória RAM (Random Access Memory - memória de acesso aleatório), ou pode ser permanente (enquanto não é "apagada" por alguém) através do armazenamento dos dados em unidades como as de disco fixo, que são meios físicos (meio magnético) localizadas no interior do gabinete do computador. Há também os disquetes, que são "discos removíveis".

O processamento dos dados é feito na CPU - Central Process Unit - unidade de processamento central (ou simplesmente processador, como o Pentium), onde a informação é tratada, sendo lida, gravada ou apagada da memória, sofrendo transformações de acordo com os objetivos que se deseja atingir com o processamento delas.

2.3.3. Saída de Dados

Os dados resultantes do processamento das informações pelo computador podem ser apresentados de inúmeras formas, e por meio de diversos dispositivos. O *monitor de vídeo* é um dos principais meios para se obter dados de saída do computador: tanto texto normal ou formatado (como em tabelas ou formulários) e gráficos podem ser apresentados ao usuário através desse dispositivo. Se quisermos que os resultados sejam apresentados em papel, podemos fazer uso de *impressoras* e/ou *plotters* (para "plotagem" de desenhos); se quisermos levar esses dados para outros computadores, podemos fazer uso, por exemplo, dos *disquetes*, ou então conectar os computadores em rede (resumidamente, ligá-los através de cabos).

2.4. O Algoritmo Implementado no Computador

Como já mencionado anteriormente, um algoritmo pode ser implementado de várias formas, inclusive através da utilização do computador. Porém, para que isso seja possível, temos que conseguir "conversar" com a máquina. Isto é possível através do uso de *programas de computador* (ou *softwares*), que é o meio pelo qual podemos controlar e nos comunicar com esse equipamento.

2.4.1. Programas de Computador

Um programa de computador pode ser definido como "*uma série de instruções ou declarações, em forma aceitável pelo computador, preparada de modo a obter certos resultados*"^[2]. Programas são gerados a partir de algoritmos.

Sendo também chamados de Softwares⁷, podemos classificá-los, de forma não exaustiva, em alguns tipos:

- **Sistemas Operacionais.** Como o próprio nome sugere, são softwares destinados à operação do computador. Tem como função principal controlar os diversos dispositivos do computador, e servir de comunicação intermediária entre o computador e os outros programas normalmente utilizados, o que facilita muito o desenvolvimento e criação desses outros programas. O antigo *DOS*, o *Windows (95/98/2000/NT/XP)* e o *LINUX* são exemplos de sistemas operacionais para microcomputadores. Também podemos citar o *OS/2*, a *IBM*, e o *System* para computadores *Macintosh*. Um computador, qualquer que seja o seu porte, não funciona sem um sistema operacional.
- **Programas Utilitários.** São programas destinados a facilitar e agilizar a execução de certas tarefas. Existem utilitários, por exemplo, para diagnosticar a situação do computador e seus diversos dispositivos (como o *Norton Utilities*), para compactar arquivos (como o *WinArj* e o *WinZip*), para realização de cópias de segurança ("backups"), etc.
- **Programas Aplicativos.** São os programas destinados a nos oferecer certos tipos de serviços, e pode-se incluir nesta categoria os processadores de texto (*Microsoft Word*, *WordPerfect*), as planilhas eletrônicas (*Microsoft Excel*, *Lotus-123*, *Quatro Pro*), programas gráficos (*CAD*, *Adobe PhotoShop*, *Corel Draw*), etc.
- **Compiladores e Interpretadores.** São programas utilizados para construir outros programas, e se caracterizam pelo tipo de linguagem empregada para isso. A maior parte dos outros tipos de programas citados acima são criados a partir do uso de compiladores (veja item *Linguagens de Programação* para maiores informações).

Para implementar os algoritmos em computador pode-se utilizar programas como planilhas eletrônicas ou compiladores e/ou interpretadores, baseados numa linguagem de programação. Tudo dependerá do problema a ser resolvido. Será visto a seguir algo sobre as linguagens de programação, ferramenta básica para "ensinar" o algoritmo para o computador.

2.4.2. Linguagens de Programação

As linguagens de programação existem para que seja possível comunicar ao computador as instruções daquilo que é desejado que ele realize em termos de processamento de dados. Na

⁷ Programa de computador e software tem um significado intercambiável. Usa-se mais o primeiro termo para aquilo que estamos construindo a partir dos algoritmos, e o segundo é mais utilizado fazendo referência a um programa de computador acabado e operacional.

verdade, o computador só entende uma linguagem, denominada *linguagem de máquina*, que é extremamente complicada para ser usada diretamente. Os programas são então escritos em outro tipo de linguagem, de mais fácil utilização, para então serem traduzidos para a linguagem de máquina e executados pelo computador. São basicamente duas as categorias principais de linguagem de programação: as *linguagens de baixo nível* e as *linguagens de alto nível*.

As *linguagens de baixo nível* são quase que praticamente a própria linguagem de máquina, porém utilizando uma simbologia mais apropriada e menos complexa (mnemônicos). Exemplo: a *linguagem assembly*. Uma instrução em linguagem assembly é equivalente a uma instrução em linguagem de máquina. Na Figura 15 é possível perceber a complexidade da linguagem de máquina (só números na base hexadecimal⁸) se comparada com a linguagem assembly, embora esta também tenha certo grau de dificuldade, porém menor. Uma característica dessas linguagens é que, para poderem ser utilizadas, é necessário que se conheça a arquitetura interna do computador, especialmente as características de seu processador (como Intel 486, Pentium, Pentium 2, etc.).

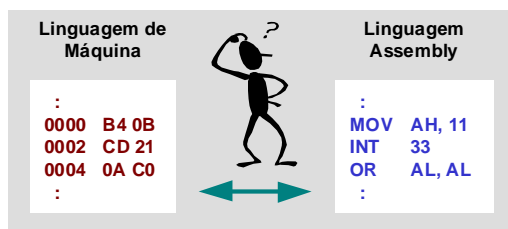


Figura 15: Comparação entre a sintaxe da linguagem de máquina e a da linguagem assembly (microprocessador 8088/8086⁹).

As *linguagens de alto nível*, ao contrário das linguagens de baixo nível, não estão "preocupadas" com a arquitetura interna do computador, e sim em fornecer as estruturas e comandos prontos para se resolver o problema desejado, tendo uma sintaxe que lembra a da linguagem natural. Na Figura 15, a instrução MOV AH, 11 indica que se está atribuindo o valor 11 ao registrador AH do microprocessador 8088/8086. Esse tipo de instrução não aparece nas linguagens de alto nível, pois para a sua utilização não é necessário conhecimento da arquitetura interna do computador ou de hardware (eventualmente, apenas de forma superficial).

As seguintes linguagens são ditas de alto nível: C, Pascal, Visual Basic, Clipper, Cobol, Fortran, Prolog, LISP, etc. (existem centenas). Uma instrução numa dessas linguagens pode corresponder a dezenas, centenas ou mesmo milhares de instruções em linguagem de máquina. Quem realiza a "tradução" das instruções escritas numa linguagem de programação para instruções em linguagem de máquina são os montadores (para as de baixo nível), os compiladores e os interpretadores (esse dois para as de alto nível) - veja a Figura 16.

Compiladores e interpretadores traduzem programas escritos em linguagem de alto nível de maneira diferente. Um *interpretador* lê uma instrução do programa de cada vez, traduz para a linguagem de máquina e a executa, linha por linha do programa. Exemplo de interpretador: a linguagem BASIC¹⁰. O *compilador*, por sua vez, lê todas as instruções do programa e faz a sua tradução para linguagem de máquina de uma só vez, para só depois executar (fazer funcionar) o programa. Exemplos de compiladores: para as linguagens C, Pascal, Cobol, Fortran, etc.

⁸ Normalmente nós trabalhamos com números na base decimal, isto é, de 0 até 9 (10 dígitos), que é natural para nós. No computador, as informações são tratadas usando-se a base binária (0 e 1 - 2 dígitos), sendo a base hexadecimal (0, 1, 2, ..., 9, A, B, C, D, E, F - 16 dígitos) uma maneira mais compacta para representação das informações binárias.

⁹ Esse microprocessador era utilizado nos antigos PC-XTs.

¹⁰ As planilhas eletrônicas (como o MS-Excel) podem, sob certo ponto de vista, ser consideradas também como interpretadores, pois nós podemos programá-las: cada célula é programada utilizando-se uma fórmula (instrução), que é testada para verificar se seu conteúdo está correto, antes de se poder passar para uma próxima célula.

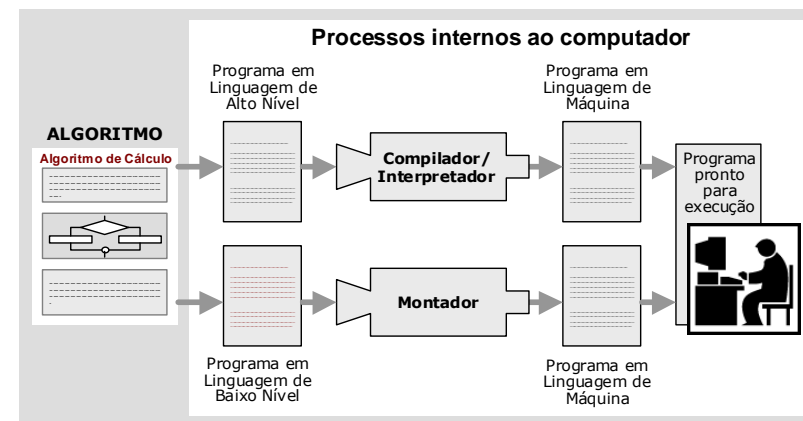


Figura 16: Processo de transformação de um algoritmo em um programa de computador.

2.5. Técnicas a Serem Utilizadas

2.5.1. Representação Textual - O Português Estruturado

Como mencionado no item anterior, programa-se computadores através das linguagens de programação. É para essa representação, essencialmente textual, que os algoritmos devem ser convertidos. Entretanto, o algoritmo em si deve ser criado antes, e também representado de uma forma voltada para implementação, mas independente da linguagem que será utilizada. Como mostrado na Figura 17, existem vários níveis de abstração¹¹ nas representações utilizadas.

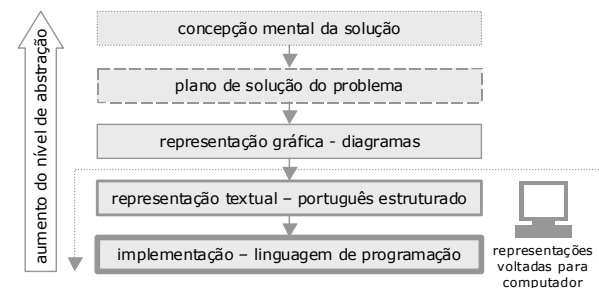


Figura 17: Níveis de abstração das representações de algoritmos.

O mais baixo nível de abstração, e por isso com o maior nível de detalhamento, é o próprio programa de computador. Será visto um nível imediatamente anterior: uma forma textual que leva em consideração as particularidades associadas à escrita de programas, porém de forma um pouco menos formal do que uma linguagem de programação propriamente dita. Utilizar-se-á o *Português Estruturado* como representação textual para descrever os algoritmos a serem estudados. Note que o fluxograma, visto no capítulo anterior, pode representar um algoritmo que não necessariamente será implementado em computador.

¹¹ **Abstração:** representação de determinada realidade considerando apenas as características essenciais necessárias em determinado contexto. Maior abstração significa menos detalhamento - abstrair = tirar (apenas o essencial).

Definição

O *Português Estruturado*, chamado também de Portugol por alguns autores, é o que se pode chamar de uma pseudolinguagem de programação: possui uma notação que lembra as diversas linguagens de programação existentes, porém incorpora características de nossa língua - o português - tornando-se assim mais fácil de compreender¹². É uma técnica formal, até certo ponto, de notação para expressão de algoritmos, com as seguintes características importantes:

- permite uma argumentação convincente e concisa (isto é, clara, objetiva e não prolixa);
- afasta a possibilidade de ambigüidade de tal maneira que, dado um algoritmo descrito segundo essa técnica, e um estado inicial, a sua execução leva sempre ao mesmo estado final.

Sintaxe e Semântica

Em toda a linguagem, as frases construídas envolvem dois aspectos: a sintaxe e a semântica. A sintaxe tem a ver com a forma, e a semântica com o conteúdo ou significado. Sendo o português também uma linguagem, analisemos a frase sintaticamente correta¹³ mostrada na [Figura 18](#). A semântica correta dessa frase é indicar que no referido local existe uma venda de frangos já mortos, e não frangos "deprimidos" ou "anêmicos". Esse tipo de ambigüidade não pode existir quando se estiver tratando da sintaxe e semântica de linguagens de programação, ou na construção de algoritmos.

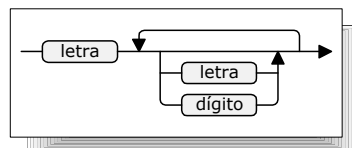


Figura 18: Frase sintaticamente correta, mas semanticamente dúbia.

No português estruturado a sintaxe é definida, e **a forma apresentada deve ser aceita e respeitada como padrão**. Para cada declaração ou comando, a semântica está devidamente explicada e clara. Ao longo desse material, isso será visto com maiores detalhes.

Exemplo 1: sintaxe de um identificador no português estruturado.

Um elemento básico de uma linguagem é o *identificador*, usado para dar nome a variáveis, funções e outras entidades. A sua sintaxe é definida conforme o diagrama¹⁴ da [Figura 19](#).



Identificadores válidos:

A, B1, BC3D, A4, z1976, j, soma, CONTADOR.

Identificadores não válidos:

2ab, 3A, 00D.

Figura 19: Sintaxe de um identificador no português estruturado

Qualquer caminho seguido no diagrama acima levará a construção de um identificador válido, isto é, com a sintaxe correta (considerando letra como as letras do alfabeto¹⁵, e dígito como os dígitos do sistema decimal).

¹² Pode-se dizer que se trata de uma mistura do português com as linguagens de programação Algol e Pascal.

¹³ Possui sujeito, verbo e predicado, está com a concordância correta, etc.

¹⁴ O diagrama mostrado é baseado no diagrama BNF - Backus-Naur Form. Ele deve ser lido da esquerda para a direita, e a leitura no sentido contrário só é possível se existir um caminho explícito naquela direção.

¹⁵ Vamos utilizar apenas as letras do alfabeto (a, b, ..., z, A, B, ..., Z) e alguns caracteres especiais (-, _, \$). Cada linguagem de programação tem a sua regra de construção de identificadores: em algumas é possível utilizar vários caracteres especiais, além das listadas aqui. Usar caracteres acentuados, ou ç, não é permitido praticamente em todas as linguagens de programação.

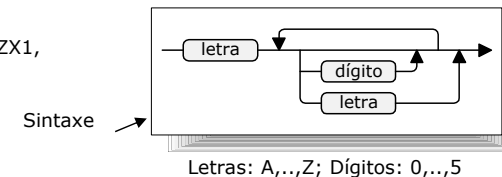
A sintaxe acima descrita diz que um identificador pode conter tanto letras como dígitos em seu corpo, mas deve sempre iniciar com uma letra.

Exercício 1: construção de identificadores sintaticamente corretos

Diga quais identificadores da lista abaixo são corretos, considerando a sintaxe expressa pelo diagrama ao lado.

Identificadores:

AT3, Cv12, X3B, SOMA, 5A, Z, Z1, Z1x, ZX1, B2222Y, B7Y.



Solução

Identificadores válidos:

X3B, Z, Z1, B2222Y.

Identificadores não válidos:

AT3, Cv12, SOMA, 5A, Z1x, ZX1, B7Y.

Observação:

A sintaxe acima descrita diz que um identificador pode conter tanto letras (apenas maiúsculas) como dígitos (variando de 0 a 5), mas deve sempre iniciar com uma letra, sendo essa seguida (ou não) tanto por dígitos quanto por letras. Entretanto, apenas uma segunda letra é possível, e na última posição, o que obrigatoriamente encerra a definição do identificador.

Estrutura básica de um Algoritmo em Português Estruturado

Um algoritmo em português estruturado segue a estrutura mostrada na figura ao lado: sempre delimitado entre as palavras início e fim, e entre elas, nessa ordem, vêm as declarações (de variáveis) e as instruções (seqüências simples, alternativas e repetições).

```

início
<declarações>
<instruções>
fim
  
```

2.5.2. Representação Gráfica

Podemos utilizar também representações gráficas para construir algoritmos. Em muitas situações, é mais fácil e cômodo utilizar gráficos ao invés de texto para exprimir uma determinada lógica antes de passar para uma representação mais detalhada, como o português estruturado. Outras vezes é mais adequado o uso da combinação das duas representações.

Faremos uso de duas representações gráficas: o *Fluxograma* (já visto no Capítulo 1) e o *Diagrama de Chapin*¹⁶ - [Figura 20](#). À medida que for sendo apresentado o português estruturado, usado como representação textual, serão mostradas em paralelo as representações gráficas equivalentes. Daremos maior ênfase à representação gráfica utilizando o fluxograma.

¹⁶ Também referenciado como diagrama de Nassi-Schneiderman.

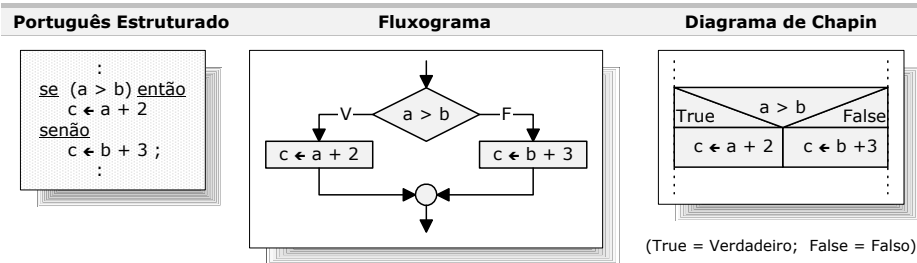


Figura 20: Exemplo de representações textual e gráfica de algoritmos.

2.6. Tipos e Estruturas de Dados

2.6.1. Conceito

Como mencionado anteriormente, os dados manipulados pelo computador são armazenados na sua memória, tanto na volátil (RAM) como na fixa (disco fixo ou rígido, também chamado de *winchester*). De agora em diante, quando se estiver falando de tipos e estruturas de dados, estará se falando de como esses dados são armazenados na memória volátil do computador.

É através de variáveis que se consegue guardar e manipular informações no computador. O conceito de variável é semelhante ao que já se conhece da matemática: uma quantidade que pode assumir qualquer valor dentro de um certo conjunto de valores, e que pode ter esse valor alterado em algum instante ao longo do tempo. Pois bem: são nas variáveis dos diversos tipos que os dados manipulados pelo computador são armazenados. Uma variável pode guardar valores de determinado *tipo de dado* (inteiro, real, caractere, etc.), e quando se estiver falando de um conjunto de variáveis com um significado particular, como um vetor ou uma matriz, estará se falando de uma *estrutura de dados*.

Portanto, para se armazenar dados no computador é preciso definir as variáveis que irão conter estes dados. As variáveis são nomeadas (através de identificadores - Exercício 1) e estão associadas às posições na memória do computador onde os dados de tipos determinados são guardados.

Na Figura 21 é mostrado simbolicamente como é feito esse armazenamento. Nela foram definidas duas variáveis (com os identificadores x e y), associadas às posições na memória onde cada dado de um certo tipo é guardado: na "gaveta x " só é possível armazenar números inteiros, e na "gaveta y " só podem ser colocados números reais.

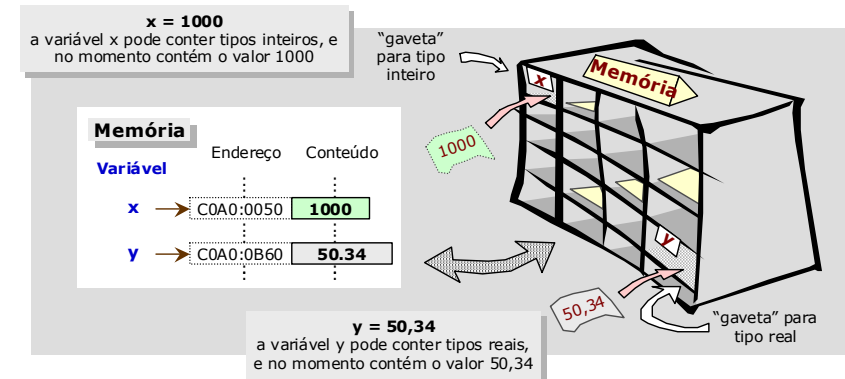


Figura 21: Como os dados são armazenados na memória do computador.

Mas por quê deve-se definir tipos de dados? De forma muito resumida, o tipo de dado está associado à quantidade de espaço na memória que o computador irá disponibilizar para armazená-lo. Seja o número π . Para guardar esse número real de forma exata, considerando todas as casas decimais, seria necessária uma quantidade infinita de memória. Mas para a solução de problemas práticos, não é necessário conhecer o valor de π com 1000 casas decimais! Dependendo do problema, $\pi = 3.1416 - 4$ casa decimais - é mais do que suficiente. É esse o raciocínio empregado na definição dos tipos de dados no computador: quantos algarismos significativos serão utilizados para representar um número; isso define os espaços de memória requerido para armazenar o tipo de dado no caso numérico.

Outro aspecto importante em relação a dados e variáveis, além do tamanho, é a compatibilidade de tipos. Se uma variável é definida como sendo de determinado tipo, o espaço reservado para ela na memória só pode receber dados desse tipo (ou de sub-tipos no caso das linguagens de programação).

A analogia mostrada na Figura 22 facilita a compreensão do exposto no parágrafo anterior, sem ser necessário entrar em detalhes sobre a exata representação interna de dados em computadores. Nessa figura, as caixas do tipo real e do tipo inteiro são espaços reservados para receber dados dos respectivos tipos - elas representam as variáveis. Como veremos a seguir, tais espaços reservados são definidos através da declaração de variáveis. Analisando as setas na figura, nota-se que a caixa do tipo real pode receber tanto números reais como inteiros ($\mathbb{Z} \subset \mathbb{R}^{17}$), ao passo que se for tentado forçar a colocação de um número real na caixa para tipo inteiro, só a parte do número correspondente ao tronco de cone entrará nessa caixa: o restante da informação é perdido. Exemplo: se for tentado colocar o valor de π (3.14159...) na caixa para tipo inteiro, apenas a parte inteira consegue entrar - a parte fracionária é eliminada - e o conteúdo passa a ser 3.

¹⁷ O conjunto dos números inteiros - \mathbb{Z} - está contido no conjunto dos números reais - \mathbb{R} .

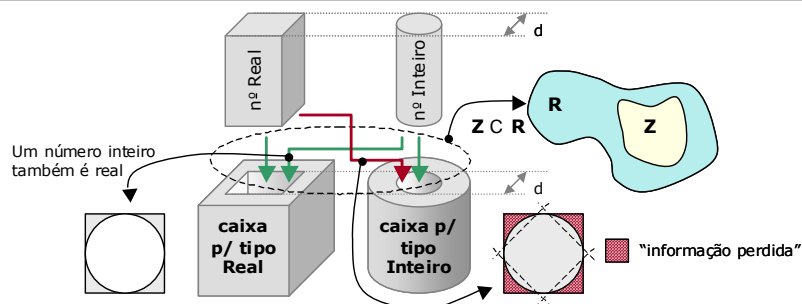


Figura 22: Dados são armazenados no computador conforme seu tipo. A tentativa de guardar um valor de tipo não compatível com o tipo da variável pode ocasionar perda de informação.

2.6.2. Tipos Básicos de Dados

Como já discutido anteriormente, o computador trabalha com variáveis de diversos tipos para manipular os dados utilizados nos programas elaborados a partir de algoritmos. São quatro os tipos básicos que serão utilizados para declarar variáveis:

- **Inteiro**: qualquer número inteiro, negativo, nulo ou positivo. Ex.: -5, 0, 237.
- **Real**: qualquer número real, negativo, nulo ou positivo. Ex.: -3.5, 0.0, 3.7.
- **Caractere**: qualquer conjunto de caracteres alfanuméricos. Ex.: "AB", "XYZ", "Abacate".
- **Lógico**: conjunto de valores *falso* (.F.) ou *verdadeiro* (.V.) em proposições lógicas.

A partir desses tipos básicos podem-se definir estruturas de dados mais complexas, mas isso será visto apenas mais adiante.

2.7. Variáveis

2.7.1. Definição

Em termos computacionais, uma *variável* representa um local ou endereço na memória do computador onde se pode armazenar qualquer valor de um conjunto de valores possíveis para o tipo básico definido para a variável em questão - Figura 23. O nome de uma variável é um *identificador*, tal como definido no Exemplo 1.

Ao se definir um nome de uma variável, devemos procurar usar nomes que transmitam de alguma maneira o significado da variável. Por exemplo, se um professor construir um algoritmo para cálculo da média dos alunos a partir das suas notas, não é razoável usar uma variável chamada *x* para guardar a média e *y* para guarda a nota. O algoritmo pode ficar confuso. É mais conveniente fazer o seguinte: usar uma variável chamada *media* para guardar a média, e *nota* para guardar a nota; é melhor para o entendimento do algoritmo.

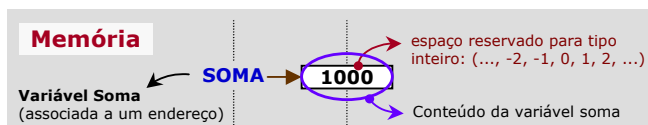


Figura 23: Uma variável está associada a um local na memória, definido por um endereço, que pode armazenar um valor do tipo da variável.

Como o próprio nome sugere, uma variável pode ter seu valor alterado em algum instante ao longo do tempo¹⁸. Isso é comum, já que durante a execução dos algoritmos dados estão sendo manipulados e transformados de diversas maneiras através do uso das variáveis.

2.7.2. Declaração

Aqui cabe uma pergunta: como é que o computador sabe o local ou endereço onde "colocar" ou "pegar" a variável que irá utilizar/processar? Não sabe. É preciso "avisá-lo" de alguma forma que será criada uma variável, e que para tal será necessário que seja reservado um local para guardar o valor que a ela será atribuído. Isso é feito através do que se denomina *declaração de variável*, cuja sintaxe no português estruturado está mostrada na Figura 24.

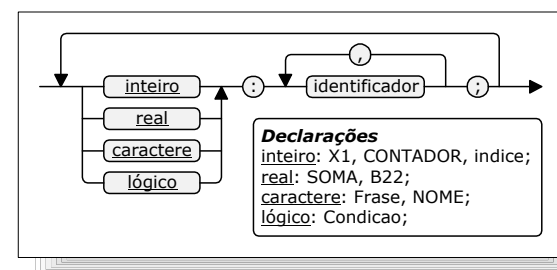


Figura 24: Declaração de variáveis no português estruturado.

Palavras-Chave

São palavras ditas reservadas do português estruturado, com sintaxe e semântica bem definida. Todas as palavras-chave do português estruturado aparecerão nos algoritmos grifadas (ou em negrito quando for possível utilizá-lo). Exemplo: **inteiro** (*inteiro é uma palavra reservada que define um tipo de dado*).

É preciso notar que, ao se declarar uma variável, o valor que ela contém *não é conhecido*. Algumas linguagens de programação definem um valor padrão (como zero para variáveis numéricas) como conteúdo inicial da variável logo após a declaração; outras não definem nada, ficando como valor inicial da variável o valor que estiver no local da memória para ela reservado no momento da sua declaração. Por isso, muito cuidado: nunca utilize uma variável em um algoritmo (e principalmente em um programa de computador) sem ter certeza de ter atribuído antes algum valor para ela (no item 2.9.1, página 42, veremos como isso é feito).

Não há declaração de variáveis nos diagramas!

Nas representações gráficas (fluxograma e diagrama de Chapin) não se inclui as declarações das variáveis, pois elas não estão associadas especificamente à programação de computador (veja novamente a Figura 17). Essas representações não são tão formais quanto o português estruturado, que já está num formato adequado para ser traduzido para uma linguagem de programação, onde essas declarações são obrigatórias.

2.8. Operadores e Expressões

Algoritmos implementados em computador geralmente envolvem cálculos matemáticos - expressões aritméticas - e muitos pontos de decisão e controle - expressões lógicas ou relacionais. Portanto, é necessário antes saber como trabalhar com essas expressões.

¹⁸ Em contraposição à variável, existe o conceito de **constante** que, ao contrário, não pode ter seu valor alterado ao longo da execução do algoritmo ou programa. As linguagens de programação trabalham com esse conceito, mas não vemos no português estruturado nenhuma construção equivalente por não ser essencial na construção dos algoritmos que serão vistos.

2.8.1. Operadores e Expressões Aritméticas

Serão utilizados basicamente os seguintes operadores aritméticos (Tabela 1 e Tabela 2):

Símbolo	Significado	Exemplo
+	soma	$2 + 4 (= 6)$
- (unário)	número negativo	-3
- (binário)	subtração	$4 - 2 (= 2)$
*	multiplicação	$2 * 4 (= 8)$
/	divisão	$6 / 3 (= 2)$
$\sqrt{a + b}$ ou $\text{raiz}(a + b)$	raiz quadrada	$\sqrt{4} (= 2)$
$(a + b)^n$ ou $(a + b)**n$ ou $(a + b)^n$	potência	$3^2 = 3^2 = 3**2 (= 9)$

Tabela 1: Operadores aritméticos utilizados no português estruturado - geral.

Símbolo	Significado	Exemplo
$\text{mod}, m \text{ mod } i$ (m e i são inteiros)	resto (módulo) da divisão inteira de m por i	$5 \text{ mod } 2 = 1$ $14 \text{ mod } 4 = 2$
$\text{div}, m \text{ div } i$ (m e i são inteiros)	quociente da divisão inteira de m por i	$5 \text{ div } 2 = 2$ $14 \text{ div } 4 = 3$

Tabela 2: Operadores aritméticos utilizados na aritmética com inteiros.

Além dos operadores, também será feito uso de funções matemáticas, algumas delas listadas na Tabela 3.

Função	Sintaxe	Exemplo
seno, cosseno, tangente e arcotangente do ângulo x	$\text{sen}(x)$, $\text{cos}(x)$, $\text{tg}(x)$ e $\text{arctg}(x)$	$\text{sen}(45) = 0.7071$
exponencial de x (e^x)	$\text{exp}(x)$	$\text{exp}(2) = 7.389056$
valor absoluto de x	$\text{abs}(x)$ ou $ x $	$\text{abs}(-2) = 2$; $\text{abs}(3) = 3$
sinal de x (-1 se $x < 0$, 1 se $x > 0$)	$\text{sinal}(x)$	$\text{sinal}(-3) = -1$
parte inteira de x: o maior número inteiro que é menor ou igual a x	$\text{int}(x)$ ou $\lfloor x \rfloor$	$\text{int}(3.45) = 3$
menor número inteiro que é maior ou igual a x	$\lceil x \rceil$	$\lceil 4.23 \rceil = 5$

Tabela 3: Algumas funções matemáticas utilizadas no português estruturado.



Flexibilidade do Português Estruturado

É possível introduzir novos operadores ou nomes de funções para adaptar a linguagem às necessidades específicas da área de aplicação do problema a ser resolvido, desde que bem definidos sintaticamente e semanticamente, sem ambigüidades. Isso é permitido no português estruturado por se tratar de uma pseudolinguagem, sem o formalismo de uma linguagem de programação, onde temos, algumas vezes, que nos restringir aos recursos por ela pré-definidos, conforme a especificação do fabricante do compilador ou interpretador.

Expressões aritméticas são aquelas cujos operadores são aritméticos, e cujos operandos são constantes ou variáveis do tipo numérico (inteiro ou real), ou funções matemáticas.

Quando uma expressão aritmética possui diversos operadores, a ordem e a sequência em que as diversas operações são realizadas são definidas pela prioridade de cada operador. Uma operação

com prioridade de execução maior do que outra significa que a primeira será avaliada (ou realizada) antes, como na seguinte expressão aritmética, onde sabe-se que $2 + 3 * 2 = 8$, pois aplicando as regras de prioridade (Tabela 4), $2 + 3 * 2 = 2 + (3 * 2)$ - a multiplicação é feita antes da adição, ou seja, o operador multiplicação tem prioridade maior. Para se alterar prioridades, utilizam-se os parênteses, e podemos ter $(2 + 3) * 2 = 10$, pois os parênteses têm maior prioridade que qualquer outro operador. Quando os operadores possuem a mesma prioridade, as respectivas operações são realizadas da esquerda para a direita.

Prioridade	Operador
("00")	+, - (unários - <u>se definido!</u> - veja nota abaixo)
1ª	** ou ^
2ª	*, /, div, mod
3ª	+, - (binários)

Tabela 4: Prioridade entre operadores aritméticos

Exemplo 2: expressões aritméticas

- a) $2 + 2$ b) $XPTO / 5$ c) $\text{raiz}(X) - 3$ d) $X**2 + 3$
e) $y^3 * 2$ f) $3**2 / 5$ g) $2 * \text{Nota}$ h) $4 * \cos(x) - 1$

Exemplo 3: prioridade em operações aritméticas

Como expressões aritméticas são colocadas sempre de forma sequencial em uma linha de instrução, as prioridades dos operadores ditam a sequência das operações, como mostrado abaixo. Para alterar a prioridade, utilizam-se parênteses.

- a) $8 / 2 * 3 = 4 * 3 = 12$ b) $12 / 2 ^ 2 = 12 / 4 = 3$
c) $4 / 4 - 2 = 1 - 2 = -1$ d) $3 - 6 / 3 = 3 - 2 = 1$
e) $4 / (4 - 2) = 4 / 2 = 2$ f) $(3 - 6) / 3 = -3 / 3 = -1$



O Operador "-" Unário

Em alguns ambientes de programação, como na planilha Excel (mas não no VBA), define-se um novo operador: o operador unário "-". Um operador é unário quando possui apenas um operando, e binário quando possui dois. Exemplos:

- Operadores binários: $2 - 3$, $3 + 4$, ..., <operando 1> operador <operando 2>.
- Operadores unários: -2 , $+4$, ..., operador <operando 1>.

