ALGORITMOS

AUTOR
MAIRUM CEOLDO ANDRADE

ALGORITMOS

AUTOR DO ORIGINAL
MAIRUM CEOLDO ANDRADE

1ª EDIÇÃO
SESES
RIO DE JANEIRO 2015

Conselho editorial FERNANDO FUKUDA, SIMONE MARKENSON, JEFERSON FERREIRA FAGUNDES

Autor do original MAIRUM CEOLDO ANDRADE

Projeto editorial ROBERTO PAES

Coordenação de produção RODRIGO AZEVEDO DE OLIVEIRA

Projeto gráfico PAULO VITOR BASTOS

Diagramação FABRICO

Revisão linguística ADERBAL TORRES BEZERRA

Imagem de capa SHUTTERSTOCK

Todos os direitos reservados. Nenhuma parte desta obra pode ser reproduzida ou transmitida por quaisquer meios (eletrônico ou mecânico, incluindo fotocópia e gravação) ou arquivada em qualquer sistema ou banco de dados sem permissão escrita da Editora. Copyright SESES, 2015.

Dados Internacionais de Catalogação na Publicação (CIP)

A553a Andrade, Mairum Ceoldo
Algoritmos / Mairum Ceoldo Andrade
Rio de Janeiro : SESES, 2014.
128 p. : il.

ISBN 978-85-60923-78-6

1. Lógica. 2. Programação. 3. Tipos de dados. 4. Estrutura de dados homogêneas I. SESES. II. Estácio.

CDD 005.1

Diretoria de Ensino — Fábrica de Conhecimento
Rua do Bispo, 83, bloco F, Campus João Uchôa
Rio Comprido — Rio de Janeiro — RJ — CEP 20261-063

Sumário

Prefácio 7

1. Fundamentos de lógica e algoritmos 10

Definição de algoritmo 11
Solução de problemas 13
Etapas para construção de um algoritmo 14
Representação de algoritmos 16
Construção de um algoritmo 23
Lógica, lógica de programação e programa 26

2. Estrutura sequencial 32

Características da estrutura sequencial 33
Comandos de início e fim 33
Variáveis 36
Comando de entrada de dados – LEIA 39
Comando de saída de dados – IMPRIMA 44
Operadores aritméticos e lógicos 46

3. Estruturas de decisão 56

Características de estrutura de decisão 57
Comando condicional simples 59
Comando condicional composto 63
Comando condicional aninhado 67
Comando condicional múltiplo 74

4. Estruturas de repetição 84

Características da estrutura de repetição 84
Comando de repetição com variável de controle - PARA 86
Comando de repetição com teste lógico no início - ENQUANTO 92
Comando de repetição com teste lógico no fim - FAÇA...ENQUANTO 98
Quando utilizar cada estrutura de repetição 103

5. Estrutura de dados homogêneas 108

Estruturas homogêneas e heterogêneas de programação 109
Tipo String 110
Matriz unidimensional (vetor) 111
Matriz bidimensional (matriz) 120

Prefácio

Prezados(as) alunos(as)

Na disciplina de Algoritmos vamos estudar os conceitos básicos e aprender a estruturar o pensamento para o desenvolvimento e implementação de softwares ou programas. Essa disciplina é a base para o aprendizado de qualquer programador, pois introduz a lógica de programação, suas estruturas e seus principais conceitos.

Não deixe de tirar todas as dúvidas. Aprenda e pratique com afinco tudo o que envolve essa disciplina, pois o que aprenderá o acompanhará durante toda sua vida como desenvolvedor de *software*.

Na unidade 1, vamos estudar os conceitos e fundamentos da lógica e dos algoritmos, bem como algumas formas estruturadas para resolução de problemas.

Na unidade 2, vamos estudar a organização e a estrutura básica de um algoritmo, aprender como escrever nossos primeiros programas e entender os comandos indispensáveis em qualquer programa interativo.

Na unidade 3, aprenderemos os comandos ou estruturas condicionais, que nos permitirão criar desvios dentro de nossos programas, possibilitando a implementação de uma infinidade de problemas.

Na unidade 4, vamos aprender as estruturas de repetição, as quais nos auxiliarão a construir programas mais longos sem a necessidade de escrever um volume exagerado de código, por meio da repetição e do controle de determinado bloco.

Na unidade 5, finalizaremos nossa programação com o estudo das matrizes unidimensionais ou bidimensionais, as quais nos permitirão tratar um volume de dados maior e de forma mais organizada.

Ao final da unidade, você será capaz de resolver praticamente qualquer problema utilizando as linguagens algorítmicas ou de programação estudadas. Porém, não esqueça, serão necessárias dedicação e atenção para entender esse novo mundo.

Bons estudos!

1

Fundamentos de lógica e algoritmos

1 Fundamentos de lógica e algoritmos

Nesta unidade, aprenderemos o que é um algoritmo e para que ele serve. Basicamente veremos como ele está inserido em nosso dia a dia e como pode nos auxiliar na resolução de problemas. Durante toda nossa vida, deparamo-nos com diversos problemas para os quais precisamos estruturar uma sequência de passos e etapas para alcançarmos a solução de forma desejada, porém fazemos isso de maneira desestruturada e empírica.

Aprenderemos técnicas e linguagens para documentar e organizar os passos de solução de problemas. Iniciaremos com técnicas e linguagens naturais e evoluiremos até chegarmos às linguagens de programação, que permitem ensinar ao computador como resolver determinados problemas, para nos auxiliar em nossas tarefas.

OBJETIVOS

- Entender como se define um algoritmo.
- Compreender as etapas necessárias para construir um algoritmo.
- Compreender as linguagens para escrever algoritmos.
- Ser capaz de construir algoritmos simples fazendo uso da linguagem natural.
- Ser capaz de nomear símbolos.

REFLEXÃO

Você se lembra dos algoritmos que aprendeu na escola para resolver os problemas de matemática? Falando dessa forma provavelmente não, mas com certeza você já construiu e já utilizou vários algoritmos, como, por exemplo, as regras para encontrar o maior divisor comum, a descrição do trajeto que você escreveu para seu amigo chegar a sua casa ou a receita de um bolo que você seguiu.

1.1 Definição de algoritmo

Alguns conceitos e nomes podem parecer muito complexos ou filosóficos, quando vistos por sua definição formal ou pelas palavras de um matemático. Porém, muitas vezes, estes complexos conceitos estão presentes em nosso dia a dia, sem que ao menos percebamos.

Desde o Ensino Fundamental aprendemos, vários algoritmos matemáticos. Um exemplo de que todos devem se lembrar é o algoritmo utilizado para calcular o MDC (máximo divisor comum), também conhecido como Algoritmo de Euclides.

O Algoritmo de Euclides é composto de um conjunto de instruções que devem ser executadas sucessivamente para encontrar de forma eficiente o máximo divisor comum entre dois números diferentes de zero.

Caso não se lembre desse algoritmo, segue abaixo a descrição:

> Sejam AB e CD os dois números que não são primos entre si. É necessário, então, encontrar a máxima medida comum de AB e CD.
> Se CD mede AB, então é uma medida comum já que CD se mede a si mesmo. É manifesto que também é a maior medida, pois nada maior que CD pode medir CD. Mas, se CD não mede AB, então algum número restará de AB e CD, o menor sendo continuamente resto do maior e que medirá o número que o precede. Porque uma unidade não ficará pois se assim não for, AB e CD serão primos entre si [Prop. VII.1], o que é contrário ao que se supôs.
> Portanto, ficará algum número que medirá o número que o precede. E seja CD a medir BE deixando EA menor que si mesmo, e seja EA medindo DF deixando FC menor que si mesmo, e seja FC medida de AE. Então, como FC mede AE e AE mede DF, FC será então medida de DF. E também se mede a si mesmo. Portanto, também medirá todo o segmento CD, e CD mede BE. Então, CF mede BE e também mede EA. Assim mede todo o segmento BA e também mede CD. Isto é, CF mede tanto AB como CD, pelo que é uma medida comum de AB e CD.
> Afirmo que também é a maior medida comum possível, porque, se não o fosse, então um número maior que CF mede os números AB e CD. Seja este G. Dado que G mede CD e CD mede BE, G também mede BE. Além disso, mede todo o segmento BA pelo que mede também o resíduo AE. E AE mede DF pelo que G também mede DF. Mede ainda todo o segmento DC pelo que mede também o resíduo CF, ou seja, o maior mede o menor, o que é impossível.

Portanto, nenhum número maior que CF pode medir os números AB e CD. Então, CF é a maior medida comum de AB e CD, o que era o que se queria demonstrar. — Euclides. Elementos VII.2

<http://pt.wikipedia.org/wiki/Algoritmo_de_Euclides>

Achou um pouco difícil de entender, veja abaixo a descrição em uma linguagem um pouco mais moderna:

1. Dados dois número A e B, divida A por B e obtenha o resto R1.
2. Se o resto R1 for zero, então o MDC de A e B é B.
3. Se R1 for diferente de zero, divida B por R1 e obtenha o resto R2.
4. Se o resto R2 for zero, então o MDC de A e B é R1.
5. Se R2 for diferente de zero, então divida R1 por R2 e obtenha o resto R3.
6. Se o resto R3 for zero, então o MDC de A e B é R2.
7. Se R3 for diferente de zero, repita os dois passos anteriores é que o novo resto obtido seja igual a zero.

Veja abaixo a aplicação do Algoritmo Euclidiano[1] para obter o MDC entre 15 e 3.

	DIVIDENDO	DIVISOR	QUOCIENTE	RESTO
PASSO 1	32	6	5	2
PASSO 2	6	2	3	0

No passo 1, dividimos 32 por 6 e obtivemos resto igual a 2; como é diferente de zero passamos para o passo 2. Dividimos 6 pelo resto 2 e obtemos resto 0. Como o resto é zero, o MDC entre 32 e 6 é 2.

Podemos então descrever o algoritmo como uma sequência finita de instruções, definidas de maneira clara e sem ambiguidade, de forma que possa ser executada diretamente pelo seu leitor.

1 Os Elementos de Euclides é um tratado matemático e geométrico composto de 13 livros, escrito pelo matemático Euclides por volta do século 300 a.C. Nele são definidas toda a base da geometria euclidiana e a teoria elementar dos números.

CONCEITO

Algoritmo é uma sequência finita de instruções, definidas de maneira clara e sem ambiguidade, de forma que possa ser executada e reproduzida diretamente pelo seu interpretador ou leitor.

Outro exemplo clássico de algoritmo com os quais você provavelmente já teve contato são as receitas culinárias, as quais são um conjunto de passos ou instruções necessárias para se cozinhar o desejado.

1.2 Solução de problemas

Constantemente em nosso dia a dia, encaramos diversos problemas, desde os mais simples até os mais complexos. Provavelmente a maioria das pessoas gasta boa parte de suas vidas solucionando problemas. Porém, algumas vezes, despendemos grande quantidade de tempo e esforço na busca destas soluções, sem a certeza de obtê-las de forma eficiente.

Quando temos um problema buscamos identificar a origem e a maneira de solucioná-lo da forma mais rápida possível. Muitas vezes encontramos problemas mais complexos ou problemas que provavelmente iremos encarar novamente, o que nos remete a uma solução mais estruturada e detalhada, até para que possamos compartilhar com outras pessoas nossa resolução. Para obtermos clareza na solução, uma ferramenta muito útil são os algoritmos.

Nesses casos, quase sempre criamos um algoritmo de forma inconsciente, ou seja, idealizamos um conjunto de passos estruturados, que executaremos de forma sequencial para solução de um problema, como, a criação de uma lista de tarefas.

Cada indivíduo possui sua forma de estruturar a elucidação do problema, e quando necessário de documentar esta, seja por meio de um texto corrido, uma lista de tarefas, de uma receita, de fluxogramas etc. Com o objetivo de padronizar a forma como criamos as nossas soluções de problemas, vamos ver nessa disciplina formas e métodos padronizados internacionalmente para tal. Dessa forma, facilitamos o entendimento para a reprodução de nossas soluções.

O matemático George Polya, em seu livro "How to Solve it" de 1945 (POLYA 1945), propõe um método estruturado para a resolução de problemas baseado em quatro etapas:

1. Entender
2. Planejar
3. Executar
4. Verificar

Basicamente, na etapa Entender, como o próprio nome já sugere, devemos obter uma melhor compreensão do problema, identificando quais são as questões e variáveis existentes e verificando se existem informações suficientes para entendê-lo, e buscar uma solução.

A etapa seguinte, Planejar, consiste em estudar a melhor forma de resolver problema. É basicamente identificar o melhor método a ser aplicado; geralmente o método da divisão em problemas menores auxilia nesse processo. Algumas técnicas comuns a ser utilizadas nessa etapa são: adivinhação, lista ordenada, eliminação de possibilidades, simetria e semelhança com outros casos, causa e efeito, análise sistêmica, estruturação de uma equação, desenhos, busca por padrões etc.

A terceira etapa consiste em executar o plano realizado na etapa anterior e efetivamente solucionar o problema.