Nessa situação, a prioridade do operador unário "-" é maior do que a do binário, e também maior do que o operador de multiplicação, divisão e potenciação. Assim, as seguintes relações são verdadeiras:

- $-1^2 + 1^2 = (-1)^2 + 1 = 1 + 1 = 2$ (aqui, "-" é unário, com a maior prioridade)
- $1^2 - 1^2 = 1^2 - (1)^2 = 1 - 1 = 0$ (aqui, "-" é binário, com prioridade menor do que a da potenciação)

Na convenção matemática que normalmente se utiliza não é definido tal uso do operador "-".¹⁹

Exercício 2: expressões aritméticas - 1.

Calcule o resultado das seguintes expressões aritméticas:

- a) $12 + 3^2 - 9 * 6 / 2 * 3$ b) $3 * 8 / 2 - 1 * 4$ c) $3 / 25 + 2 / 5 ^ 2$

Solução

- a) $12 + 9 - 9 * 6 / 2 * 3 = 12 + 9 - 81 = -60$
b) $(3 * 8 / 2) - (1 * 4) = (24 / 2) - 4 = 12 - 4 = 8$
c) $(3 / 25) + (2 / 25) = 5 / 25 = 1 / 5 = 0.2$

¹⁹ Operacionalmente, isto seria o caos, e igualdades como a seguinte seriam sempre verdade: $-x^2 = x^2$.

Exercício 3: expressões aritméticas - 2.

Converte para o português estruturado as seguintes expressões aritméticas:

$$a) \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad b) \quad x - \frac{y^2 - 1}{z - 3} + w$$

Solução

$$a) \quad (-b + \text{raiz}(b^2 - 4*a*c)) / (2*a) \\ b) \quad x - (y^2 - 1) / (z - 3) + w$$

2.8.2. Operadores e Expressões Lógicas

Os operadores lógicos que utilizaremos no português estruturado são mostrados na [Tabela 5](#) e na [Tabela 6](#). Esses operadores são utilizados em *expressões lógicas*, que são aquelas cujos operadores são lógicos ou relacionais (vistos a seguir), e cujos operandos são outras expressões, constantes ou variáveis do tipo lógico, que assumem um de dois valores: verdadeiro - .V. - ou falso - .F. (utilizar-se-á pontos antes e depois para enfatizar o fato de se tratar de valor lógico).

Operador lógico	e			ou		
Significado	conjunção			disjunção não exclusiva		
Tabela Verdade	A	B	A e B	A	B	A ou B
	.F.	.F.	.F.	.F.	.F.	.F.
	.F.	.V.	.F.	.F.	.V.	.V.
	.V.	.F.	.F.	.V.	.F.	.V.
	.V.	.V.	.V.	.V.	.V.	.V.

[Tabela 5:](#) Operadores lógicos no português estruturado.

Operador lógico	xou			não	
Significado	disjunção exclusiva			negação	
Tabela Verdade	A	B	A xou B	A	não A
	.F.	.F.	.F.	.F.	.V.
	.F.	.V.	.V.	.V.	.F.
	.V.	.F.	.V.		
	.V.	.V.	.F.		

[Tabela 6:](#) Operadores lógicos no português estruturado.

Em uma mesma expressão utilizando operadores lógicos, a prioridade de execução das operações é a seguinte:

Prioridade	1º	2º	3º	4º
Operador	não	e	ou	xou

Exemplo 4: operadores lógicos.

Expressão	Quando eu não saio?
Se chover <u>e</u> relampejar, eu não saio.	Somente quando chover e relampejar ao mesmo tempo (apenas 1 possibilidade).
Se chover <u>ou</u> relampejar, eu não saio.	Somente quando chover, somente quando relampejar ou quando chover e relampejar ao mesmo tempo (3 possibilidades).
Se chover <u>xou</u> relampejar, eu não saio.	Somente quando chover, ou somente quando relampejar (2 possibilidades).

Exemplo 5: expressões utilizando operadores lógicos

Para A = .V., B = .F. e C = .F., as expressões abaixo fornecem os seguintes resultados²⁰:

- a) não A; \Rightarrow não .V. = **.F.**
 b) A e B; \Rightarrow .V. e .F. = **.F.**
 c) A ou B xou B ou C; \Rightarrow (.V. ou .F.) xou (.F. ou .F.) = .V. xou .F. = **.V.**
 d) não (B ou C); \Rightarrow não (.F. ou .F.) = não .F. = **.V.**
 e) não B ou C; \Rightarrow (não .F.) ou .F. = .V. ou .F. = **.V.**

Exercício 4: expressões utilizando operadores lógicos

Para A = .V., B = .V. e C = .F., qual o resultado da avaliação das seguintes expressões:

- a) A e B ou A xou B; b) A ou B e A e C; c) A ou C e B xou A e não B;

Solução

- a) **.V. e .V. ou .V. xou .V.**
 .V. ou .V. xou .V.
 .V. xou .V.
 .F.
 b) **.V. ou .V. e .V. e .F.**
 .V. ou **.V. e .F.**
 .V. ou .F.
 .V.
 c) **.V. ou .F. e .V. xou .V. e não .V.**
 .V. ou **.F. e .V. xou .V. e .F.**
 .V. ou .F. xou .F.
 .V. xou .F.
 .V.

2.8.3. Operadores Relacionais

Utilizam-se os operadores relacionais, mostrados na [Tabela 7](#), para realizar comparações entre valores *do mesmo tipo básico*. Tais valores são representados por constantes, variáveis ou expressões, geralmente aritméticas. O resultado dessas comparações sempre é um valor lógico: .V. ou .F.. Todos esses operadores têm a *mesma prioridade* na avaliação de expressões relacionais, sendo, portanto, calculados da esquerda para a direita.

²⁰ No item a desse exemplo os parênteses foram utilizados apenas para deixar claro quais operações que estão sendo realizadas primeiro. Isso porque, da forma que foram colocados, não alteram (nem deveriam alterar!) a expressão.

Operador	Significado
=	igual a
>	maior que
<	menor que
≠, <>	diferente de
≥, >=	maior ou igual a
≤, <=	menor ou igual a

Tabela 7: Operadores relacionais



Geralmente não é possível dois ou mais operadores relacionais consecutivos em uma expressão, pois o resultado de uma comparação sempre é um valor lógico, o que traria problemas numa expressão como a abaixo:

$4 > 3 > 2 \Rightarrow (4 > 3) > 4 \Rightarrow .V. > 2 \Rightarrow ?$

Isso é resolvido colocando-se a expressão acima na seguinte forma:

$(4 > 3) \text{ e } (3 > 2) \Rightarrow .V. \text{ e } .V. \Rightarrow .V.$

A expressão abaixo, com vários operadores relacionais, é possível, pois é mantida a compatibilidade de tipos:

$4 > 3 = .F. \Rightarrow (4 > 3) = .F. \Rightarrow .V. = .F. \Rightarrow .F.$

Exemplo 6: expressões relacionais.

- a) $2 * 4 = 24 / 3$
 $8 = 8$
 .V.
- b) $15 \bmod 4 < 19 \bmod 6$
 $3 < 1$
 .F.
- c) $3 * 5 \bmod 4 \leq 3 * 2 / 0.5$
 $15 \bmod 4 \leq 9 / 0.5$
 $3 \leq 18$
 .V.
- d) $2 + 8 \bmod 7 \geq 3 * 6 - 15$
 $2 + 1 \geq 18 - 15$
 $3 \geq 3$
 .V.

Exemplo 7: expressões lógicas com operadores relacionais e lógicos.

- a) $(2 > 4) \text{ e } (24 > 3)$
 .F. .V.
 .F.
- b) $(15 \bmod 4 > 2) \text{ ou } (19 \bmod 6 < 2)$
 $(3 > 2) \text{ ou } (1 < 2)$
 .V. .V.
 .V.

Exercício 5: expressões lógicas com operadores relacionais e lógicos.

Seja A = .F. e B = .V., calcule as expressões abaixo:

- a) $(A \text{ ou } 5 > 4) \text{ e } (24 > 3 \text{ e } B)$
- b) $(B \text{ ou } 17 \bmod 4 > 2) \text{ ou } (9 \bmod 6 < 2)$

Solução

- a) $(.F. \text{ ou } .V.) \text{ e } (.V. \text{ e } .V.)$
 .V. .V.
 .V.
- b) $(.V. \text{ ou } (1 > 2)) \text{ ou } (3 < 2)$
 $(.V. \text{ ou } .F.) \text{ ou } .F.$
 .V. .F.
 .V.

2.8.4. Prioridade entre Operadores

Em expressões com operadores de diferentes tipos existe uma ordem que deve ser seguida ao se executar cada operação. Por exemplo, na expressão aritmética $2 + x * 2$, sabe-se que a operação de multiplicação deve ser realizada antes da operação de adição. Na expressão $2 < 3 - 1$ deve-se subtrair 1 de 3 antes de se realizar a comparação.

Para os diversos tipos de operadores/operações vistos existem prioridades a serem respeitadas, as quais estão estabelecidas na [Tabela 8](#).

Prioridade	Operador
1ª	parênteses e funções
2ª	operadores aritméticos: +, - (unários) (se definidos!) ** ou ^ *, /, div, mod +, - (binários)
3ª	operadores relacionais: <, ≤ (<=), =, ≥ (>=), >, ≠ (<>)
4ª	operadores lógicos: não e ou xor

Tabela 8: Prioridade entre os vários tipos de operadores.

Da mesma forma que para os operadores aritméticos, quando as prioridades dos operadores são as mesmas em uma expressão, as operações são feitas da *esquerda para a direita*.

Exemplo 8: prioridade entre operadores de vários tipos.

Observe a sequência em que cada operação foi executada, e verifique o efeito do uso dos parênteses no item b para alterar a prioridade das operações na expressão. Note que a diferença do resultado é total.

- a) $5 * 3 + \text{raiz}(16) < 2 * 3 ^ 2$
 $5 * 3 + 4 < 2 * 3 ^ 2$
 $5 * 3 + 4 < 2 * 9$
 $15 + 4 < 18$
 $19 < 18$
 .F.
- b) $5 * 3 + \text{raiz}(16) < (2 * 3) ^ 2$
 $5 * 3 + 4 < 6 ^ 2$
 $5 * 3 + 4 < 36$
 $15 + 4 < 36$
 $19 < 36$
 .V.

Exercício 6: prioridade entre operadores.

Qual será o resultado de cada expressão abaixo:

- a) $36 / 2 - 3 \geq 6 + 3 ^ 2$
- b) $36 / 2 - 3 \geq (6 + 3) ^ 2$

Solução

- a) $36 / 2 - 3 \geq 6 + 9$
 $18 - 3 \geq 6 + 9$
 $15 \geq 15$
 .V.
- b) $36 / 2 - 3 \geq 9 ^ 2$
 $36 / 2 - 3 \geq 81$
 $18 - 3 \geq 81$
 $15 \geq 81$
 .F.

Exercício 7: expressões mistas.

Calcule o resultado das seguintes expressões:

- a) $.F. \text{ ou } 20 \bmod (18/3) = (21/3) \bmod 2$
 $.F. \text{ ou } 20 \bmod 6 = 7 \bmod 2$
 $.F. \text{ ou } 3 = 3$
 $.F. \text{ ou } .V.$
 .V.

- b) não .V. ou $3*2/3 > 15 - 35 \bmod 7$
 .F. ou $9/3 > 15 - 0$
 .F. ou $3 > 15$
 .F. ou .F.
 .F.
- c) não $(5 <> 10/2 \text{ ou } .V. \text{ e } 2 - 5 > 5 - 2 \text{ ou } .V.)$
não $(5 <> 5 \text{ ou } .V. \text{ e } -3 > 3 \text{ ou } .V.)$
não (.F. ou .V. e .F. ou .V.)
não (.F. ou .F. ou .V.)
não (.F. ou .V.)
não (.V.)
 .F.
- d) $2*4 <> 4 + 2 \text{ ou } 2 + 3 * 5/3 \bmod 5 <> 0$
 $16 <> 6 \text{ ou } 2 + 15/3 \bmod 5 <> 0$
 .V. ou $2 + 5 \bmod 5 <> 0$
 .V. ou $2 + 0 <> 0$
 .V. ou $2 <> 0$
 .V. ou .V.
 .V.

2.8.5. Expressões Literais

São aquelas formadas por operadores, variáveis ou constantes do tipo caractere que, depois de avaliadas, geram um resultado caractere. Veremos duas expressões básicas: concatenação (operador +) e a atribuição (já visto anteriormente).

Exemplo 9: expressões Literais

- a) "Transporte " + "aéreo" resulta em "Transporte aéreo"
 b) "Relo" + "joaria" resulta em "Relojoaria"
 c) $a \leftarrow$ "Expressão" (a variável a recebe o valor literal "Expressão")
 d) $b \leftarrow$ "Literal" (a variável b recebe o valor literal "Literal")
 e) $a + b$ resulta em "Expressão Literal"

Note que, nos exemplos acima, existe um espaço após a palavra *Transporte* e antes da palavra *Literal*, garantindo a separação das palavras. Se não houvesse esse espaço, as palavras estariam "grudadas": *Transporteaéreo* e *ExpressãoLiteral*.

Exercício 8: expressões Literais

- a) "Altamirando" + " de " + "Souza" resulta em _____
 b) "B" + "o" + "I" + "a" resulta em _____
 c) $a \leftarrow$ "Algoritmo "
 d) $b \leftarrow$ "Solução"
 e) $c \leftarrow$ "da "
 f) $a + c + b$ resulta em _____

2.9. Comandos Básicos

Já vimos como declarar as variáveis que serão utilizadas pelo algoritmo. Agora veremos como atribuir valores a essas variáveis, como fazer para fornecer dados ao algoritmo (entrada de

dados) e como o algoritmo nos passará o resultado por ele gerado (saída de dados). Com esses comandos básicos já teremos condição de montar os primeiros algoritmos utilizando o português estruturado.

2.9.1. Atribuição

O comando de atribuição é utilizado para definir o valor de uma variável, e para tal é utilizado o símbolo de atribuição " \leftarrow ", conforme a sintaxe mostrada na Figura 25. O significado do comando de atribuição está representado na Figura 26. Nesse comando, a expressão do lado direito é avaliada primeiro e o seu resultado é atribuído à variável do lado esquerdo. Essa expressão pode ser matemática, lógica, relacional, literal, etc., mas é importante que o resultado da expressão seja de tipo compatível com o tipo da variável (recore a discussão realizada em torno da Figura 22).

Seja a sequência de atribuições mostra ao lado. O valor 3 é inicialmente atribuído à variável x . No comando seguinte, é calculada a expressão $(x + 3)$, que resulta no valor 6, que então é atribuído a x . É importante notar que do lado esquerdo do símbolo de atribuição só pode estar presente um identificador de uma variável.

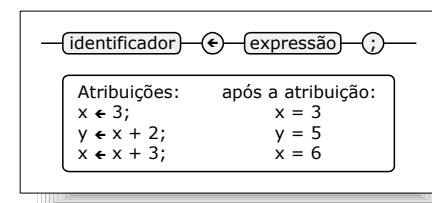


Figura 25: Sintaxe do comando de atribuição.

inteiro: x ;
 $x \leftarrow 3$;
 $x \leftarrow x + 3$;

No comando de atribuição, um valor é colocado no local da memória reservado para a variável.

$x \leftarrow 1000$;

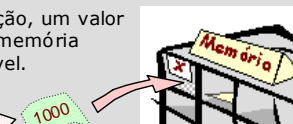


Figura 26: Significado do comando de atribuição.

Exemplo 10: atribuições.

- a) $A \leftarrow 3$;
 d) $TEM \leftarrow .F.$;
 g) $X1 \leftarrow \text{raiz}(\text{delta})$;
- b) $K \leftarrow K + 1$;
 e) $PI \leftarrow 3.1416$;
 h) $RESTO \leftarrow N \bmod 2$;
- c) $\Delta \leftarrow B^2 - 4*A*C$;
 f) $NOME \leftarrow$ "ABACATE";
 i) $MARCA \leftarrow .V.$;

Exercício 9: atribuições.

Coloque ao lado de cada comando de atribuição o valor assumido por cada variável no seguinte algoritmo:

Exemplo 12: algoritmo para cálculo das raízes distintas de uma equação de 2º grau.

Faça um algoritmo que calcule as seguintes expressões:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{e} \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Os valores de a , b e c devem ser lidos, e os resultados obtidos para x_1 e x_2 devem ser impressos.

Solução

O algoritmo em português estruturado é o seguinte²¹:

```
{ Esse algoritmo calcula o valor de x, dados os valores de a, b e c }
```

início

real: x1, x2, a, b, c; { declaração das variáveis }

leia (a, b, c); { entrada dos valores de a, b e c }

x1 ← (-b + raiz(b^2 - 4*a*c)) / (2*a); { calculo do valor de x1 }

x2 ← (-b - raiz(b^2 - 4*a*c)) / (2*a); { calculo do valor de x2 }

imprima ("x1 = ", x1, "x2 = ", x2); { saída do valor de x }

fim.

(A 1)

Convém observar que esse algoritmo só é válido para valores de a , b e c tais que a relação $\Delta \geq 0$ seja verdadeira, onde $\Delta = b^2 - 4ac$ (discriminante). Ou seja, os valores de a , b e c fornecidos devem atender essa condição previamente.

Exercício 11: representação gráfica de algoritmo.

Faça a representação gráfica (fluxograma e diagrama da Chapin) do algoritmo do [Exemplo 12](#).

Solução

Fluxograma:

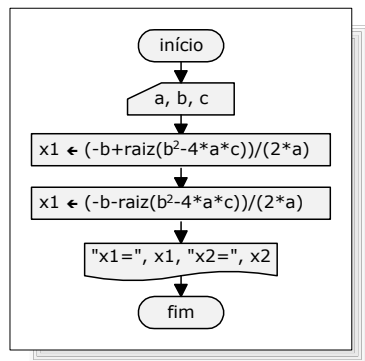
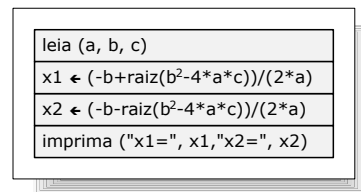


Diagrama de Chapin

**2.10. Documentando seu Algoritmo - Comentários**

Até agora vimos algoritmos simples, sem maior complexidade e de tamanho reduzido. Nessa situação, é fácil, através de uma simples leitura, entender o seu significado. Entretanto, à medida que os algoritmos crescem, com um maior número de variáveis e estruturas de controle de fluxo, é natural que o seu entendimento por uma pessoa que não o tenha desenvolvido (ou mesmo pela própria pessoa que o construiu, após algum tempo sem vê-lo) seja difícil, especialmente se soluções particulares tiverem sido utilizadas no algoritmo em questão.

²¹ Nesse algoritmo foram incluídos comentários. Sua utilização é explicada mais adiante.

Para minimizar tal tipo de problema é importante deixar indicações do que foi feito no próprio algoritmo, como o significado de algumas das variáveis e o porquê de algumas construções. Isso é realizado através da inserção de *comentários* ao longo do algoritmo, que são textos colocados entre chaves ("{" e "}") mas que não fazem parte do algoritmo propriamente dito: apenas explicam as linhas (ou conjunto de linhas) perto das quais foram colocados. Por exemplo, no algoritmo (A 2) existe um comentário inicial explicando o que faz o algoritmo, e outros comentários explicando o significado de algumas linhas.

```
{ Esse algoritmo calcula o valor de um polinômio de 3º. grau p3(x), dados os valores }  
{ de x, a0, a1, a2 e a3 }
```

início

real: p3, { valor do polinômio }

x, { valor de x para calculo do polinômio }

a0, a1, a2, a3; { coeficientes do polinômio }

(A 2)

leia (x, a0, a1, a2, a3); { entrada dos coeficientes do polinômio e do valor de x }

p3 ← a0 + a1*x + a2*x^2 + a3*x^3; { calcula o valor do polinômio em x }

imprima ("p3(" , x , ") = ", p3); { saída do valor de p3 (x) }

fim.

Como se trata apenas de um exemplo, alguns desses comentários poderiam ser deixados de lado se estivéssemos documentando um algoritmo na prática, como no caso do comando leia (x, a0, a1, a2, a3), cujo significado é óbvio. Isso deve ficar a critério do bom senso do construtor do algoritmo.

Exercício 12: algoritmo para calcular a resultante de duas forças.

Monte um algoritmo que calcule a resultante de duas forças, conforme mostrado na [Figura 30](#).

Os dados de entrada são os módulos das forças, F_1 e F_2 , e os ângulos a_1 e a_2 . O resultado é o módulo da força resultante F_R e seu ângulo a_R .

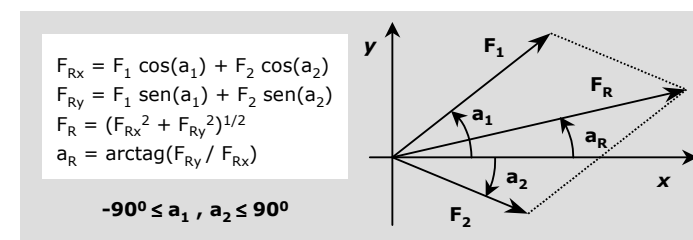


Figura 30:
Resultante entre duas forças.

Escreva o algoritmo usando o português estruturado e o fluxograma, e considere que os valores fornecidos para os ângulos a_1 e a_2 sempre atenderão a restrição indicada na figura (isto é, não é preciso testar se os valores estão corretos).

Solução*Português Estruturado*

{ Esse algoritmo calcula o valor da força resultante, dadas duas forças quaisquer. }

início

real: A1, A2, AR, F1, F2, FRX, FRX, FR;

leia (A1, A2, F1, F2);

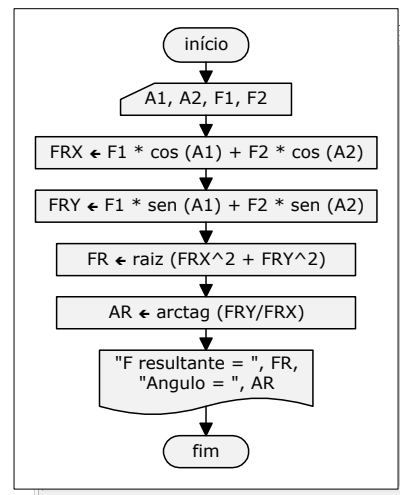
$FRX \leftarrow F1 * \cos(A1) + F2 * \cos(A2);$

$FRY \leftarrow F1 * \sin(A1) + F2 * \sin(A2);$

$FR \leftarrow \text{raiz}(FRX^2 + FRY^2);$

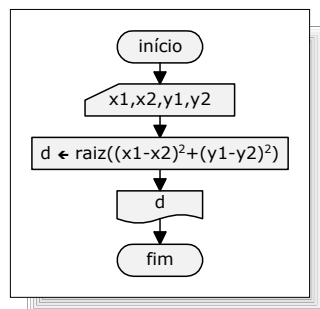
$AR \leftarrow \text{arctag}(FRY/FRX);$

imprima ("FR = ", FR, "AR = ", AR);

fim.*Fluxograma*

2.11. Do Português Estruturado para as Linguagens de Programação

Para se ter uma idéia de como é o processo de conversão do algoritmo para uma linguagem de programação, a seguir é mostrado um algoritmo descrito em fluxograma e passado para o português estruturado - [Figura 31](#). Na sequência, são apresentados exemplos de implementação desse algoritmo através de quatro linguagens de programação diferentes: Pascal, C, Basic e Fortran - [Figura 32](#) até [Figura 35](#).

Fluxograma**Português Estruturado**

{ Este programa calcula a distância entre dois pontos }
 { dadas as suas coordenadas (x1,y1) e (x2, y2). }

início

real x1, x2, y1, y2, d;

leia(x1,y1,x2,y2);

$d \leftarrow \text{raiz}((x1-x2)^2 + (y1-y2)^2);$

imprima("Distância = ", d);

fim.

[Figura 31](#): Algoritmo para cálculo da distância entre dois pontos.

C

```

/* Este programa calcula a distância entre dois pontos */

#include <stdio.h> /* para usar a função printf() */
#include <math.h> /* para usar a função pow() */

void main(){

/* Declaração de variáveis */
float x1, y1, x2, y2, d;

/* Entrada de dados */
clrscr(); /* limpa a tela (antigo Turbo C)*/
printf("Digite (x1,y1), separados por virgula: ");
scanf("%f,%f",&x1, &y1);
printf("\nDigite (x2,y2), separados por virgula: ");
scanf("%f,%f",&x2, &y2);

/* Calcula a distância */
d = pow(pow(x1-x2,2)+pow(y1-y2,2),.5);

/* Saída de dados */
printf("\nDistancia = %8.5f",d);

}
  
```

[Figura 32](#): Programa C para cálculo da distância entre dois pontos

BASIC

```

REM Este programa calcula a distancia entre dois pontos

REM Entrada de dados
input "Digite x1: "; x1
input "Digite y1: "; y1
input "Digite x2: "; x2
input "Digite y2: "; y2

REM Calculo da distancia
d= ((x1-x2)^2+(y1-y2)^2)^.5

REM Saida de dados
print "Distância = ";d
  
```

[Figura 33](#): Programa Basic²² para cálculo da distância entre dois pontos.

²² Liberty BASIC 2.02.

PASCAL

```
{ Este programa calcula a distância entre dois pontos. }
{ Como não existe o operador de potência no Pascal, isso }
{ é feito através da seguinte operação: }
{   x^y = exp(y*ln(x)) }

program Dist2Pontos;

{ Declaração de variáveis }
var
    x1, y1, x2, y2, dx, dy, d: double;

begin

    { Entrada de dados }
    write('Entre com a cordenadas (x,y) do ponto 1: ');
    readln(x1,y1);
    write('Entre com a cordenadas (x,y) do ponto 2: ');
    readln(x2,y2);

    { Cálculo da distância }
    dx := abs(x1-x2);
    dy := abs(y1-y2);
    d := sqrt(exp(2*ln(dx)) + exp(2*ln(dy)));
    { No Pascal, como no C, não existe o operador de potenciação. Para calcular }
    { a^n utiliza-se exp(n*ln(a)) }

    { Saída de dados }
    writeln;
    writeln('Distância = ',d:8:5);

end.
```

Figura 34: Programa Pascal para cálculo da distância entre dois pontos**FORTRAN**

```
C      Este programa calcula a distância entre dois pontos.
C
      program Dist2Pontos

C      Declaração de variáveis reais de precisão simples: real *4
      real *4 x1, x2, y1, y2, d

C      Leitura dos pontos
      write (*,*) 'Entre com a coordenada (x,y) do ponto 1: '
      read (*,*) x1,y1
      write (*,*) 'Entre com a coordenada (x,y) do ponto 2: '
      read (*,*) x2,y2

C      Cálculo da distância
      d = ((x1 - x2)**2 + (y1 - y2)**2)**.5

C      Saída de dados
C      O comando write utiliza o comando de formatação 2, onde:
C      a: indica um texto de qualquer tamanho
C      F8.5: indica um número real com 8 dígitos de tamanho
C            e 5 casas após a vírgula
      write (*,2) 'Distância = ',d
      format(a,F8.5)

      stop
      end
```

Figura 35: Programa Fortran para cálculo da distância entre dois pontos.**Referências do Capítulo**

- [1] Norton, P.: "Introdução à Informática", Makron Books do Brasil, 1997.
- [2] Velloso, F.C.: "Informática - Conceitos Básicos", Editora Campus Ltda, 1997.
- [3] Forbellone, A. L. V.; Eberspächer, H. F.: "*Lógica de Programação - A Construção de Algoritmos e Estruturas de Dados*", Makron Books do Brasil Editora Ltda, São Paulo, Brasil,1993.
- [4] Guimarães, A. M. e Lages, N. A. C.: "*Algoritmos e Estruturas de Dados*", Livros Técnicos e Científicos Editora S.A, Rio de Janeiro, Brasil, 1994.

3. Estruturas de Controle

3.1. Recordando

No Capítulo 1 foram abordados vários aspectos da solução de problemas, dentre eles a construção dos algoritmos da solução. Foram vistas, conceitualmente, as estruturas básicas utilizadas na construção dos algoritmos numa forma gráfica - o fluxograma.

No Capítulo 2 abordaram-se alguns aspectos associados à implementação dos algoritmos em computador. Foi visto como declarar e atribuir valores a variáveis, como utilizá-las em expressões de vários tipos (aritméticas, lógicas, relacionais) e como realizar a entrada e saída de dados em um algoritmo. Iniciou-se o estudo do português estruturado.

Neste capítulo serão abordadas novamente as estruturas de controle básicas vistas no Capítulo 1, porém com mais detalhes e utilizando a representação textual do português estruturado. Isso permitirá a construção de algoritmos voltados para a implementação de programas de computador que possam resolver problemas com os mais variados níveis de complexidade. Através dessas estruturas - sequência, alternativa/seleção e repetição - e a combinação delas, exemplificado de forma gráfica na [Figura 1](#), é possível construir os diversos algoritmos.

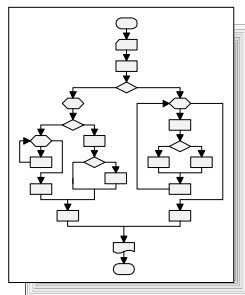


Figura 1: A combinação das estruturas básicas de controle permite descrever os diversos níveis de complexidade dos algoritmos.

3.2. Estrutura Sequencial e Blocos de Comandos

Uma **estrutura sequencial** é um conjunto de ações executadas numa sequência linear, de cima para baixo (ou da esquerda para a direita), na mesma ordem em que foram escritas. A sintaxe da estrutura sequencial no português estruturado está representada na [Figura 2](#), e na [Figura 3](#) são mostradas também as representações gráficas correspondentes.

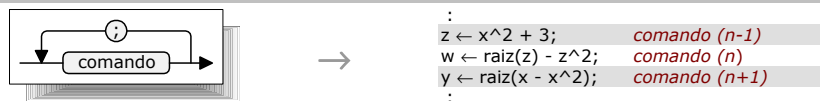
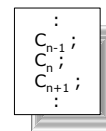


Figura 2: Sintaxe da estrutura sequencial no português estruturado

Notar que no português estruturado um comando termina apenas no ponto e vírgula, mesmo que, por vezes, ele se estenda por algumas linhas, como veremos adiante para algumas ocorrências das estruturas de seleção e repetição. Isso é verdade também para algumas linguagens de programação, como o Pascal.

Português Estruturado



Fluxograma

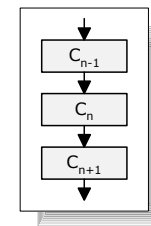


Diagrama de Chapin

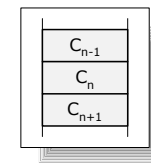


Figura 3: Representação textual e gráfica da estrutura sequencial.

Um **bloco de comando** pode ser considerado como um trecho de estrutura sequencial que aparece ao longo do algoritmo e que agrupa um certo número de comandos que deverão ser executados em conjunto, ou "em bloco". Pode ser constituído por apenas um comando (caso mais simples) ou por vários comandos, conforme definido na sintaxe mostrada na [Figura 4](#). O uso do bloco de comando ficará evidente nas próximas estruturas a serem estudadas.

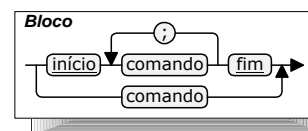


Figura 4: Sintaxe de um bloco de comandos.

Observação: Um bloco de comandos pode conter apenas um comando. Nesse caso, as palavras *início* e *fim* não são necessárias: o comando sozinho indica que se trata de um bloco com comando único. Isso deve ser atentamente observado nas estruturas que veremos adiante.

3.3. Estrutura de Seleção

3.3.1. Alternativas Simples e Composta

Esse tipo de estrutura corresponde a alternativa já vista no Capítulo 1. A sintaxe da alternativa simples/composta no português estruturado está representada na [Figura 5](#). Também são mostradas as representações gráficas correspondentes na [Figura 6](#). Note-se a diferença entre as duas alternativas: a alternativa simples prevê ações apenas quando a condição é verdadeira, nada sendo feito se esta for falsa - não há a cláusula *senão*.

Na [Figura 6](#) e no **Error! Reference source not found.**, é possível perceber o significado do bloco de comandos definido no item anterior. Esse conceito é muito importante: o aluno deve se certificar que entendeu a definição e o uso de blocos de comandos.

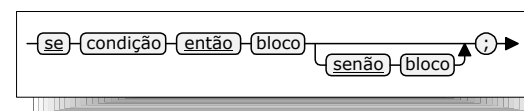


Figura 5: Sintaxe da alternativa simples/composta no português estruturado.

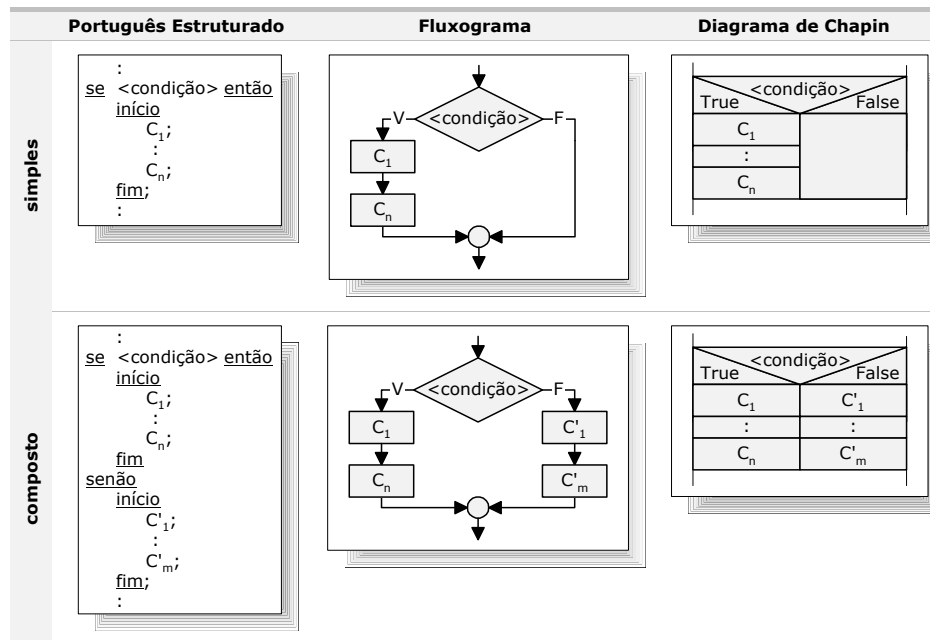


Figura 6: Representação textual e gráfica das alternativas simples e composta²³.

Exemplo 13: alternativa.

Nesse exemplo temos uma alternativa composta com o bloco referente à cláusula então - bloco verdade - com vários comandos delimitados entre as palavras início e fim. Na cláusula senão - bloco falsidade - não foi necessário o uso das palavras início e fim, pois tem-se apenas um comando (e não um conjunto) sendo executado se a condição for falsa (veja o item 3.2).

Exercício 13: alternativa.

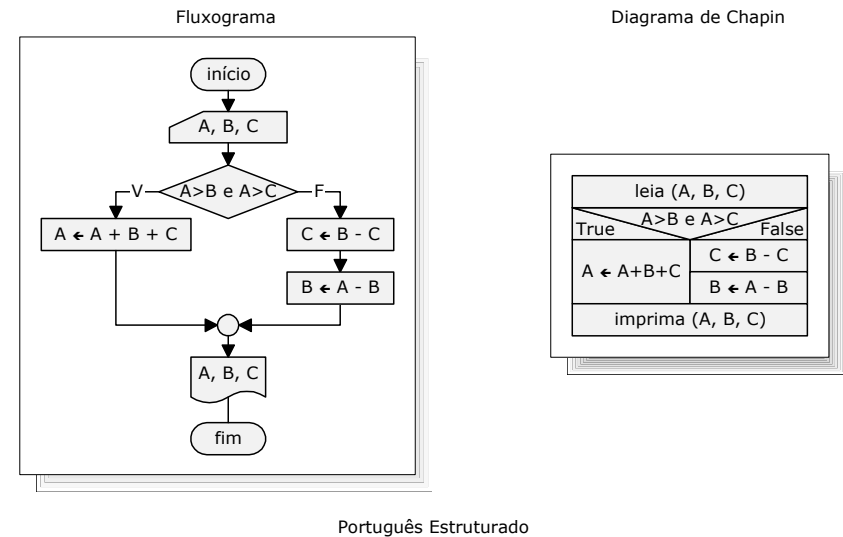
Determinado problema é resolvido através da modificação dos valores iniciais de 3 variáveis, A, B e C. Caso A seja maior que os outros valores, B e C, o seu valor é modificado para o resultado da expressão $A + B + C$, e B e C permanecem inalterados. Caso contrário, B passa a ter o valor da expressão $A - B$ (A e B originais) e C passa a ter o valor da expressão $B - C$ (B e C originais - atenção!), e A permanece inalterado. Os resultados finais são os valores de A, B e C, alterados ou não, conforme essas regras. Monte o algoritmo que descreve a solução desse problema usando as três representações: fluxograma, diagrama de Chapin e português estruturado.

```

:
se (x^2 - 1) > 0 então
  início
    q ← p + (n + t)/2;
    r ← p - (n + t)/3;
  fim
senão
  z ← raiz (p + t);
:

```

Solução



Português Estruturado

```

início
  inteiro: A, B, C;
  leia (A, B, C);
  se (A > B e A > C) então
    A ← A + B + C
  senão
    início
      C ← B - C;
      B ← A - B;
    fim;
  imprima (A, B, C);
fim.

```

Na Figura 7 é possível visualizar a implementação da alternativa composta em várias linguagens de programação.

²³ O diagrama de Chapin foi desenhado no VISIO 4.5, onde a língua é o inglês. True significa Verdadeiro (.V.), e False é Falso (.F.).

PORT. ESTRUTURADO	C	BASIC
<pre> se (a > b) então início a ← a + 2; b ← b - 1; fim senão a ← a + 3; </pre>	<pre> if (a > b) { a = a + 2; b = b - 1; } else a = a + 3; </pre>	<pre> if (a > b) then a = a + 2 b = b - 1 else a = a + 3 end if </pre>
PASCAL	FORTRAN	
<pre> if (a > b) then begin a := a + 2; b := b - 1; end else a := a + 3; </pre>	<pre> if (a.gt.b) then a = a + 2 b = b - 1 else a = a + 3 endif </pre>	

Figura 7: Alternativa composta implementada em várias linguagens de programação.

Fazer Exercício Avulso 1.

3.3.2. Seleção Múltipla

Na seleção múltipla, o número de alternativas não é limitado a duas, como no caso das alternativas simples e composta. Ao contrário dessas, na seleção múltipla a decisão para determinar a alternativa a ser escolhida não se restringe à avaliação apenas de uma expressão lógica: pode ser qualquer outra expressão que forneça como resultado um valor dentro dos tipos válidos no português estruturado.

O valor obtido no cálculo da expressão é então comparado com determinados conjuntos de valores, cada um desses conjuntos associados a uma alternativa que conduz a execução de um bloco de comandos. Se o valor calculado na expressão coincidir com algum valor dentro do conjunto de valores associados a uma alternativa, então apenas o bloco correspondente a essa alternativa é executado. O comando para a seleção múltipla no português estruturado é o comando **escolha**, cuja sintaxe está definida na Figura 8, e com as representações textual e gráfica correspondentes mostradas na Figura 9.

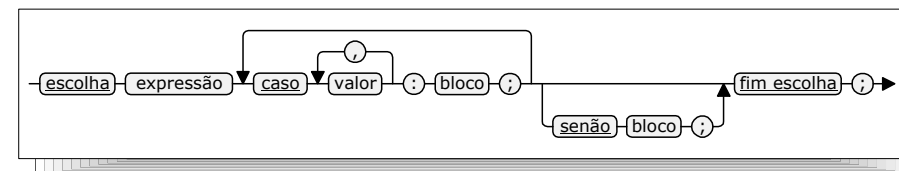


Figura 8: Sintaxe da seleção múltipla no português estruturado

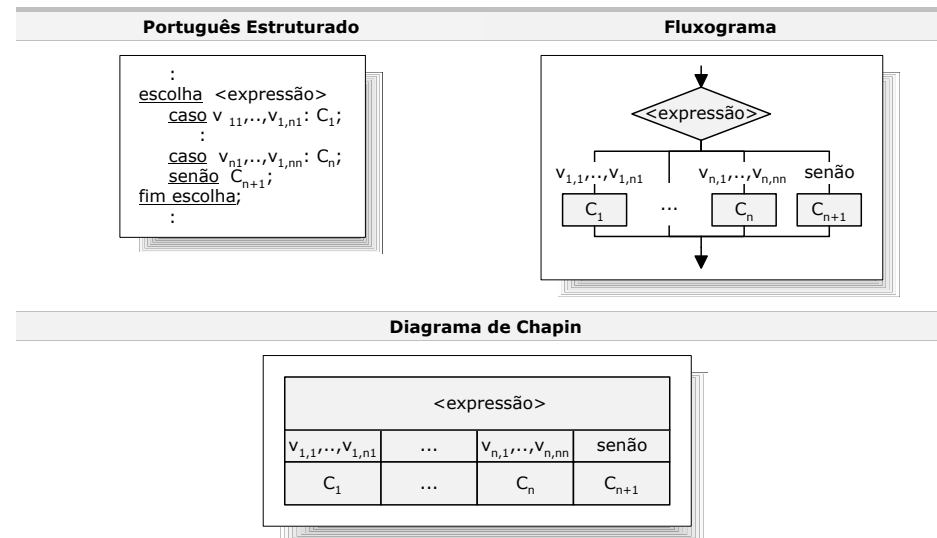


Figura 9: Representação textual e gráfica da seleção múltipla - comando escolha.

Nota: C_i (i=1,...,n,n+1) representa um bloco de comandos.

Notar a existência da cláusula **senão**, que indica o que será feito se o resultado da expressão não coincidir com nenhum valor listado nos diversos casos. Se essa cláusula não existir, ou não for utilizada (em branco), nada será feito pelo comando escolha.



Representação de Faixa de Valores

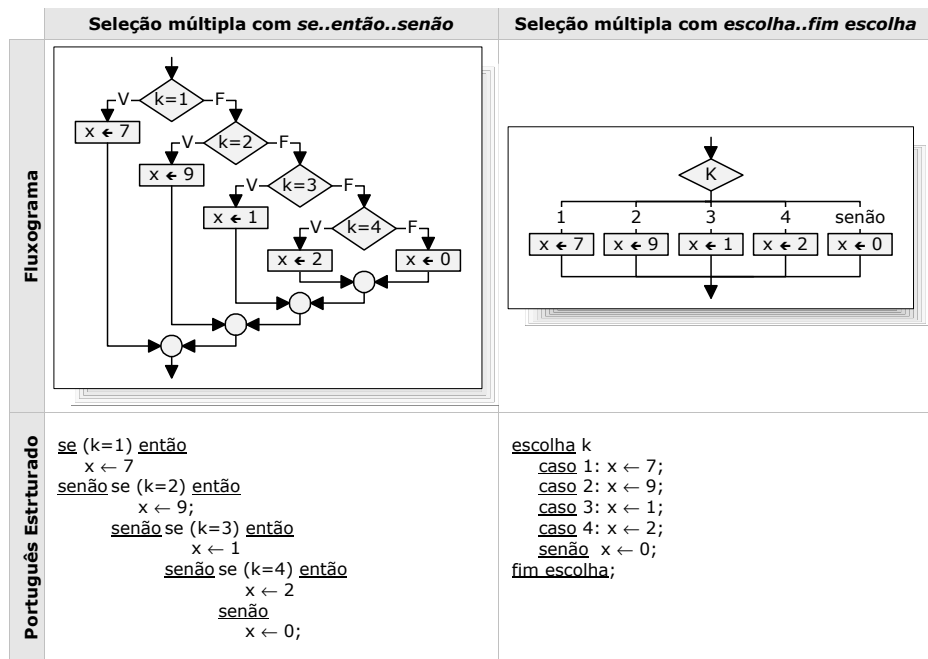
Vamos utilizar a seguinte notação, válida para números inteiros:

n..m representará *n*, (*n*+1), ... , (*m*-1), *m*

Exemplo: 1..5 representa o conjunto 1, 2, 3, 4, 5.

Exemplo 14: seleção múltipla 1.

Exemplo da seleção múltipla implementada tanto pelo comando *escolha* como através de alternativas compostas.



Exemplo 15: seleção múltipla 2.

Na seleção múltipla ao lado, se o valor resultante da expressão $(k-1)$ for igual a 1, 2, 3 ou 4, um dos blocos de comandos associado a um dos casos será executado. Caso contrário, isto é, caso $(k-1)$ seja diferente de 1, 2, 3 ou 4, o bloco associado à cláusula *senão* é que será executado.

```

:
escolha (k - 1)
  caso 1, 2: q ← p + n / 2;
  caso 3, 4:
    início
      q ← p + n / 2;
      t ← q^2;
    fim;
  senão z ← p + t;
fim escolha;
:
          
```

Na [Figura 10](#) são mostradas as implementações da seleção múltipla em várias linguagens de programação.

PORT. ESTRUTURADO	FORTRAN ²⁴	BASIC ²⁵
<pre> escolha n caso 1, 2: imprima("n=1 ou 2"); caso 3..10: imprima("3 <= n <= 10"); caso >=11: imprima("n >= 11"); senão imprima("n < 1"); fim escolha; </pre>	<pre> select case (n) case (1,2) write (*,*) 'n=1 ou 2' case (3:10) write (*,*) '3 <= n <= 10' case (11:) write (*,*) 'n >= 11' case default write (*,*) 'n < 1' end select </pre>	<pre> select case n case 1,2 print "n=1 ou 2" case 3 to 10 print "3 <= n <= 10" case is >= 11 print "n >= 11" case else print "n < 1" end select </pre>
PASCAL	C	
<pre> if (n < 1) then writeln('n=1 ou 2') else if (n>=11) then writeln('n=1 ou 2') else case n of 1,2: writeln('n=1 ou 2'); 3..10: writeln('3 <= n <= 10'); end; </pre>	<pre> if (n < 1) printf("\nm < 1"); else switch (n) { case 1: case 2: printf("\nn=1 ou 2"); break; case 3: case 4: case 5: case 6: case 7: case 8: case 9: case 10: printf("\n3 <= n <= 10"); break; default: printf("\nn1 <= n"); } </pre>	

Figura 10: Alternativa composta implementada em várias linguagens de programação. Notar que nas linguagens C e Pascal não é possível o uso de comparações nos casos. Na linguagem Pascal padrão não existe cláusula correspondente ao *senão* da seleção múltipla do português estruturado.

Exercício 14: seleção múltipla.

Construa uma seleção que atribua valores a uma variável caractere denominada *COR* conforme o valor da variável inteira *K*. Essa atribuição deve ser feita de acordo com a tabela ao lado.

Faça o trecho de algoritmo correspondente em português estruturado, em fluxograma e diagrama de Chapin.

K	COR
1, 2, 3 ou 4	Verde
5 ou 8	Roxo
7, 11 ou 15	Azul
outros inteiros	Branco

Solução

Português Estruturado

```

:
escolha K
  caso 1..4: COR ← "Verde";
  caso 5, 8: COR ← "Roxo";
  caso 7, 11, 15: COR ← "Azul";
  senão COR ← "Branco";
fim escolha;
:
          
```

²⁴ Implementações mais antigas dessa linguagem não possuem esta estrutura de seleção múltipla.

²⁵ Rapid-Q Basic.

Fluxograma

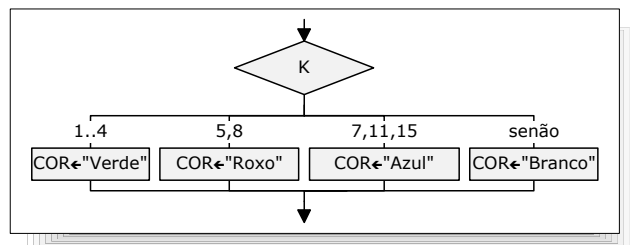
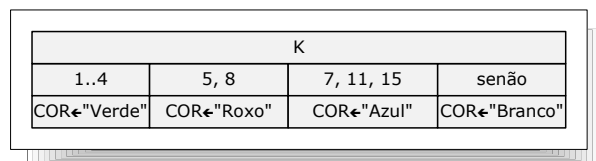


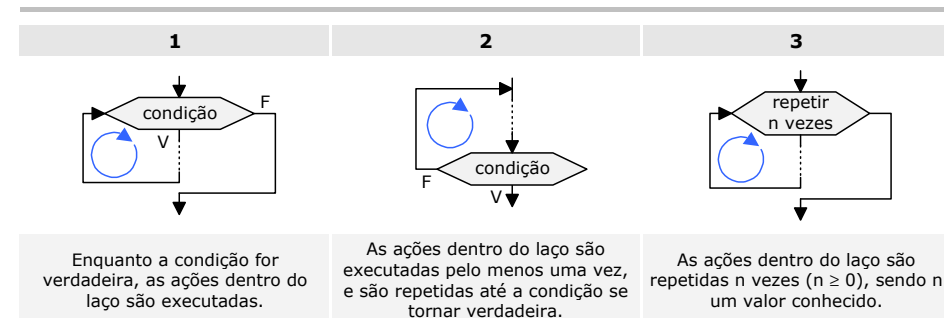
Diagrama da Chapin



Fazer Exercício Avulso 2.

3.4. Estrutura de Repetição

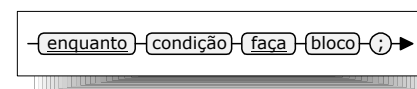
Foi visto no Capítulo 1 três tipos de estruturas de repetição utilizadas em fluxogramas, reproduzidos novamente na [Figura 11](#). No português estruturado é possível representar esses mesmos tipos. É o que será visto nos próximos itens.



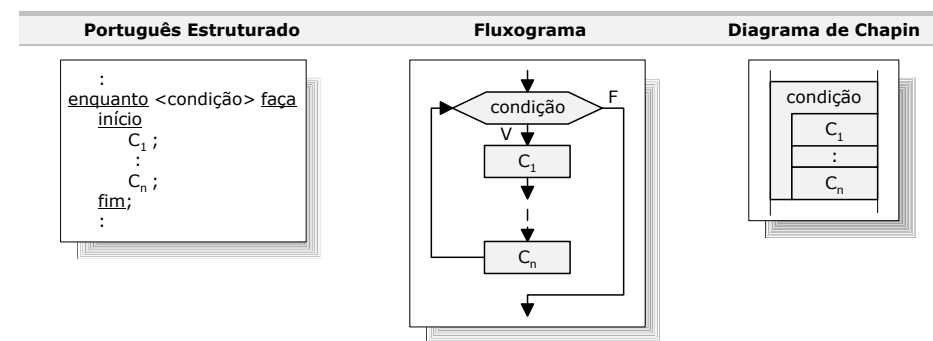
[Figura 11](#): Estruturas de repetição utilizadas em fluxogramas.

3.4.1. Repetição Controlada por Condição no Início

Essa estrutura de repetição, com a condição no início, corresponde à repetição do tipo 1 na [Figura 11](#). A sintaxe dessa estrutura no português estruturado é apresentada na [Figura 12](#), e na [Figura 13](#) são mostradas as representações textual e gráfica correspondentes.



[Figura 12](#): Sintaxe da estrutura de repetição controlada por condição no início no português estruturado



[Figura 13](#): Representação textual e gráfica da estrutura de repetição controlada por condição no início.

Exemplo 16: repetição controlada por condição no início

A repetição nesse exemplo calcula o seguinte somatório:

$$S = 1 + 2 + 3 + \dots + N$$

Notar que o teste é feito no início da repetição e que a variável utilizada no teste (K) deve ser conhecida antes do teste ser realizado: ela deve ser previamente inicializada (isso é feito no comando " $K \leftarrow 1$;").

```

:
leia (N);
S ← 0;
K ← 1; {inicialização de K}
enquanto ( K ≤ N ) faça
  início
    S ← S + K;
    K ← K + 1;
  fim;
:

```

Note a semelhança conceitual entre essa repetição e aquela do exemplo do capítulo 1, onde um tanque era cheio utilizando-se um balde. Imagine aqui o mesmo tipo de problema: desejasse encher um tanque, mas dispõe-se, para cada viagem à torneira, de um balde com capacidade diferente: na primeira viagem, um balde de 1 litro, na segunda um balde de 2 litros, e assim por diante, até a viagem N, quando a capacidade do balde é de N litros. Nessa linha de raciocínio, o valor de S obtido nesse somatório seria a quantidade de água colocada no tanque: S acumula os termos (ou "enche") ao longo da repetição.

A Figura 14 contém exemplos de implementações da repetição controlado no início em várias linguagens.

PORT. ESTRUTURADO	FORTRAN 77	FORTRAN 90
<pre> S ← 0; K ← 1; enquanto (K ≤ N) faça início S ← S + K; K ← K + 1; fim; </pre>	<pre> S = 0 k = 1 if (k.gt.n) goto 4 S = S + k k = k + 1 goto 3 continue </pre>	<pre> S = 0 k = 1 do while(k.leq.n) S = S + k k = k + 1 end do </pre>
BASIC	PASCAL	C
<pre> s = 0 k = 1 while (k<=n) s = s + k k = k + 1 wend </pre>	<pre> S := 0; k := 1; while (k<=n) do begin S := S + k; k := k + 1; end; </pre>	<pre> S = 0; k = 1; while (k<=n) { S = S + k; k = k + 1; } </pre>

Figura 14: Repetição controlada no início implementada em várias linguagens.

**Somatórios**

Um somatório onde cada termo é função apenas de sua posição pode ser representado da seguinte forma:

$$S = \sum_{k=1}^n f(k) \quad (= f(1) + f(2) + \dots + f(k) + \dots + f(n))$$

Se $S = 1 + 2 + \dots + N$, então $f(k) = k$ e $S = \sum_{k=1}^n k$ - visto no exemplo anterior

Para que o algoritmo possa calcular o somatório desse tipo, é preciso determinar o termo geral, $f(k)$. Note também que o valor de k sempre varia de um em um nesse tipo de representação de somatório.

Exercício 15: repetição controlada por condição no início - português estruturado

Escreva o trecho de algoritmo, em português estruturado, para fazer o somatório dos N primeiros termos ($N \geq 0$) do seguinte somatório:

$$S = 1 + 1/3 + 1/5 + 1/7 + \dots + \text{<termo N>}$$

Faça conforme sugerido na observação acima e no Exemplo 16: determine primeiro o termo geral $f(k)$.

Solução

```

:
leia (N);
S ← 0;
K ← 1;
enquanto (K ≤ N) faça
  início
    S ← S + 1/(2*K - 1);
    K ← K + 1;
  fim;
:

```

```

:
leia (N);
S ← 0;
K ← 1;
enquanto (K ≤ 2*N - 1) faça
  início
    S ← S + 1/K;
    K ← K + 2;
  fim;
:

```

Termo	K	1/(2*K-1)
1	1	1
2	2	1/3
3	3	1/5
:	:	:
N	N	1/(2*N-1)

Termo	K	1/K
1	1	1
2	3	1/3
3	5	1/5
:	:	:
N	2*N-1	1/(2*N-1)

Exercício 16: repetição controlada por condição no início - diagramas

Faça os diagramas (fluxograma e Chapin) correspondentes ao Exercício 15.

Solução

Fluxograma

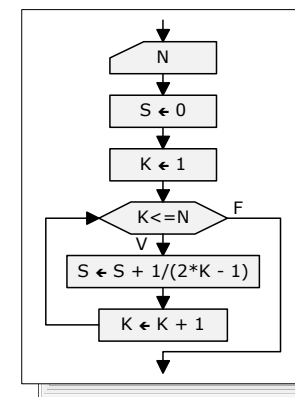
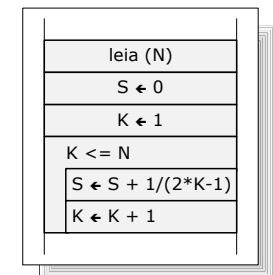


Diagrama de Chapin



3.4.2. Repetição Controlada por Condição no Final

Nesse tipo de estrutura, a condição que controla a repetição é colocada após o bloco de comando, correspondendo à repetição do tipo 2 na Figura 11. Isso faz com que ao menos uma iteração seja realizada antes de ser feito o primeiro teste da condição. A sintaxe no português estruturado é mostrada na [Figura 15](#), e na [Figura 16](#) são mostradas também as representações textual e gráficas correspondentes.

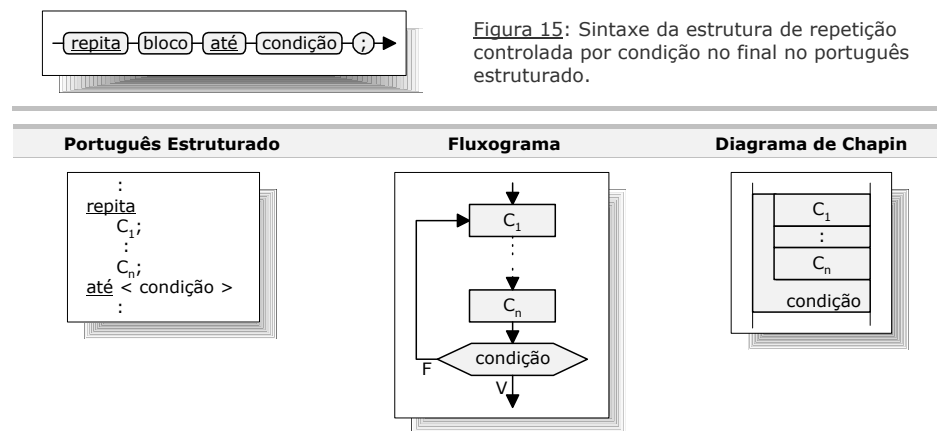
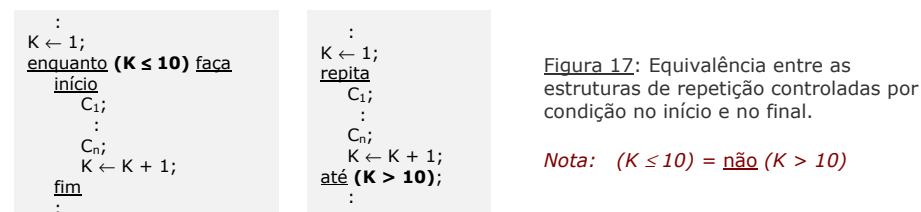


Figura 16: Representação textual e gráfica da estrutura de repetição controlada por condição no final.

Deve ser notado que essa repetição é executada até que a condição se torne verdadeira, ao contrário da repetição com teste no início (*enquanto ... faça ...*). É possível utilizar esses dois tipos de estruturas para construir repetições equivalentes, sendo que a condição de teste em uma delas é a negação da condição que controla a outra, como exemplificado na [Figura 17](#).



Exemplo 17: repetição controlada por condição no final

A repetição neste exemplo calcula o seguinte somatório:

$$S = 1 + 2 + 3 + \dots + N, N > 0$$

Notar que o teste é feito no final, indicando que o somatório pára de ser executado no momento em que o valor de K supera o valor de N. Aqui também é necessário que a variável utilizada no teste (K) seja conhecida antes de se iniciar a repetição: ela deve ser previamente inicializada (isto é feito no comando $K \leftarrow 1$;).

```

:
leia (N);
S ← 0;
K ← 1;
repita
  S ← S + K;
  K ← K + 1;
até ( K > N );
:

```



Sobre a Repetição Controlada no Final

No [Exemplo 17](#), se $N = 0$ a repetição irá fornecer como resultado $S = 1$, que não é o resultado correto! Utilizando a repetição controlada no início o resultado será coerente: $S = 0$ para $N = 0$. Ou seja, dependendo do problema, um ou outro tipo de construção é o mais adequado para realizar determinada repetição.

A [Figura 18](#) contém exemplos de implementações da repetição controlada no final em várias linguagens.

PORT. ESTRUTURADO	FORTRAN	
<pre>S ← 0; K ← 1; <u>repita</u> S ← S + K; K ← K + 1; <u>até</u> (K>N);</pre>	3	<pre>S = 0 k = 1 S = S + k k = k + 1 if (k.leq.n) goto 3</pre>
BASIC	PASCAL	C
<pre>s = 0 k = 1 Do s = s + k k = k + 1 Loop while k<=n</pre>	<pre>S:=0; k:=1; repeat S := S + k; k := k + 1; until (k>n);</pre>	<pre>S=0; k=1; do { S = S + k; k = k + 1; } while (k<=n)</pre>

Figura 18: Repetição controlada no final implementada em várias linguagens.

Exercício 17: repetição controlada por condição no final - português estruturado

Escreva o trecho de algoritmo, em português estruturado, para fazer o somatório dos N primeiros termos do seguinte somatório:

$$S = 1 + 16 + 49 + 100 + \dots$$

Supor que $N > 0$ sempre.

Solução

```

:
leia (N);
S ← 0;
K ← 1;
repita
  S ← S + (3*K - 2)^2;
  K ← K + 1;
até (K > 3*N-2);
:

```

```

:
leia (N);
S ← 0;
K ← 1;
repita
  S ← S + K^2;
  K ← K + 3;
até (K > 3*N-2);
:

```

Termo	K	$(3 \cdot K - 2)^2$	Termo	K	K^2
1	1	1	1	1	1
2	2	16	2	4	16
3	3	49	3	7	49
:	:	:	:	:	:
N	N	$(3 \cdot N - 2)^2$	N	$3 \cdot N - 2$	$(3 \cdot N - 2)^2$

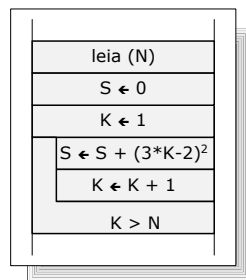
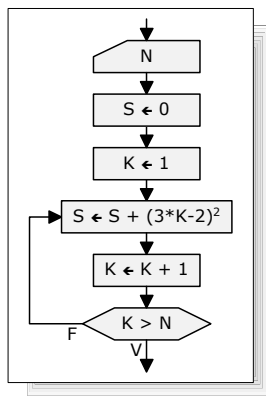
Exercício 18: repetição controlada por condição no final - diagramas

Faça os diagramas (fluxograma e Chapin) correspondentes ao Exercício 17.

Solução

Fluxograma

Diagrama de Chapin



Fazer Exercício Avulso 3.

3.4.3. Repetição Com Variável de Controle

A repetição com variável de controle é útil quando o número de iterações a serem realizadas é conhecido ou pré-fixado, não sendo necessário a utilização de um teste para verificar o momento em que a repetição deve terminar. Corresponde à repetição do tipo 3 na Figura 11. Uma variável de controle (v) varia entre dois extremos (i e l) num passo p , da seguinte forma: $v = i, i + p, i + 2p, \dots, i + np$, onde $i + np$ é o último valor para o qual se verifica $v = i + np \leq l$. Conhecendo-se os extremos de variação da variável de controle, e também o passo de variação, o número de repetições fica determinado.

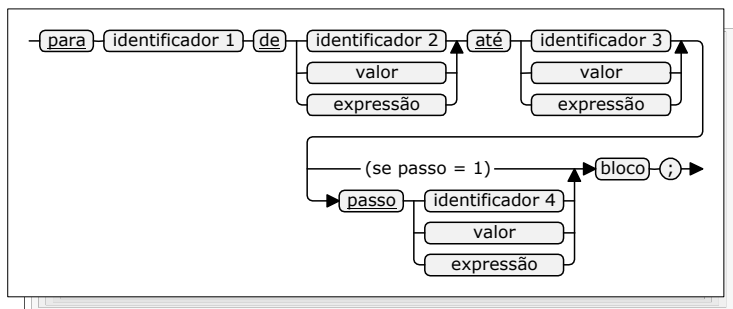


Figura 19: Sintaxe da estrutura de repetição controlada por variável de controle no português estruturado.

A sintaxe no português estruturado é mostrado na figura Figura 19, e na Figura 20 são mostradas também as representações textual e gráficas correspondentes. Notar que, na sintaxe desse comando, quando o passo for igual a um (1) não é necessário explicitar o valor do passo: ele pode ser omitido.

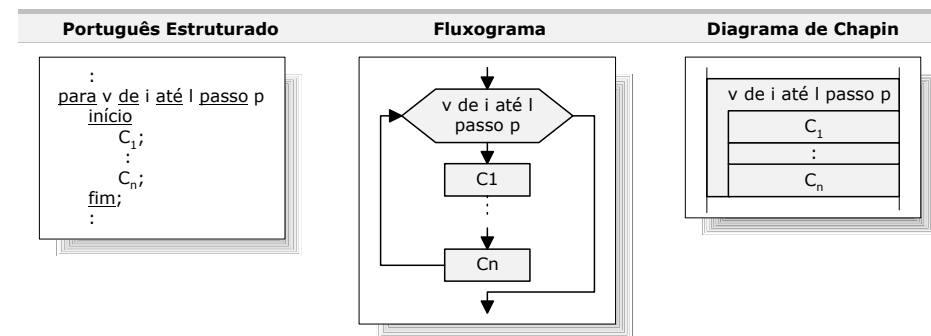


Figura 20: Representação textual e gráfica da estrutura de repetição controlada por variável de controle.

Sobre a Variável de Controle

As variáveis de controle numa repetição desse tipo nunca devem ser modificadas pelos comandos executados dentro da repetição. Caso isso aconteça, o comportamento da repetição pode ser alterado de forma inesperada e imprevisível, não atingindo o objetivo desejado.

Na Figura 21 é possível comparar três estruturas realizando a mesma repetição. Obviamente, existem situações onde o uso de determinado tipo de construção é melhor ou mais conveniente do que outro. Ao longo do curso isso ficará mais evidente.

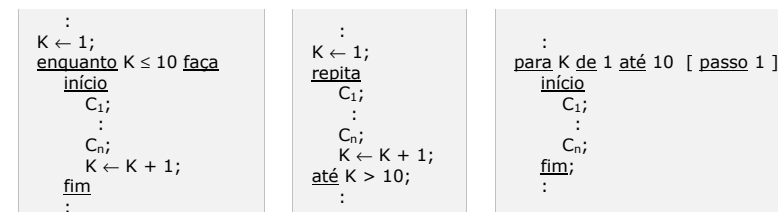


Figura 21: Equivalência entre as estruturas de repetição controladas por condição no início, no final e por variável de controle.

Observação: o texto entre colchetes ("[" e "]") é opcional: pode ou não ser colocado.

Exemplo 18: repetição controlada por variável de controle

A repetição neste exemplo calcula o seguinte somatório:

$$S = 1 + 2 + 3 + \dots + N$$

Notar que o teste é feito implicitamente: o somatório pára de ser executado no momento em que o valor de K supera o valor de N . A inicialização da variável de controle K é feita no próprio comando *para*.

```

:
leia (N);
S ← 0;
para K de 1 até N [passo 1]
  S ← S + K;
:

```

A [Figura 22](#) contém exemplos de implementações da repetição com variável de controle em várias linguagens.