Na quarta e última etapa, devemos verificar a solução encontrada. Para isso, necessário validar todas as entradas possíveis apresentadas no problema e identificar possíveis pontos de falhas.

CONEXÃO

Para praticar o método de solução de problemas, acesse os endereços a seguir:
Torre de hanoi: <http://www.ufrgs.br/psicoeduc/hanoi/>
Teste de Einstein: <http://rachacuca.com.br/teste-de-einstein/ >

1.3 Etapas para construção de um algoritmo

Para que sejamos capazes de construir um algoritmo que seja claro e sem ambiguidade, precisamos criar certa estrutura e seguir algumas regras e passos. Nessa etapa, vamos entender como devemos nos organizar para que consigamos criar os algoritmos da melhor forma possível.

As etapas para a construção de um algoritmo são bem parecidas com as etapas para solução de problemas proposta por Polya, apresentadas no item anterior. A Figura 1 apresenta esta relação.

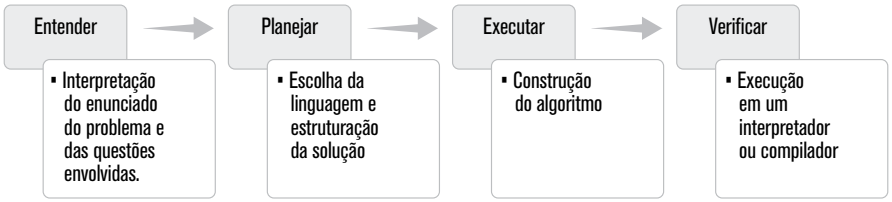


Figura 1 – Relação das etapas de solução de problemas de Polya com as etapas de construção de um algoritmo

Uma primeira diferença significante é a identificação e escolha da melhor linguagem para a construção do algoritmo, ou seja, qual a melhor forma de expressarmos a solução de nosso problema. Geralmente utilizamos a linguagem natural, ou seja, a nossa linguagem para descrever o problema. Esse tipo de linguagem permite a descrição e estruturação de quaisquer algoritmos, porém ela pode ser muito extensa de forma a dificultar a padronização e remoção de ambiguidades conforme desejamos, por isso geralmente optamos por linguagens mais estruturadas e formais. Algumas podem ser muito próximas de nossa linguagem natural, outras visuais, outras próximas das linguagens de máquina, como as linguagens de programação. Durante este curso, trabalharemos com linguagens dos três modelos citados, e principalmente veremos as diferenças e semelhanças entre elas.

É importante ter em mente que em uma primeira análise a linguagem que conhecemos é sempre a melhor, porém precisamos saber que há linguagens mais eficientes para a descrição de determinados tipos de problemas, para determinadas ocasiões. Esse é o motivo de existirem diversas linguagens de programações, que basicamente funcionam como diferentes línguas, como português, espanhol, inglês, alemão, as quais se utilizam de termos e regras diferenciados para descrever um mesmo objeto ou uma mesma situação. Tendo isso em mente fica claro que, se aprendermos os conceitos da lógica e da solução de problemas (algoritmos), ficará muito mais fácil navegarmos pelas diferentes linguagens e aprendê-las, conforme nossas necessidades.

Outro ponto diferente é a verificação; para os algoritmos e linguagens, geralmente existem programas ou ambientes de desenvolvimento. Utilizando estes am-

bientes, podemos testar e validar o correto funcionamento da solução implementada. Esses programas podem ser de dois tipos: interpretadores ou compiladores.

Interpretadores são programas que validam e executam diretamente os códigos na linguagem apresentada e apresentam o resultado em tela de saída. Os compiladores possuem um processo um pouco mais complexo. Eles convertem os códigos apresentados em linguagem de máquina, gerando um programa executável. Após a geração desse programa, temos de executá-lo para verificar seu funcionamento. A diferença básica para o usuário final é que, se usarmos interpretadores, iremos sempre precisar deles para executar ou realizar a solução; já nos compiladores, após criado o programa executável, não precisaremos mais deles, pois utilizaremos os programas gerados, independentemente de onde ou como foram desenvolvidos.

1.4 Representação de algoritmos

Como explicado anteriormente, um algoritmo pode ser escrito utilizando diferentes linguagens. Vamos conhecer melhor três tipos diferentes, a Linguagem Natural, a Linguagem Gráfica e as Pseudolinguagens.

No decorrer do curso, usaremos sempre os diferentes tipos de linguagens para compreender a melhor forma de estruturar e apresentar o algoritmo.

1.4.1 Linguagem natural

Como já dito, a linguagem natural é a linguagem do cotidiano. A escolha das palavras e termos utilizados dependem diretamente da pessoa que está escrevendo e da compreensão de quem lê. Sabemos que na linguagem natural uma mesma palavra pode ter diversos significados como, por exemplo:

Pilha: pilha pode ser um monte de objetos, ou pilha de objetos, ou pode ser uma pilha elétrica, que fornece energia para algum equipamento, também conhecida como bateria. Mas bateria também pode ser o instrumento musical de percussão.

A essa característica das linguagens naturais damos o nome de ambiguidade léxica. Porém, como vimos anteriormente, queremos que um algoritmo não seja ambíguo; logo, precisamos definir algumas regras para utilização.

1. Utilize frases curtas.
2. Use somente um verbo em cada frase, sempre no infinitivo ou no imperativo.

3. Evite palavras ambíguas.
4. Detalhe todas as etapas.

Por exemplo, como podemos descrever o algoritmo para fazermos um bolo utilizando a linguagem natural? Basicamente, por meio de uma receita de bolo que encontramos em qualquer livro de receitas, conforme segue abaixo:

Ingredientes:
2 xícaras (chá) de açúcar
3 xícaras (chá) de farinha de trigo
4 colheres (sopa) de margarina bem cheias
3 ovos
1 1/2 xícara (chá) de leite aproximadamente
1 colher (sopa) de fermento em pó bem cheia

Modo de preparo:
1. Bata as claras em neve.
2. Reserve.
3. Bata bem as gemas com a margarina e o açúcar.
4. Acrescente o leite e farinha aos poucos sem parar de bater.
5. Por último, agregue as claras em neve e o fermento.
6. Coloque em forma grande de furo central untada e enfarinhada.
7. Preaqueça o forno a 180° C por 10 minutos.
8. Asse por 40 minutos.

Perceba, na descrição dos ingredientes, que, quando uma palavra com possível ambiguidade aparece, é necessário removê-la. Por exemplo, em "4 colheres (sopa) de margarina bem cheias", é necessário especificar o tipo de colher a ser utilizada: no caso, a de sopa, pois, caso contrário, a pessoa que fará a receita pode se confundir e utilizar uma colher utilizada para sobremesa, café ou chá, já que existem diferentes tipos de colheres.

1.4.2 Linguagem gráfica

As linguagens gráficas são baseadas em imagens e esquemas, de forma a tentar facilitar a visualização e o entendimento das etapas e processos. Quando utilizamos uma linguagem gráfica de forma livre, esta também não está isenta de

ambiguidades, pois depende da criatividade e qualidade de quem está criando os desenhos para representar as atividades e os processos dos algoritmos ou para a resolução de problemas.

Um exemplo claro são os manuais de montagens que recebemos quando compramos alguns equipamentos, por exemplo, um ventilador, uma cadeira ou um móvel. A Figura 2 é um exemplo de que, muitas vezes, quando vamos montar algo que compramos, sempre sobram muitas peças, porque não entendemos as instruções ou por colocarmos porcas e parafusos em lugares errados.

Figura 2 – Exemplo de utilização de linguagem gráfica

Para evitar tais problemas, utilizaremos uma padronização de linguagem gráfica conhecida como Fluxograma.

O fluxograma é um diagrama padronizado utilizado para representações esquemáticas. Ele possui diversos elementos gráficos e é baseado na norma ISO 5807, que os padroniza, facilitando seu aprendizado e compreensão. Esse fluxograma é composto basicamente de figuras geométricas que representam as possíveis diferentes etapas de um processo, conectadas por setas que indicam a sequência das etapas e suas transições.

Veja na Figura 3 um exemplo de um fluxograma para um domingo com a família.

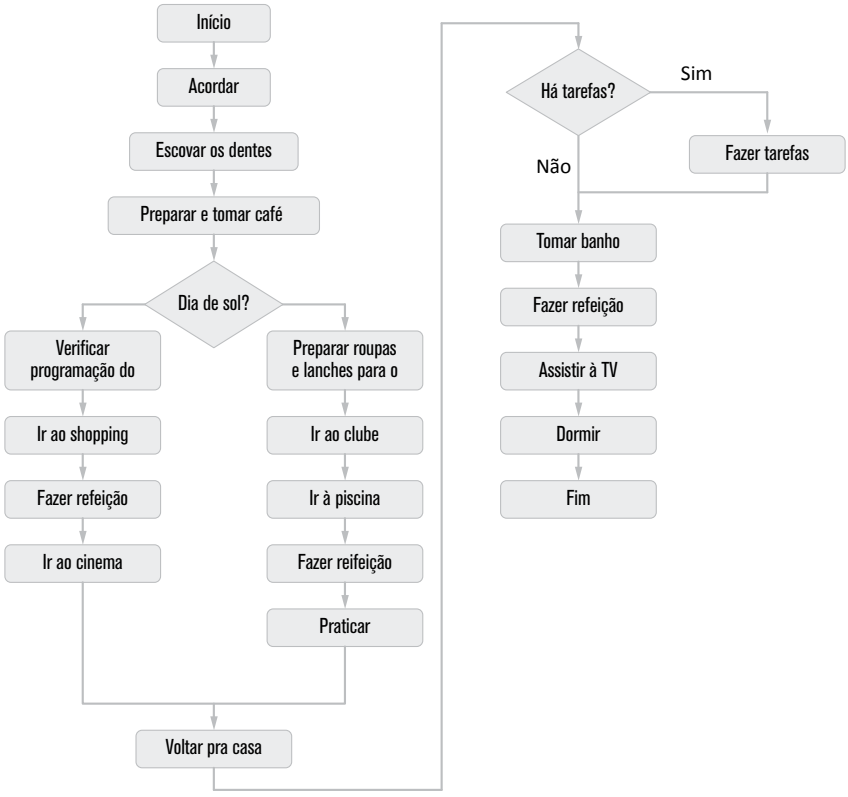


Figura 3 – Exemplo de fluxograma para domingo em família

Analisando o fluxograma acima, podemos começar a entender esse processo. Não se preocupem com os detalhes, pois os aprenderemos durante a disciplina. O fluxograma apresenta as tarefas sequencialmente, com uma indicação de início, e as setas indicando o fluxo das ações. Sempre que há uma possibilidade, utilizamos uma forma geométrica diferente para identificar essa possiblidade, e apresentamos as atividades das duas possibilidades conforme resposta à pergunta. Após percorrer todo o processo, encontraremos a indicação fim.

Para auxiliar na compreensão, vamos descrever o fluxograma acima utilizando linguagem natural.

Para um domingo com a família, siga os passos: acorde de manhã, escove os dentes, prepare e tome seu café, verifique se o dia está ensolarado. Em caso afirmativo, prepare as roupas e lanches para o clube, pegue a família e vá para o clube; lá, vá à piscina, depois faça uma refeição, depois pratique esportes e na

sequência, volte para casa. Caso o dia não esteja ensolarado, verifique a programação do cinema, pegue a família e vá para o *shopping*, faça uma refeição, vá ao cinema e volte para casa. Após chegar em casa, verifique se há tarefas a fazer, em caso positivo , faça-as. Após terminar, tome um banho; em caso negativo, tome um banho. Após tomar banho, faça uma refeição. Após a refeição, assista à TV e depois vá dormir. Assim, termina seu domingo.

1.4.3 Pseudolinguagem

Pseudolinguagem é uma forma de abstrair os problemas existentes na linguagem natural, de forma a criar um padrão para descrição e estruturação de algoritmos. Podemos entendê-la como uma forma intermediária entre a linguagem natural e as linguagens de programação. Ela foi criada com o objetivo de facilitar o aprendizado da lógica de algoritmos ou lógica de programação, devendo o aprendiz focar na resolução do problema ou lógica do problema, e não na estrutura e formalismo de como representá-lo de forma clara e sem ambiguidade.

Originalmente, as pseudolinguagens foram concebidas apenas para estudo e aprendizado, sem a utilização de sistemas computacionais, ou seja, não era possível executá-las em um computador para testar. Porém, com o avanço e a facilidade de acesso às tecnologias, foram criadas ferramentas para testar e validar automaticamente os algoritmos feitos utilizando as pseudolinguagens. Com isso, essas pesudolinguagens passam a ter características de linguagens, necessitando de regras sintáticas e semânticas bem definidas e estruturadas.