PORT. ESTRUTURADO	FORTRAN 77	FORTRAN 90
<pre>S ← 0; para K de 1 até N S ← S + K;</pre>	<pre>S = 0 do 5 k=1,n,1 S = S + k continue 5</pre>	<pre>S = 0 do k=1,n,1 S = S + k enddo</pre>
BASIC	PASCAL	C
<pre>s = 0 for k = 1 to n s = s + k next k</pre>	<pre>S:=0; for k:=1 to n do S := S + k;</pre>	<pre>S=0; for(k=1;k<=n;k++) S = S + k;</pre>

Figura 22: Repetição com variável de controle implementada em várias linguagens.

Exercício 19: repetição controlada por variável de controle - português estruturado.

Escreva o trecho de algoritmo, em português estruturado, para fazer o somatório dos N primeiros termos da seguinte expressão:

$$S = 1 + 5 + 9 + 13 + 17 + \dots$$

Solução

<pre>: leia (N); S ← 0; para K de 1 até N S ← S + (4*K - 3); :</pre>	<pre>: leia (N); S ← 0; para K de 1 até 4*N - 3 passo 4 S ← S + K; :</pre>
--	--

Exercício 20: repetição controlada por variável de controle - diagramas.

Faça os diagramas (fluxograma e Chapin) correspondentes ao Exercício 19.

Solução

Fluxograma

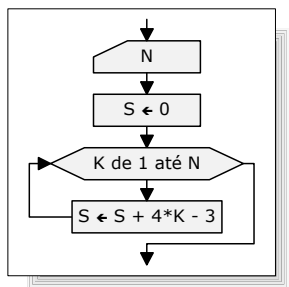
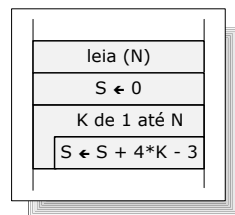


Diagrama de Chapin

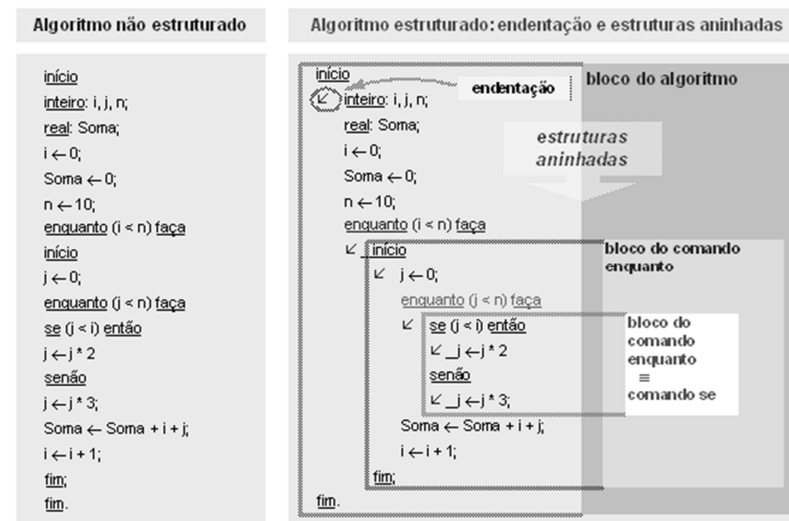


Fazer Exercício Avulso 4.

3.5. Aninhamento e Endentação

Deve-se ressaltar que algoritmos expressos através do português estruturado não devem ser escritos de forma aleatória, mas sim de forma organizada e legível, facilitando ao máximo o seu

entendimento. Além do uso de comentários, citados no Capítulo 2, existe uma forma de estruturação do algoritmo através do aninhamento correto de blocos e do uso da endentação.



igura 23: Comparação entre um algoritmo não estruturado e outro estruturado.

Para entendermos esses conceitos, compare os dois algoritmos mostrados na [igura 23](#), escritos utilizando o português estruturado. Não é difícil perceber que o algoritmo da direita permite uma leitura mais clara e rápida, facilitando a compreensão.

Aninhar significa colocar um bloco de comandos dentro de outro de maneira adequada, com o intuito de construir uma estrutura mais complexa. Na [igura 23](#) temos um comando *se* (alternativa) contido dentro de um comando *enquanto* (repetição), que por sua vez está contido dentro de um bloco de comandos dentro de outro comando *enquanto*. Deve ser observado que um bloco de comandos deve sempre iniciar e terminar dentro do bloco mais externo ao qual pertence. Na [Figura 24](#) é possível visualizar o significado do aninhamento realizado da forma correta.

A endentação consiste em realizar deslocamentos para a direita do texto de modo a ressaltar a estruturação do algoritmo, como o aninhamento de blocos. Tanto na [igura 23](#) como na [Figura 24](#) é possível perceber as estruturas aninhadas corretamente devido ao uso da endentação, que deixa visível cada um dos blocos.

É possível notar que mesmo um algoritmo pequeno pode se tornar de difícil compreensão se a sua estruturação não for feita de forma correta, ou se nenhuma endentação for realizada. Isso vai ficando cada vez mais crítico a medida que o algoritmo cresce em tamanho e complexidade. Portanto, deve se tornar um costume o uso da endentação e o correto aninhamento dos blocos de comandos.

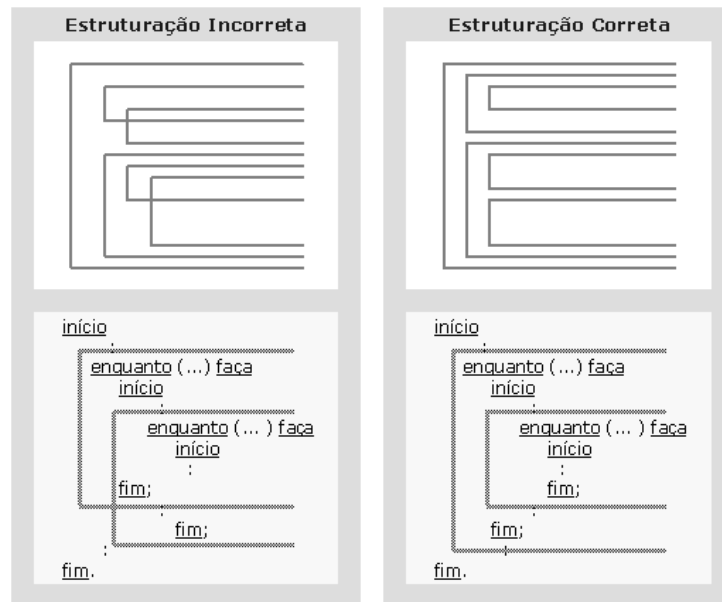


Figura 24: Aninhamento de blocos e comandos.

3.6. Comandos de Desvio de Fluxo

3.6.1. Comando Abandone

O comando *abandone* é geralmente utilizado dentro de repetições (*enquanto*, *repita*, *para*), e funciona da seguinte maneira: quando ele é encontrado, o fluxo do algoritmo passa para o primeiro comando logo após a repetição. Além disso, ele aparece sempre associado a uma alternativa (comando *se*), o que introduz um segundo teste para saída da repetição. Deve ser usado com critério e organização para não deixar o algoritmo difícil de compreender e manter.

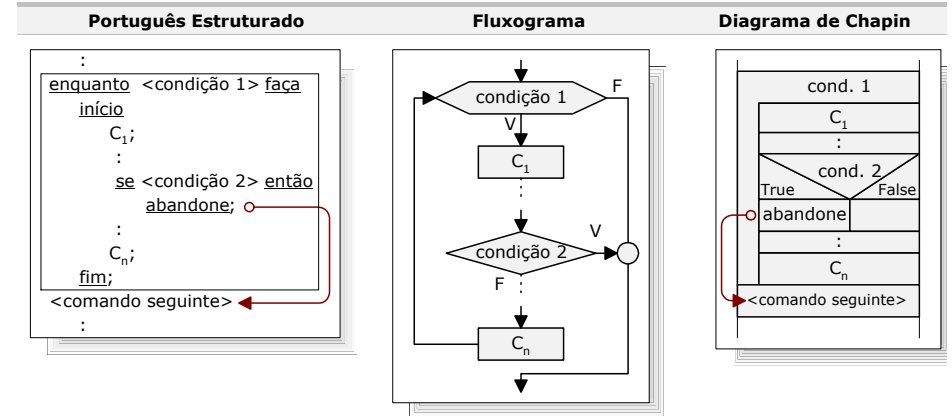
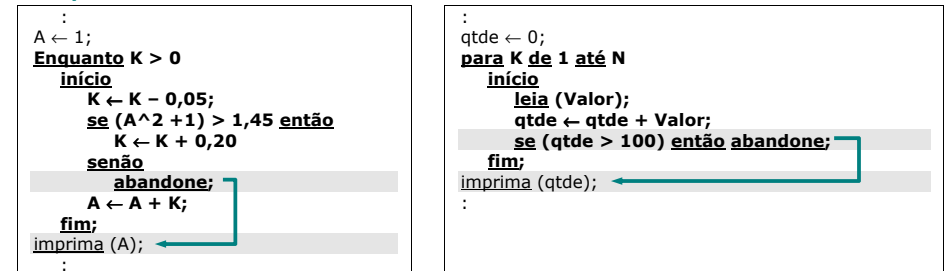


Figura 25: Representação textual e gráfica do comando abandone.

Exemplo 19: comando abandone.



Exercício 21: algoritmo utilizando o comando abandone.

Escreva um algoritmo que leia as idades de um conjunto de 100 pessoas utilizando o comando de repetição *para*. Entre as idades lidas, o algoritmo deve checar se existem pelo menos 3 pessoas com mais de 70 anos. Se isso acontecer, a pesquisa deve ser interrompida e uma mensagem deve ser emitida informando que a condição especificada foi atendida. Se isso não acontecer, o algoritmo deve imprimir o número de pessoas maiores de 70 anos encontradas no conjunto de 100 pessoas (que será menor ou igual a 3).

Solução

```

início
    inteiro: K, Idade, Maior70;
    Maior70 ← 0;
    para K de 1 até 100
        início
            leia (Idade);
            se (Idade > 70) então Maior70 ← Maior70 + 1;
            se (Maior70 = 3) então abandone;
        fim;
    se (Maior70 = 3) então
        imprima ("Foram encontradas 3 pessoas com mais de 70 anos")
    senão
        imprima ("Foram encontradas apenas ", Maior70, " pessoas com idade superior a 70 anos");
    fim.

```

3.6.2. Desvio Incondicional

Existe um comando para desviar incondicionalmente o fluxo do algoritmo. É um comando que, em diversas linguagens de programação, possui uma sintaxe muito semelhante: é o comando *goto* - "vá para". Ele é sempre utilizado em conjunto com uma "etiqueta" (*label* em inglês), que indica o ponto no algoritmo para onde o fluxo de execução deve desviar.

Entretanto, esse comando possui um potencial enorme para desestruturar algoritmos, podendo torná-los de difícil compreensão e manutenção, devendo ser usado apenas em casos especiais, quando nenhuma outra construção resolve o problema. Por esse motivo, esse comando não será visto em maior detalhe. Como exemplo, na [Figura 26](#) é possível comparar dois trechos de algoritmos equivalentes, escritos com e sem o comando *goto*, para entender o perigo do uso indiscriminado desse comando: ele facilita a construção do que se chama comumente de "código spaghetti" ou "macarrônico".

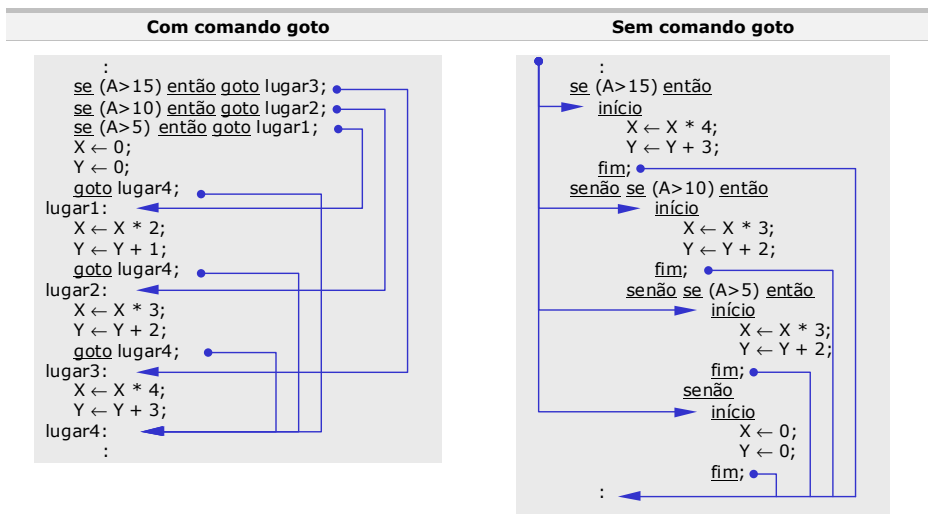


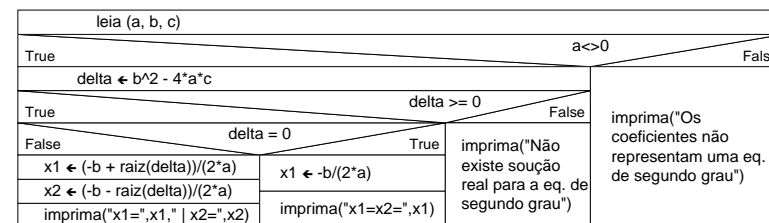
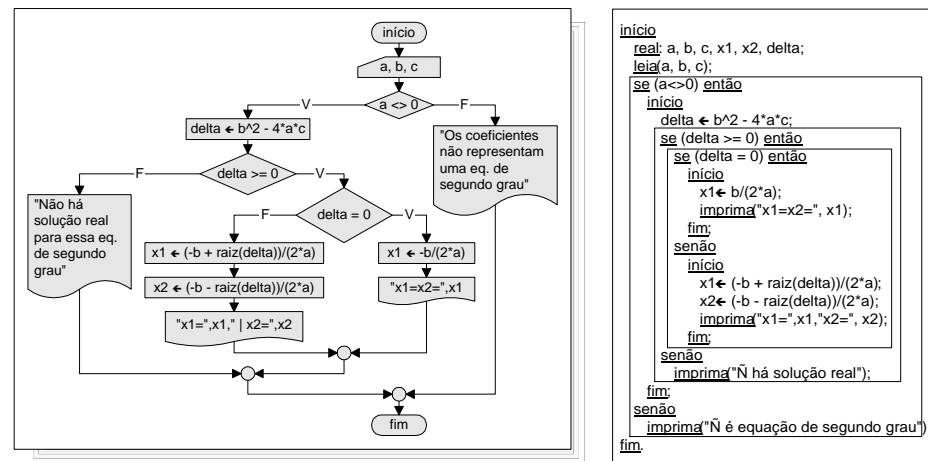
Figura 26: Comparação entre um trecho de algoritmo escrito utilizando o comando *goto* e outro equivalente escrito de forma estruturada, sem esse comando.

Referências do Capítulo

- [5] Forbellone, A. L. V.; Eberspächer, H. F.: "Lógica de Programação - A Construção de Algoritmos e Estruturas de Dados", Makron Books do Brasil Editora Ltda, São Paulo, Brasil, 1993.
- [6] Guimarães, A. M. e Lages, N. A. C.: "Algoritmos e Estruturas de Dados", Livros Técnicos e Científicos Editora S.A, Rio de Janeiro, Brasil, 1994.
- [7] Hehl, M. E.: "Linguagem de Programação Estruturada FORTRAN 77", Editora McGraw-Hill Ltda, São Paulo, Brasil, 1986.
- [8] Gottfried, B. S.: "Programação em Pascal", Editora McGraw-Hill de Portugal Lda, Lisboa, Portugal, 1988.
- [9] Kernighan, B. W. e Ritchie, D. M.: "C - A Linguagem de Programação", Editora Campus Ltda, Rio de Janeiro, Brasil, 1986.
- [10] Função Ajuda dos Compiladores Fortran Force 2.0, Visual Basic (Microsoft) e Rapid-Q Basic.

EXERCÍCIOS AVULSOS**Exercício Avulso 1: Eq. de 2ª Grau**

Construir um algoritmo, em fluxograma e português estruturado, para resolver a equação de 2ª grau: $ax^2 + bx + c = 0$ pela fórmula de Bhaskara. Os três casos possíveis ($\Delta > 0$, $\Delta = 0$ e $\Delta < 0$) devem ser considerados; apenas soluções reais são desejadas.

Solução

Exercício Avulso 2: seleção múltipla

Elabore um algoritmo que, dada a idade de um competidor, classifique-o em uma das seguintes categorias:

infantil A: 5-7 anos
 infantil B: 8-10 anos
 juvenil A: 11-13 anos
 juvenil B: 14-17 anos
 sênior: maiores do que 18 anos

O algoritmo deve também verificar se a idade do competidor está na faixa permitida para a competição, que vai dos 5 anos até 25 anos, inclusive.

Fazer esse algoritmo em fluxograma e português estruturado.

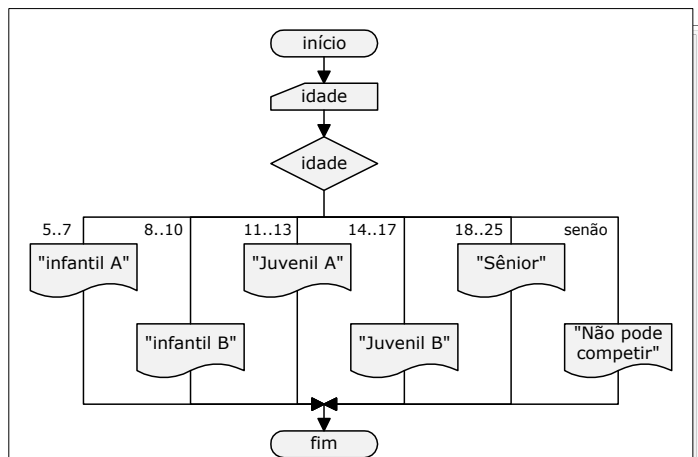
Solução

Português Estruturado:

```

início
  inteiro: idade;
  leia(idade);
  escolha idade
    caso 5..7: imprima("Infantil A");
    caso 8..10: imprima("Infantil B");
    caso 11..13: imprima("Juvenil A");
    caso 14..17: imprima("Juvenil B");
    caso 18..25: imprima("Sênior");
    senão imprima("Não pode competir");
  fim escolha;
fim.
  
```

Fluxograma:

**Exercício Avulso 3: comando repita**

Supondo que o valor de N também possa ser zero, faça a correção nos diagramas do Exercício 18 para que essa situação seja corretamente tratada: $S=0$ p/ $N=0$.

Solução

Fluxograma

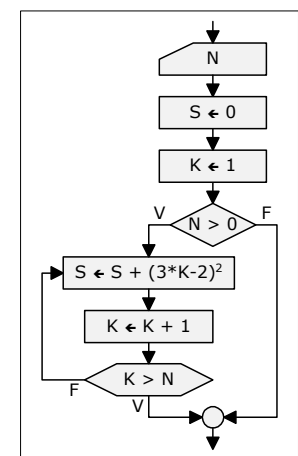
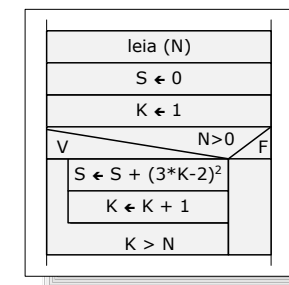


Diagrama de Chapin



Exercício Avulso 4: múltiplas alternativas

Escreva um algoritmo, em fluxograma e português estruturado, que leia N conjuntos de dados, cada um destes conjuntos contendo a altura e o código do sexo de uma pessoa (código = 1 p/ masculino, código = 2 p/ feminino). Esse algoritmo deve calcular e imprimir:

- a maior e a menor altura lida;
- a média de altura das mulheres;
- a média de altura global;

Solução

```

início
  inteiro: K, Cod, N, NM;
  real: H, SomaH, SomaHM, MaiorH, MenorH;
  {--- Inicialização de Variáveis ---}
  SomaH ← 0; SomaHM ← 0; NM ← 0;
  leia(N);
  { Coleta e análise dos dados }
  para K de 1 até N
    início
      leia(Cod, H);
      SomaH ← SomaH + H;
      se (k=1) então
        início
          MenorH ← H;
          MaiorH ← H;
        fim;
      senão
        início
          se (H > MaiorH) então MaiorH ← H;
          se (H < MenorH) então MenorH ← H;
        fim;
      se (Cod = 2) então
        início
          SomaHM ← SomaHM + H;
          NM ← NM + 1;
        fim;
      fim;
  imprima("Maior altura:", MaiorH, "Menor altura:", MenorH);
  imprima("Altura Média da Mulheres:", SomaHM/NM);
  imprima("Altura Média:", SomaH/N);
fim.

```

4. Modularização de Algoritmos

4.1. Dividir para Conquistar

Algoritmos podem ser simples, descritos por algumas poucas linhas em português estruturado, ou maiores e mais complexos, constituídos de vários comandos e estruturas de controle. Em tais situações, técnicas para dividir o algoritmo em vários trechos menores, porém integrados, são utilizadas para melhor administrar o tamanho e a complexidade. É mais fácil trabalhar com vários trechos reduzidos e menos complexos do que com um algoritmo inteiro, complexo e monolítico²⁶.

Algoritmo Monolítico	Algoritmo utilizando função
<pre> início real: A, B, X, Y, Z; inteiro: N, M, Fat, K; : leia (N, M, A, B); {assume-se que N, M ≥ 0 } K ← 1; {Cálculo do fatorial de N} Fat ← 1; enquanto K < N faça início Fat ← Fat * (K + 1); K ← K + 1; fim; : X ← (A / Fat)^2; : K ← 1; {Cálculo do fatorial de M} Fat ← 1; enquanto K < M faça início Fat ← Fat * (K + 1); K ← K + 1; fim; : Y ← B + 10*Fat; : K ← 1; {Cálculo do fatorial de M+N} Fat ← 1; enquanto K < (M + N) faça início Fat ← Fat * (K + 1); K ← K + 1; fim; : Z ← Fat / (A + B); : fim. </pre>	<pre> início real: A, B, X, Y, Z; inteiro: N, M; leia (N, M, A, B); {assume-se que N, M ≥ 0 } : X ← (A / Fatorial(N))^2; : Y ← B + 10 * Fatorial(M); : Z ← Fatorial(M+N) / (A+B); : fim. inteiro Fatorial (inteiro: N) início inteiro: K, Fat; K ← 1; Fat ← 1; enquanto K < N faça início Fat ← Fat * (K + 1); K ← K + 1; fim; Fatorial ← Fat; {valor de retorno} fim. </pre>

Figura 27: Comparação de um algoritmo monolítico com outro equivalente utilizando função: torna-se flagrante a vantagem do uso de funções quando o mesmo tipo de cálculo é necessário ser realizado repetidas vezes.

Tomemos como exemplo um algoritmo que resolve um determinado problema matemático onde o cálculo do fatorial de um número é realizado várias vezes, em pontos diferentes do algoritmo. Na Figura 27, o algoritmo da esquerda é uma versão da implementação da solução utilizando uma

²⁶ Uma técnica "desenvolvida" pelo general e imperador romano Júlio César: dividir para conquistar. Obviamente, em outro contexto.

construção monolítica. Note que para calcular o fatorial de N, M e (N + M) foi necessário repetir 3 vezes o mesmo trecho de algoritmo. Se fosse necessário o cálculo do fatorial em mais lugares, em cada um deles seria necessário reescrever o trecho que calcula o fatorial, e o algoritmo ficaria muito grande e repetitivo.

Existe ainda um outro problema. Imagine se, por algum motivo, fosse necessário alterar a maneira como se calcula o fatorial. Nesse caso, seria necessário ir a cada ponto do algoritmo onde esse cálculo é realizado e efetuar a modificação, correndo-se o risco de se cometer erros em um ou outro ponto, ou mesmo esquecer-se de se fazer a modificação em algum lugar. Isto não seria difícil de acontecer se esse cálculo tivesse que ser realizado, por exemplo, em 30 lugares diferentes do algoritmo.

Esse tipo de problema é solucionado através do uso de funções e/ou procedimentos, que são partes ou trechos de algoritmos que executam uma determinada operação a partir de dados de entrada ou parâmetros. Uma função é um algoritmo que retorna como resultado um único valor colocado justamente no ponto onde a função é "chamada", sendo ali utilizado como se a própria função fosse uma variável ou constante. O procedimento se distingue da função por não retornar nenhum valor, apenas processando dados e fornecendo seus resultados de outra forma.

Um exemplo de utilização de função é mostrado no algoritmo da direita na Figura 27, onde os vários trechos para cálculo do fatorial de um número são substituídos por chamadas a uma única função definida apenas uma vez em um único local.

4.2. Funções

Como mencionado acima, uma função é um algoritmo construído para resolver um problema específico, fornecendo como resultado um único valor, geralmente calculado a partir de dados de entrada - [Figura 28](#). Normalmente o nome da função fornece alguma indicação do tipo de cálculo que ela realiza, como a função *Fatorial* na Figura 27. Note nesta figura (algoritmo da direita) que, para utilizar a função *Fatorial*, foi realizada o que se denomina "chamada de função", que significa escrever o nome da função no ponto onde se deseja utilizar o resultado que ela retorna. Após o identificador da função são passados, entre parênteses, os dados necessários para realizar o cálculo, que no caso da função *Fatorial* é um número inteiro não negativo.

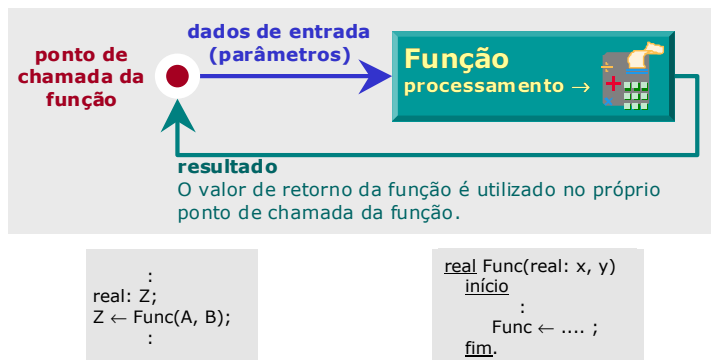


Figura 28: Esquema básico de funcionamento de uma função.

A sintaxe utilizada para escrever uma função é definida na [Figura 29](#). Deve ser notado que, para uma função retornar um valor, ele deve ser explicitamente definido. No português estruturado isso é feito através da atribuição de um valor, constante ou resultado de uma expressão ao identificador da função dentro de seu bloco, como se esta fosse uma variável (veja a seta no exemplo da Figura 29). É imprescindível que isso seja feito, e o valor atribuído deve ser compatível com o tipo definido para a função (se a função *Soma1* é real, S pode ser real ou inteira, mas nunca caractere ou lógica).

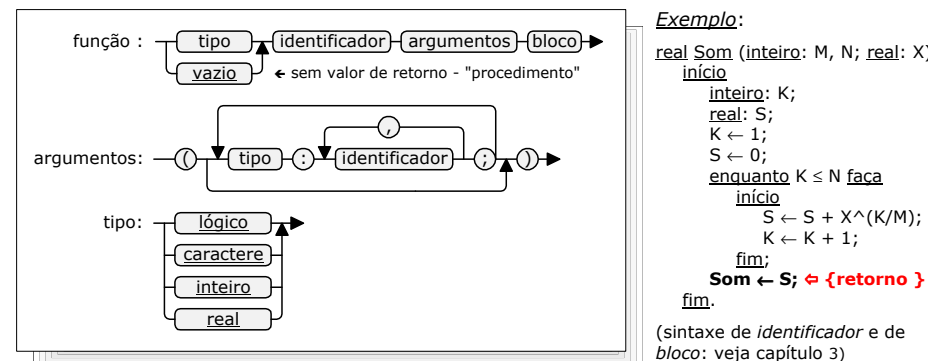


Figura 29: Sintaxe de função no português estruturado

Outra observação importante: o identificador (nome) da função não pode ser utilizado no bloco que a define em expressões (como *Som > 3*) ou do lado direito de atribuições (como *Z ← Som + 1*). Por esse motivo, na função *Som* da Figura 29 foi utilizada a variável S ao invés de do identificador *Som*. Mas existe uma exceção: quando se faz uso recursivo da função, isto é, chama-se a função dentro dela mesma. Para tal, existem algumas regras que serão vistas mais adiante (Recursividade de Funções, página 86).

A primeira linha de uma função é o seu cabeçalho, onde o nome e o tipo da função, bem como a lista dos argumentos que receberão os dados de entrada da função, são definidos. A quantidade de argumentos pode ser qualquer, inclusive nenhum, e é a partir dos dados por eles recebidos que a função executa o seu cálculo ou processamento. As variáveis na lista de argumentos não precisam ser declaradas para serem utilizadas no bloco da função, pois isso já é feito implicitamente no próprio cabeçalho. Qualquer outra variável que não esteja na lista de argumentos e que seja utilizada na função deve ser declarada, e não pode possuir um identificador que seja igual ao identificador de algum dos argumentos da função, ou da própria função, o que geraria duplicidade de declarações.

Em algumas linguagens, como o C, não se utilizam procedimentos (próximo item), mas apenas funções. Entretanto, nessa linguagem são previstas funções que não retornam valores, mas que apenas executam um determinado processamento de dados como se fosse um procedimento. Nessa situação, para representar uma função que não tem valor de retorno, defini-se esta função como sendo do tipo *vazio*²⁷ (veja novamente a sintaxe descrita na [Figura 29](#)), não sendo necessário definir dentro do bloco da função o valor de retorno, já que ele não existe. Dessa forma, ter-se-á duas formas de se implementar o conceito de procedimentos: utilizando funções do tipo *vazio*, ou procedimentos de forma explícita, como descrito a seguir.

Uma observação final: o bloco de uma função, ao contrário de outras estruturas de controle, sempre deverá ser delimitado pelas palavras início e fim, mesmo que contenha apenas um comando, como mostrado ao lado. O mesmo comentário é válido para os procedimentos (próximo item).

```

real Func (real: X)
início
  Func ← x^3 + x;
fim.
  
```

4.3. Procedimentos

Procedimentos, também conhecidos como subrotinas, são operacionalmente idênticos às funções a menos do valor de retorno, que não existe nesse caso - [Figura 30](#). A sintaxe para construção de um procedimento é definida na [Figura 31](#). Linguagens como Pascal e Visual Basic implementam

²⁷ Na linguagem C, trata-se do tipo void.

procedimentos (*procedure* e *sub*, respectivamente), enquanto a linguagem C só implementa funções, incluindo as do tipo vazio que fazem o papel de procedimentos.

Como não há valor de retorno, um procedimento é utilizado através de uma chamada simples em uma linha de código. Exemplo: no procedimento abaixo é realizado um cálculo passando-se os dados através das variáveis A e B, sendo o resultado retornado através das variáveis C e D (por isso se utiliza passagem por referência para essas duas variáveis - veja próximo item)

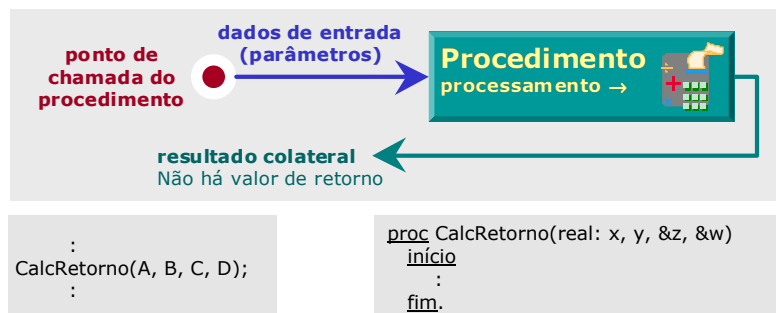
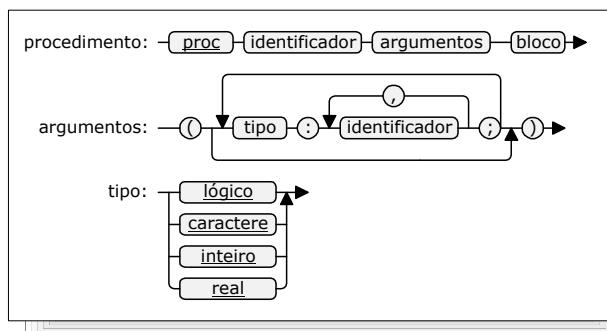


Figura 30: Esquema básico de funcionamento de um procedimento.



Exemplo:

```
proc ImpV (inteiro: M, N, P)
início
  imprima("M=", M);
  imprima("N=", N);
  imprima("P=", P);
fim.
```

(sintaxe de *identificador* e de *bloco*: veja capítulo 4)

Figura 31: Sintaxe de procedimento no português estruturado.

4.4. Passagem de Parâmetros

Funções e procedimentos (módulos), na maior parte das vezes, realizam seu processamento a partir de parâmetros de entrada passados para os argumentos. Existem duas modalidades de passagem de parâmetros para uma função ou procedimento: por *valor* e por *referência*. Na passagem de parâmetros por valor é feita uma cópia do valor passado (obtido através de uma constante, do conteúdo de uma outra variável ou mesmo do resultado de uma expressão) para o conteúdo do respectivo argumento do módulo, de tal modo que não há nenhuma alteração ou influência em qualquer variável do algoritmo que "chamou" o módulo. A passagem por valor pode ser feita como exemplificado na Figura 32 para a função fatorial.

Constante	Variável	Variável	Expressão
:	:	:	:
Z ← Fatorial (4);	Z ← Fatorial (N);	Z ← Fatorial (K);	Z ← Fatorial (2*K + 2);
:	:	:	:
Z contém o fatorial de 4	Z contém o fatorial de N	Z contém o fatorial de K	Z contém o fatorial de (2*K + 2)

Figura 32: Exemplo de chamada de função (N e K são inteiros, não negativos)

No caso da passagem por referência, há duas diferenças principais:

- o valor deve ser passado apenas através de uma variável;
- se o valor do parâmetro for alterado dentro do módulo, o conteúdo da variável utilizada para passagem do valor também é alterado.

A Figura 33 ilustra o que acontece quando se utiliza cada um dos tipos de passagem de parâmetro. Note que na passagem por referência, representada pela colocação do símbolo & na frente do argumento no cabeçalho do módulo, tanto a variável do algoritmo principal (K) como o argumento do módulo (N) compartilham o mesmo endereço, ou *referência*, na memória, o que significa que o seu conteúdo é comum: modificando-se o valor do argumento na função, o valor da variável do algoritmo principal, passada como parâmetro, também é alterada. Na passagem por valor, o argumento recebe uma cópia do valor da variável, mas em outro local da memória reservada para esse argumento. Por esse motivo, alterações no conteúdo do argumento dentro da função ou procedimento não tem nenhum reflexo na variável do algoritmo quando a passagem é por valor.

Analisemos agora o exemplo mostrado na Figura 34, onde temos passagem de parâmetro por valor. Podemos observar nessa figura o uso de uma variável N no algoritmo principal, e de um argumento N no cabeçalho da função *Fatorial*. Muita atenção: *são entidades distintas, apesar de utilizarem o mesmo identificador!* O escopo da variável N no algoritmo principal é apenas o bloco onde ela é declarada. Argumentos de funções, como o argumento N da função *Fatorial*, tem como escopo²⁸ o bloco da função, recebendo os valores passados através da chamada da função. Dessa forma, alterações no valor de N dentro da função *Fatorial* não tem nenhum efeito na variável N do algoritmo principal, pois são variáveis diferentes, com escopos diferentes. Isso é válido para qualquer parâmetro *passado por valor*.

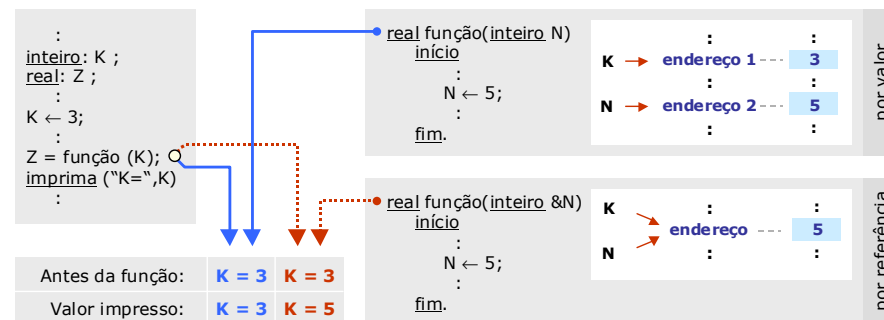


Figura 33: Passagem por valor e por referência: comportamento da referência às variáveis.

²⁸ De forma resumida, *escopo* significa a região onde a variável existe ou pode ser usada.

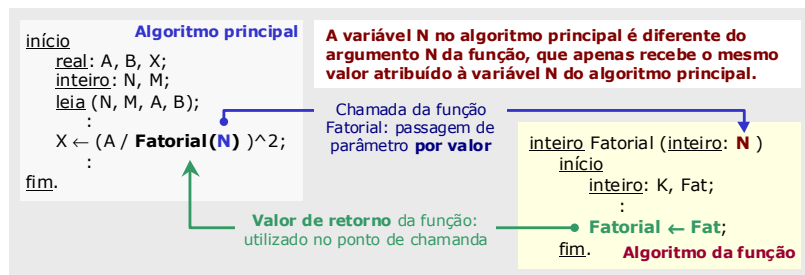


Figura 34: "Chamada de função" no português estruturado.

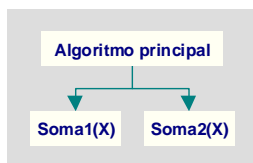
Exemplo 20: um algoritmo utilizando duas funções.

No algoritmo abaixo, temos o programa principal fazendo uso de duas funções: Soma1(x) e Soma2(x).

```

início
  real: X;
  inteiro: N;
  leia (N);
  X ← Soma1(3) + Soma1(N);
  X ← X + Soma2(N+1) + Soma2(N+2);
  imprima ("X=", X);
fim.

```



```

real Soma1 (inteiro: N)
  início
    inteiro: K;
    real: S; S ← 0;
    para K de 1 até N
      S ← S + 1/(K + 1);
    Soma1 ← S; {valor de retorno}
  fim.

```

```

real Soma2 (inteiro: N)
  início
    inteiro: K;
    real: S; S ← 0;
    para K de 1 até N
      S ← S + (K + 1)^2/K^3;
    Soma2 ← S; {valor de retorno}
  fim.

```

Exercício 22: algoritmo utilizando função com passagem por valor.

Faça um algoritmo que imprima o somatório abaixo para $N = 4, 5, 6$ e 7 . Esse somatório deve ser calculado através de uma função (passagem por valor), e o algoritmo principal deve utilizar o comando *para* na realização da impressão.

$$S = 1 + 2 + 3 + \dots + N$$

Solução

Algoritmo principal:

```

início
  inteiro: N;
  para N de 4 até 7
    imprima ("S =", Soma(N));
  fim.

```

Função:

```

real Soma (inteiro: N)
  início
    inteiro: K, S;
    S ← 0;
    para K de 1 até N
      S ← S + K;
    Soma ← S; {valor de retorno}
  fim.

```

Exercício 23: algoritmo utilizando função com passagem por referência

Construa um algoritmo que leia duas variáveis A e B inteiras, imprima os valores lidos, troque os valores das variáveis entre elas através da chamada a um módulo (função e procedimento: construir os dois) com passagem por referência e imprima o resultado.

Solução

Algoritmo Principal

```

início
  inteiro: A, B;
  leia (A, B);
  imprima ("A =", A, "B =", B);
  troca (A, B);
  imprima ("Valores trocados:");
  imprima ("A =", A, "B =", B);
fim.

```

Função e Procedimento

```

vazio Troca (inteiro: &X1, &X2)
  início
    inteiro: Temp;
    Temp ← X1;
    X1 ← X2;
    X2 ← Temp;
  fim.

proc Troca (inteiro: &X1, &X2)
  início
    inteiro: Temp;
    Temp ← X1;
    X1 ← X2;
    X2 ← Temp;
  fim.

```

4.5. Exemplos de Módulos nas Linguagens de Programação

Nesse item são mostradas implementações da modularização em diversas linguagens de programação. A Figura 35 mostra um algoritmo modularizado em português estruturado, e da Figura 36 até a Figura 39 tem-se exemplos da implementação desse algoritmo em várias linguagens.

Português Estruturado
<pre> { Função que calcula a média de dois valores. } { A passagem de parâmetros é por valor. } real media(real: x, y) início media = (x + y)/2 fim; { Procedimento que troca o conteúdo de duas variáveis. } { A passagem de parâmetros é por referência. } proc troca(real: &x, &y) início real: t; t = x; x = y; y = t; fim; { *** Programa Principal *** } início inteiro: a, b; imprima("Digite a e b: "); leia (a, b); troca (a, b); imprima("Valores trocados: a = "; a; " b = "; b); imprima ("Media dos valores = "; media(a, b)) end. </pre>

Figura 35: Algoritmo utilizando função e procedimento.

PASCAL

```

program Main;

var a, b: integer;

{ Função que calcula a média de dois valores. }
{ Passagem de parâmetros por valor. }
function media(x, y: double): double;
begin
    media := (x + y) / 2;
end;

{ Procedimento que troca os valores de duas variáveis. }
{ Passagem de parâmetros por referência. }
procedure troca(var x, y: integer);
var t: integer;
begin
    t := x;
    x := y;
    y := t;
end;

{ Programa principal }
begin
    write('Entre com a e b: ');
    readln(a, b);
    troca(a, b);
    writeln('Valores trocados: a=', a, ' b=', b);
    writeln('Valor médio: ', media(a, b):5:2);
    readln(a);
end.

```

Figura 36: Programa em PASCAL implementando o algoritmo da Figura 35.**BASIC**

```

' Função que calcula a média de dois valores
' A passagem de parâmetros é por valor.
function media(x as double, y as double) as double
    media = (x + y) / 2
end function

' Subrotina (procedimento) que troca o conteúdo
' de duas variáveis.
' A passagem de parâmetros é por referência.
sub troca(byref x as double, byref y as double)
    dim t as double
    t = x
    x = y
    y = t
end sub

' *** Programa Principal ***

input "Digite a: "; a
input "Digite b: "; b

troca a, b      ' Os parâmetros não precisam ficar entre
                ' parênteses na chamada do procedimento.

print "Valores trocados: a = "; a; " b = "; b
print "Media dos valores = "; media(a, b)

```

Figura 37: Programa em BASIC implementando o algoritmo da Figura 35.**C**

```

#include <stdio.h>

/* Funcao que calcula a media de dois valores */
float media(float x, float y){
    return (x + y) / 2;
}

/* Funcao que troca os valores de duas variaveis */
/* Passagem por referencia via ponteiros */
void troca (int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
}

/* Programa Principal*/
void main(){
    int a, b;
    printf("Digite a e b: ");
    scanf("%i %i",&a,&b);
    troca(&a, &b);
    printf("\nValores trocados: a = %i; b = %i",a,b);
    printf("\nValore medios: %.2f",media(a,b));
}

```

Figura 38: Programa em C implementando o algoritmo da Figura 35.

	FORTTRAN
C	Função que calcula a média de dois valores <pre> real function media(x, y) integer x, y media = 0.5*(x + y) return end </pre>
C	Subrotina (procedimento) que troca os valores de duas variáveis. No FORTRAN, toda vez que o parâmetro da função é uma variável sua passagem é feita por referência, caso contrário (expressão ou constante) é feita por valor.
C	<pre> subroutine troca(x, y) integer x, y, t t = x x = y y = t return end </pre>
C	Programa principal <pre> program Main integer a, b real media write (*,*) 'Entre com a e b: ' read (*,*) a, b write (*,1) 'Valores lidos: a=',a,' b=',b 1 format(a,I3,a,I3) call troca(a,b) write (*,1) 'Valores trocados: a=',a,' b=',b write (*,2) 'M,dia dos valores: ', media(a,b) 2 format(a,f5.2) stop end </pre>

Figura 39: Programa em FORTRAN implementando o algoritmo da Figura 35.

4.6. Estrutura de Programas – Modularização

Como já comentado no início desse capítulo, a administração da complexidade e tamanho dos algoritmos e, conseqüentemente, dos programas de computador é realizada através do uso de módulos (funções e procedimentos). Isso se denomina modularização, isto é, a divisão do algoritmo global em algoritmos menores e relacionados entre si, com funcionalidades bem definidas, como exemplificado na Figura 40. É possível notar nessa figura que um módulo pode chamar outro módulo, e assim por diante, constituindo uma estrutura de chamada de módulos.

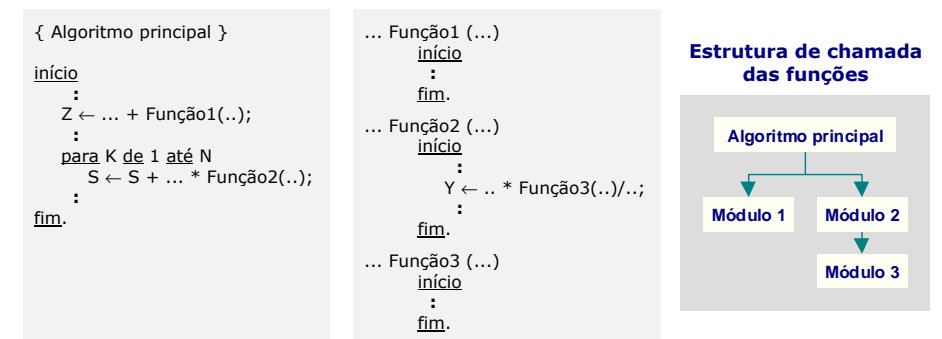


Figura 40: Algoritmo dividido em módulos menores.

Outro aspecto importante é o fato de partes de um algoritmo implementadas como módulos poderem ser úteis na construção de outros algoritmos para solução de diferentes tipos de problemas. Se não houvesse algum tipo de modularização, a reutilização não seria possível. A Figura 41 ilustra, de forma esquemática, como isso acontece.

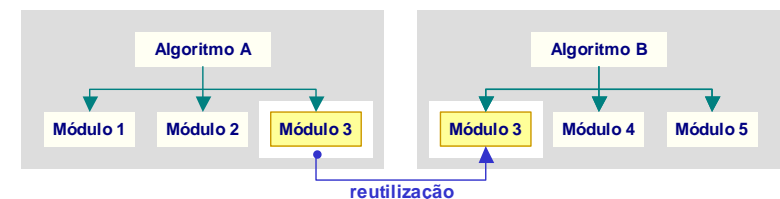


Figura 41: Exemplo de reutilização - o módulo 3 poderia ser a função Fatorial, que aparece na solução de diversos problemas matemáticos.

4.7. Recursividade de Funções

A recursividade de uma função é a possibilidade dela chamar a si mesma para realizar o cálculo desejado. Ao se escrever uma função recursiva, é essencial que ela inclua uma condição de finalização. Isso evita que a chamada recursiva continue indefinidamente. Porém, enquanto a condição de finalização não for satisfeita, a função simplesmente chama a si mesma no local apropriado, tal como faria qualquer outra função.

O exemplo clássico dessa característica é a implementação da função Fatorial na forma recursiva. O fatorial de um número n pode ser definido recursivamente como sendo $n! = n \cdot (n-1)!$, em que

$1! = 1$. Assim sendo, a condição de finalização da chamada recursiva ocorre quando n for igual a um (1). O algoritmo da função Fatorial implementada recursivamente é mostrado na [Figura 42](#).

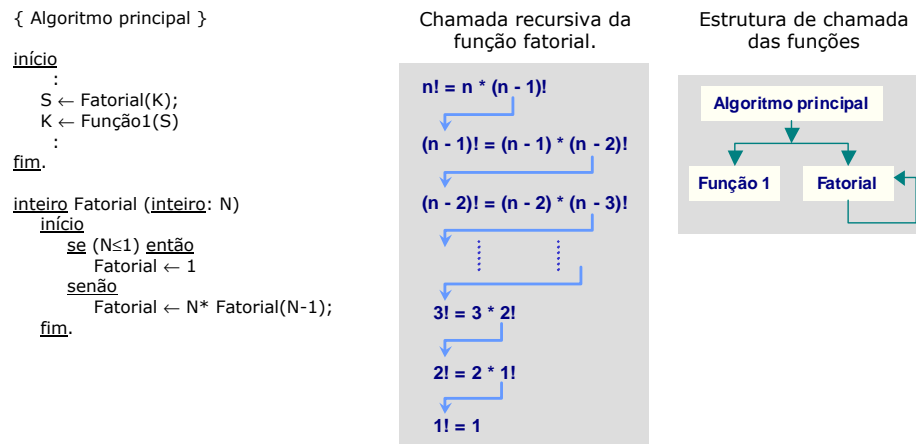


Figura 42: Exemplo de função implementada recursivamente.

Exercício 24: implementação de função recursiva

Construa um algoritmo que calcule o seguinte somatório para $n = 6, 7$ e 8 :

$$S = 1 + x + x^2 + x^3 + x^4 + \dots + x^n, \quad -1 \leq x \leq 1$$

Esse algoritmo deve ser implementado utilizando uma função recursiva que irá calcular o somatório, sendo os parâmetros de entrada dessa função os valores de x e n .

O algoritmo deverá ler o valor de x e imprimir o valor do somatório para cada um dos valores de n .

Dica: O somatório acima pode ser expandido da seguinte forma:

$$\begin{aligned} S &= 1 + x + x^2 + x^3 + x^4 + \dots + x^n &&= F(n, x) = \\ &= 1 + x (1 + x + x^2 + x^3 + \dots + x^{n-1}) &&= 1 + x F(n-1, x) = \\ &= 1 + x (1 + x (1 + x + x^2 + \dots + x^{n-2})) &&= 1 + x (1 + x F(n-2, x)) = \\ &= \vdots &&\vdots \\ &= 1 + x (1 + x (1 + x (1 + x (1 + x (\dots x (1 + x (1)) \dots))) && \\ &= 1 + x (1 + x (1 + \dots x (1 + x F(0, x)) \dots) && \end{aligned}$$

Ou seja, o somatório S pode ser expresso recursivamente da seguinte forma:

$$F(n, x) = 1 + x F(n-1, x), \text{ com } F(0, x) = 1 \text{ (condição de finalização: } n = 0)$$

Solução

```

início
  real: X;
  inteiro: K;
  leia (X);
  para K de 6 até 8
    imprima ("S(", K, ") =" , SomaRec(K, X) );
fim.

```

```

real SomaRec (inteiro: N; real: X)
  início
    se (N ≤ 0) então { esse sinal garante que, caso N seja negativo, }
      SomaRec ← 1 { a função não entre em "loop infinito" }
    senão
      SomaRec ← 1 + X * SomaRec(N-1, X);
  fim.

```

Referências do Capítulo

- [11] Forbellone, A. L. V.; Eberspächer, H. F.: "*Lógica de Programação - A Construção de Algoritmos e Estruturas de Dados*", Makron Books do Brasil Editora Ltda, São Paulo, Brasil, 1993.
- [12] Guimarães, A. M. e Lages, N. A. C.: "*Algoritmos e Estruturas de Dados*", Livros Técnicos e Científicos Editora S.A, Rio de Janeiro, Brasil, 1994.
- [13] Hehl, M. E.: "*Linguagem de Programação Estruturada FORTRAN 77*", Editora McGraw-Hill Ltda, São Paulo, Brasil, 1986.
- [14] Gottfried, B. S.: "*Programação em Pascal*", Editora McGraw-Hill de Portugal Lda, Lisboa, Portugal, 1988.
- [15] Kernighan, B. W. e Ritchie, D. M.: "*C - A Linguagem de Programação*", Editora Campus Ltda, Rio de Janeiro, Brasil, 1986.
- [16] Função Ajuda dos Compiladores Fortran Force 2.0, Visual Basic (Microsoft) e Rapid-Q Basic.

EXERCÍCIOS AVULSOS

Exercício Avulso 5: chamada de funções (1)

Construa um algoritmo (português estruturado) que leia as coordenadas de dois pontos ((x_1, y_1) e (x_2, y_2)) e imprima a distância entre eles. Utilizar uma função para o cálculo da distância.

Solução

```

início
    real: x1, x2, y1, y2;
    leia(x1, x2, y1, y2);
    imprima ("Distancia = ", Distancia(x1, y1, x2, y2);
fim.

real Distancia (real: a1,b1,a2,b2)
início
    Distancia ← raiz((a1-a2)2+(b1-b2)2);
fim.

```

Exercício Avulso 6: chamada de funções (2)

Construa um algoritmo (português estruturado) que monte uma tabela de temperaturas, sendo que a primeira coluna contém a temperatura em graus Celsius variando 0 a 100 C, de 10 em 10, e a segunda coluna contém a temperatura em graus Fahrenheit. Utilizar uma função para retornar a temperatura em Fahrenheit dada a temperatura em graus Celsius.

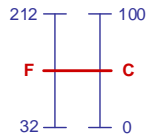
Solução

```

início
    inteiro: C;
    imprima ("-----");
    imprima ("| Celsius | Fahrenheit |");
    imprima ("-----");
    para C de 0 até 100 passo 10
        imprima ("| "; C; " | "; Fahrenheit(k), " | ");
    imprima ("-----");
fim.

real Fahrenheit (real: C)
início
    Fahrenheit ← 9*C/5+32;
fim.

```



Exercício Avulso 7: chamada múltipla de funções

Construa um algoritmo que calcule o seguinte somatório:

$$S = \frac{dF}{dx}(1) + \frac{dF}{dx}(2) + \frac{dF}{dx}(3) + \dots + \frac{dF}{dx}(N)$$

A função $dF/dx(x)$ é a derivada primeira de $F(x)$, e deve ser calculada pela seguinte expressão, obtida a partir da definição de derivada:

$$\frac{dF}{dx}(x) = \frac{F(x+e) - F(x)}{e}$$

onde e é um número muito pequeno: $e = 10^{-6}$.

A função $F(x)$ é:

$$F(x) = x^3 + x + 1/x$$

Solução

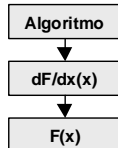
```

início
    inteiro: K, N;
    real: S;
    S ← 0;
    leia(N);
    para K de 1 até N
        S ← S + dFdx(K);
    imprima ("S = ", S );
fim.

real dFdx(real: X)
início
    dFdx ← ((F(x + 1e-6)-F(x))/1e-6;
fim.

real F(real: X)
início
    F ← x^3 + x - 1/x;
fim.

```



Exercício Avulso 8: chamada de funções (1)

Implementar recursivamente uma função que calcula o termo da série de Fibonacci conforme a sua posição na série:

Termo	1	1	2	3	5	8	13	21	34	55	...
Posição K	1	2	3	4	5	6	7	8	9	10	...

Utilizando essa função, monte um algoritmo que imprima os N primeiros termos dessa série.

Solução

```

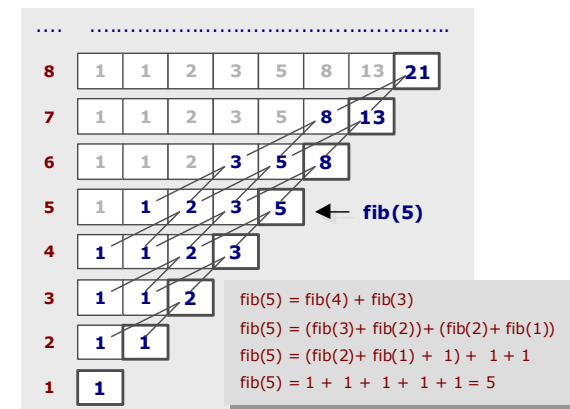
Para K = 1 ou 2: T(K) = 1
Para K > 2: T(K) = T(K-1) + T(K-2)

{ Algoritmo }

início
    inteiro: K, N;
    leia (N);
    para K de 1 até N
        imprima(Fibo(K));
fim.

inteiro Fibo(inteiro: K)
início
    se (K<=2) então
        Fibo ← 1
    senão
        Fibo ← Fibo(K-1) + Fibo(K-2);
fim.

```



5. Introdução às Estruturas de Dados

5.1. Estruturas de Dados Homogêneas

5.1.1. Definição

Imagine o seguinte problema:

Um professor possui uma turma com 40 alunos e necessita calcular a média bimestral de cada um deles baseado no resultado de duas provas. Para isso irá construir um algoritmo que leia todas as notas das provas de cada aluno, calcule todas as médias para só depois imprimi-las.

Utilizando apenas as variáveis simples vistas até agora, o algoritmo poderia ser como o mostrado na [Figura 1](#). Observando-se esse algoritmo, nota-se que para um problema simples foi necessário um número muito grande de linhas (mais de 80) para implementar a solução devido ao grande número de variáveis simples manipuladas.

Nesse ponto podemos nos perguntar: será que não existe uma maneira melhor de manipular dados em problemas como o citado acima? A resposta é sim. Para problemas desse tipo utilizam-se estruturas de dados ditas homogêneas, que nada mais são que conjuntos de dados de mesmo tipo. Vetores e matrizes, também vistos na matemática, são exemplos desse tipo de estrutura.

```

início
{ São necessárias 80 variáveis para guardar as duas notas de todos os alunos }
real: N1A1, N1A2, N1A3, ..., N1A40, N2A1, N2A2, N2A3, ..., N2A40;
{ São necessárias 40 variáveis para guardar as médias dos alunos }
real: Media1, Media2, ..., Media40;
{ É necessário ler as 80 notas dos alunos }
leia (N1A1, N1A2, N1A3, ..., N1A40, N2A1, N2A2, N2A3, ..., N2A40);
{ Cálculo das médias dos alunos: são necessária 40 linhas de atribuições }
Media1 ← (N1A1+N2A1)/2;      { 1 }
Media2 ← (N1A2+N2A2)/2;      { 2 }
:
:
Media40 ← (N1A40+N2A40)/2;    { 40 }
{ Impressão dos resultados: 40 linhas com o comando imprima }
imprima ("Media Aluno 1 = ", Media1);    { 1 }
imprima ("Media Aluno 2 = ", Media2);    { 2 }
:
:
imprima ("Media Aluno 40 = ", Media40);  { 40 }
fim
  
```

Figura 1: Algoritmo para cálculo das médias de alunos utilizando apenas variáveis simples.

5.1.2. Vetores

Conceito

Um vetor é uma estrutura de dados homogênea onde cada elemento ou componente pode ser considerado como uma variável simples. Cada elemento do vetor é referenciado por um índice, que define sua posição relativa dentro da estrutura.

Para se entender melhor o conceito de vetor, seja o seguinte exemplo: um edifício com vários andares, um apartamento por andar. O edifício pode ser visualizado como sendo um vetor onde cada apartamento corresponde a um componente desse vetor. Esse componente (ou "variável") pode conter pessoas (ou "valores do tipo pessoa"), sendo identificado por um número (número ou "índice" do apartamento). O número de pessoas que um apartamento contém em um determinado momento pode ser entendido como sendo um "valor atribuído" àquele apartamento. Esse exemplo pode ser visualizado na [Figura 2](#).

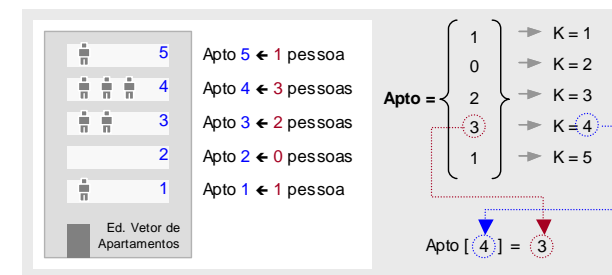


Figura 2: Visualização do conceito de vetor

Sintaxe

A sintaxe da declaração de vetores no português estruturado é mostrada na [Figura 3](#). Na [Figura 22](#) é apresentado um exemplo, em português estruturado e em algumas linguagens de programação, da declaração (ou definição) de um vetor e da atribuição de valores aos seus componentes (cada elemento do vetor recebe o valor de seu índice).

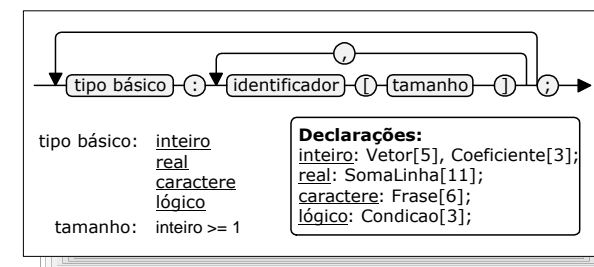


Figura 3: Exemplo da sintaxe para declaração de vetores no português estruturado

No algoritmo da [Figura 3](#) cada componente do vetor *Apto* pode ser considerado como sendo uma variável simples do tipo inteiro. No português estruturado, o índice de um vetor inicia-se em 1, ao passo que na linguagem C o primeiro componente de um vetor tem índice 0. Começar o índice de vetores em 0, em 1 ou em outro valor qualquer é convenção de cada linguagem, podendo ser alterado em algumas delas se o programador assim desejar (veja o exemplo para a linguagem Pascal).

PORT. ESTRUTURADO		FORTRAN	C
<pre> início inteiro: K, Apto[5]; para K de 1 até 5 Apto [K] ← 5; fim. </pre>	1	<pre> program Main integer k, a(5) do 1 k=1,5 a(k) = k continue stop end </pre>	<pre> void main() { int K, Apto[5]; for (K=0; K<5; K++) Apto[K] = 5; } </pre>
BASIC		PASCAL	
<pre> dim k as integer, a(5) as integer for k = 1 to 5 a(k) = k next k </pre>		<pre> program Main; var k: integer; a: array [5..10] of integer; begin for k := 5 to 10 do a[k] := k; end. </pre>	

Figura 4: Declaração e uso de vetores em diversas linguagens.

Com a utilização de vetores, pode-se construir o algoritmo para cálculo da média bimestral de 40 alunos, mostrado na Figura 1, da maneira mostrada na Figura 5.

```

início
{ São necessários 2 vetores para guardar as duas notas de todos os alunos }
real: K, Nota1[40], Nota2[40];
{ É necessário 1 vetor para guardas as médias dos alunos }
real: Media[40];
{ É necessário ler as 2 notas dos 40 dos alunos }
para K de 1 até 40
  leia (Nota1[K], Nota2[K]);
{ Cálculo das médias dos alunos }
para K de 1 até 40
  Media[K] ← (Nota1[K] + Nota2[K]) / 2;
{ Impressão dos resultados }
para K de 1 até 40
  imprima ("A média do Aluno ", K, " é ", Media[K]);
fim.

```

Figura 5: Algoritmo para cálculo da médias de alunos utilizando vetores.

É possível notar que o número de linhas executáveis do algoritmo (aquelas que não são comentários, nem delimitadoras de blocos, como início e fim) reduziu-se de 83 para apenas 8 na versão utilizando vetores.

Outro exemplo de algoritmo utilizando vetores pode ser visto na Figura 6, onde um vetor denominado *Apto* é criado possuindo cinco (5) componentes ($K = 1, 2, \dots, 5$), e o conteúdo de cada um deles é definido através da leitura do valor correspondente.

É preciso observar um ponto importante na Figura 6. Ao se declarar o vetor são definidos os endereços na memória onde serão armazenados os valores de cada componente, porém esses valores continuam indefinidos. Em determinadas linguagens, o compilador pode atribuir um valor padrão (como 0) à variável no momento da sua declaração; em outras, nada é feito, e um valor aleatório qualquer ("lixo da memória") contido previamente naquele endereço passa a ser o conteúdo da variável. É através do comando de atribuição (\leftarrow) que o valor a ser armazenado na variável é definido explicitamente, ou através da leitura do valor utilizando o comando *leia*, conforme feito nessa figura.

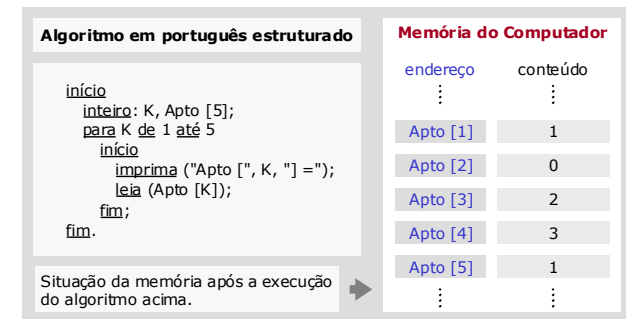


Figura 6: Configuração da memória após execução do algoritmo/programa de computador.

Outro tipo de algoritmo onde o uso de vetores é bem útil é o de ordenação de variáveis. São esses algoritmos que permitem pegar um conjunto de números e colocá-los em ordem crescente ou decrescente, ou um conjunto de palavras e ordená-las alfabeticamente. O algoritmo em português estruturado mostrado na Figura 7 é um exemplo deste tipo de algoritmo. Nesse caso, trata-se de ordenação pelo *método de seleção*.

```
-- Este algoritmo ordena um conjunto de N números em ordem crescente --
```

```

início
{ É necessário um vetor para guardar o conjunto de números }
real: Aux, J, K, Numero[100];
{ Leitura do conjunto de N números }
leia (N); { N ≤ 100 }
para K de 1 até N
  leia (Numero[K]);
{ Ordenação dos números em ordem crescente }
para J de 1 até N-1 { pesquisa N-1 pilhas }
  para K de (J+1) até N { procura números menores que o do topo da pilha }
    se (Numero[K] < Numero[J]) então { coloca menor no topo da pilha }
      início
        Aux ← Numero[K];
        Numero[K] ← Numero[J];
        Numero[J] ← Aux;
      fim;
  fim;
{ Impressão dos números ordenados }
para K de 1 até N
  imprima ("Numero[", K, "] = ", Numero[K]);
fim.

```

Figura 7: Algoritmo para ordenação de números através do método de seleção.

A ilustração do processo de ordenação pelo método de seleção que esse algoritmo realiza é mostrada na Figura 8.

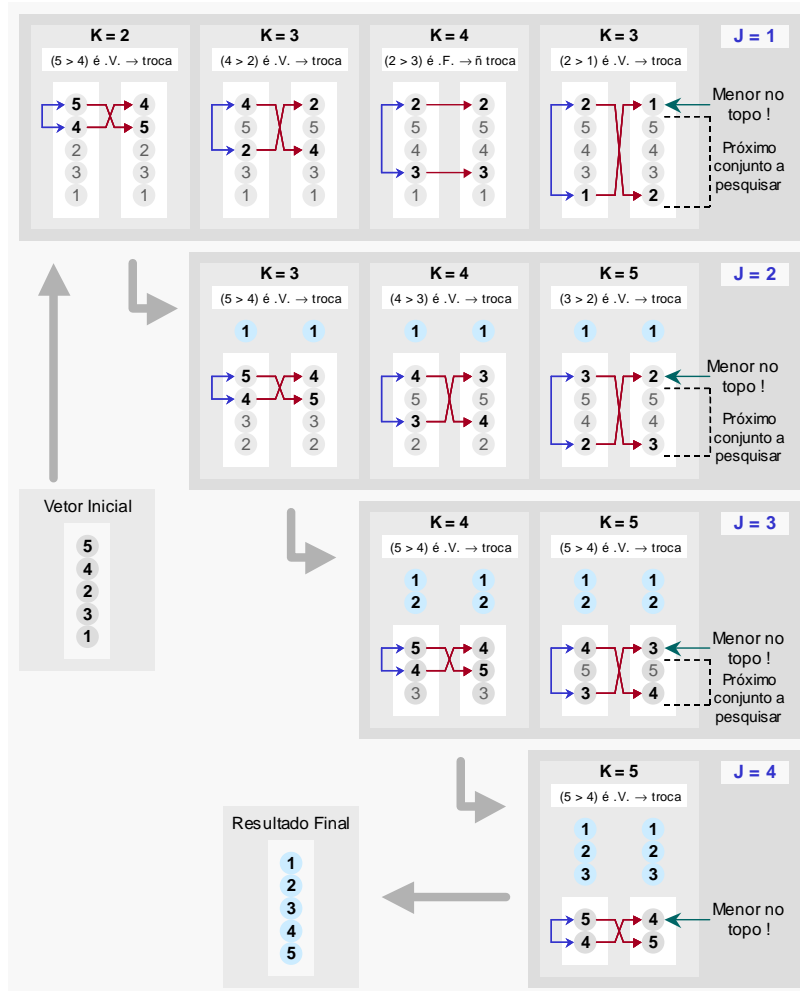


Figura 8: Visualização gráfica da repetição composta que ordena os números através do método de seleção.