Em uma linguagem, a semântica é o significado das palavras e a sintaxe é a forma e a relação entre as palavras na formação de frase, para que façam sentido. Por exemplo, vamos analisar a frase “A maçã caiu da árvore“. Saber o significado das palavras, por exemplo, saber que maçã é aquele fruto arredondado de cor avermelhada ou verde, geralmente rajada de amarelo, com sabor doce e levemente ácido, proveniente de uma árvore de pequeno porte, com galhos pendentes, chamada de macieira, é conhecer a semântica das palavras. Para que a frase faça sentido, precisamos estruturá-la com um substantivo, depois um verbo seguido de um advérbio, para que a sintaxe necessária seja escrita

Quando extrapolamos isso para as pseudolinguagens ou linguagens de programação, temos que a semântica são os comandos ou palavras reservadas que nós podemos utilizar, e a sintaxe é a forma exata como devemos dispor cada palavra e símbolo para que a interpretação faça sentido.

É importante ter consciência de que, quando utilizamos ferramentas automatizadas para validar ou interpretar as pseudolinguagens ou linguagens de programação, devemos seguir as regras sintáticas e semânticas exatas, caso contrário a ferramenta não será capaz de compreender seu algoritmo. A falta de uma vírgula ou de um ponto é o suficiente para que a ferramenta acuse que seu algoritmo ou pseudocódigo esteja incorreto.

As pseudolinguagens são conhecidas também como linguagens algorítmicas, e existem diversas disponíveis para uso, como o PORTUGOL, o ILA e o UAL. Nessa disciplina, utilizaremos o UAL, mas veremos a seguir uma breve apresentação das demais.

PORTUGOL é uma ferramenta utilizada para o aprendizado de linguagem algorítmica desenvolvida pelo Departamento de Engenharia Informática do Instituto Politécnico de Tomar em Portugal. Trata-se de um *software* distribuído sob licença livre ou GNU, apresentado na Figura 4, utilizado para fins acadêmicos e educacionais. Um grupo de professores utilizou uma linguagem algorítmica em português, padronizada para definição dos comandos. Por sua simplicidade e facilidade de uso, o PORTUGOL é muito utilizado academicamente nos países de língua portuguesa, incluindo o Brasil.

CONEXÃO

Para baixar o *software* e conhecer a linguagem PORTUGOL, acesse: <http://www.dei.estt.ipt.pt/portugol/>.

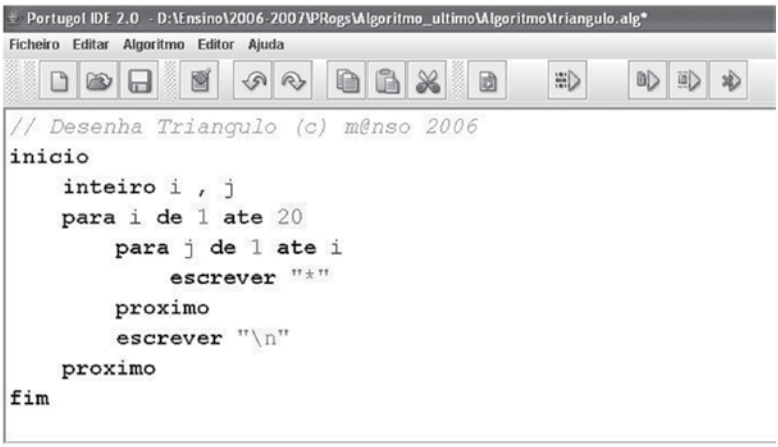


Figura 4 – Ambiente de desenvolvimento PORTUGOL (retirado de: <http://www.dei.estt.ipt.pt/portugol/node/2>)

O ILA, ou Interpretador de Linguagem Algorítmica, é um interpretador, muito semelhante ao PORTUGOL, desenvolvido por um grupo brasileiro da Universidade do Vale do Rio dos Sinos ou UNISINOS. Basicamente trata-se de um interpretador, pequeno e versátil, utilizado para testar algoritmos utilizando-se o português estruturado como linguagem.

CONEXÃO

Para saber mais sobre o ILA e baixar o interpretador, acesse: <http://professor.unisinos.br/wp/crespo/ila/>.

O UAL, ou *Unesa Algorithmic Language*, "é um interpretador animado que, através da execução e visualização das etapas de um algoritmo, auxilia no aprendizado do aluno iniciante" (SPALLANZANI; MEDEIROS & FILHO). Basicamente é um interpretador como os anteriores, que utiliza uma linguagem adaptada das linguagens Pascal e C, porém com termos e vocábulos em português. Foi desenvolvido como Projeto Final do curso de Informática da Universidade Estácio de Sá. O ambiente de desenvolvimento para Windows é apresentado na Figura 5.

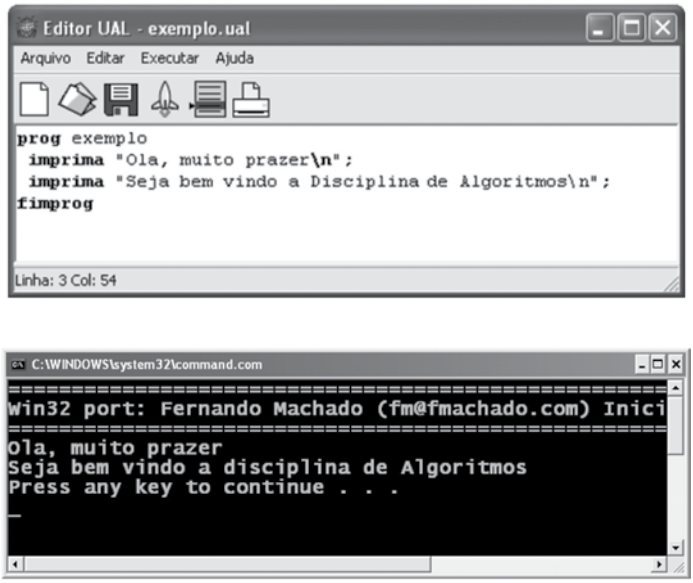


Figura 5 – Interpretador UAL

No decorrer da disciplina, aprenderemos a semântica e sintaxe da linguagem UAL aplicada à criação dos algoritmos. Veremos que ela é bem simples e didática, permitindo o aprendizado de uma forma estruturada e a compreensão simples e visual da ferramenta.

CONEXÃO

Para conhecer mais e baixar o interpretador, acesse: <http://geocities.ws/ual_language/ual.html>.

1.5 Construção de um algoritmo

Vimos anteriormente as etapas de resolução de problema de Polya e como elas podem ser aplicadas na construção de um algoritmo. Com base nas quatro etapas de Polya, vamos nos aprofundar na estruturação e construção de um algoritmo.

Considere o seguinte o problema.

Você esquece constantemente de pagar contas e, por isso, resolve desenvolver um programa para auxiliá-lo no cálculo dos juros. Para isso, é necessário ler o valor da prestação que deveria ter sido paga e o valor dos juros para calcular o novo valor da prestação com juros e exibi-la.

No primeiro momento, é necessário compreender o problema; para tal, deve-se identificar no enunciado os tipos de dados que serão processados e qual a sua origem; compreender o que deve ser feito e qual o conteúdo do dado de saída. Para o exemplo apresentado, podemos identificar o seguinte:

a) As informações serão digitadas no teclado (origem), sendo compostas por dois dados numéricos: o valor da prestação e o valor dos juros.
b) O algoritmo deverá calcular o novo valor da conta, já com os juros, que também será um valor numérico.
c) O algoritmo deverá mostrar o novo valor em tela (saída).

No segundo momento, é necessário identificar as operações e ações a serem executadas sobre os dados para obter o resultado final esperado. Dentre estas operações, podemos listar:

- Entradas e saídas de dados;
- Variáveis e constantes necessárias;
- Cálculos;
- Decisões através de comparações;
- Ciclos ou repetições.

Após identificar as ações necessárias, é importante identificar as semelhanças com outros problemas ou programa já realizados, dessa forma é possível aproveitar o trabalho já realizado e evitar erros. Para o exemplo apresentado, podemos identificar o seguinte:

a) São necessárias duas operações de entrada de dados, uma para ler o valor da conta e uma para ler o valor dos juros;
b) É necessário realizar um processamento para calcular juros e acrescer este valor na conta;
c) É necessária uma operação de saída para apresentar o novo valor da conta.

No terceiro momento, é necessário escolher a linguagem a ser utilizada e estruturar a sequência de execução das ações conforme a necessidade da linguagem. Para o exemplo apresentado, podemos escolher a linguagem natural e descrever o problema da seguinte forma:

1. Início
2. Obtenha o valor da conta
3. Obtenha o índice de juros
4. Multiplique o índice de juros pelo valor da conta
5. Some o valor dos juros ao valor da conta
6. Exiba o novo valor da conta
7. Fim

No quarto momento, é necessário analisar a solução realizada e refletir sobre alguns aspectos, como:

- A solução foi satisfatória?
- Existe uma forma mais simples para se resolver o problema?
- A solução, ou parte dela, poderá ser utilizada para resolver outros problemas?

Geralmente, essa etapa é realizada de forma superficial ou simplesmente não é realizada. Os programadores costumam dar pouca importância quando percebem que o objetivo foi alcançado e simplesmente passam para o próximo, mesmo que alguém perceba que existe uma melhor forma para resolver o problema. Nesse sentido, é muito comum ouvirmos frases como: "Está funcionando", "Se está funcionando, não mexa". Veremos no decorrer do curso que esta etapa está ligada a algumas etapas de engenharia de *software* e à qualidade de *software*, que são primordiais para garantir sistemas e programas de qualidade e com correto funcionamento evitando falhas e *bugs* indesejados.

É importante estar claro que o simples fato de "funcionar" não é o suficiente para garantir que temos um programa correto e eficiente, pois alguns programas podem funcionar para determinadas situações e para outras não. Além disso, é necessário analisar a velocidade e o consumo de recursos, como que memória e processador o programa está utilizando, antes de garantir que temos um problema solucionado, um programa eficiente funcionando.

Analisando o problema proposto, podemos identificar o seguinte:

a) A solução apresentada foi satisfatória;
b) Como o problema é simples, não há solução melhor, porém poderíamos realizar soluções mais genéricas para atender a contas que possuam formas mais complexas de cálculo dos juros;
c) O processamento realizado neste algoritmo é muito usual e pode ser utilizando para muitos outros algoritmos.

Vimos anteriormente de forma direta algumas considerações e preocupações que o programador deve sempre ter em mente no momento de estruturação e criação de um programa. É claro que esta forma simplificada oferece subsídios para a solução de problemas simples e o desenvolvimento de programas com complexidade baixa. Como já citado, no decorrer do curso serão estudadas formas mais completas e complexas de conceber e estruturar um sistema computacional, considerando todos os seus aspectos. Por hora, a estrutura proposta é o suficiente para a nossa disciplina de algoritimo. Por isso, tenha ela sempre em mente.

1.6 Lógica, lógica de programação e programa

Para entendermos os termos com mais clareza, vamos terminar esta unidade compreendendo melhor alguns conceitos, tais como: lógica, lógica de programação e conceito de programa.

Lógica

Como vimos anteriormente, a lógica é um ramo da filosofia que estuda e cuida das regras de estruturação do pensamento, do uso do raciocínio no estudo e na solução de problemas. Ela apresenta formas e técnicas para estruturação e argumentação utilizadas na solução de problemas.

Ela pode ser desenvolvida por meio do estudo dos métodos formais e estruturados e principalmente através da prática e do desenvolvimento do raciocínio.

Lógica de programação

A lógica de programação é a aplicação dos conceitos e práticas da lógica na utilização das linguagens de programação para o desenvolvimento de algoritmos na solução de problemas, respeitando regras da lógica matemática, aplicadas pelos programadores durante o processo de construção do *software*.

Conceito de programa

Programa é um algoritmo escrito ou codificado, que utiliza linguagem de programação. É composto por um conjunto de entradas, que são processadas e suas saídas resultantes. O processamento é realizado por meio de um conjunto de instruções e funções que serão convertidas em linguagem de máquina e interpretadas ou executas por um computador, para a realização de uma ou mais tarefas.

A obra e o legado de John Von Neumann

[...] Na área da computação, o nome de Von Neumann está geralmente associado à ideia de *arquitetura de von Neumann*, ou seja, à estrutura, hoje considerada clássica, de computadores digitais com programa armazenado na própria memória e, portanto, passível de automodificação e de geração por outros programas. Suas principais contribuições estão nas áreas de arquitetura de computadores, princípios de programação, análise de algoritmos, análise numérica, computação científica, teoria dos autômatos, redes neurais, tolerância a falhas, sendo o verdadeiro fundador de algumas delas. Saiba mais sobre o legado de von Neumann. Acesse o artigo que traz sua trajetória e as contribuições deste estudioso da computação, que deixou sua marca na história das grandes invenções tecnológicas. [...]

Acesse: <http://dx.doi.org/10.1590/S0103-40141996000100022 >

Trecho retirado do artigo de Tomasz Kowaltowski, Estud. av. vol.10 no.26 São Paulo Jan./Apr. 1996, disponível no endereço: < http://dx.doi.org/10.1590/S0103-40141996000100022 >

REFLEXÃO

Nessa primeira unidade, estudamos o que é um algoritmo, para que ele serve e como o utilizamos para resolver problemas – basicamente através da construção de um programa. Para isso, utilizamos uma linguagem, que pode ser linguagem natural, linguagem gráfica, pseudolinguagem ou linguagem de programação.

É importante que todos os conceitos vistos nessa unidade sejam bem assimilados e entendidos. Nas unidades seguintes aprenderemos como utilizar a sintaxe e a semântica das linguagens para a construção de algoritmos e programas. A estrutura e as regras dessas linguagens são bem simples e fáceis de visualizar, porém a capacidade de solução de problemas e suas aplicações são infinitas, e por isso o completo conhecimento e o controle dessas ferramentas serão obtidos por meio da prática, da solução dos problemas propostos. Nessa disciplina, não é necessário memorização, mas estruturação lógica do conhecimento aliada às estruturas que iremos estudar.

Esperamos ter despertado o desejo por este aprendizado e conhecimento, para que possamos exercitar e aprender com efetividade.

LEITURA

Conheça um pouco mais sobre o portugol:

<http://www.dei.estt.ipt.pt/portugol/node/32>
<http://orion.ipt.pt/~aulasi/ip/04-decisao/help/index.html>

Conheça uma variação do portugol – o WEBportugol:
<http://siaiacad17.univali.br/webportugol/>
<http://www.univali.br/webportugol>

Conheça um pouco mais sobre o UAL:
<http://geocities.ws/ual_language/ual.html>
<http://anitalopes.com>

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E.e A. V. Fundamentos da programação de computadores: Algoritmos, Pascal, C/C++ e Java. 2. ed. São Paulo: Pearson Education, 2008.

CRESPO, Sérgio. Linguagem ILA. Disponível em: <http://professor.unisinos.br/wp/crespo/ila/> . Acesso em: 26 abr. 2014.

FORBELLONE, A.L. V; EBERSPACHER, H. Lógica de programação. 3. ed. São Paulo: Makron Books, 2005.

MANSANO, Antonio; MARQUES, Célio; DIAS, Pedro. Portugol.Disponível em: <http://www.dei.estt.ipt.pt/portugol/>. Acesso em: 25 abr. 2014.

PUGA, S.; RISSETTI, G. Lógica de programação e estrutura de dados: com aplicações em Java. 1. ed. São Paulo: Pearson Education, 2003.

POLYA, G. How to Solve It. Princeton University Press. 1945.

SPALLANZANI, Adriana Sayuri; MEDEIROS, Andréa Teixeira de; FILHO, Juarez Muylaert, Linguagem UAL. Disponível em: <http://geocities.ws/ual_language/ual.html>. Acesso em: 25 abr. 2014.

NO PRÓXIMO CAPÍTULO

No próximo capítulo, aprenderemos como fazer nossos primeiros algoritmos ou programas utilizando as linguagens propostas. Para tal, veremos a estruturação dessas linguagens e seus comandos mais básicos, para início e fim de um programa, entrada e saída de dados, e a criação e atribuição de variáveis.

2

Estrutura sequencial

2 Estrutura sequencial

Aprendemos anteriormente o que é um algoritmo e como estruturá-lo. Vamos agora ver na prática como escrever esses algoritmos utilizando lingua gens predefinidas.

Aprenderemos a utilizar três diferentes linguagens: o UAL, para reforçarmos conceitos utilizando nossa língua nativa, para facilitar desta forma o entendimento, linguagem que poderíamos classificar como pseudolinguagem; o C++, que será a linguagem base de programação estruturada para o aprendizado de linguagem, com comandos simples de fácil entendimento; e o fluxograma, que permite uma visualização gráfica dos conceitos e do funcionamento dos comandos.

Fique muito atento pois essa unidade apresenta os subsídios que serão utilizados em todos os algoritmos e programas que você fizer e serão utilizados em todos os outros comandos.

OBJETIVOS

- Definir variáreis.
- Conhecer e listar os tipos de dados.
- Construir algoritmos de forma sequencial, utilizando comandos de atribuição, entrada e saída, e operadores aritméticos.
- Conhecer os utilizar operadores aritméticos.
- Identificar os operadores relacionais e lógicos.

REFLEXÃO

Estudamos anteriormente as quatro etapas para a construção de um algoritmo e para a resolução de um problema. Agora não falaremos mais delas diretamente, mas é muito importante que as tenha sempre em mente para a construção dos algoritmos e programas e principalmente para a realização das atividades propostas.

Em outras disciplinas, você deve ter aprendido sobre a estrutura básica do computador, em que temos o programa e os dados sempre em memória (RAM) durante sua execução. Falaremos sobre a criação de variáveis e tipos de dados, o que tem uma relação direta no uso da memória, pois são os dados utilizados pelo programa. Caso tenha dúvidas sobre a estrutura

do computador, bem como sobre memória, volte às disciplinas específicas e tenha estes conceitos claros para facilitar o entendimento.

2.1 Características da estrutura sequencial

Antes de falar da estrutura sequencial vamos, relembrar o conceito básico de um programa de computador, para que possamos entender como ele é criado. Temos três momentos fundamentais em um programa, representados na Figura 6 abaixo.

Figura 6 – Estrutura básica de um programa

Com isso em mente temos de entender que nosso sistema sempre precisará obter os dados de algum lugar, processar estes dados e fornecê--los processados para algum lugar. Geralmente a entrada provém dos dispositivos de entrada, principalmente do teclado, ou dos dispositivos de armazenamento, como o disco rígido ou HD. O processamento é composto pela lógica implementada por você, e a saída geralmente é dada nos dispositivos de saída, sendo o principal a tela ou o disco rígido (HD), que também é muito utilizado.

Analisando a estrutura básica, podemos perceber que trata-se de uma sequência de etapas a ser executada, o que nos leva ao entendimento da estrutura sequencial. É um tipo de estrutura em que os comandos são executados em uma ordem preestabelecida, onde cada comando só é executado após o anterior ser finalizado, ou seja, de forma sequencial.

A forma como iremos estruturar os comandos para preestabelecer a ordem de execução é definida pela sintaxe da linguagem, e a disposição sequencial dos diferentes comandos irá compor e implementar a lógica necessária para a resolução do problema.

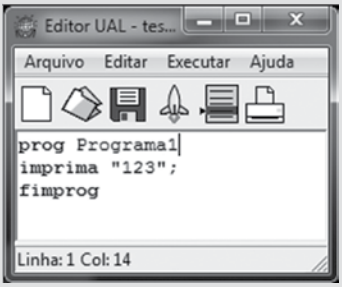
2.2 Comandos de início e fim

A primeira etapa para se estruturar um algoritmo ou programa é delimitá-lo, ou seja, definir claramente o ponto de início do algoritmo ou programa e o

ponto final deste. Nessa disciplina, veremos que isso coincidirá com a primeira e última linha escritas, porém, no decorrer do curso, você perceberá que nem sempre isso será assim, por isso é importante ter este conceito bem entendido.

Toda linguagem possui um comando ou representação para determinar o início e o fim. Veremos isso nas linguagens propostas.

No UAL, indicamos o início do algoritmo através da palavra reservada prog seguida de um nome do programa, nome qualquer a ser definido por você. Esse nome deve ser composto por uma palavra apenas, sem uso de espaços. Para delimitar e indicar o fim do programa, utilizamos a palavra reservada fimprog, com a qual o computador sabe que o programa está encerrado, o que dispara o processo de remoção do programa e dos dados associados a ele da memória. Abaixo são representados a estrutura e um exemplo.

LINGUAGEM UAL	EXEMPLO
prog <nomeDoPrograma> ... fimprog	

Nas definições e apresentações de estruturas, sempre que utilizamos um conteúdo entre os sinais de maior e menor (< >) como em <nomeDoPrograma> estamos indicando que é algo que deve ser definido por você, programador, de acordo com seu programa ou aplicação. Nesse caso, como pode ser visto no exemplo, no momento de programação deve ser substituído pelo nome do seu programa, que no caso do exemplo é "primeiro".

No C++, o início do programa é indicado pelas palavras reservadas int main() **{** e o final pela palavra reservada **}**. Em C++, as chaves (**{ }**) são utilizadas para delimitar um conjunto de códigos ou os códigos inseridos dentro de um comando. As palavras reservadas **int main** () indicam o início da função ou conjunto de códigos principais, que serão executados no início do programa. Essa estrutura indica a criação de uma função, porém isso não é o foco de nossa disciplina e essa estruturação será estudada em uma próxima disciplina de programação. Portan-

to não se preocupe em entendê-la agora. Veja a seguir a estrutura e um exemplo.

LINGUAGEM C++	EXEMPLO

#include <iostream>
using namespace std;
int main()
{
...
}
	Programa1.cpp
	`#include <iostream>`
	`using namespace std;`
	`int main() {`
	`  cout << "123";`
	`}`

Perceba que há, tanto na definição quanto no exemplo, as duas linhas iniciais não tratam do início do programa, mas, sim, de definições da linguagem, que o pacote de funções predefinidas da linguagem chamado "*iostream*" será utilizado e que o *namespace* padrão "std" será utilizado. Não é necessário entender agora o que é o pacote de funções de funções predefinidas ou o *namespace*, mas ambos são necessários para o funcionamento do programa.

No fluxograma, é necessário utilizar os símbolos de início e fim que são representados por retângulos com os cantos arredondados com os termos início e fim, em cada um, indicando visualmente os pontos do desenho esquemático, conforme é ilustrado a seguir.

FLUXOGRAMA	EXEMPLO

2.3 Variáveis

O principal objetivo dos algoritmos computacionais é a manipulação de dados para gerar informações, as quais podem ser especificadas pelo usuário ou geradas ao longo da execução do algoritmo. Para a manipulação de informações nos algoritmos, é necessária a utilização de um recurso denominado variáveis.

Em um programa, quando queremos manipular ou processar um dado por meio de um conjunto de transformações ou associações sobre ele, precisamos que este dado seja armazenado temporariamente na memória. Uma variável nada mais é que um espaço na memória do computador onde podemos guardar um dado temporariamente e alterá-lo de acordo com nossa necessidade. Para o computador, o ideal é que trabalhemos indicando o endereço de memória onde os dados estão, mas, para facilitar nossa vida, as linguagens de programação abstraem esta complexidade por meio da criação da variável.

De acordo com o tipo de informação que queremos armazenar ou manipular, é necessário especificar o tipo que será a variável. Esse conceito é conhecido como tipo de dados. Um tipo de dados determina quais tipos de operações, quais tipos de valores podem ser manipulados pelas variáveis e para o computador, este tipo de dados informa como ele deve guardar isso na memória.

CONCEITO

Uma variável é um elemento sintático do algoritmo que tem como funcionalidade armazenar um determinado valor. O valor associado à variável depende do tipo de informação que se deseja manipular, assim diversos tipos de valores podem ser atribuídos às variáveis.

2.3.1 Tipos de dados

Durante nossa vida, principalmente na vida escolar, aprendemos a lidar com diferentes tipos de dados de informações e aprendemos as diferentes formas de tratá-los e as suas operações associadas. Por exemplo, em matemática, aprendemos a lidar com números e que existem diferentes tipos ou conjuntos de números, como os naturais (N), os inteiros (Z), os reais (R) e os complexos (C); em português, aprendemos a lidar com dados literais, que são as letras e palavras. Associados a esses tipos de dados, aprendemos as operações soma,

divisão, multiplicação, potenciação, a junção de um conjunto de letras para forma uma palavra, a junção de palavras para formar uma frase etc. Veremos que em algoritmos e programação existem, da mesma forma, diferentes tipos de dados que podemos manipular e diferentes tipos de operações que podem ser utilizadas associadas aos tipos de dados.

c de dados primitivos, apenas para contextualizar os tipos de dados compostos são criados a partir de tipos de dados primitivos e serão estudados mais detalhadamente no decorrer do curso.

Os tipos de dados primitivos são aqueles fornecidos nativamente nas linguagens de programação, e não obrigatoriamente serão os mesmos em todas as linguagens. Usaremos os tipos mais comuns listados abaixo.

TABELA – TIPO DE DADOS

TIPO	DESCRIÇÃO	UAL	C++
Inteiro	Números inteiros	int	int
Real	Números fracionários ou de ponto flutuante	real	float
Caractere	Composto por um ou mais caractere ou também conhecido como alfanumérico	string	char (apenas um caractere)
Lógico	Tipo lógico que pode assumir os valores Verdadeiro ou Falso	logico	bool

2.3.2 Declaração e atribuição de variáveis

Para que possamos utilizar uma variável, um algoritmo ou programa, precisamos primeiro declará-la. Para o computador, essa é uma etapa muito importante, pois é nesse momento que o computador reserva o espaço de memória necessário para a variável, o espaço e o local que não será alterado enquanto a variável existir.

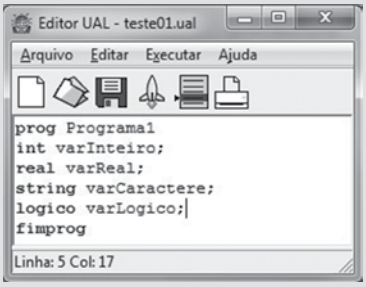
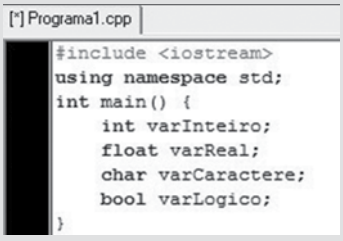
Para que possamos declarar uma variável, primeiro precisamos definir que tipo de dado ela irá armazenar, pois, quando se define um tipo de variável, informamos ao computador o quanto de memória será necessário "separar" e que tipos de operações poderão ser realizadas com o dado que será armazenado.