Notação Simplificada

Ao trabalhar com vetores no português estruturado é possível utilizar a notação simplificada mostrada na Tabela 9. Essa notação é válida apenas para as situações mostradas nessa tabela, onde a variável Vetor aparece sozinha nos comandos *leia*, *imprima* e de atribuição.

Operação	Notação Normal	Notação simplificada
leitura do vetor	para K de 1 até N leia (Vetor[K]);	leia (Vetor);

Operação	Notação Normal	Notação simplificada
impressão do vetor	para K de 1 até N imprima (Vetor[K]);	imprima (Vetor);
inicialização de todo o vetor com o mesmo valor	para K de 1 até N Vetor[K] ← 0;	Vetor ← 0;

Tabela 9: Notação simplificada para comandos com vetores no português estruturado.

Exemplo 21: atribuição e uso de índices.

```

início
  inteiro: K, VE[6];
  caractere: CA[6];
  VE[1] ← 1; VE[2] ← 1; VE[3] ← 2;
  VE[4] ← 2; VE[5] ← 5; VE[6] ← 6;
  CA[1] ← "SEG"; CA[2] ← "TER";
  CA[3] ← "QUA"; CA[5] ← "SEX";
  CA[6] ← "SAB";
  para K de 1 até 6 passo 2
    imprima (CA[VE[K]]);
    imprima (CA[VE[VE[K]]]);
fim.

```

O que será impresso pelo algoritmo ao lado?

Solução

Após a atribuição dos valores aos componentes de cada vetor (região sombreada no algoritmo), obtém-se o seguinte:

VE:	1	1	2	2	5	6
k	1	2	3	4	5	6
CA:	SEG	TER	QUA	?	SEX	SAB

Com esta configuração, os valores impressos pelo comando *imprima* dentro da repetição (inserida no retângulo) são os mostrados na tabela ao lado.

K	CA[VE[K]]	Valor Impresso
1	CA[VE[1]] = CA[1]	SEG
3	CA[VE[3]] = CA[2]	TER
5	CA[VE[5]] = CA[5]	SEX

O comando *imprima* após a repetição imprime a palavra "SEG", pois:

CA[VE[VE[3]]] = CA[VE[2]] = CA[1] = "SEG"

Exemplo 22: cálculo de notas de alunos.

Um professor tem uma turma de 80 alunos e deseja ler e imprimir as notas de cada aluno, juntamente com a média da turma, que é calculada.

Solução

Para ilustrar a utilização dos três comandos de repetição (*para*, *enquanto* e *repita*), tem-se os seguintes algoritmos:

Algoritmo utilizando o comando *enquanto*

```

início
  inteiro: K;
  real: Media, Soma, Nota[80];
  Soma ← 0;
  K ← 1;
  enquanto (K ≤ 80) faça
    início
      imprima ("Nota[",K,"]=");
      leia (Nota[K]);
      Soma ← Soma + Nota[K];
      K ← K + 1;
    fim;
  Media ← Soma/80;
  K ← 1;
  enquanto (K ≤ 80) faça
    início
      imprima (Nota[K]," (" ,Media,")");
      K ← K + 1;
    fim;
  fim.

```

Algoritmo utilizando o comando *repita*

```

início
  inteiro: K;
  real: Media, Soma, Nota[80];
  Soma ← 0;
  K ← 1;
  repita
    imprima ("Nota[",K,"]=");
    leia (Nota[K]);
    Soma ← Soma + Nota[K];
    K ← K + 1;
  até (K > 80);
  Media ← Soma/80;
  K ← 1;
  repita
    imprima (Nota[K]," (" ,Media,")");
    K ← K + 1;
  até (K > 80);
  fim.

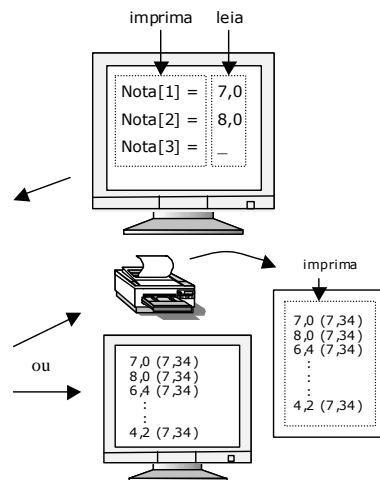
```

Algoritmo utilizando o comando *para*

```

início
  inteiro: K;
  real: Media, Soma, Nota[80];
  Soma ← 0;
  para K de 1 até 80
    início
      imprima ("Nota[",K,"]=");
      leia (Nota[K]);
      Soma ← Soma + Nota[K];
    fim;
  Media ← Soma/80;
  para K de 1 até 80
    imprima (Nota[K]," (" ,Media,")");
  fim.

```

**Exercício 25: alteração dos elementos de um vetor.**

Construa um algoritmo que leia um vetor qualquer com N elementos inteiros ($N \leq 10$), e altere seus elementos da seguinte forma:

- se o elemento for par, multiplique-o por 3;
- se o elemento for ímpar e positivo, multiplique-o por dois;

O vetor modificado deve ser impresso.

Solução

```

início
  inteiro: K, N, V[10];
  leia(N);
  para K de 1 até N
    início
      leia (V[K]);
      se (V[K] mod 2 = 0) então
        V[K] ← V[K]*3
      senão
        se (V[K] > 0) então V[K] ← V[K]*2;
    fim;
  para K de 1 até N
    imprima ("V[",K,"] = ", V[K]);
  fim.

```

Exercício 26: somatório dos elementos de um vetor.

Construa um algoritmo que leia um vetor qualquer com N elementos inteiros ($N \leq 10$) e faça o somatório dos elementos pares de ordem ímpar. O resultado deve ser impresso

Solução

```

início
  inteiro: S, K, N, V[10];
  leia(N);
  S ← 0;
  para K de 1 até N
    início
      leia (V[K]);
      se (V[K] mod 2 = 0) e (K mod 2 = 1) então
        S ← S + V[K];
    fim;
  imprima ("Soma = ", S);
  fim.

```

Exercício 27: múltiplos testes

Construa um algoritmo, em português estruturado, que leia um vetor com N elementos inteiros ($N \leq 20$) e calcule o seguinte:

- a média aritmética dos elementos de ordem (índice) par;
- a média aritmética dos elementos que são ímpares;
- a soma dos elementos positivos.

Os resultados devem ser impressos.

Solução

```

início
  inteiro: K, N, V[20], NEOP, NEIMP;
  real: SEOP, SEIMP, SEPOS;
  leia(N);
  NEOP ← 0;
  NEIMP ← 0;      { Colocar comentários }
  SEOP ← 0;
  SEIMP ← 0;
  SEPOS ← 0;
  para K de 1 até N
    início
      leia (V[K]);
      se (K mod 2 = 0) então
        início
          SEOP ← SEOP + V[K];
          NEOP ← NEOP + 1;
        fim;
      se (V[K] mod 2 = 1) então
        início
          SEIMP ← SEIMP + V[K];
          NEIMP ← NEIMP + 1;
        fim;
      se (V[K] > 0) então
        SEPOS ← SEPOS + V[K];
      fim;
    fim;
  imprima ("Méd. elem. ordem par = ", SEOP/NEOP);
  imprima ("Méd. elem. ímpares = ", SEIMP/NEIMP);
  imprima ("Soma elem. positivos = ", SEPOS);
fim.

```

5.1.3. Matrizes**Conceito**

Uma matriz é uma estrutura de dados homogênea onde cada elemento ou componente pode ser considerado como uma variável simples. Cada elemento da matriz é referenciado por dois índices, que definem sua posição relativa dentro da estrutura. Visualizando a matriz como uma estrutura bi-dimensional, o primeiro índice se refere à linha, e o segundo à coluna.

Para se entender melhor o conceito de matriz, seja o seguinte exemplo: um edifício com vários andares, vários apartamentos por andar. O edifício pode ser visualizado como sendo uma matriz onde cada apartamento corresponde a um componente dessa matriz. Esse componente (ou "variável") pode conter pessoas (ou "valores do tipo pessoa"), sendo identificado por um número (número ou "índice" do apartamento). O número de pessoas que um apartamento contém em um determinado momento pode ser entendido como sendo um "valor atribuído" àquele apartamento. Essa analogia é mostrada na [Figura 9](#).

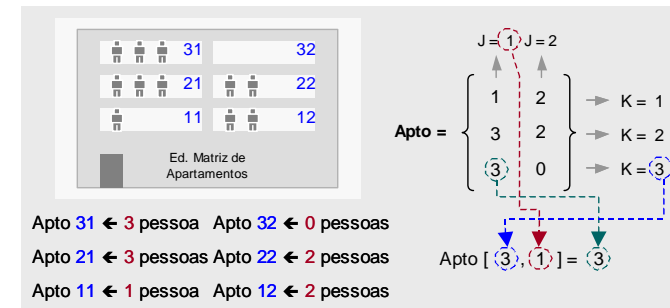


Figura 9: Visualização do conceito de matriz

```

{-- Este algoritmo lê uma matriz 3 x 2 --}

```

```

início
  inteiro: J, K, Apto[ 3, 2];
  { Leitura dos elementos da matriz }
  para K de 1 até 3
    para J de 1 até 2
      início
        {solicita elemento da matriz }
        imprima ("Apto[",K," ",J,"]:");
        {lê elemento da matriz }
        leia (Apto[K,J]);
      fim;
    fim;
  fim.

```

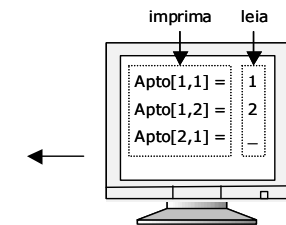


Figura 10: Algoritmo para leitura dos elementos de uma matriz.

Um algoritmo em português estruturado para atribuir valores a cada um dos componentes da matriz *Apto* mostrada no exemplo da Figura 9 da pode ser visto na [Figura 10](#).

Sintaxe

A sintaxe da declaração de vetores no português estruturado é mostrada na [Figura 11](#).

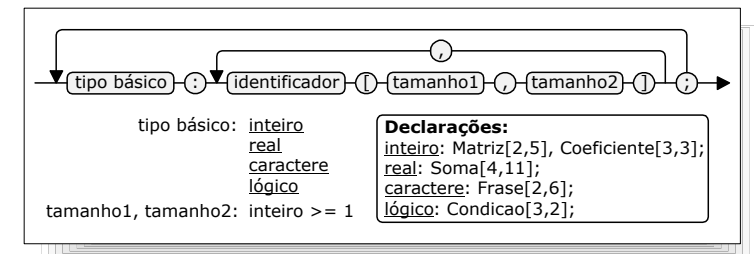


Figura 11: Exemplo da sintaxe para declaração de matrizes no português estruturado

No português estruturado e em algumas outras linguagens de programação a declaração de uma matriz e a atribuição de valores aos seus componentes pode ser feito conforme as sintaxes mostradas na [Figura 12](#), onde cada elemento da matriz recebe o valor da soma de seus índices.

PORT. ESTRUTURADO	FORTTRAN	C
<pre> início inteiro: I,J,Apto[4,5]; para I de 1 até 4 para J de 1 até 5 Apto[I,J] ← I+J; fim. </pre>	<pre> program Main integer i,j,apto(4,5) do 1 i=1,4 do 2 j=1,5 apto(i,j) = i+j continue continue stop end </pre>	<pre> void main() { int i, j, apto[4][5]; for (i=0; i<4; i++) for (j=0; j<5; j++) apto[i][j] = i+j; } </pre>
BASIC	PASCAL	
<pre> dim i as integer, j as integer dim apto(4,5) as integer for i = 1 to 4 for j = 1 to 5 apto(i,j)=i+j next j next i </pre>	<pre> program Main; var i,j: integer; apto: array [1..4,1..5] of integer; begin for i:=1 to 4 do for j:=1 to 5 do apto[i,j] := i+j; end. end. </pre>	

Figura 12: Declaração e uso de matrizes em diversas linguagens.

No algoritmo da Figura 12, cada componente da matriz *Apto* pode ser considerado como sendo uma variável simples do tipo inteiro. No português estruturado os índices de uma matriz iniciam-se em 1, ao passo que na linguagem C a primeira linha e a primeira coluna têm índices 0. Começar o índice de vetores ou matrizes em 0, em 1 ou em outro valor qualquer é convenção de cada linguagem, podendo ser alterado em algumas delas (como no Pascal), se o programador assim desejar.

Notação Simplificada

Ao trabalhar com matrizes no português estruturado, é possível utilizar a notação simplificada mostrada na Tabela 10. Esta notação é válida apenas para as situações mostradas nessa tabela, onde a variável Matriz aparece sozinha nos comandos *leia*, *imprima* e de atribuição.

Operação	Notação Normal	Notação simplificada
leitura da matriz m x n	<pre> para K de 1 até M para J de 1 até N leia (Matriz[K,J]); </pre>	<i>leia</i> (Matriz);
impressão da matriz	<pre> para K de 1 até M para J de 1 até N imprima (Matriz[K,J]); </pre>	<i>imprima</i> (Matriz);
inicialização de toda a matriz com o mesmo valor	<pre> para K de 1 até M para J de 1 até N Matriz[K,J] ← Y; </pre>	Matriz ← Y;

Tabela 10: Notação simplificada para comandos com matriz no português estruturado

Exemplo 23: utilização de matriz – atribuição de valores.

O que será impresso pelo algoritmo abaixo ?

```

início
  inteiro: J, K, M1[3,4];
  caractere: N1[2,2];
  J ← 2;
  para K de 1 até 3
    início
      M1[K,J] ← 2;
      M1[K,J+2] ← 2;
      M1[K,J-1] ← 1;
      M1[K,J+1] ← 1;
    fim;
  para K de 1 até 2
    para J de 1 até 2
      se (K=J) então
        N1[K,J] ← "A"
      senão
        N1[K,J] ← "Z";
    imprima (M1,N1);
fim.

```

Solução

As matrizes M1 e N1 serão:

M1:

	1	2	3	4
1	1	2	1	2
2	1	2	1	2
3	1	2	1	2

N1:

	1	2
1	A	Z
2	Z	A

Será impresso o seguinte:

1	2	1	2
1	2	1	2
1	2	1	2
A	Z		
Z	A		

Exercício 28: somatório dos elementos de uma matriz.

Construa um algoritmo que leia uma matriz N x M ($N, M \leq 10$) e faça o somatório dos elementos onde a soma dos índices seja par. O resultado desse somatório deve ser impresso.

Solução

```

início
  inteiro: S, I, J, N, M, Mat[10, 10];
  leia(N, M);
  S ← 0;
  para I de 1 até N
    para J de 1 até M
      leia (Mat[I, J]);
  para I de 1 até N
    para J de 1 até M
      se ((I+J) mod 2 = 0) então
        S ← S + Mat[I, J];
  imprima ("Soma = ", S);
fim.

```

```

início
  inteiro: S, I, J, N, M, Mat[10, 10];
  leia(N, M);
  S ← 0;
  para I de 1 até N
    para J de 1 até M
      leia (Mat[I, J]);
      se ((I+J) mod 2 = 0) então
        S ← S + Mat[I, J];
  fim
  imprima ("Soma = ", S);
fim.

```

Exercício 29: transposta de uma matriz.

Construa um algoritmo que obtenha a transposta de uma matriz 10 x 10 utilizando apenas uma variável do tipo matriz. A matriz lida e a sua transposta devem ser impressas.

Solução

```

início
  inteiro: Aux, I, J, Mat[10, 10];
  leia(Mat);
  imprima(Mat);
  para I de 2 até 10
    para J de 1 até i-1
      início
        Aux ← Mat[I, J];
        Mat[I, J] ← Mat[J, I];
        Mat[J, I] ← Aux;
      fim;
  imprima(Mat);
fim.

```

Exercício 30: somatório dos elementos de uma matriz.

Dada uma matriz MAT 5 x 4, construa um algoritmo que some os elementos de cada linha e guarde os resultados no vetor SOMALINHA. Em seguida, o algoritmo deve calcular o somatório de

todos os elementos da Matriz através da soma dos componentes do vetor SOMALINHA, guardando o resultado na variável TOTAL. O somatório de cada linha e o somatório total devem ser impressos. As expressões para cada um destes somatórios são as seguintes:

$$\text{SomaLinha}_k = \sum_{j=1}^4 \text{MAT}_{kj}, k=1,2,3,4,5 \quad \text{Total} = \sum_{k=1}^5 \text{SomaLinha}_k$$

Solução

```

início
  inteiro: J, K, TOTAL, SOMALINHA[5], MAT[5,4];
  SOMALINHA ← 0; { Cada elemento recebe o valor 0 }
  TOTAL ← 0;
  leia (MAT);
  para K de 1 até 5
    início
      para J de 1 até 4
        SOMALINHA[K] ← SOMALINHA[K] + MAT[K,J];
      imprima (SOMALINHA[K]);
      TOTAL ← TOTAL + SOMALINHA[K];
    fim;
  imprima (TOTAL);
fim.

```

Na figura ao lado é mostrado um exemplo da aplicação do algoritmo, onde a matriz MAT é inicializada com determinados valores, o total de cada linha é calculado, e o total de toda a matriz é obtido através da soma dos totais de cada linha.

	1	2	3	4		SomaLinha
1	1	6	1	-2	→	6
2	0	-1	0	2	→	1
3	1	4	1	4	→	10
4	-2	2	9	3	→	12
5	1	-4	1	2	→	0
						TOTAL: 29

Exercício 31: utilização de matriz – modificação dos elementos.

Construa um algoritmo para um programa que:

- leia uma matriz quadrada M_{ij} 5×5 ;
- divida cada elemento de uma linha pelo elemento da diagonal principal (onde $i = j$) se este for diferente de zero;
- faça cada elemento da linha igual a zero se o elemento da diagonal principal for também zero;
- imprima a matriz modificada.

Solução

```

início
  inteiro: I, J;
  real: Diagonal, M[5, 5];
  leia (M);
  para I de 1 até 5
    início
      Diagonal ← M[I, I];
      para J de 1 até 5
        se Diagonal = 0 então
          M[I, J] ← 0
        senão
          M[I, J] ← M[I, J]/Diagonal;
    fim;
  imprima (M);
fim.

```

	1	2	3	4	5
1	1	6	1	-2	-2
2	0	-1	0	2	2
3	1	4	2	4	4
4	-2	2	9	0	3
5	1	-4	1	2	-2

Matriz M lida

	1	2	3	4	5
1	1	6	1	-2	-2
2	0	1	0	-2	-2
3	0.5	2	1	2	2
4	0	0	0	0	0
5	-0.5	2	-0.5	-1	1

Matriz M modificada

Exercício 32: utilização de matriz – multiplicação de matrizes.

O mecanismo de multiplicação de uma matriz $A_{m \times p}$ por outra $B_{p \times n}$, fornecendo como resultado a matriz $C_{m \times n}$, é representado pela seguinte expressão:

$$C_{ij} = \sum_{k=1}^p A_{ik} * B_{kj}$$

$$i = 1, 2, \dots, m; \quad j = 1, 2, \dots, n$$

A figura ao lado mostra um exemplo de multiplicação de matrizes.

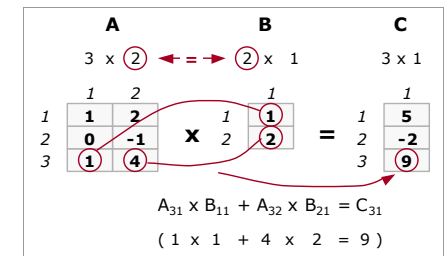
Construa o algoritmo para realizar a multiplicação de uma matriz genérica A $m \times p$ por outra B $p \times n$.

Solução

```

início
  inteiro: I, J, K, M, N, P;
  real: A[10,10], B[10,10], C[10,10];
  leia (M, N, P);
  { Lê a matriz A }
  para I de 1 até M
    para J de 1 até N
      início
        C[I,J] ← 0;
        para K de 1 até P
          C[I,J] ← C[I,J] + A[I,K]* B[K,J];
        fim;
  { Imprime o resultado }
  para I de 1 até M
    para J de 1 até N
      imprima (C[I,J]); { imprime a linha I }
  fim.

```



5.2. Estruturas de Dados Heterogêneas

5.2.1. Definição

No item anterior vimos um exemplo de um problema onde um professor necessitava calcular as médias de seus alunos a partir de duas notas bimestrais. Para melhor resolver esse tipo de problema foram introduzidas as estruturas de dados homogêneas, como os vetores, que permitiram agrupar em uma só variável composta (estrutura) um conjunto de variáveis simples do mesmo tipo (vetores Nota1, Nota2 e Média), facilitando a manipulação desses dados. O resultado foi o algoritmo mostrado na [Figura 5](#).

Vamos imaginar agora que o problema passe a ser calcular a média anual dos alunos, onde quatro (4) médias bimestrais estão envolvidas. Além disso, também será determinada a situação do aluno conforme a sua média:

- se média < 5.0: aluno reprovado - "Reprovado";
- se $5.0 \leq \text{média} < 7.0$: aluno fará exame final - "Exame final";
- se $7.0 \leq \text{média}$: aluno aprovado - "Aprovado";

Note que nesse problema estão envolvidas variáveis de tipos diferentes (real e caractere), associadas a um determinado aluno, definido pelo seu nome. Seria conveniente se pudéssemos agrupar todas as informações de determinado aluno em um mesmo local, como numa ficha de cadastro. Isso é possível através da utilização de *Registros, que são entidades que permitem agrupar dados de diferentes tipos, referenciados como campos, em uma mesma estrutura*.

5.2.2. Registros

Conceito

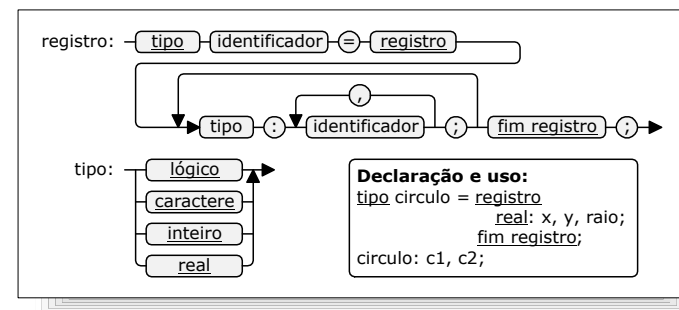
Um registro é uma estrutura de dados heterogênea composta por diversos elementos - os campos (variáveis) - que podem ser de diferentes tipos. Voltando ao problema descrito no item anterior, a estrutura de dados desejada para conter os dados de determinado aluno seria algo como mostrado na [Figura 13](#).

[Figura 13](#): Conjunto de registros de alunos em um cadastro eletrônico de notas escolares.

No exemplo acima, todos os dados de um determinado aluno estão agrupados em uma mesma ficha, representando o seu registro. Para consultar os dados de um determinado aluno, basta localizar a sua ficha ou registro no cadastro de alunos.

Declaração de Registros

A sintaxe da declaração de uma variável do tipo registro é mostrada na [Figura 14](#). Normalmente é definido um tipo criado pelo próprio usuário com a estrutura desejada para o registro, para depois se declarar as variáveis utilizando esse tipo. Exemplos de registros em algumas linguagens podem ser vistos na [Figura 15](#). As variáveis declaradas como componentes do registro são denominadas *campos* do registro.



[Figura 14](#): Sintaxe da declaração de registros. Note-se que primeiro se cria um novo tipo de variável registro (tipo circulo), para depois utilizar esse tipo criado para declarar as variáveis desse tipo de registro (c1 e c2).

PORT. ESTRUTURADO	C
<pre> tipo RegistroFuncionario = registro inteiro: ID; caractere: Nome, Endereco; inteiro: Telefone; caractere: DataDeAdmissao; fim registro; RegistroFuncionario: empregado; </pre>	<pre> struct registro_funcionario { int id; char nome[20]; char endereco[30]; long int telefone; char data_de_admissao[11]; } struct registro_funcionario empregado; </pre>
VISUAL BASIC	PASCAL
<pre> Type RegistroFuncionario ID As Integer Nome As String * 20 Endereco As String * 30 Telefone As Long DataDeAdmissao As Date End Type Dim Empregado AS RegistroFuncionario </pre>	<pre> Type RegistroFuncionario = record ID: integer; Nome: string[20]; Endereco: string[30]; Telefone: longint; DataDeAdmissao: string; end; Empregado: RegistroFuncionario; </pre>

[Figura 15](#): Declaração e uso de registros em diversas linguagens de programação²⁹.

Exercício 33: declaração de registro.

Declare, em português estruturado, a variável Aluno como sendo do tipo registro correspondente à ficha definida na [Figura 13](#).

Solução

```

tipo RegistroAluno = registro
  caractere: Nome;
  real: nota1, nota2, nota3, nota4, media_anual;
  caractere: Situação;
fim registro;

RegistroAluno: Aluno;

```

²⁹ O FORTRAN não disponibiliza tal tipo de estrutura de dados.

Trabalhando com os Campos do Registro

Quando se deseja definir o conteúdo de um campo específico de um registro, esse campo pode ser referenciado através da notação *variável_registro.campo*, como exemplificado na [Figura 16](#).

```
{ Esse algoritmo calcula a distância entre dois pontos no plano}
início
    tipo ponto =registro
        real: x, y;
        fim registro;

    ponto: P1, P2;
    real: distancia;
    leia (P1, P2); { Notação simplificada equivalente a leia(P1.x,P1.y,P2.x,P2.y) }
    distancia ← raiz((P1.x - P2.x)^2+(P1.y - P2.y)^2);
    imprima("Distância entre P1 e P2: ", distancia);
fim.
```

Figura 16: Manipulação de um registro.

Exercício 34: manipulando registros (1).

Baseando-se na Figura 16, construa um algoritmo que calcule a distância de dois pontos no espaço. Para tal, defina os pontos através de registros.

Solução

```
{ Esse algoritmo calcula a distância entre dois pontos no espaço}
início
    tipo ponto =registro
        real: x, y, z;
        fim registro;

    ponto: P1, P2;
    real: distancia;
    leia (P1, P2);
    distancia ← raiz((P1.x - P2.x)^2+(P1.y - P2.y)^2+(P1.z - P2.z)^2);
    imprima("Distância entre P1 e P2: ", distancia);
fim.
```

Exercício 35: manipulando registros (2).

Construa um algoritmo que, dada as coordenadas espaciais (x,y,z) dos centros de duas esferas e os respectivos raios, verifique se existe alguma intersecção entre elas. Para tal, defina as esferas utilizando registros com os campos que forem necessários.

Solução

```
{ Esse algoritmo verifica se há intersecção entre duas esferas }
início
    tipo esfera = registro
        real: x, y, z, r;
        fim registro;

    ponto: E1, E2;
    real: distancia;
    leia (E1, E2);
    distancia ← raiz((E1.x - E2.x)^2+(E1.y - E2.y)^2+(E1.z - E2.z)^2) - (E1.r + E2.r);
    se (distancia <= 0) então
        imprima("Há intersecção")
    senão
        imprima("Não há intersecção");
fim.
```

Um exemplo na Linguagem C

A estrutura é o equivalente em C de um registro. Para se criar uma estrutura, utiliza-se o comando *struct*. Sua forma geral é mostrada na [Figura 17](#). O *nome_da_estrutura* é equivalente a

um nome de um tipo, nesse caso um tipo criado pelo usuário. A *<lista de variáveis_estrutura>* é opcional, e contém nomes de variáveis declaradas como sendo do tipo *nome_da_estrutura*.

Sintaxe	Exemplo
<pre>struct nome_estrutura { tipo_1 variável_1; tipo_1 variável_1; ... tipo_n variável_n; } <lista1 variáveis_estrutura>; struct nome_estrutura <lista2 variáveis_estrutura>;</pre>	<pre>struct tipo_endereco { char rua[50]; int numero; char bairro[20]; char cidade[30]; char sigla_estado[3]; long int CEP; } empresa; struct tipo_endereco residencia;</pre>

Figura 17: Exemplo de uma estrutura no C.

Na [Figura 18](#) tem-se um programa em C que utiliza estruturas para preencher uma ficha. O programa declara uma variável *ficha* do tipo *ficha_pessoal* e preenche os seus dados. O exemplo mostra como podemos acessar um elemento de uma estrutura (basta usar o ponto: *.*), e mostra que uma estrutura pode ter como campo uma outra estrutura (campo *endereco* na estrutura *ficha_pessoal*).

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

struct tipo_endereco {
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
};

struct ficha_pessoal {
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;
};

void main (void) {
    struct ficha_pessoal ficha;
    strcpy (ficha.nome,"Luiz Osvaldo Silva");
    ficha.telefone=4921234;
    strcpy (ficha.endereco.rua,"Rua das Flores");
    ficha.endereco.numero=10;
    strcpy (ficha.endereco.bairro,"Cidade Velha");
    strcpy (ficha.endereco.cidade,"Belo Horizonte");
    strcpy (ficha.endereco.sigla_estado,"MG");
    ficha.endereco.CEP=31340230;
}
```

Figura 18: Manipulação de registros/estruturas na linguagem C.

Matrizes de Registros

Um registro é como qualquer outro tipo de dado. Podemos, portanto, definir matrizes de registros. Para exemplificar, seja o seguinte problema: calcular a distância percorrida por uma partícula que passa pelos *n* pontos do plano p_1, \dots, p_n , nessa ordem, sempre percorrendo a distância entre dois pontos em linha reta, como sugerido na [Figura 19](#).

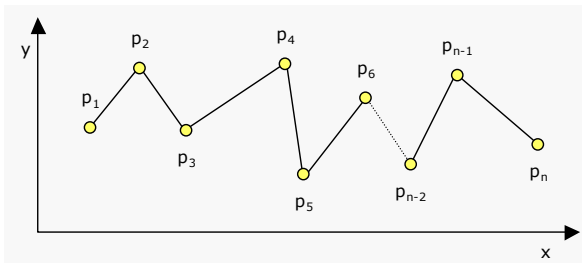


Figura 19: Trajetória a ser percorrida por uma partícula: qual é a distância total do percurso?

O algoritmo em português estruturado para implementar a solução do problema exposto na Figura 19 é o da Figura 20.

```

início
  tipo ponto = registro
    real: x, y;
  fim registro;

  ponto: P[100];
  real: distancia;
  inteiro: k, n;
  leia (n); {n <= 100}
  para k de 1 até n
    leia (P[k]);
  distancia ← 0;
  para k de 1 até n-1
    distancia ← distancia + raiz( (P[k+1].x - P[k].x)^2 + (P[k+1].y - P[k].y)^2 );
  imprima("Distância percorrida entre P1 e PN: ", distancia);
fim.

```

Figura 20: Algoritmo que implementa a solução do problema exposto na Figura 19.

Exercício 36: matriz de registros.

Construa o algoritmo, em português estruturado, para calcular a distância percorrida por uma partícula que passa pelos n pontos do espaço p_1, \dots, p_n , $n \leq 10$, nessa ordem, sempre percorrendo

a distância entre dois pontos em linha reta. Determine também entre quais pontos está o menor e o maior trecho percorrido (é um problema semelhante ao exposto na Figura 19).

Solução

```

início
  tipo ponto = registro
    real: x, y, z;
  fim registro;

  ponto: P[100];
  real: distancia, t, tmaior, tmenor, kmaior, kmenor;
  inteiro: k, n;
  leia (n); {n <= 100}
  para k de 1 até n
    leia (P[k]);
  distancia ← 0;
  para k de 1 até n-1
    início
      t ← raiz((P[k+1].x - P[k].x)^2 + (P[k+1].y - P[k].y)^2 + (P[k+1].z - P[k].z)^2);
      se k=1 então
        início
          tmenor = t;
          tmaior = t;
          kmenor = 1;
          kmaior = 1;
        fim;
      senão se t > tmaior então
        início
          tmaior = t;
          kmaior = k;
        fim;
      senão se t < tmenor então
        início
          tmenor = t;
          kmenor = k;
        fim;
      distancia ← distancia + t;
    fim;
  imprima("Distância total entre P1 e PN: ", distancia);
  imprima("Menor trecho: ", tmenor, " (entre os pontos "; kmenor; " e "; kmenor+1, ")");
  imprima("Maior trecho: ", tmaior, " (entre os pontos "; kmaior; " e "; kmaior+1, ")");
fim.

```

Referências do Capítulo

- [17] Forbellone, A. L. V.; Eberspächer, H. F.: "Lógica de Programação - A Construção de Algoritmos e Estruturas de Dados", Makron Books do Brasil Editora Ltda, São Paulo, Brasil, 1993.
- [18] Guimarães, A. M. e Lages, N. A. C.: "Algoritmos e Estruturas de Dados", Livros Técnicos e Científicos Editora S.A, Rio de Janeiro, Brasil, 1994.
- [19] Hehl, M. E.: "Linguagem de Programação Estruturada FORTRAN 77", Editora McGraw-Hill Ltda, São Paulo, Brasil, 1986.
- [20] Gottfried, B. S.: "Programação em Pascal", Editora McGraw-Hill de Portugal Lda, Lisboa, Portugal, 1988.
- [21] Kernighan, B. W. e Ritchie, D. M.: "C - A Linguagem de Programação", Editora Campus Ltda, Rio de Janeiro, Brasil, 1986.
- [22] Função Ajuda dos Compiladores Fortran Force 2.0, Visual Basic (Microsoft) e Rapid-Q Basic.

6. Ponteiros e Alocação Dinâmica de Memória

Nota ⇒ Neste capítulo será tratado o assunto Ponteiros utilizando-se apenas exemplos na Linguagem C, visto que ela os utiliza largamente, deixando-se de lado as construções equivalentes em português estruturado. Construções equivalentes existem em algumas outras linguagens, como a linguagem Pascal. Portanto, se o leitor não estiver trabalhando com a Linguagem C/C++, ele pode deixar este capítulo de lado.

6.1. Ponteiros

6.1.1. Conceito

Definição

Ponteiro é uma variável que contém o endereço de outra variável. Também é chamado de apontador em algumas referências.

Como Funcionam

Variáveis do tipo *int* guardam inteiros. Variáveis do tipo *float* ou *double* guardam números reais (ou ponto flutuante); variáveis *char* guardam caracteres; ponteiros guardam endereços de memória.

Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é o pedaço de papel onde o endereço foi anotado. Tendo esse endereço anotado, você pode depois utilizá-lo para encontrar seu amigo. O C funciona de forma semelhante. Pode-se anotar o endereço de qualquer variável numa variável do tipo ponteiro para posteriormente utilizá-la. Da mesma maneira, uma agenda onde são guardados endereços de vários amigos poderia ser vista como sendo uma matriz de ponteiros no C. Outra analogia: uma gaveta em determinado armário contendo uma anotação indicando em que outra gaveta do mesmo armário se encontra determinado objeto é também um ponteiro, pois "aponta" para o local onde o objeto está guardado, como exemplificado na [Figura 21](#) e na [Figura 22](#).

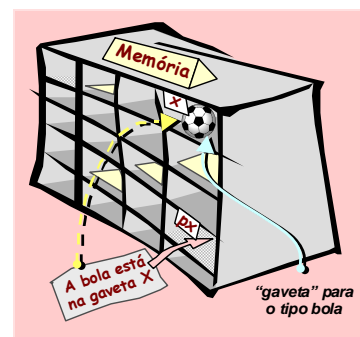


Figura 21: O ponteiro (px) guarda o "endereço" onde se pode encontrar determinado objeto (no caso, o endereço onde está a "bola").

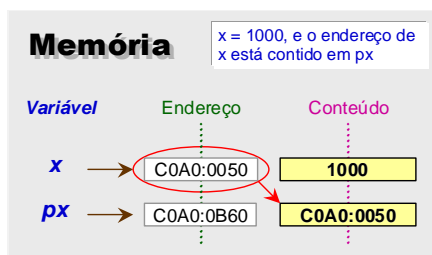


Figura 22: Significado do ponteiro na memória do computador.

Um ponteiro também tem tipo. Quando você anota um endereço de um amigo, você o trata diferente de quando você anota o endereço de uma empresa. Apesar do endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.), eles indicam locais cujos conteúdos são diferentes: os dois endereços são ponteiros de tipos diferentes. No C, quando se declara um ponteiro, informa-se ao compilador para que tipo de variável ele vai apontar. Por exemplo, um ponteiro *int* aponta para um inteiro, isto é, guarda o endereço de uma variável inteira. Um ponteiro *double* aponta para uma variável real de dupla precisão.

Ponteiros são muito utilizados, pois em muitas situações são o único recurso para expressar determinada computação, como:

- construção de listas encadeadas;
- alteração de argumentos de funções (que são passados por valor);
- aumento da eficiência do código em determinadas situações, como operações com arranjos;
- passagem de funções como argumentos de outras funções;
- realizar acesso direto a posições de memória específicas em determinados hardwares, etc;
- otimização do uso de memória (alocação dinâmica).

6.1.2. Declaração

A forma geral para a declaração de um ponteiro no C é a seguinte:

*tipo_do_ponteiro *nome_da_variável;*

É o asterisco (*) colocado antes do nome da variável que indica ao compilador que aquela variável não vai guardar um valor de determinado tipo, mas sim um endereço para uma variável daquele tipo. Exemplos:

```
int *pt; // pt apontará para uma variável do tipo int
char *temp, *pt2; // temp e pt2 apontarão para variáveis do tipo char
```

Exercício 37: declaração de ponteiros.

Declare, na sintaxe do C, as variáveis inteiras k e j, as variáveis reais x e y com precisão simples e a variável real z com precisão dupla. Na sequência, declare ponteiros para essas variáveis, nomeando-os com o mesmo nome das variáveis para as quais apontam, acrescidos da letra p na frente.

Solução

```
int k, j, *pk, *pj;
float x, y, *px, *py;
double z, *pz;
```

6.1.3. Ponteiros e Endereços

Operadores

Uma vez que o ponteiro contém o endereço de uma variável, esta pode ser manipulada indiretamente através do ponteiro. Para tal, os operadores descritos a seguir são utilizados.

Operador Unário & ("endereço de")

Esse operador retorna o endereço do operando, que só pode ser uma variável ou um elemento de um arranjo³⁰. Se px é o ponteiro para a variável x, então:

`px = &x;`

atribui o endereço de x à variável ponteiro px, como exemplificado na [Figura 23](#) (veja também o item [Inicialização de Ponteiros](#)).

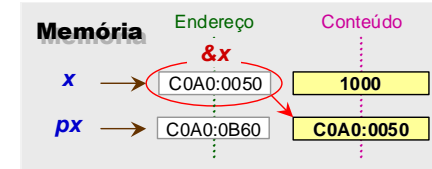


Figura 23: Significado do operador &.

Operador Unário * ("referência a")

Este operador trata seu operando como um endereço, e o utiliza para buscar o conteúdo da variável para qual aponta. Se y possui o mesmo tipo que a variável x, então

```
y = *px;
```

atribui a y o conteúdo da variável para a qual px aponta, que é a variável x, conforme ilustrado na [Figura 24](#).

O operador * é também chamado de operador de indireção.

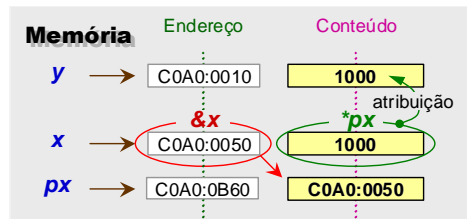


Figura 24: Significado do operador de indireção * - *px é o conteúdo de x.



O Símbolo "*"

Apesar do símbolo ser o mesmo, o operador * (multiplicação) nada tem a ver com o operador * (referência de ponteiros): além de representarem operações completamente distintas, o primeiro é binário, ao passo que o segundo é unário pré-fixado (antes do operando).

Inicialização de Ponteiros

Sejam os exemplos de declaração abaixo:

```
int *pt;
char *temp, *pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro, e o segundo declara dois ponteiros para caracteres. Como acontece a toda variável do C, a declaração apenas reserva um espaço na memória para guardar o conteúdo, mas o conteúdo em si continua desconhecido. Isto significa que esses ponteiros apontam para um lugar qualquer. Este lugar pode estar, por exemplo, na região da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas circunstâncias pode levar a um travamento do micro, ou a algo pior. O ponteiro deve ser sempre inicializado (ser apontado para algum lugar conhecido) antes de ser usado. Isto é de extrema importância !

Para atribuir um valor a um ponteiro recém-criado bastaria atribuir a ele um endereço de memória. O problema está em saber de antemão que valor poderia ser esse. É muito difícil saber o endereço de cada variável que usamos, mesmo porque esses endereços são determinados pelo compilador na hora da compilação e alocados durante a execução do programa. É melhor deixar que o compilador faça este trabalho por nós. Isso é feito, por exemplo, ao se utilizar o operador & apresentado acima. Assim sendo, a inicialização de um ponteiro poderia ser como mostrado abaixo:

```
int *pk, k;
pk = &k // pk passa a apontar para o endereço de k, definido
        // pelo próprio compilador
```

Ponteiro para Ponteiro

Criar um ponteiro para um ponteiro é como anotar o endereço de um papel que tem o endereço da casa de um amigo. Pode-se declarar um ponteiro para um ponteiro da seguinte forma:

```
tipo_do_ponteiro **nome_da_variável;
```

Exemplos:

```
int **pi; // ponteiro p/um ponteiro p/uma variável do tipo int
char **pc; // ponteiro p/um ponteiro p/uma variável do tipo char
```

Para manipular o conteúdo de um ponteiro para um ponteiro, utiliza-se a indireção múltipla:

```
int x, *px, **ppx;
px = &x;
ppx = &px;
**ppx = 2; // equivale a x = 2; ** significa "dupla indireção"
```

Ainda é possível declarar um ponteiro para ponteiro para ponteiro, ou então um ponteiro para ponteiro para ponteiro para ponteiro, e assim por diante. A lógica é a mesma que a apresentada no exemplo acima. A utilização desse recurso, entretanto, restringe-se a casos muitos específicos ou especiais.

Advertência

Ponteiros podem proporcionar grandes soluções para diversos problemas, porém se usados sem disciplina e cuidado podem ser uma excelente forma de criar programas impossíveis de entender. Além disso, a criação de ponteiros que apontem para lugar algum (ponteiro não inicializado) gera erros por vezes difíceis de identificar, com efeitos aleatórios e imprevisíveis.

Como já citado no item *Inicialização de Ponteiros*, o principal cuidado ao se usar um ponteiro é saber sempre para onde ele está apontando. Ou seja, nunca utilize um ponteiro sem que ele tenha sido inicializado. O pequeno programa abaixo é um exemplo de como **não** usar um ponteiro:

```
void main () { /* Programa incorreto */
int x,*p;
x=13;
*p=x; /* para onde aponta p ? */
}
```

O que vai acontecer ao executar esse programa é imprevisível. O ponteiro p está apontando para qualquer lugar da memória, e o número 13 é gravado em um lugar desconhecido. Com um número apenas talvez não se observe nenhum problema. Porém, se vários números forem gravados em posições aleatórias na memória, não vai demorar muito para ocorrer algum problema com o micro.

Exemplo 24: uso dos operadores unários "*" e "&"(1).

A sequência

```
px = &x;
y = *px;
```

atribui a y o mesmo valor atribuído no comando

```
y = x;
```

Para a construção acima, deve-se declarar as variáveis da seguinte maneira

```
int x, y;
int *px; /* significa que *px é equivalente a um int, e px um
          ponteiro para int */
```

Exemplo 25: uso dos operadores unários "*" e "&"(2).

```
int x, *px; // declaração
px = &x; // inicializa o ponteiro com o endereço de x
*px = 0; // atribui 0 a x, sendo equivalente a x = 0
y = *px + 1; // equivalente a y = x + 1;
*px+=1; // equivalente a x+=1 (ou x=x+1;)
(*px)++; // equivalente a x++
          //(veja item Incremento e Decremento, página 117);
```

Exemplo 26: uso dos operadores unários "*" e "&"(3).

Seja a sequência abaixo:

```
int count=10;
int *pt;
pt=&count;
```

Foi criado um inteiro count com o valor 10 e um apontador para um inteiro pt. A expressão &count representa o endereço de count, o qual é armazenado em pt. Note que não foi alterado o valor de count, que continua valendo 10. Como foi colocado um endereço em pt, ele está agora "liberado" para ser usado. Pode-se, por exemplo, alterar o valor de count usando pt. Para tanto utiliza-se o operador de indireção *. No exemplo acima, uma vez que fizemos pt = &count a expressão *pt é equivalente ao próprio count. Isto significa que, se quisermos mudar o valor de count para 12, basta fazer *pt=12.

Exercício 38: uso dos operadores unários "*" e "&"(1).

O programa abaixo imprime uma série de valores.

```
#include <stdio.h>
void main(){
    int num, valor;
    int *p;
    num=55;
    p=&num;
    valor=*p;
    printf ("Valor1 = %d\n",valor);
    printf ("Valor da variavel apontada = %d\n",*p);
    *p=*p+3;
    printf ("Valor2 = %d\n",*p);
    (*p)--;
    printf ("Valor3 = %d\n",*p);
    valor = num - *p;
    printf ("Valor4 = %d\n",valor);
}
```

Complete com os valores corretos as lacunas referentes ao resultado do programa acima:

```
Valor1 =      55
Valor da variavel apontada =      55
Valor2 =      58
Valor3 =      57
Valor4 =      0
```

Exercício 39: uso dos operadores unários "*" e "&"(2).

```
#include <stdio.h>

void main(){
    int x, *px, **ppx, **ppx2;
    px = &x;
    ppx = &px;
    *px = 2;
    printf("\nres = %d",**ppx);
    **ppx=3;
    printf("\nres = %d",*px);
    ppx2=ppx; //apontam p/o mesmo local
    *px = **ppx2 + x;
    printf("\nres = %d\n\n",x);
}
```

Complete as lacunas com o valor que será impresso por cada um dos comandos printf do programa ao lado.

res= ____ (2)

res= ____ (3)

res= ____ (6)

Exercício 40: uso dos operadores unários "*" e "&"(3).

Construa um programa em C que faça o seguinte somatório:

$$S = 1 + 1/2 + 1/3 + 1/4 + \dots + 1/N$$

Para implementá-lo, defina as variáveis que normalmente seriam utilizadas (k, n e s) e os ponteiros para essas variáveis (pk, pn e ps). Feito isso, o restante do algoritmo deve ser montado utilizando apenas os ponteiros ao invés da variáveis.

Solução

```
#include <stdio.h>
void main(){
    int k, n; // contador e número de termos
    int *pk, *pn; // ponteiros para k e n
    double s, *ps; // somatório e seu ponteiro

    pk = &k;
    pn = &n;
    ps = &s;

    // Complete o programa

    printf("Entre com n: ");
    scanf ("%d",pn);
    *ps = 0;

    for(*pk=1; *pk<=*pn; (*pk)++)
        *ps+=1.0/(*pk);

    printf("\nsoma para n= %d , %lf\n",*pn,*ps);
}
```

6.1.4. Operações

O conhecimento de algumas operações possíveis com ponteiros é útil para a solução de determinados problemas e para manipulação de algumas estruturas de dados como os arranjos (vetores e matrizes). Veremos algumas dessas operações a seguir.

Igualdade

Dados dois ponteiros p1 e p2, pode-se igualá-los fazendo a seguinte atribuição:

```
p2 = p1;
```

Isso significa que p2 passará a apontar para o mesmo lugar que p1 após essa atribuição (veja ppx2 no [Exercício 39](#)). Caso se queira que a variável apontada por p2 tenha o mesmo conteúdo da variável apontada por p1 (veja [Figura 25](#)), deve-se fazer :

```
*p2 = *p1;
```

Incremento e Decremento

As operações de incremento e o decremento também são aplicáveis a ponteiros. Ao incrementar-se um ponteiro ele passa a apontar para o endereço seguinte ao da variável para a qual apontava originalmente. Nesse caso, o tipo do ponteiro determina o incremento: ao incrementar um ponteiro para *int*, ele "anda" 2 bytes na memória (tamanho de uma variável tipo *int* é 2 bytes); ao incrementar um ponteiro para um tipo *double*, ele "anda" 8 bytes na memória. O decremento funciona de forma semelhante, porém no sentido inverso. Supondo que p é um ponteiro, as operações são escritas como p++ (incremento) e p-- (decremento).

Exemplo:

```
int x, *px;
px = &x;
px++;      /* px passa a apontar para um endereço 2 bytes após o
           endereço de x */
```

Atenção! Estamos tratando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Para incrementar o conteúdo da variável apontada pelo ponteiro, utiliza-se a operação (*p)++. Exemplo:

```
int x=2, *px;
px = &x;
(*px)++;    // x passa a valer 3 após esse comando
```

Adição e Subtração

Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15 unidades ("andar" 15 posições na memória "para frente"). Basta fazer:

```
p = p + 15;
```

Para usar o conteúdo do ponteiro 15 posições adiante sem alterar p, pode-se referenciá-lo da seguinte forma (supondo x como *int* e p um ponteiro para *int*):

```
x = *(p + 15);
```

A subtração tem o mesmo mecanismo. Para pegar o conteúdo duas posições antes da posição apontada por p, deve-se fazer:

```
x = *(p - 2);
```

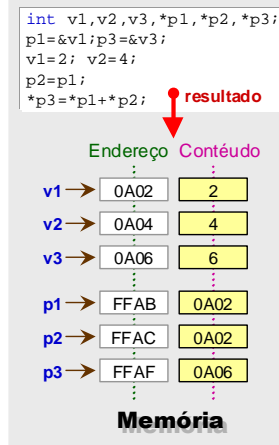


Figura 25: Igualdade entre ponteiros e conteúdos apontados por ponteiros.

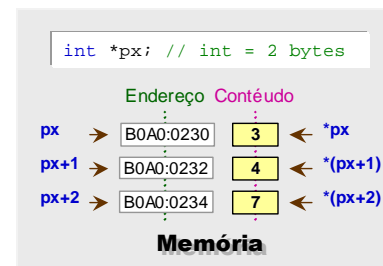


Figura 26: Aritmética de ponteiros – a soma de um ponteiro com um número inteiro *n* desloca o ponteiro *n* posições adiante na memória, sendo cada deslocamento equivalente ao tamanho do tipo da variável apontada.

Comparação

Outra operação eventualmente útil é a comparação entre dois ponteiros. Os operadores de comparação podem verificar se ponteiros são iguais ou diferentes (== e !=) ou verificar qual ponteiro aponta para uma posição mais alta *na memória* (>, <, >= e <=). A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer: p1>p2. Exemplo:

```
int *p1, *p2;
scanf("%i %i", p1, p2);
if (p1 == p2)
    printf("Mesma posicao na memoria.\n");
else
    printf("Posicoes diferente na memoria.\n");
```

Operações Não Aplicáveis

Há operações que *não* podem ser efetuados em ponteiros. Não se pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair *float* ou *doubles* de ponteiros.

6.1.5. Ponteiros e Argumentos de Funções

No C, parâmetros são sempre passados para funções através do mecanismo conhecido como "passagem por valor": os argumentos representam apenas cópias dos parâmetros passados, e assim sendo estes não podem ser alterados pela função chamada.

(1) Não funciona	(2) Funciona
<pre>: troca(a, b); : void troca(int x, int y){ int temp; temp =x; x = y; y = temp; }</pre>	<pre>: troca(&a, &b); : void troca(int *x, int *y){ int temp; temp =*x; *x = *y; *y = temp; }</pre>

Figura 27: Passagem de parâmetros por valor e por referência no C.

Para situações em que é necessário alterar os valores de argumentos, a solução é passar ponteiros como parâmetros. Dessa forma, a função que recebe um ponteiro para determinada variável terá condições de alterar o valor do conteúdo do endereço referente à essa variável. O exemplo a abaixo ilustra esse mecanismo através da função troca(), definida para trocar os conteúdos das duas variáveis passadas como parâmetro.

A opção (1) não funciona, pois neste caso a função troca() manipula apenas cópias de a e b (colocadas em x e y), não sendo possível alterar essas variáveis; isso é ilustrado na [Figura 28](#).

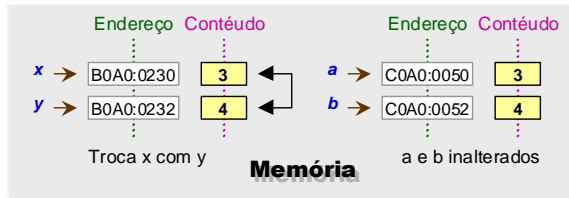


Figura 28: Passagem de parâmetros por valor na função troca() - não há alteração das variáveis a e b.

A opção (2) trabalha com ponteiros, recebendo os endereços das variáveis a e b, o que possibilita alterar o conteúdo destes endereços. O resultado final é a troca efetiva dos conteúdos dessas variáveis, como ilustrado na Figura 29.

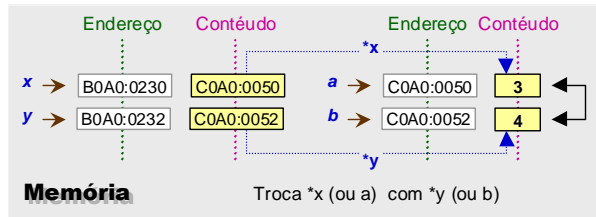


Figura 29: Passagem de parâmetros "por referência"³¹ através de ponteiros na função troca() - é possível o acesso e a modificação dos conteúdos das variáveis a e b.

Exercício 41: ponteiro como argumento de função.

Construa um programa que imprima os N primeiros termos de uma progressão aritmética de razão K e primeiro termo a₁. Para imprimir essa P.A., utilize sempre uma variável x, cujo primeiro valor será a₁, e cujos valores subsequentes devem ser obtidos através da chamada à função inc(*y, m) (sem retorno - void) que incrementa o valor da variável inteira apontada por y em m unidades.

Solução

```
#include <stdio.h>

void inc(int *y, int m){
    (*y)+=m;
}

main(){
    int al,j,k,n,x;
    printf("Entre com a1, k e n:");
    scanf("%d,%d,%d", &al, &k, &n);
    x = al;
    for(j=1; j<=n; j++){
        print("%d ",x);
        inc(&x, k);
    }
}
```

³¹ É por valor ainda, mas como esse "valor" é uma referência...

6.1.6. Ponteiros e Arranjos

Relação

Em C, ponteiros e arranjos (ou arrays) tem um estreito relacionamento. Operações com índices em arranjos podem ser feitas através de ponteiros, sendo, em geral, mais rápidas.

Seja a declaração:

```
int a[10];
```

Se pa é um ponteiro para inteiro, declarado como:

```
int *pa;
```

então a atribuição

```
pa = &a[i]; // 0 <= i < 10
```

faz com que pa aponte para o i-ésimo elemento de a (isto é, contém o endereço de a[i]).

Armazenamento na Memória

Quando um arranjo é declarado na seguinte forma:

```
tipo_da_variável *nome_da_variável [tam1][tam2] ... [tamN];
```

o compilador C calcula o tamanho, em bytes, necessário para armazenar esse arranjo. Esse tamanho é:

(tam1 x tam2 x ... x tamN) x tamanho_do_tipo

O compilador então aloca (reserva) essa quantidade de bytes em um espaço livre de memória. O nome da variável declarada é na verdade um ponteiro para o tipo da variável do arranjo, e aponta para o primeiro elemento do arranjo.

Vetores e Ponteiros

Para vetores, a notação

nome_da_variável [índice]

é absolutamente equivalente a:

*(nome_da_variável + índice)

Dessa equivalência vem a explicação do porquê, na linguagem C, da indexação começar no número zero. Num vetor vet, o primeiro elemento está em *(vet + 0), que é o equivalente a vet[0]. Ou seja: devemos ter o índice do primeiro elemento do vetor igual a zero.

Exemplo 27: vetores e ponteiros (1).

O programa abaixo define um vetor com três elementos e o imprime de duas formas: referenciando cada elemento através da notação vetorial (a[i]) e através do uso de ponteiros (*(pa+i)).

```
#include <stdio.h>

#define N 3 /* Define o simbolo N como sendo 3 */

void main() {
    int i, *pa, a[N];
    for(i=0;i<N;i++)
        a[i] = i+1;
    printf("\nImpressao do vetor:");
    for(i=0;i<N;i++)
        printf("\n%3d",a[i]);
    pa = &a[0]; /* aponta p/ o inicio do vetor */
    printf("\nImpressao do vetor por ponteiro:");
    for(i=0;i<N;i++)
        printf("\n%3d",*(pa+i));
}
```

Impressao do vetor:

```
1
2
3
Impressao do vetor por ponteiro:
1
2
3
```

← RESULTADO DO PROGRAMA:

Note a equivalência:

$a[i] \Leftrightarrow *(pa+i)$, com $pa=&a[0]$

Exemplo 28: vetores e ponteiros (2).

O programa abaixo cria um vetor **a** com 3 elementos inteiros ($a = \{1, 2, 3\}^T$), e imprime uma tabela com o endereço na memória e o conteúdo de cada um desses elementos. O endereço é um número na base hexadecimal³²: dígitos 0,...,9,A,...,F}. Como cada elemento ocupa 2 bytes, esse é o deslocamento na memória da posição de um elemento para outro. A formatação %p no último printf é utilizada para imprimir endereços.

³² Nós utilizamos normalmente a base decimal (10 dígitos: 0,1,...,9) na matemática, sendo uma coisa "natural" para nós. Para o computador, é "natural" trabalhar na base hexadecimal ($16 = 2^4$ dígitos), pois ela está relacionado com a base binária (0 ou 1), que é a informação básica armazenada/tratada por um computador.

```
#include <stdio.h>

#define N 3

void main() {

    int i, *p, *pa, a[N]; /* o tipo int ocupa 2 bytes na memória */

    for(i=0;i<N;i++)
        a[i] = i+1;

    pa = &a[0]; /* aponta p/ o inicio do vetor */
    printf("variavel  endereco  valor\n");
    printf("-----\n");
    for(i=0;i<N;i++) {
        p = pa+i; /* p aponta para a[i] */
        printf(" a[%d]      %p      %d  \n",i , p, *p);
    }
}
```

variavel	endereco	valor
a[0]	FFCA	1
a[1]	FFCC	2
a[2]	FFCE	3

← RESULTADO DO PROGRAMA:

Notar que, na aritmética hexadecimal, tem-se:
 $\&pa[0] = (pa + 0) = FFCA + 0 \text{ bytes} = FFCA$
 $\&pa[1] = (pa + 1) = FFCA + 2 \text{ bytes} = FFCC$
 $\&pa[2] = (pa + 2) = FFCA + 4 \text{ bytes} = FFCE$

Exercício 42: vetores e ponteiros.

Construa um algoritmo que leia um vetor qualquer com N elementos ($N \leq 10$), e altere seus elementos da seguinte forma:

- se o elemento for par, multiplique-o por 4;
- se o elemento for impar e positivo, multiplique-o por 3;

O vetor deve ser modificado e impresso através apenas do uso de ponteiros para seus elementos.

Solução

```
#include <stdio.h>
void main(){
    int *pv, v[10], i, n;
    pv = &v[0];
    // Complete o programa
    printf("n: "); scanf("%d",&n);
    printf("Forneca os elementos do vetor:\n");
    for(i=0; i<n; i++){
        scanf("%i",pv+i);
        if (*(pv+i)%2==0)
            *(pv+i)*=4;
        else
            if (*(pv+i)>0)
                *(pv+i)*=3;
    }
    for(i=0; i<n; i++)
        printf("V[%i] = %i\n",i,*(pv+i));
}
```

Ponteiros como vetores

Como consequência do que foi visto até agora, é possível então indexar um ponteiro qualquer. O programa mostrado a seguir, utilizando um ponteiro para uma matriz, é possível e correto.

```
#include <stdio.h>
void main () {
    /* É possível declarar e atribuir valores simultaneamente */
    int mat[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int *p;
    p=mat;
    /* Note que p[2] equivale a *(p+2) */
    printf("O terceiro elemento do vetor e: %d",p[2]);
}
```

Note que, no programa acima, $p[2]$ é equivalente a $*(p+2)$.

Matrizes e Ponteiros

Seja a matriz declarada na seguinte forma:

*tipo_da_variável *nome_da_variável [tam1][tam2];*

Assim sendo, a notação

nome_da_variável [índice1] [índice2]

é absolutamente equivalente a:

**(nome_da_variável + índice1*tam2 + índice2)*

Exemplo 29: matrizes e ponteiros

O programa abaixo define uma matriz $Mat_{2 \times 3}$ e a imprime utilizando a notação matricial. Utilizando ponteiros, imprime uma tabela com o endereço na memória e o conteúdo de cada um dos elementos dessa matriz.

```
#include <stdio.h>
#define M 2
#define N 3

void main(){
    int i, j, *p, *pa, a[M][N];

    for(i=0;i<M;i++)
        for(j=0;j<N;j++)
            a[i][j] = i*N + j + 1;
    printf("\n\nImpressao da Matriz:\n ");
    for(i=0;i<M;i++) {
        printf("\n");
        for(j=0;j<N;j++)
            printf("%2d",a[i][j]);

    }
    pa = &a[0][0]; /* aponta p/ o inicio da matriz */
    printf("\n\nvariavel  endereco  valor\n");
    printf("-----\n");

    for(i=0;i<M;i++)
        for(j=0;j<N;j++){
            p = pa + i*N + j; /* p aponta para a[i][j] */
            printf(" a[%d][%d]    %p        %d  \n",i, j, p, *p);
        }
}
```

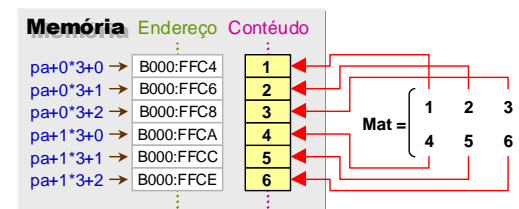
Impressao da Matriz

1	2	3
4	5	6

variavel	endereco	valor

a[0][0]	FFC4	1
a[0][1]	FFC6	2
a[0][2]	FFC8	3
a[1][0]	FFCA	4
a[1][1]	FFCC	5
a[1][2]	FFCE	6

← RESULTADO DO PROGRAMA:



Um uso importante de ponteiros é a varredura seqüencial de uma matriz. Isso é mostrado na [Figura 30](#), onde uma matriz 100×100 é inicializada atribuindo-se zero a todos os seus elementos. Duas formas são utilizadas: atribuindo zero a cada elemento da matriz, e através do uso de ponteiros. No primeiro, cada vez que se faz a atribuição $mat[i][j] = 0$, o programa tem que calcular o deslocamento para obter o ponteiro correspondente. Ou seja, o programa tem que calcular 10000 deslocamentos. No segundo, o único cálculo que é necessário ser feito pelo programa é o incremento de ponteiro. Realizar 10000 incrementos em um ponteiro é muito mais rápido que calcular 10000 deslocamentos completos.

VARREDURA SEQUENCIAL DE UMA MATRIZ**Através da Matriz**

```
void main (){
    float mat[100][100];
    int i, j;
    for(i=0;i<100;i++)
        for(j=0;j<100;j++)
            mat[i][j]=0;
}
```

Uso de ponteiros: mais eficiente

```
void main (){
    float *pmat, mat[100][100];
    int k;
    pmat=&mat[0][0];
    for(k=0;k<10000;k++){
        *pmat=0;
        pmat++;
    }
}
```

Figura 30: A inicialização dos elementos de uma matriz utilizando ponteiros é mais eficiente/rápido.

Exercício 43: ponteiro e matriz

Construa um programa que calcule a transposta de uma matriz inteira $A_{10 \times 10}$. O programa deve realizar a transposição utilizando ponteiros. O resultado deve ser impresso.

Solução

```
#include <stdio.h>
void main(){
    int *pa, a[10][10], i, j, aux;
    pa = a;
    // Complete o programa
    printf("Entre com a Matriz A:\n\n");
    for(i=0;i<10;i++){
        printf("Digite a linha %i (10 elementos):\n",i);
        for(j=0;j<10;j++)
            scanf("%d",pa+i*10+j);
    }
    for(i=1;i<10;i++){
        for(j=0;j<i;j++){
            aux = *(pa+i*10+j);
            *(pa+i*10+j) = *(pa+j*10+i);
            *(pa+j*10+i)=aux;
        }
    }
    printf("Matriz A transposta:\n");
    for(i=0;i<10;i++) {
        printf("\n");
        for(j=0;j<N;j++)
            printf("%5d",*(pa+i*10+j));
    }
}
```

Vetores e Matrizes de ponteiros

É possível criar vetores ou matrizes de ponteiros como se faz para qualquer outro tipo de variável. Uma declaração de vetor ou matriz de ponteiros poderia ser:

```
int *pvet[10];
float *pmat[5][6];
```

No caso acima, pvet é um vetor que armazena 10 ponteiros para inteiros, e pmat armazena 30 ponteiros para variáveis reais de precisão simples.

Alguns Cuidados com a Notação

Há uma diferença entre nomes de arranjos e de ponteiros que deve ser notada: um ponteiro é uma variável, mas o nome de um arranjo não é uma variável, mas sim uma constante contendo o endereço do início do bloco de memória que armazena a matriz. Isso significa que não se consegue alterar o endereço que é apontado pelo "nome do arranjo". Veja o que é correto e o que não é no trecho de código C abaixo envolvendo um arranjo unidimensional (vetor).

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;
/* as operações a seguir não são válidas */
vetor = vetor + 2; /* ERRADO: vetor não é variável */
vetor++;          /* ERRADO: vetor não é variável */
vetor = ponteiro; /* ERRADO: vetor não é variável */
/* as operações abaixo são válidas */
ponteiro = vetor; /* CERTO: ponteiro é variável */
ponteiro = vetor+2; /* CERTO: ponteiro é variável */
```

6.1.7. Ponteiros para Caracteres

Cadeias de caracteres podem ser manipuladas utilizando-se ponteiros. Exemplos:

```
char *txt; /* ponteiro para caracter */
txt = "Isto eh um teste"; /* Atribui o ponteiro da cadeia a txt*/
printf("%s", txt); /* Imprime "Isto eh um teste" */
printf("%c", *(txt+3)); /* Imprime "o" (quarta posição na cadeia)*/
*(txt+4) = *(txt+5); /* faz o quinto caracter igual ao sexto
                     /* caracter da cadeia */
printf("%s", txt); /* Imprime "Istoeeh um teste" */
```

Como observado acima, nomes de strings (cadeias de caracteres) são do tipo `char*`, ou seja, ponteiros.

Exemplo 30: ponteiros e strings

A função `StrCpy()` abaixo funciona como a função `strcpy()` da biblioteca padrão do C.

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

StrCpy (char *destino, char *origem){
    while (*origem){ /* a condição torna-se falsa para /0 */
        *destino=*origem;
        origem++;
        destino++;
    }
    *destino='\0';
}

void main () {
    char str1[100],str2[100],str3[100];
    printf ("Entre com uma string: ");
    gets (str1);
    StrCpy (str2,str1);
    StrCpy (str3,"Voce digitou a string ");
    printf ("\n\n%s%s",str3,str2);
}
```

No Exemplo 30 há alguns pontos a se destacar. Para permitir a alteração do string *str2*, o primeiro argumento da função *StrCpy* (*destino*) é um ponteiro para um string (veja item 6.1.5 *Ponteiros e Argumentos de Funções*). É dessa mesma maneira que as funções do C, como *scanf()*, *gets()*, *strcpy()* e outras funcionam. Passando o ponteiro, a função tem como alterar o conteúdo da variável por ele apontado. No comando *while (*origem)* considera-se o fato de que o string termina com o caractere '\0' como critério de parada, cujo valor é 0 (zero)³³. Ao se fazer os incrementos *origem++* e *destino++*, lembre-se de que a passagem de parâmetros no C é sempre por valor: essas variáveis são apenas cópias dos parâmetros passados. Assim sendo, quando alteramos o ponteiro *origem* na função *StrCpy()*, o ponteiro *str2* permanece inalterado na função *main()*.