Na maioria das linguagens, as variáveis precisam ser declaradas antes de serem usadas.

Para se declarar uma variável, usamos a seguinte sintaxe:

```
tipo <nomeDaVariável>
```

Seguem abaixo alguns exemplos de declaração de variáveis:

TIPO	UAL	C++
Inteiro		
Real		
Caractere		
Logico		

Após declarar uma variável, podemos utilizá-la da forma que desejarmos. Geralmente, a primeira ação relacionada a uma variável é a atribuição de valores, a qual é feita através de um comando de atribuição que diz ao computador para copiar um valor para o espaço de memória daquela variável. A seguir, veja o comando de atribuição das linguagens.

LINGUAGEM UAL	LINGUAGEM C++
<variável> <- <valor>	<variável> = <valor>

No UAL, a atribuição é feita pelo símbolo menor seguido de menos <- e em C++ pelo símbolo =. Para atribuir é importante saber o tipo da variável, pois só é possível atribuir valores de tipos compatíveis com o tipo da variável, ou seja, para uma variável do tipo inteiro, é possível apenas atribuir valores inteiros; se tentarmos atribuir valores fracionados ou literais teremos um erro de programação. Seguem alguns exemplos de atribuição de valores.

LINGUAGEM UAL	LINGUAGEM C++

varInteiro <- 213;
varReal <- 2.5;
varCaractere <- “Ola”;
varLogico <- falso;
	varInteiro = 213;
	varReal = 2.5;
	varCaractere = “E”;
	varLogico = false;

Observe que os tipos numéricos são informados diretamente, e os literais são informados entre aspas. A divisão de parte inteira e fracionada dos reais é feita com ponto. Os tipos lógicos são verdadeiro e falso para o UAL e true e false para o C++.

2.4 Comando de entrada de dados – LEIA

Retomando a estrutura básica de um programa, a primeira etapa é a entrada de dados. A entrada de dados mais utilizada e que trataremos nesta disciplina como fonte de dados é o teclado. Em um segundo momento, os resultados são apresentados em dispositivos de saída como monitor ou impressora. Durante o processo de construção de algoritmos, o programador pode contar com instruções específicas para carregar informações fornecidas pelo usuário e para apresentar resultados na tela – essas instruções são chamadas de comandos de entrada e saída.

Para a leitura de informações, o comando utilizado para entrada de dados é conhecido como leia. Assim, com este comando, é possível transferir uma in-

formação digitada pelo usuário com o teclado para uma determinada variável no escopo do algoritmo. Por outro lado, para a apresentação de uma informação na tela, é utilizado um comando de saída específico denominado escreva.

Iniciamos, assim, o uso de funções ou comandos predefinidos, como o comando leia, que trata-se de uma palavra reservada da linguagem, que já possui uma ação predeterminada associada, que no caso é a leitura de dados do dispositivo de entrada padrão.

Vejamos abaixo como utilizar esses comandos nas diferentes linguagens.

COMANDO LEIA		
	LINGUAGEM UAL	LINGUAGEM C++
SINTAXE	**leia** <nomeDaVariavel>;	**cin** >> <nomeDaVariavel>;
DESCRIÇÃO	Palavra reservada leia seguida do nome da variável. É possível ler apenas uma variável de cada vez, seguido de ponto e vírgula.	Palavra reservada cin seguida do símbolo obrigatório >> seguido do nome da variável, seguido de ponto e vírgula.
EXEMPLOS	**leia** v1; **leia** v2; **leia** v3;	**cin** >> v1; **cin** >> v2; **cin** >> v3;

É importante nos atentarmos à necessidade de declarar a variável antes de sua leitura, ou seja, para que seja utilizada uma variável com o comando de leitura, é obrigatório que a variável utilizada tenha sido declarada anteriormente.

Fluxograma:

O símbolo anterior representa o comando que libera o teclado para que o usuário digite o dado que lhe for solicitado. Como já apresentado, consideramos sempre o teclado como o dispositivo de entrada padrão.

Dentro do símbolo, virá o nome da variável que receberá o dado que for digitado.

Se você tiver mais de um dado para ser digitado, poderá usar a vírgula para separar os nomes das varáveis.

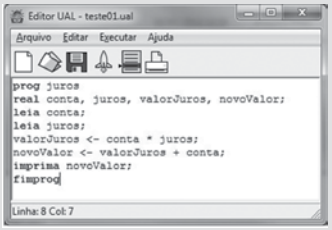
Antes de continuarmos e estudarmos os comandos de saída de dados, vamos reforçar alguns conceitos. No início desta aula, vimos as três etapas principais de um software: Entrada, Processamento e Saída. Com o que aprendemos, já é possível estruturarmos um pouco melhor os conceitos associados a essas etapas.

Você se lembra do algoritmo de cálculo de juros que construímos ao final da Unidade 1? Caso não se lembre, volte e releia. Vamos agora identificar os componentes que já aprendemos para resolver aquele problema.

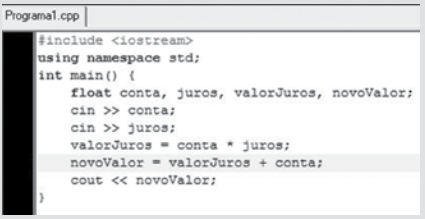
DADOS DE ENTRADA	
quantidade: 2 tipos: real, real	nomes: conta, juros
DADOS INTERMEDIÁRIOS	
quantidade: 2 tipos: real, real	nomes: valor Juros, novoValor
PROCEDIMENTO	
1 – obter conta, juros 2 – valorJuros <- conta * juros 3 – novoValor <- valor + valorJuros 4 – exibir novo Valor	

Veja o algoritmo implementado.

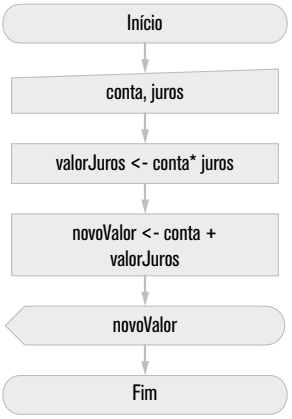
LINGAUGEM UAL



LINGAUGEM C++



Fluxograma



Veja na Figura 7 a execução do algoritmo utilizando o Editor UAL.

Figura 7 – Resultado da execução do algoritmo de cálculo de juros

Provavelmente, se você tentar reproduzir este código no Editor UAL, pode ter algumas dificuldades para obter a saída mostrada anteriormente.

Vamos, então, ver quais podem ser os principais pontos de dificuldade:

1. Ao executar, aparentemente nada acontece, de acordo com a Figura 8 abaixo. Na verdade, o programa está aguardando a entrada do primeiro comando leia; basta entrar com o dado da conta, que no caso do exemplo anterior foi 100.

Figura 8 – Programa aguardando entrada.

2. Se você digitar o número 100, receberá uma imagem de erro, conforme a Figura 9 a seguir – isso porque a variável é do tipo real, e 100 é um número inteiro.

Figura 9 – Programa erro na leitura de variável

3. Para que o programa interprete como um valor do tipo real, é necessário informar o valor explicitando que a variável é real, mesmo que a parte fracionária seja zero; dessa forma, deve-se entrar com o valor 100.0, conforme Figura 10 a seguir.

Figura 10 – Forma correta de entrar com uma variável do tipo real.

4. Basta informar o valor da taxa de juros, lembrando que também é do tipo real, e o programa lhe retornará o novo valor da conta.

2.5 Comando de saída de dados – IMPRIMA

Ao executar o programa anterior, percebemos que não é muito fácil utilizá-lo, pois não temos informações do que devemos fazer. Se exibirmos uma mensagem antes da leitura dos dados, fica bem mais fácil sua utilização. Para exibir conteúdo, como é feito ao final do programa, utilizamos o comando **imprima**.

O comando escreva pega o conteúdo da memória interna e joga no dispositivo de saída padrão, que no nosso caso é o monitor, logo ele irá imprimir ou mostrar o conteúdo na tela do usuário. Vejamos a seguir a estrutura do comando imprima.

COMANDO ESCREVA		
	LINGUAGEM UAL	LINGUAGEM C++
SINTAXE	**imprima** <conteúdo>;	**cout** << <conteúdo>;
DESCRIÇÃO	Palavra reservada imprima seguida de uma variável, de um conjunto de caracteres ou de variáveis ou conjuntos de caracteres separados por virgula., seguido de ponto e vírgula.	Palavra reservada cout seguida do símbolo <<, seguido de uma variável, de um conjunto de caracteres ou de variáveis ou conjuntos de caracteres separados pelo símbolo <<, seguido de ponto e vírgula.
EXEMPLOS	**imprima** v1; **imprima** "Variável: ", v1; **imprima** v1, " - ", v2;	**cout** << v1; **cout** << "Variável: " << v1; **cout** << v1 << " - " << v2;

Perceba que, diferentemente do comando leia, o comando escreva permite que se utilize mais de uma variável no mesmo comando, obrigando apenas o uso de um separador. Ele permite ainda utilizar em um mesmo comando variá-

veis e constantes, conjuntos de caracteres.

Fluxograma:

O símbolo anterior representa o comando que joga o conteúdo indicado na tela do usuário, considerando sempre a tela do usuário como o dispositivo de saída padrão.

Dentro do símbolo, virá o nome da variável, conteúdo ou expressão, que será mostrada. Se você tiver mais de um dado para ser mostrado, poderá usar a vírgula para separar os nomes das variáveis, ou conteúdos.

Já que conhecemos melhor o comando de escrita, que tal melhorarmos um pouco o programa UAL para cálculo dos juros? Vamos exibir algumas mensagens que facilitam o uso do programa. Veja a Figura 11 a seguir com o novo código e o resultado.

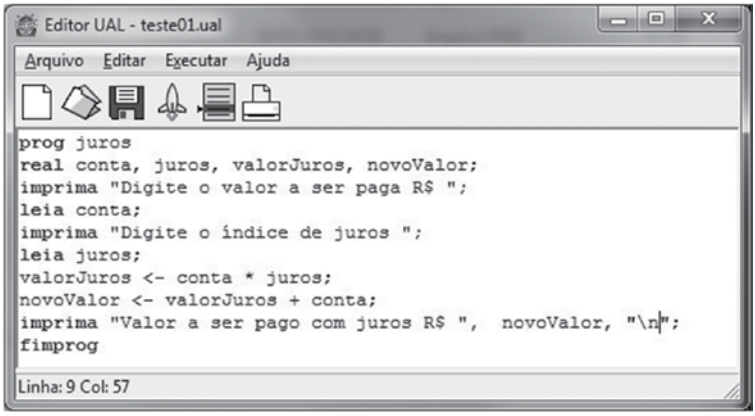


Figura 11 – Novo código e resultado do programa de cálculo de juros.

Devemos concordar que a solução ficou bem melhor e mais elegante. Perceba que no último comando de escrita utilizamos um símbolo diferente "\n". Esse símbolo realiza a quebra de linha ou insere uma linha na impressão. Compare a última linha do resultado com o anterior. No anterior, quando não tínhamos o \n, o texto do programa UAL "*Press any key to continue...*" está na mesma linha logo após o valor calculado; já neste exemplo o texto está uma linha abaixo, desta forma conseguimos estruturar melhor a apresentação. Além desses outros símbolos, podem ser utilizados para formatar melhor a saída estes símbolos estão presentes na tabela ASCII, você pode consultá-la para identificar novas possibilidades e testá-los.

CONEXÃO

Para conhecer mais símbolos, consulte a tabela ASCII: <http://en.wikipedia.org/wiki/ASCII>

CONCEITO

Símbolo \n – equivalente à tecla enter em um editor de texto.
Símbolo \t – equivalente à tecla tab em um editor de texto.

2.6 Operadores aritméticos e lógicos

Os computadores podem ser encarados como grandes máquinas capazes de realizar uma quantidade imensa de operações por segundo. Na evolução histórica da computação, os primeiros algoritmos foram criados com o intuito de solucionar complexas expressões em poucos segundos. As expressões são formadas essencialmente por operadores, operandos e parênteses. De acordo com o tipo de operador utilizado, podemos criar expressões denominadas aritméticas, relacionais ou lógicas, conforme estudaremos a seguir.

2.6.1 Expressões aritméticas

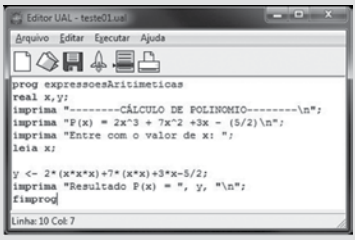
As expressões aritméticas utilizam os operadores aritméticos da matemática tradicional para criar expressões capazes de resolver os mais variados tipos de

funções. Nesse grupo, os principais operadores utilizados são:

OPERADOR	LINGUAGEM UAL	LINGUAGEM C++
Soma	+	+
Subtração	-	-
Multiplicação	/	/
Divisão	*	*
Resto da divisão	%	%

A Figura 12 apresenta a codificação de um algoritmo que utiliza expressões aritméticas para resolver um polinômio.