Exercício 44: ponteiros e strings

Construa um programa que leia um string com até 30 caracteres e troque todas as suas vogais pela letra x. Utilize ponteiros para realizar a troca. Imprima o string original e o string alterado logo abaixo. Lembre-se que um string termina com o caractere "/0", cujo valor é zero: use esse fato para determinar o tamanho dele.

Solução

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

void main() {
    char *s[30], *ps;
    ps = &s[0];
    printf("\nString: ");
    gets(s);
    printf("\n\nantes:  %s", s);
    while(*ps) {
        switch (toupper(*ps)) {
            case 'A':
            case 'E':
            case 'I':
            case 'O':
            case 'U': *ps='x';
        }
        ps++;
    }
    printf("\n\ndepois: %s", s);
}
```

6.1.8. Ponteiros para Funções

Ponteiros podem também apontar para funções. Isso permite construir, por exemplo, programas onde funções são passadas como parâmetros de outras funções.

Um ponteiro para uma função tem a seguinte declaração:

```
tipo_de_retorno (*nome_do_ponteiro)();
```

ou

```
tipo_de_retorno (*nome_do_ponteiro)(declaração_de_parâmetros);
```

Não é obrigatório se declarar os parâmetros da função. Veja o exemplo a seguir.

Exemplo 31: ponteiros para funções

No programa desse exemplo, a função *Calcula()* tem como primeiro parâmetro um ponteiro para uma função do tipo *float* com um argumento também do tipo *float*. O segundo parâmetro é utilizado internamente nessa função como parâmetro da função passada.

³³ Lembre-se: zero é falso, e qualquer valor que não seja zero é verdadeiro.

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

float x2(float x);
float x3(float x);
float Calcula(float (*func)(float x), float x);

void main() {
    float x;
    x = 2;
    printf("\nO quadrado de x eh %.2f", Calcula(x2,x));
    printf("\nO cubo de x eh %.2f", Calcula(x3,x));
}

float Calcula(float (*func)(float x), float x) {
    return (*func)(x);
}

float x2(float x) {
    return x*x;
}

float x3(float x) {
    return x*x*x;
}
```

6.2. Alocação Dinâmica de Memória

6.2.1. Conceito

A alocação dinâmica de memória é uma característica importante das linguagens de alto nível, permitindo a criação de variáveis em tempo de execução, ou seja, alocar memória para novas variáveis quando o programa está sendo executado.

Até agora se tem definido variáveis de forma estática, ou seja, reservando-se o espaço da memória necessária para as variáveis através de suas declarações. Isto funciona bem quando se sabe o quanto de memória será utilizado. Mas, e se essa quantidade necessária de memória não for conhecida.

Tome-se como exemplo as definições de arranjos. A questão é: será que sempre se terá certeza do tamanho que o arranjo poderá ter durante a vida de um programa? Será que, para um determinado problema, um arranjo com 10.000 posições é o suficiente? Será que nunca acontecerá de ter que se trabalhar com o elemento 10.001? E o que acontece quando o programa nunca trabalha com mais de 100 posições do arranjo? As posições de memória restantes (9.900) não poderão ser utilizadas por outras variáveis, pois já estão reservadas. O uso de arranjos é, sem dúvida, de grande ajuda para a construção de determinados programas, mas quando é necessário que superdimensionar uma variável desse tipo por não se saber qual tamanho ela poderá ter, a sua utilização pode causar problemas no gerenciamento e uso da memória.

Outra situação onde a alocação da memória é feita dinamicamente com vantagens é no gerenciamento de memória de janelas e menus gráficos. Toda vez que um desses elementos é exibido é necessário utilizar uma certa quantidade de memória, a qual é liberada no momento em que se deixa de exibir o respectivo elemento. Se para cada uma dessas janelas ou menus fosse alocada estaticamente memória no início da execução do programa, haveria um consumo muito grande e desnecessário de espaço, com a consequente degeneração na performance do programa, ou mais provavelmente insuficiência de memória.

Para contornar tais problemas existe um tipo de alocação de memória onde é possível reservar espaço para os dados à medida que é necessário. Da mesma forma, é possível liberar posições de memória quando estas não são mais utilizadas. A este processo dá-se o nome de *alocação dinâmica*, uma vez que a memória é alocada não no início do programa, mas sim no decorrer de

sua execução. Isso torna possível, entre outras coisas, definir arranjos com os tamanhos adequados às necessidades.

6.2.2. Funções do C para Alocação Dinâmica

O padrão C ANSI define apenas 4 funções para o sistema de alocação dinâmica, disponíveis na biblioteca `stdlib.h`: `malloc()`, `calloc()`, `realloc()` e `free()`. Entretanto, existem outras funções que também são utilizadas, mas são dependentes do ambiente e do compilador. Nesse capítulo só serão vistas estas funções básicas. Se desejar saber mais sobre as funções disponíveis para alocação dinâmica, consulte o manual de seu compilador.

Antes de prosseguir, deve-se ressaltar que no processo de alocação dinâmica há uso intensivo de ponteiros, pois é preciso saber onde começa a região da memória sendo alocada: essa posição é sempre "apontada" por um ponteiro.

Função `malloc()`

Essa função serve para alocar memória, e tem o seguinte protótipo:

```
void *malloc (unsigned int num_bytes);
```

O argumento da função é o número de bytes que se deseja alocar: `num_bytes`. Essa quantidade é alocada na memória, sendo retornado como resultado um ponteiro `void*` para o primeiro byte alocado. Esse ponteiro pode ser atribuído a qualquer outro tipo de ponteiro através da operação de `cast`. Se não houver memória suficiente para alocar a memória requisitada, a função `malloc()` retorna um ponteiro nulo. Veja o exemplo de alocação dinâmica com `malloc()` na Figura 31.

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

```
void main (void){
    int *p;
    int a = 5;
    ...
    p = (int *)malloc(a*sizeof(int));
    if (!p){ // é verdade se p é nulo
        printf("Erro: Memória Insuficiente!");
        exit(1); // termina o programa
    }
    ...
}
```

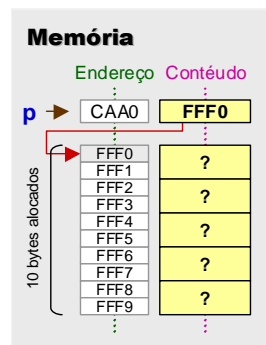


Figura 31: Alocação de memória com a função `malloc()`.

No exemplo da Figura 31, é alocada memória suficiente para guardar 5 números inteiros (pois $a = 5$). O operador unário `sizeof()` retorna o número de bytes do argumento; nesse caso, o parâmetro passado é o tipo `int`. Esse operador é útil para se saber o tamanho de tipos: para o tipo `int`, retorna o valor 2 (bytes). O ponteiro `void*` retornado por `malloc()` é convertido para um ponteiro `int*` pela operação de `cast`, sendo então atribuído a `p`. A declaração seguinte testa se a operação foi bem sucedida. Se a alocação falhar, `p` terá um valor nulo ($p = 0$), o que fará com que `!p` retorne verdadeiro ($!p = 1$). Se a alocação ocorrer sem problemas, o vetor de inteiros assim criado poderá ser utilizado normalmente indexando-se `p`: `p[0]` até `p[a-1]`.

Função `calloc()`

Essa função também é utilizada para alocar memória, tendo o seguinte protótipo:

```
void *calloc (unsigned int num, unsigned int size);
```

Ao evocar essa função uma quantidade de memória igual a `num*size` é reservada. Isso é equivalente a se alocar memória suficiente para um "vetor" com `num` objetos de tamanho `size`. A função `calloc()` retorna um ponteiro `void*` para o primeiro byte alocado. Esse ponteiro também pode ser atribuído a qualquer outro tipo de ponteiro através da operação de `cast`. Se não houver memória suficiente para alocar a memória requisitada, função `calloc()` retorna um ponteiro nulo. Veja um exemplo de alocação dinâmica com `calloc()`:

No exemplo da Figura 32, é alocada memória suficiente para guardar 5 números inteiros (pois $a = 5$). O operador unário `sizeof()` retorna o número de bytes do argumento. O ponteiro `void*` retornado por `calloc()` é convertido para um ponteiro `int*` pela operação de `cast`, sendo então atribuído a `p`. A declaração seguinte testa se a operação foi bem sucedida. Se a alocação falhar, `p` terá um valor nulo ($p = 0$), o que fará com que `!p` retorne verdadeiro ($!p = 1$). Se a alocação ocorrer sem problemas, o vetor de inteiros assim criado poderá ser utilizado normalmente indexando-se `p`: `p[0]` até `p[a-1]`.

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

```
void main (void){
    int *p;
    int a = 5;
    ...
    p = (int *)calloc(a, sizeof(int));
    if (!p){
        printf("Erro: Memória Insuficiente!");
        exit(1); // termina o programa
    }
    ...
}
```

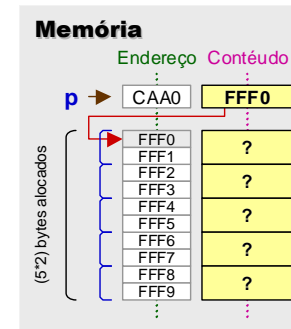
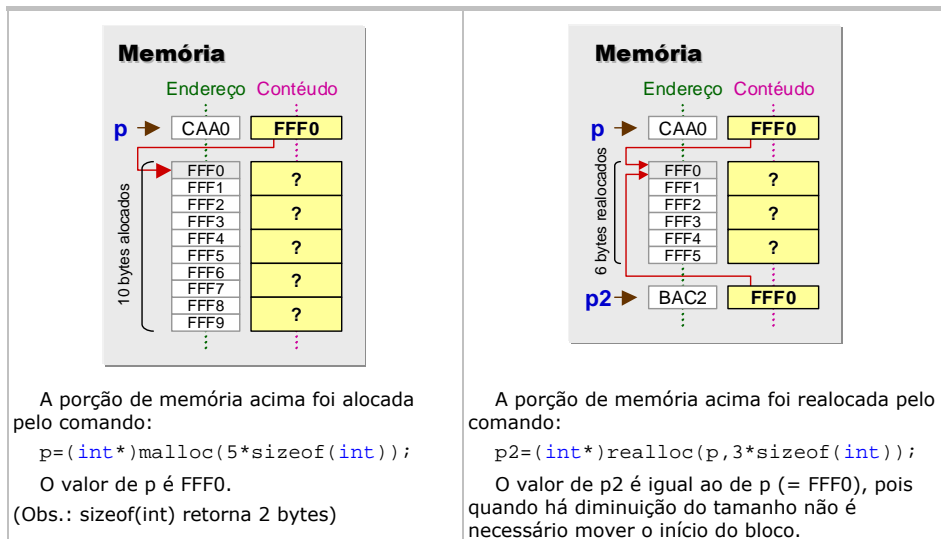


Figura 32: Alocação de memória com a função `calloc()`.

Função `realloc()`

Essa função é utilizada para realocar memória já reservada, e tem o seguinte protótipo:

```
void *realloc (void *ptr, unsigned int num_bytes);
```


Figura 33: Realocação de memória com a função `realloc()`.

A função modifica o tamanho da memória previamente alocada apontada por **ptr* (retornado por `malloc()` ou `calloc()`) para o tamanho, em bytes, especificado por *num_bytes*. O valor de *num_bytes* pode ser maior ou menor que o que o valor originalmente reservado. Um ponteiro para o bloco realocado é devolvido, pois `realloc()` pode precisar mover o bloco da posição inicialmente alocada no caso de aumento de tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco sem que nenhuma informação seja perdida. Se *ptr* for nulo, aloca a quantidade de bytes indicada por *num_bytes* e devolve um ponteiro; se *num_bytes* é zero, a memória apontada por *ptr* é liberada. Se não houver memória suficiente para a alocação, um ponteiro nulo é devolvido e o bloco original é deixado inalterado. A Figura 33 exemplifica o funcionamento da função `realloc()`.

Função `free()`

Quando alocamos memória dinamicamente é necessário que nós a liberemos quando ela não for mais necessária. Para isto existe a função `free()`, cujo protótipo é:

```
void free(void *p);
```

O parâmetro a ser passado para a função `free()` é o ponteiro para o início da memória alocada. Mas e a quantidade de memória (bytes) a ser liberada? Esse valor já é conhecido, pois quando a memória é reservada (através de `malloc()`, `calloc()` ou `realloc()`) o número de bytes alocados é guardado numa "tabela de alocação" interna. Na Figura 34 o exemplo de alocação apresentado na Figura 31 foi modificado com a inclusão da liberação de memória através da função `free()`.

Mas por que se preocupar com liberação de memória? O problema é que se a alocação de memória for sendo feita continuamente, sem a liberação quando não mais for necessária, a memória disponível vai diminuindo e pode comprometer o desempenho do programa, e até mesmo esgotar os recursos disponíveis no equipamento. Isso jogaria fora toda a vantagem de se trabalhar com alocação dinâmica de memória, onde os recursos disponíveis (memória) são utilizados apenas quando necessários.

Em resumo, assim que o bloco de memória cumprir a sua tarefa, ele deve ser imediatamente liberado para que possa ser usado posteriormente pelo próprio programa, ou por outro aplicativo.

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

void main (void){
    int *p;
    int a = 5;
    ...
    p = (int *)malloc(a*sizeof(int));
    if (!p){
        printf("Erro: Memoria Insuficiente!");
        exit(1); /* termina o programa */
    }
    ...
    free(p); /* libera a memória alocada apontada por p */
    ...
}
```

Figura 34: Liberação de memória com a função `free()`.

6.2.3. Programa Exemplo

O Programa da Figura 35 cria uma matriz qualquer *n* x *m* através de alocação dinâmica de memória. Ao se chamar as funções `init_mat()` e `print_mat()`, passa-se como parâmetros o ponteiro para o bloco de memória alocado e o tamanho de cada dimensão da matriz. Assim sendo, os ponteiros utilizados dentro de cada função estão apontando para posições dentro de uma região adequadamente reservada para a matriz *n* x *m* criada.

Figura 35: Programa exemplo.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <ctype.h>

/* Inicializa uma matriz n x m conhecido seu ponteiro.
   Assume-se que a memória correspondente já esta alocada. */
void init_mat(int valor, int *p, int n, int m) {
    int i,j;
    for (i=0; i<n; i++)
        for (j=0; j<m; j++)
            *(p+i*m+j)=valor;
}

/* Imprime uma matriz n x m conhecido seu ponteiro.
   Assume-se que a matriz já está inicializada */
void print_mat(int *p, int n, int m) {
    int i,j;
    printf("\n***");
    for (i=0; i<n; i++) {
        printf("\n");
        for (j=0; j<m; j++)
            printf("%-5d", *(p+i*m+j));
    }
    printf("\n***\n");
}
```


Figura 35: Programa exemplo.

```

/* Programa principal */
void main() {
    int valor, *p, n, m;
    clrscr(); /* Só no Borland C */
    printf("CRIAR UMA MATRIZ N x M E A INICIALIZA COM VALOR\n");
    printf("-----");

    inicio:

    /* Solicita n, m e valor */
    printf("\nForneca n e m (separados por virgula): ");
    scanf("%d,%d",&n,&m);
    printf("Forneca Valor: ");
    scanf("%d",&valor);

    /* Aloca memoria para a matriz n*m apontada por p */
    p =(int*) malloc(n*m*sizeof(int));

    /* Inicializa os elementos da matriz com valor*/
    init_mat(valor,p,n,m);

    /* Imprime a matriz inicializada */
    print_mat(p,n,m);

    /* Libera a memoria */
    free(p);

    /* Pergunta se deseja repetir o procedimento */
    printf("\nCriar/Inicializar outra matriz ? (s/n) => ");
    if ( toupper(getche()) == 'S') {
        printf("\n--- OUTRA MATRIZ -----");
        goto inicio;
    }
    printf("\n\n*** FIM DO PROGRAMA ***");
}

```

O resultado da execução desse programa exemplo está mostrado na [Figura 36](#).

TELA DE SAÍDA - DOS

```

CRIAR UMA MATRIZ N x M E A INICIALIZA COM VALOR
Forneca n e m (separados por vírgula): 2,3
Forneca Valor: 1

***
1   1   1
1   1   1
***

Criar/Inicializar outra matriz ? (s/n) => s
--- OUTRA MATRIZ -----
Forneca n e m (separados por vírgula): 3,9
Forneca Valor: 0

***
0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0
***

Criar/Inicializar outra matriz ? (s/n) => n
*** FIM DO PROGRAMA ***

```

Figura 36: Resultado da execução do programa exemplo.

Exercício 45: alocação dinâmica.

Construa um programa que leia uma matriz $n \times n$ e faça a sua transposição. Apenas uma matriz, alocada dinamicamente com exatamente o tamanho necessário, pode ser utilizada.

Solução

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

void main(){
    int i, j, n, aux, *a;
    long m;
    printf("Ordem da matriz a ser transposta: ");
    scanf("%d",&n);

    // Aloca memória para a matriz, retornando um ponteiro
    // para seu início. Calloc retorna um ponteiro do tipo
    // void. Para mudar para ponteiro para int e manipular os
    // elementos da matriz, é preciso fazer a operação de "cast".
    a = (int *) calloc(n*n, sizeof(int));

    // Imprime a quantidade de memória alocada.
    m = (long) _msize(a);
    printf("\nMemoria alocada: %i bytes\n\n",m);

    // Leitura da matriz a.
    printf("Digite a matriz por linhas, %d por linha e %d linhas:\n",n,n);
    for (i=0;i<n;i++){
        for (j=0;j<n;j++){
            scanf("%i", &a[i*n+j]);
        }
        printf("\nMatriz lida:\n\n");

        // Impressão da matriz lida.
        for (i=0;i<n;i++){
            for (j=0;j<n;j++){
                printf("%4i", *(a+i*n+j));
            }
            printf("\n");
        }
        printf("\n");

        // Transposição da matriz.
        for (i=1;i<n;i++){
            for (j=0;j<i;j++){
                aux = *(a+i*n+j);
                *(a+i*n+j) = *(a+j*n+i);
                *(a+j*n+i) = aux;
            }
        }

        // Impressão da matriz transposta
        printf("\nMatriz transposta:\n\n");
        for (i=0;i<n;i++){
            for (j=0;j<n;j++){
                printf("%4i", *(a+i*n+j));
            }
            printf("\n");
        }
        printf("\n\n");

        // Libera a memória utilizada para o bloco apontado por a.
        free(a);
    }
}
```

Referências

1. Kernighan, B.W. e Ritchie, D.M., "C - A Linguagem de Programação", Editora Campus, 1986.
2. Curso de C [<http://www.ccc.br/salav/informat/c/c.html>]

3. Linguagens de Programação - Programação Convencional em Pascal [<http://www.inf.puc-rio.br/~tecbd/treinamentos/apqs/nivel/lq/Pascal/index.htm>]
4. Algumas Notas sobre Programação em C [<http://www.ctsoft.softex.br/mauricio/tutoriais/C/index.htm>]

Apêndice A - Exemplos do Processo de Solução de Problemas Conforme Polya

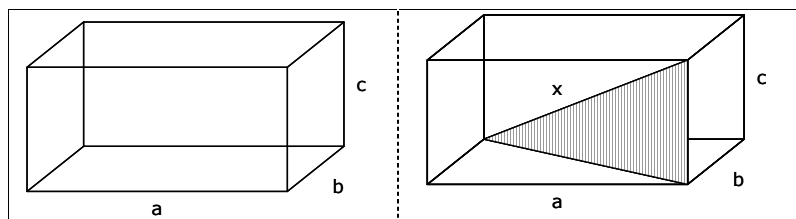
Exemplo de Solução de Problema 1

⇒ Colocação do Problema

Calcular a diagonal de um paralelepípedo retângulo do qual são conhecidos o comprimento, a largura e a altura.

1) Compreensão do problema

Figura:



Qual é a incógnita?

R.: O comprimento da diagonal de um paralelepípedo.

Quais são os dados?

R.: O comprimento, a largura e a altura do paralelepípedo.

Qual letra deve denotar a incógnita? (adotar uma notação adequada)

R.: x

Quais letras escolheria para o comprimento, a largura e a altura? (adotar uma notação adequada)

R.: a, b e c.

Qual é a condicionante que relaciona a, b, c e x?

R.: x é a diagonal do paralelepípedo no qual a, b e c são, respectivamente, o comprimento, a largura e a altura

Trata-se de um problema razoável, ou seja, a condicionante é suficiente para determinar a incógnita?

R.: Sim ele é razoável. Se conhecermos a, b e c, conheceremos o paralelepípedo. Se o paralelepípedo ficar determinado, sua diagonal também o ficará.

2) Estabelecimento de um Plano

Considere a incógnita. Conhece um problema que tenha a mesma incógnita ou outra semelhante? Nunca resolveu um problema cuja incógnita fosse um comprimento de uma linha?

R.: Claro que já resolveram esse tipo de problema. Por exemplo, calcular um lado de um triângulo retângulo.

Eis um problema correlato já resolvido. É possível utilizá-lo? Há algum elemento geométrico como o identificado acima na figura desse problema?

R.: Sim. Ver figura.

Existe alguma relação entre o elemento geométrico encontrado e a incógnita do problema?

R.: Sim. A hipotenusa do triângulo e a diagonal do paralelepípedo são coincidentes.

Conhecendo a relação entre o elemento geométrico encontrado e a incógnita do problema, é possível chegar à solução desse? Qual poderia ser o procedimento aplicado?

R.: Sim. Calculando-se a hipotenusa do triângulo a partir do teorema de Pitágoras.

Conhecendo-se o procedimento aplicável, existem dados suficientes?

R.: Um cateto é conhecido (c), e o outro pode ser facilmente calculado, pois é a hipotenusa do triângulo retângulo de lados a e b, também obtido pelo teorema de Pitágoras.

Pois bem. Agora tem-se um plano para a solução do problema!

Conceber um plano, a idéia da resolução, não é fácil. Para conseguir isto é preciso, além de conhecimentos anteriores, de bons hábitos mentais e de concentração no objetivo, contando-se, às vezes, com uma "pitada de sorte"³⁴. O objetivo desse item é tentar iniciar algo nesse sentido.



Antes de continuar, cabe a seguinte pergunta:

O que tem a ver a construção de algoritmos com o processo de resolução de problemas?

A resposta é: **TUDO!**

Observe que o plano de solução nada mais é do que um algoritmo. Cabem, portanto, as mesmas dificuldades de concepção descritas acima para o plano de solução. Por isso a importância do estudo do processo de solução de problemas.

Executar o plano (próximo passo) é muito mais fácil: paciência é o "ingrediente" mais importante.

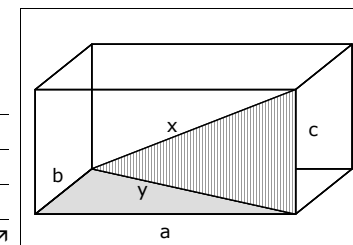
3) Execução do Plano

Incógnitas: x

Dados: a, b, c

Plano: Resolver o triângulo de lados a, b e y e, em seguida, resolver o triângulo de lados x, y e c aplicando duas vezes, sucessivamente, o teorema de Pitágoras.

Figura 7



Fórmulas:

$$1^{\text{a}} \text{ relação: } x^2 = y^2 + c^2$$

$$2^{\text{a}} \text{ relação: } y^2 = a^2 + b^2$$

$$\text{Relação Final: } x^2 = a^2 + b^2 + c^2$$

$$\text{ou: } x = \sqrt{a^2 + b^2 + c^2}$$

³⁴ Se bem que alguns dizem que a sorte sorri para quem trabalha... Faz muito sentido em nosso contexto.



Nunca deixe de fazer o retrospecto!

A maior parte dos alunos, uma vez chegada à solução do problema e escrita a demonstração, fecham os livros e passam a outro assunto. Assim fazendo, eles perdem uma fase importante e instrutiva do trabalho da resolução. Se fosse feito um retrospecto (ou revisão) da resolução completa, reconsiderando-se e reexaminando-se o resultado final e o caminho que levou até ele, seria possível consolidar o conhecimento e aperfeiçoar a capacidade de resolver problemas. Nenhum problema fica completamente esgotado.

4) Retrospecto ou Revisão

É possível verificar o resultado?

R.: Sim. Através da medição das distâncias a , b , c e x em um paralelepípedo qualquer e sua substituição na fórmula encontrada. A relação $x = (a^2 + b^2 + c^2)^{1/2}$ deve se mostrar verdadeira para qualquer medição desse tipo

Note que problemas literais apresentam uma grande vantagem sobre os problemas puramente numéricos: aqueles se prestam a um grande número de verificações, ao passo que estes não podem ser verificados, a não ser no caso particular.

Todos os dados foram utilizados? Todos os dados aparecem na fórmula que exprime a diagonal? Eles são intercambiáveis, ou seja, são simétricos?

R.: Sim; todos os dados foram utilizados. Intercambiar os dados a , b e c significa apenas mudar a posição espacial do paralelepípedo, sem alterar sua forma. Assim sendo, permutando a , b e c entre si não altera o resultado: eles são simétricos

Esse problema é da Geometria Espacial. Ele é análogo a outro problema da geometria plana. Qual? É possível aplicar o resultado obtido para esse caso particular? Quando?

R.: É análogo ao cálculo da diagonal de um retângulo de dimensões a e b (ou b e c , ou c e a). É possível utilizar a fórmula encontrada para o caso espacial desde que a terceira dimensão seja feita igual a zero.

Se todas as dimensões do paralelepípedo crescerem numa determinada proporção, a diagonal também crescerá nessa proporção? Isto é, se os lados a , b e c forem multiplicados por 5, a diagonal também o será? É possível verificar?

R.: Sim. A variação, sendo idêntica para as dimensões que definem o paralelepípedo, também o será para a diagonal

$$x = (a^2 + b^2 + c^2)^{1/2} \Rightarrow [(5a)^2 + (5b)^2 + (5c)^2]^{1/2} = [5^2(a^2 + b^2 + c^2)]^{1/2} = [5^2(x)^2]^{1/2} = 5x$$

Se as dimensões estiverem expressas em metros, a fórmula também fornecerá a diagonal em metros. Se a , b e c estiverem em centímetros, a fórmula deverá continuar válida. Você está certo disso³⁵?

R.: Sim. Supondo a dimensão como sendo d (m, cm, mm, etc), e sendo essa a dimensão de a , b e c , o teste dimensional fornece: $(d^2)^{1/2} = d$. Ou seja, a diagonal terá sempre a mesma dimensão que a , b e c tiverem.

É possível utilizar o resultado, ou o método, em algum outro problema? Como?

R.: Sim. Por exemplo: "Sendo dados o comprimento, a largura e a altura de um paralelepípedo retângulo, calcular a distância do seu centro a um dos vértices."
 Como: utilizando o resultado do problema anterior, pois a distância pedida é a metade da diagonal do paralelepípedo.

³⁵ Infelizmente, se você estiver certo, você não concorrerá a um milhão de reais!

Exemplo de Solução de Problema 2

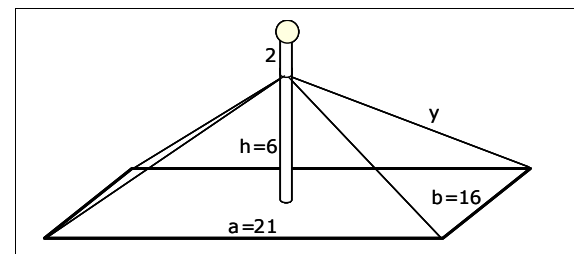
⇒ Colocação do Problema

No centro da cobertura retangular de um edifício, que tem 21 m de comprimento e 16 m de largura, instala-se um mastro de 8 m de altura. Para amarrar o mastro, precisamos de quatro cabos iguais. Estes partem do mesmo ponto, 2 m abaixo do topo do mastro, e são fixados nos quatro cantos da cobertura do edifício. Qual o comprimento de cada cabo?

Solução

1) Compreensão do problema

Figura:



Qual é a incógnita?

R.: Os comprimentos dos cabos, que são iguais.

Quais são os dados?

R.: O comprimento e a largura da cobertura, e a altura de fixação dos cabos no mastro.

Qual letra deve denotar a incógnita? (adotar uma notação adequada)

R.: y

Quais letras escolheria para o comprimento, a largura e a altura? (adotar uma notação adequada)

R.: a , b e h .

Qual é a condicionante que relaciona os dados e a incógnita?

R.: y é lado do triângulo cujos vértices são o centro da cobertura, um canto da cobertura e o ponto de fixação dos cabos.

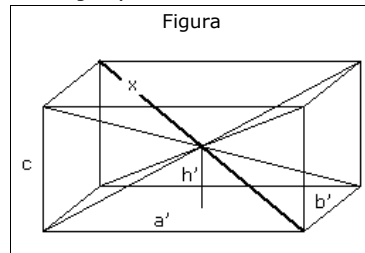
Trata-se de um problema razoável, ou seja, a condicionante é suficiente para determinar a incógnita?

R.: Sim, ele é razoável. Se conhecermos a e b , pode-se encontrar o centro da cobertura, bem como os cantos. Conhecendo-se h , tem-se o terceiro vértice

2) Estabelecimento de um Plano

Conhece um problema correlato? Considere a incógnita. Conhece um problema que tenha a mesma incógnita, ou outra semelhante? (analise através de uma figura)

R.: Sim. Acabamos de ver a solução do problema onde foi determinada a diagonal de um paralelepípedo. Analisando a figura ao lado, nota-se que o comprimento y procurado é a metade da diagonal do paralelepípedo se $a'=a$, $b'=b$ e $h'=h$. Tendo-se a' e b' , e observando-se que $c=2h'$, o cálculo da diagonal é direto.



Conhecendo-se o procedimento aplicável, existem dados suficientes?

R.: São conhecidos a , b e h . Com estes dados pode-se obter os valores para aplicação do método do problema correlato fazendo-se $a'=a$, $b'=b$ e $h'=h$, com $c=2h'$. Assim sendo, os dados são suficientes.

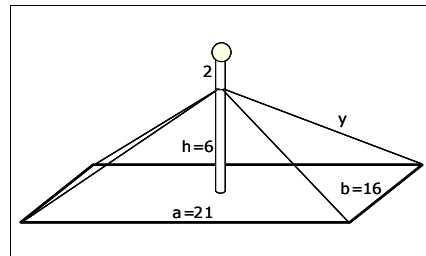
3) Execução do Plano

Incógnitas: y

Dados: a , b e h

Plano: Achar a diagonal do paralelepípedo com dimensões tais que $a'=a$, $b'=b$ e $h'=h$ ($c=2h$). Dividir o resultado por dois.

Figura 7



$$\text{Fórmula: } y = \frac{x}{2} = \frac{1}{2} \sqrt{a^2 + b^2 + c^2} = \frac{1}{2} \sqrt{a^2 + b^2 + (2h)^2}$$

$$\text{Cálculo: } y = \frac{1}{2} \sqrt{21^2 + 16^2 + (2 \cdot 6)^2} = 14.5$$

4) Retrospecto ou Revisão

É possível verificar o resultado?

R.: Sim. Através da medição das distâncias a , b , h e y e sua substituição na fórmula encontrada.

Todos os dados foram utilizados? Todos os dados aparecem na fórmula que exprime a diagonal? Eles são intercambiáveis, ou seja, são simétricos?

R.: Sim; todos os dados foram utilizados. Intercambiar a com b é possível, mas o valor de h não pode ser outro: tem que corresponder à altura do ponto de fixação sempre.

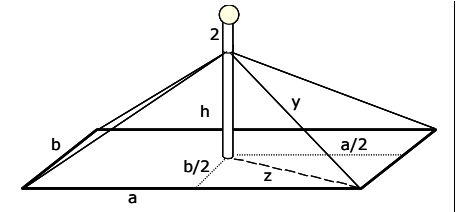
Seria possível resolver esse problema de alguma outra forma? Qual outro problema correlato poderia ser utilizado? (utilize uma figura para auxiliar no raciocínio)

R.: Sim. Através da solução do triângulo retângulo de lados y , h e z .



O valor de h é conhecido, e o de z pode ser facilmente obtido resolvendo-se o triângulo retângulo de lados

z , $a/2$ e $b/2$



Qual a fórmula derivada do plano de solução descrito na pergunta anterior? Qual a sua relação com a fórmula obtida originalmente?

R.: A solução do triângulo fornece como resultado a fórmula $y = \sqrt{(a/2)^2 + (b/2)^2 + h^2}$

A fórmula obtida inicialmente é $y = (\sqrt{a^2 + b^2 + (2h)^2})/2$

São as mesmas fórmulas, obtidas por caminhos diferentes.