LINGUAGEM UAL



LINGUAGEM C++

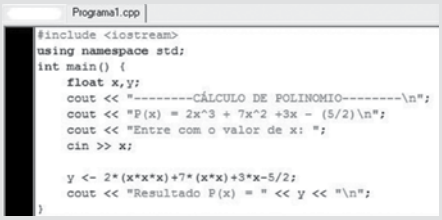


Figura 12 – Algoritmo computacional que utiliza expressões aritméticas.

A Figura 13 apresenta o resultado da execução do algoritmo.

Figura 13 – Resultado da execução do algoritmo

A precedência ou ordem de execução dos operadores aritméticos é a mesma que aprendemos na matemática. Caso seja preciso alterar a ordem, é necessário o uso de parênteses ( ) delimitando as operações que devem ser realizadas primeiro, utilizando os mesmo princípios da matemática.

2.6.2 Expressões relacionais

Explicativo

Em uma expressão relacional, o resultado produzido pela avaliação da expressão é sempre um valor lógico. Assim, uma expressão relacional resulta em um valor do tipo verdadeiro ou falso.

Quando construímos algoritmos, é muito comum a necessidade de comparar ou estabelecer relações entre determinados operandos. Para isso, podemos utilizar um conjunto específico de operadores capazes de produzir expressões relacionais. Os operadores relacionais conhecidos são:

OPERADOR	LINGUAGEM UAL	LINGUAGEM C++
maior	>	>
menor	<	<
menor ou igual	<=	<=
maior ou igual	>=	>=
igual	==	==
diferente	<>	!=

A Figura 14 demonstra a construção de um algoritmo computacional que utiliza expressões relacionais. Note, no algoritmo, que todos os operadores relacionais são expressões por meio de símbolos.

LINGUAGEM C++

Programa1.cpp

```
#include <iostream>
using namespace std;
int main() {
    int a,b;
    cout << "--------EXPRESSOES RELACIONAIS--------\n";
    cout << "Entre com o valor de a: ";
    cin >> a;
    cout << "Entre com o valor de b: ";
    cin >> b;

    cout << "\nRelacoes entre os valores "<<a<<" e "<<b;
    cout << "\nIgualdade: "<< (a == b);
    cout << "\nDiferenca: "<< (a != b);
    cout << "\n'a' e maior que 'b': "<< (a > b);
    cout << "\n'a' e menor que 'b': "<< (a < b);
    cout << "\n'a' e maior ou igual que 'b': "<< (a >= b);
    cout << "\n'a' e menor ou igual que 'b': "<< (a <= b);
    cout << "\n";
    system("pause");
}
```

Figura 14 – Algoritmo computacional que demonstra a utilização de expressões e operadores relacionais.

Nesse caso, apresentamos o exemplo apenas na linguagem C++, pois o programa que interpreta a linguagem UAL não possui suporte a impressão do tipo lógico, que é o resultado das expressões relacionais. A Figura 15 apresenta o resultado da execução do algoritmo. Perceba que o valor lógico no C++ é tratado como um número 0 para falso e 1 para verdadeiro.

Figura 15 – Resultado da execução do algoritmo

Em um programa, cada variável possui um tipo preestabelecido, de acordo com o tipo de dados visto anteriormente. Quando usamos os operadores relacionais, temos que ter o cuidado de relacionar variáveis que sejam do mesmo tipo.

Na construção de algoritmos computacionais também é possível criar expressões a partir de operadores lógicos, as quais são conhecidas como expressões lógicas. Para a criação de uma expressão lógica, é necessário utilizar operadores booleanos, em que os principais são:

OPERADOR	LINGUAGEM UAL	LINGUAGEM C++
negação (NOT)	!	!
conjunção (E)	&&	&&
disjunção (OU)	||	||
disjunção-exclusiva (XOR)	<não existe>	^

Na Figura 16, é apresentada a codificação de um algoritmo capaz de calcular a tabela verdade a partir de expressões lógicas. Nesse exemplo, é possível determinar um dos principais conceitos da lógica matemática conhecido como tabela verdade.

LINGUAGEM C++

Programa1.cpp

```
#include <iostream>
using namespace std;
int main() {
    bool a,b;
    cout << "--------EXPRESSOES LOGICAS--------\n";
    cout << "Entre com o valor logico de a: ";
    cin >> a;
    cout << "Entre com o valor logico de b: ";
    cin >> b;

    cout << "\nOperacao logia entre os valores "<<a<<" e "<<b;
    cout << "\nNegacao a: "<< (!a);
    cout << "\nNegacao b: "<< (!b);
    cout << "\nConjuncao (E): "<< (a && b);
    cout << "\nDisjuncao (OU): "<< (a || b);
    cout << "\nDisjuncao exclusiva (XOR): " << (a ^ b);
    cout << "\n";
    system("pause");
}
```

Figura 16 – Algoritmo computacional que demonstra a utilização de expressões e operadores lógicos.

A Figura 17 apresenta o resultado da execução do algoritmo.

Figura 17 – Resultado da execução do algoritmo

CONEXÃO

Para um estudo aprofundado a respeito de tabelas verdade, você poderá consultar as seguintes referências: <http://pt.wikipedia.org/wiki/Tabela_verdade>

CURIOSIDADE

Os operadores lógicos são muito importantes para a construçãode programas, principalmente para a elaboração de expressões relacionais compostas. Para complementar seu estudo, você poderá utilizar a seguinte referência:
<http://pt.wikipedia.org/wiki/ Operadores_l%C3%B3gicos>

Com isso, concluímos o estudo a respeito dos operadores e das expressões aritméticas, relacionais e lógicas. Aproveite esse momento para construir algoritmos a partir das atividades propostas.

Para demonstrar a utilização dos comandos e a estrutura estudadas anteriormente, vamos construir uma calculadora simples de quatro operações. Esse programa de utilizar da estrutura de um programa, obtendo as entradas necessárias, realização do processamento dos dados e saída. O início e o fim do programa devem estar corretamente definidos. O processamento deve ser executado utilizando as operações aritméticas.

Nessa calculadora, o usuário fornecerá dois valores numéricos para serem realizados os cálculos de soma, subtração, multiplicação e divisão. Para a leitura dos valores, será utilizado o comando leia e para a apresentação dos resultados será necessário o comando escreva. Vamos identificar os componentes que

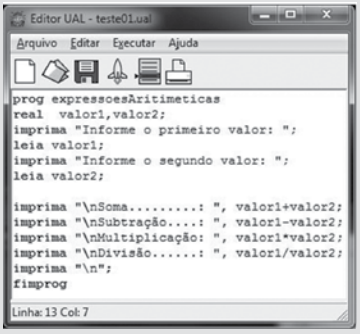
já aprendemos para resolver aquele problema. A Figura 18 apresenta o algoritmo codificado para solução do problema.

DADOS DE ENTRADA	
quantidade: 2 tipos: real, real	nomes: valor1, valor2
DADOS INTERMEDIÁRIOS	
quantidade: 4 tipos: real, real, real, real	nomes: soma, sub, mult, div

PROCEDIMENTO

1 – obter valor1, valor2
2 – soma <- valor1 + valor2
3 – sub <- valor1 - valor2
4 – mult <- valor1 * valor2
5 – div <- valor1 / valor2
6 – exibir soma, sub, mult, div

LINGUAGEM UAL



LINGUAGEM C++

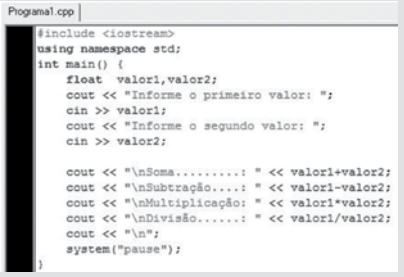


Figura 18 – Algoritmo que codifica um cálculo simples com quatro operações.

O resultado da execução do algoritmo é apresentado na Figura 19. Nessa execução, os valores fornecidos como entrada foram 10 e 20.

Figura 19 – Resultado da execução do algoritmo

ATIVIDADE

1. Construir um algoritmo capaz de calcular o consumo médio de combustível de um veículo. Para isso, o usuário deverá informar como entrada os quilômetros percorridos pelo veículo e o total de litros usados para abastecê-lo.

2. Escreva um algoritmo computacional capaz de calcular a área de um retângulo. Para isso, utilize a seguinte fórmula: área = (base*altura).

3. Elabore um algoritmo que seja capaz de realizar as seguintes conversões:
   a) litro para mililitro.
   b) quilômetros para metros.
   c) toneladas para gramas.

REFLEXÃO

Nessa segunda unidade, aprendemos os comandos básicos para a construção de algoritmos e programas. Sabemos como deve ser criado e estruturado um algoritmo, como devemos utilizar as variáveis para manipular os dados internamente e como realizar a entrada e a saída de dados. Essa estrutura básica será utilizada em todos os programas que você criar.
Faça os exercícios propostos e pratique bastante para que o entendimento e o uso da estrutura básica fiquem mais fáceis e sejam parte integrante de seu pensamento estruturado.

LEITURA

Uma forma fácil e estruturada para que você teste seus algoritmos e verifique seu funcionamento é a utilização do teste de mesa. Para aprender como realizá-lo, acesse: <http://gomeshp.plughosting.com.br/ed/testemesa.htm>.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E.e A. V. Fundamentos da programação de computadores: Algoritmos, Pascal, C/C++ e Java. 2. ed. São Paulo: Pearson Education, 2008.

FORBELLONE, A.L. V; EBERSPACHER, H. Lógica de programação. 3. ed. São Paulo: Makron Books, 2005.

PUGA, S.; RISSETTI, G. Lógica de programação e estrutura de dados: com aplicações em Java. 1. ed. São Paulo: Pearson Education, 2003.

POLYA, G. How to Solve It. Princeton University Press. 1945.

SPALLANZANI, Adriana Sayuri; MEDEIROS, Andréa Teixeira de; FILHO, Juarez Muylaert, Linguagem UAL. Disponível em <http://geocities.ws/ual_language/ual.html>. Acesso em: 25 abr. 2014.

NO PRÓXIMO CAPÍTULO

Agora que você já sabe construir um algoritmo ou programa vamos aprender no próximo capítulo a utilizar as estruturas condicionais. Essas estruturas vão nos permitir realizar decisões dentro dos algoritmos e transportar para dentro dos algoritmos ou programas diversas situações reais de nosso dia a dia.

3

Estruturas de decisão

3 Estruturas de decisão

Em nossa vida, geralmente nos deparamos com problemas que, em determinadas ações, estão condicionadas a um acontecimento, como, por exemplo: se eu não trabalhar este final de semana, então poderei ir à praia; se chover, não poderei ir à piscina; se eu ganhar na loteria, pagarei todas as minhas contas.
Todas as situações expostas dependem da conjunção subordinativa condicional/ partícula expletiva/ pronome reflexivo/ conectivo ou qualquer outra denominação que se queira dar 'se'. Provavelmente, não conseguiríamos viver sem o se, algumas vezes usamos até alguns senões. Os algoritmos e as linguagens de programação foram feitos para resolver problemas do dia a dia, logo precisam também desta estrutura condicional para que possam atender a um conjunto real de problemas.
Nesta unidade, estudaremos a estrutura condicional ou de decisão composta pelos comandos se ... então ... senão.

OBJETIVOS

- Utilizar operadores relacionais e lógicos.
- Conhecer as estruturas de decisão.
- Construir algoritmos utilizando se ... então ... senão.
- Construir algoritmos utilizando comandos de decisão aninhados.

REFLEXÃO

Na unidade anterior, estudamos operadores lógicos e relacionais. À primeira vista, eles não parecem ter muita utilidade em um programa de estrutura básica como estudado até aqui, porém são de extrema importância para que possamos tomar decisões em algoritmo ou programa. Associados à estrutura de decisão que será estudada nesta unidade, eles permitem a criação de diferentes caminhos dentro de nossos programas, permitindo-nos resolver problemas mais complexos.

3.1 Características de estrutura de decisão

A codificação de um algoritmo computacional é baseada em uma estrutura sintática e sua execução é sempre realizada de maneira linear, ou seja, o processo de execução começa na primeira linha do algoritmo e segue linha após linha, até a instrução de fim do algoritmo.

Durante a construção de soluções algorítmicas, é comum o programador avaliar o conteúdo das variáveis para tomar decisões. Uma das decisões mais empregadas é o desvio condicional do fluxo de execução do algoritmo. Assim, de acordo com o valor de uma variável, o algoritmo computacional poderá executar instruções diferentes. Em outras palavras, um determinado trecho do algoritmo será executado apenas quando o valor deuma variável satisfizer uma determinada condição. Para a elaboração destes tipos de desvios lógicos na execução dos algoritmos, são utilizadas as estruturas condicionais (ASCENCIO e EDILENE, 2002).

Existem quatro maneiras distintas de codificar *estruturas condicionais* em linguagem algorítmica que são conhecidas como:

- Condicional simples
- Condicional composta
- Condicional aninhada
- Condicional múltipla

O aprendizado do comando condicional ou de decisão nos possibilita vislumbrar o grau de complexidade das tarefas que os algoritmos poderão executar a partir de agora. Não se assuste com o tamanho dos programas nessa unidade. É assim mesmo, eles vão crescendo e ganhando complexidade conforme aprendemos novos comandos e funções.

Condicional simples: é a forma mais simples de decisão; os comandos do bloco condicional serão executados somente se a condição de teste do comando condicional for verdadeira.

Condicional composta: nessa forma, existem dois blocos de comando, um que será executado caso a condição de teste do comando condicional seja verdadeira, e outro que será executado caso o resultado do teste seja falso.

Condicional aninhada: após a execução do teste, será executado um bloco de comandos se o resultado do teste for verdadeiro, podendo ter um outro teste

(aninhado); caso não seja, um outro teste (encadeado) será feito até que todas as possibilidades de respostas sejam contempladas.

Condicional múltipla: permite a verificação de múltiplas possibilidades para uma mesma variável executando diferentes blocos de código, de acordo com seu valor. Permite especificar quantos valores sejam necessários para uma variável.

CONCEITO

Teste: O teste é uma expressão relacional que compara valores dos seus operandos..

Nesse momento, é muito importante termos de forma clara o que é o teste. É uma expressão que utiliza os operadores relacionais e lógicos vistos na unidade anterior para realizar uma comparação ou avaliação de uma expressão, resultando sempre em valor lógico. Veja alguns exemplos:

OPERAÇÕES OU TESTES		
SIGNIFICADO	MATEMÁTICA	EXEMPLO - UAL
Maior	>	2 * 8 > 4
Menor	<	2 < 9
Igual	=	3 == 7 – 4
Maior ou igual	≥	9 >= 5
Menor ou igual	≤	4 <= 10
Diferente	≠	8 <> 5
Conjunção (e)	∧	n>=100 && n<=200
Disjunção (ou)	∨	n<100 || n>200
Negação (não)		!p

Caso não lembre como funciona a conjunção, a disjunção e a negação, segue a tabela verdade com as respetivas saídas para cada possível combinação de entradas.

TABELA VERDADE

NÃO

E	S
V	F
F	V

E

E1	E2	S
V	V	V
V	F	F
F	V	F
F	F	F

OU

E1	E2	S
V	V	V
V	F	V
F	V	V
F	F	F

3.2 Comando condicional simples

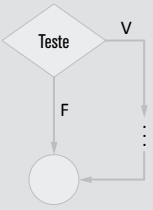
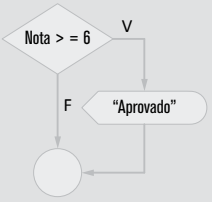
A estrutura condicional que pode ser definida em linguagem algorítmica e de programação é conhecida como se. Dessa forma, um determinado bloco de instruções do algoritmo será executado se (e apenas se) uma condição de teste for satisfeita. As linguagens de programação utilizam a instrução equivalente denominada if.

Uma estrutura condicional simples é utilizada para especificar que um bloco de instruções será executado apenas após a avalição de uma condição. O resultado da avalição será sempre um valor lógico, ou seja, verdadeiro ou falso. Dessa forma, o bloco de instruções será executado quando a avaliação da expressão resultar em um valor verdadeiro.

A seguir, são apresentados a estrutura e um exemplo de uso para o UAL, C++ e Fluxograma. O exemplo compara uma nota – se a nota for maior ou igual a seis, é impresso no dispositivo padrão de saída a palavra Aprovado.

LINGUAGEM UAL	EXEMPLO
se (<teste>) **{** <bloco de comandos> **}**	**se** (nota >= 6) **{** **imprima** "Aprovado"; **}**

LINGUAGEM C++	EXEMPLO
if (<teste>) **{** <bloco de comandos> **}**	**if** (nota >= 6) **{** **cout <<** “Aprovado“; **}**

FLUXOGRAMA	EXEMPLO
	

A melhor forma de aprender e entender o uso das estruturas de programação é praticando. Vamos a um exemplo completo, apresentado na Figura 20. O programa recebe um número inteiro como entrada e verifica se é positivo ou negativo.

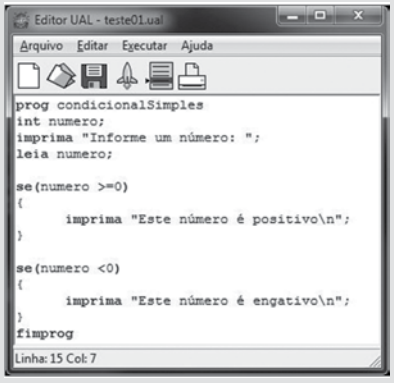
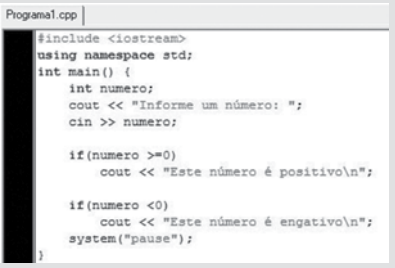
LINGUAGEM UAL	LINGUAGEM C++
	

Figura 20 – Algoritmo com comando condicional simples que verifica se um número é positivo ou negativo.

Perceba que no exemplo C++ não foram utilizados os parênteses antes e depois do bloco de comandos internos do comando condicional. Em C++, quando temos apenas uma linha de comando, os parênteses para delimitar o início e o fim de bloco de comandos é opcional e pode ser omitido, porém é importante que se tenha muita atenção ao omitir os parênteses: apenas a primeira linha de comando após o comando condicional será executada caso ele seja verdadeiro, que no caso do exemplo é o comando cout.

O resultado da execução do algoritmo é apresentado na Figura 21 e na Figura 22. No resultado deste exemplo, é importante notar que temos duas possibilidades de saída diferente: a mensagem "Este número é positivo" ou "Este número é negativo" será exibida de acordo com o valor informado pelo usuário. Assim, o algoritmo realiza o desvio do fluxo de execução a partir do valor definido na entrada de dados.

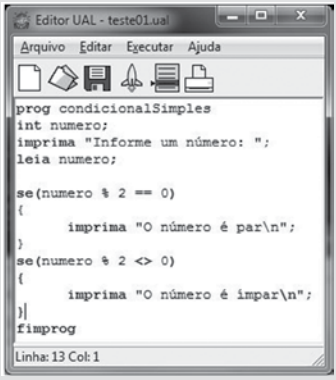
Figura 21 – Resultado da execução do algoritmo para um valor positivo.

Figura 22 – Resultado da execução do algoritmo para um valor negativo.

A Figura 23 a seguir apresenta a codificação um algoritmo computacional capaz de verificar se um número informado pelo usuário é par ou ímpar. Para isso, será utilizado no algoritmo o operador aritmético %, que é necessário para determinar o resto de uma divisão inteira. Um número é definido como par quando a divisão deste número por 2 resultar em resto zero, caso contrário este número é ímpar. Devemos nos atentar ao fato de estar utilizando mais de uma expressão dentro do teste. De acordo com a precedência dos operadores a operação módulo será executada primeiramente seguida pela operação de

comparação. Lembre-se de que, se quisermos alterar a ordem de precedência, precisaremos delimitar as operações com parênteses, porém isso não é necessário para este exemplo.

LINGUAGEM UAL



LINGUAGEM C++

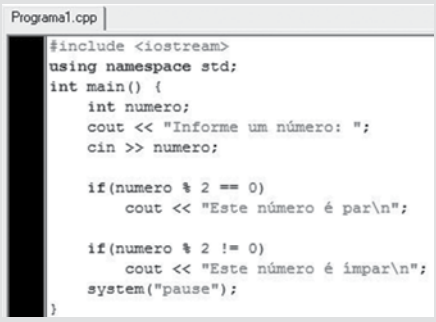


Figura 23 – Algoritmo com comando condicional simples que verifica se um número é par ou ímpar.

As Figura 24 e 25 apresentam o resultado da execução do algoritmo para os possíveis valores de entrada. Na Figura 6, é demonstrada a entrada de um valor par, e na figura 7, a entrada de um valor ímpar.

Figura 24 – Resultado da execução do algoritmo com entrada par.

Figura 25 – Resultado da execução do algoritmo com entrada ímpar.

3.3 Comando condicional composto

A estrutura condicional composta é um complemento da estrutura condicional simples, dessa maneira, com a condicional composta, é possível especificar o que será executado pelo algoritmo quando a expressão resultar em um valor verdadeiro e também definir o que será realizado quando ocorrer um resultado falso na expressão. Como uma expressão condicional pode resultar apenas em um valor lógico verdadeiro ou falso, a estrutura condicional composta permite codificar o comportamento do fluxo de execução do algoritmo em todas as situações possíveis.

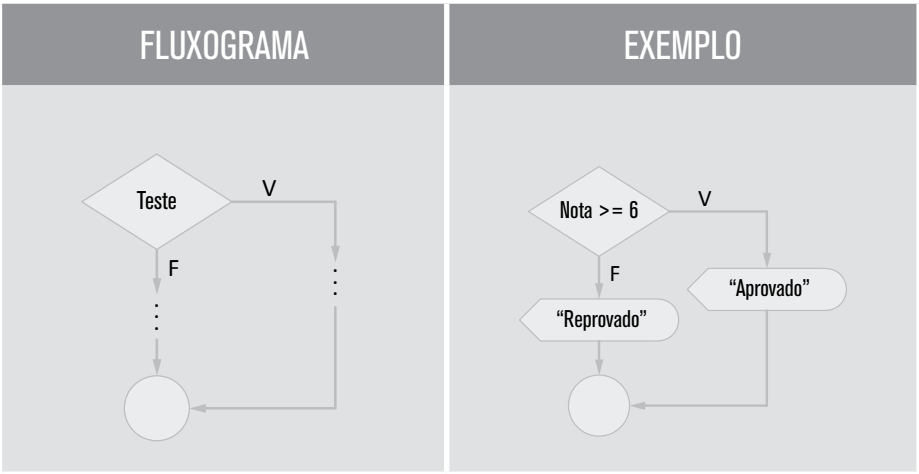
O trecho da estrutura condicional que especifica o que será realizado quando a expressão resultar em um valor falso é conhecido como **senão**. Nas linguagens de programação o bloco **senão** é chamado de **else**. A seguir, são apresentados a estrutura e um exemplo de uso para o UAL, C++ e Fluxograma do comando condicional composto. O exemplo compara uma nota – se a nota for maior ou igual a seis, é impresso no dispositivo padrão de saída a palavra Aprovado, caso contrário deverá ser escrito Reprovado.

LINGUAGEM UAL	EXEMPLO

se (<teste>)
{
  <bloco de comandos>
}
senão
{
  <bloco de comandos>
}
	se (nota >= 6)
	{
	  imprima "Aprovado";
	}
	senão
	{
	  imprima "Reprovado";
	}

LINGUAGEM C++	**EXEMPLO**

if (<teste>)
{
  <bloco de comandos>
}
else
{
  <bloco de comandos>
}
	if (nota >= 6)
	{
	  cout << "Aprovado";
	}
	else
	{
	  cout << "Reprovado";
	}



Veja na Figura 26 o exemplo de como podemos codificar o algoritmo anterior de verificação se um número é par ou ímpar utilizando o comando condicional composto.

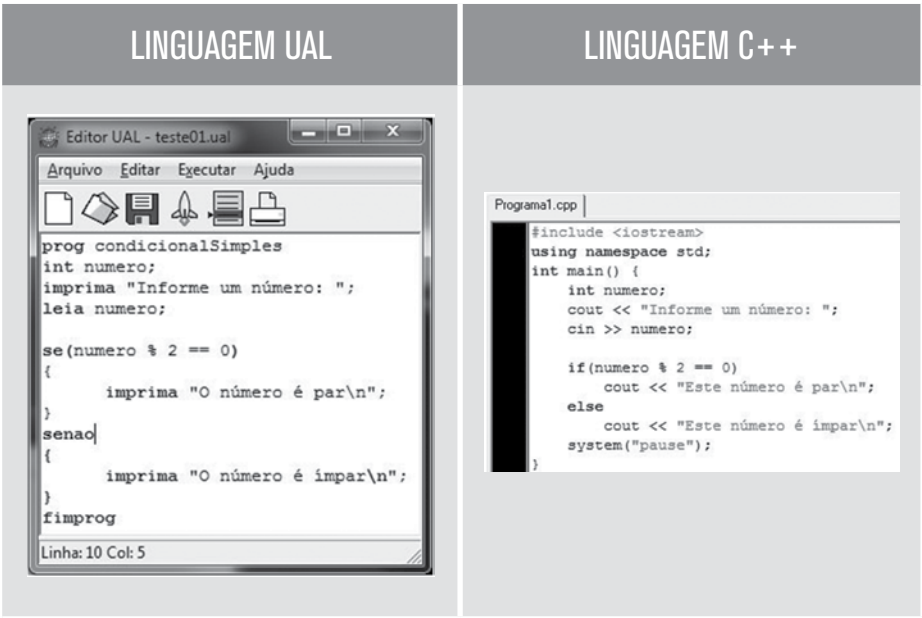


Figura 26 – Algoritmo com comando condicional composto que verifica se um número é par ou ímpar.

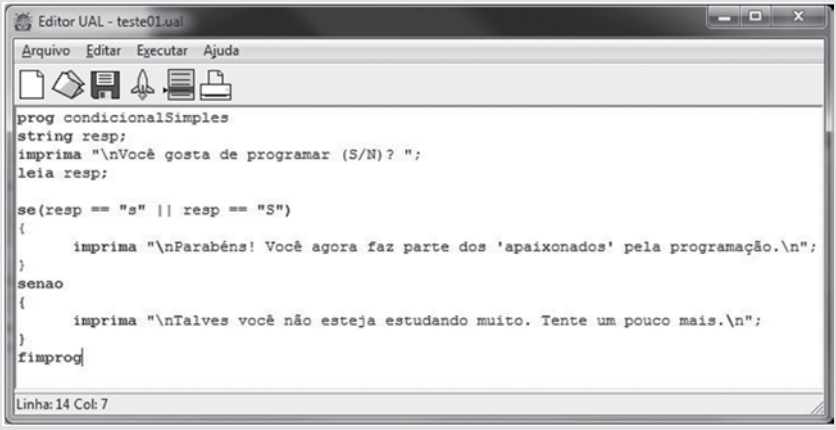
O resultado da execução do algoritmo, para os possíveis valores de entrada, é apresentado na Figura 27 e na Figura 28.

Figura 27 – Resultado da execução do algoritmo com entrada par

Figura 28 – Resultado da execução do algoritmo com entrada ímpar

Veja o exemplo codificado na Figura 29.

LINGUAGEM UAL



Programa1.cpp

```
#include <iostream>
using namespace std;
int main() {
    char resp;
    cout << "\nVocê gosta de programar (S/N)? ";
    cin >> resp;

    if(resp == 's' || resp == 'S')
        cout << "\nParabéns! Você agora faz parte dos 'apaixonados' pela programação.\n";
    else
        cout << "\nTalves você não esteja estudando muito. Tente um pouco mais.\n";
}
```

Figura 29 – Algoritmo com comando condicional composto utilizando composição de condições de teste.

Nesse exemplo, utilizamos uma entrada literal, e, neste caso, precisamos de alguns cuidados adicionais. Através da mensagem inicial, informamos ao usuário quais são opções de valores de entrada, delimitando assim as possibilidades, apesar de ainda haver possibilidade de variações de entrada. Mesmo tendo delimitado que a entrada deve ser S para sim e N para não, o usuário pode digitar os caracteres minúsculos ou maiúsculos, sendo assim precisamos tratar estes dois casos em nosso programa. Perceba que, para isso, utilizamos uma composição de condição dentro do teste do comando condicional. No caso específico, utilizamos o operador relacional OU, que verifica se um caso ou outro é verdadeiro – sendo um dos dois lados da operação verdadeiro, o resultado será verdadeiro. Em várias situações, será necessário utilizar uma composição de testes, tenha apenas o cuidado de avaliar corretamente o operador entre as composições e se necessário delimitá-las com parênteses. A Figura 30 e a Figura 31 apresentam como resultado da execução a entrada 's' e 'N', respectivamente.

Figura 30 – Resultado da execução do algoritmo para entradas

Figura 31 – Resultado da execução do algoritmo para entrada N

3.4 Comando condicional aninhado

A estrutura condicional aninhada é um tipo de estrutura em que uma condicional é definida no interior do bloco de instruções da outra. Em outras palavras, quando construímos estruturas condicionais que contém no interior do código outras estruturas condicionais, chamamos de *estrutura condicional aninhada*. O objetivo deste tipo de estrutura é avaliar todas as possibilidades de uma condição a partir dos operandos e operadores utilizados na expressão.

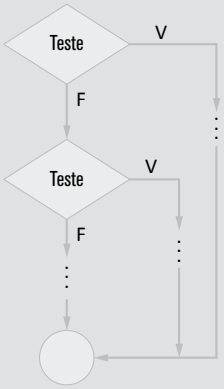
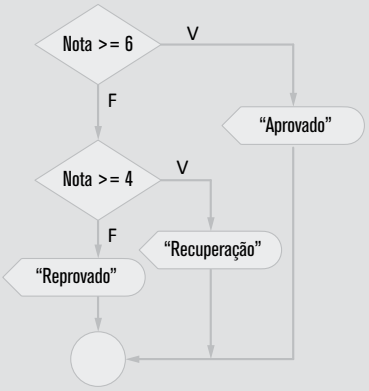
A seguir, são apresentados a estrutura e um exemplo de uso para o UAL, C++ e Fluxograma do comando condicional composto. O Exemplo compara uma nota – se a nota for maior ou igual a seis, é impresso no dispositivo padrão de saída a palavra Aprovado, caso contrário é verificado novamente a nota; se a nota for maior que quatro, deverá ser escrito Recuperação e, caso contrário, deverá ser escrito Reprovado.

LINGUAGEM UAL	EXEMPLO

se (<teste>)
{
  <bloco de comandos>
}
senão
{
  se (<teste>)
  {
  <bloco de comandos>
  }
  senão
  {
  <bloco de comandos>
  }
}
	se (nota >= 6)
	{
	  imprima “Aprovado“;
	}
	  se (nota >= 4)
	  {
	  imprima imprima “Recuperação“;
	  }
	  senão
	  {
	  imprima “Reprovado”;
	  }
	}

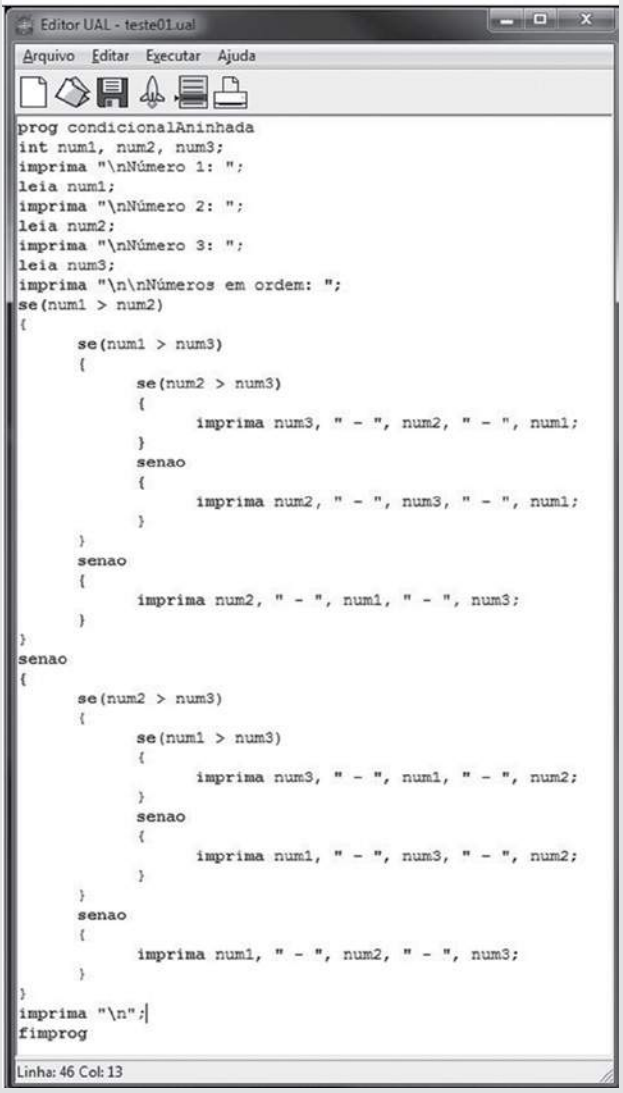
LINGUAGEM C++	EXEMPLO

if (<teste>)
{
  <bloco de comandos>
}
else
{
  if (<teste>)
  {
  <bloco de comandos>
  }
  else
  {
  <bloco de comandos>
  }
}
	if (nota >= 6)
	{
	  cout << “Aprovado“;
	}
	else
	{
	  if (nota >= 4)
	  {
	  cout << “Recuperação“;
	  }
	  else
	  {
	  cout << “Reprovado”;
	  }
	}

FLUXOGRAMA	EXEMPLO
	

Para demonstrar a codificação de uma estrutura condicional aninhada, vamos elaborar um algoritmo que tem como objetivo ordenar um conjunto de três elementos numéricos do tipo inteiro informados pelo usuário. Isto posto, o usuário especificará por meio do teclado três valores e o algoritmo será capaz de exibi-los em ordem crescente. Para a codificação do exemplo, foi utilizada uma estrutura condicional aninhada, em que é possível identificar a utilização de estruturas condicionais encadeadas. A Figura 32 demonstra sua codificação.

LINGUAGEM UAL



Programa1.cpp

```
#include <iostream>
using namespace std;
int main() {
    int num1, num2, num3;
    cout << "\nNúmero 1: ";
    cin >> num1;
    cout << "\nNúmero 2: ";
    cin >> num2;
    cout << "\nNúmero 3: ";
    cin >> num3;
    cout << "\n\nNúmeros em ordem: ";
    if(num1 > num2)
    {
        if(num1 > num3)
        {
            if(num2 > num3)
                cout << num3<<" - "<<num2<<" - "<<num1;
            else
                cout << num2<<" - "<<num3<<" - "<<num1;
        }
        else
        {
            cout << num2<<" - "<<num1<<" - "<<num3;
        }
    }
    else
    {
        if(num2 > num3)
        {
            if(num1 > num3)
                cout << num3<<" - "<<num1<<" - "<<num2;
            else
                cout << num1<<" - "<<num3<<" - "<<num2;
        }
        else
        {
            cout << num1<<" - "<<num2<<" - "<<num3;
        }
    }
    cout << "\n";
}
```

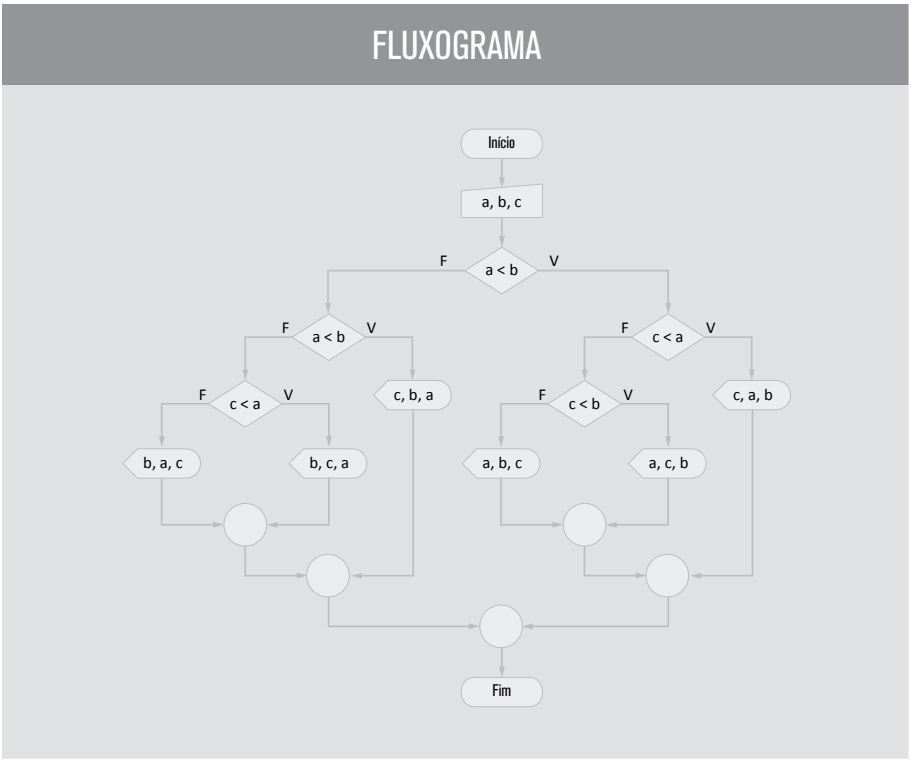


Figura 32 – Algoritmo com comando condicional aninhada que ordena 3 números.

Para facilitar a visualização e o entendimento dos comandos condicionais aninhados, adicionamos o fluxograma para visualização gráfica. Sempre que tiver dificuldades no entendimento de um código, utilize recursos gráficos, como o fluxograma ou o teste de mesa, para facilitar a compreensão. O resultado da execução do algoritmo pode ser visualizado na Figura 33.

Figura 33 – Resultado da execução do algoritmo

O próximo exemplo apresentará a codificação de um algoritmo que simula o processo de autenticação de um sistema. A autenticação de usuários é geralmente conhecida como *login* e necessita de um nome de usuário e senha para validação dos dados. A listagem de código da Figura 34 apresenta a codificação do algoritmo, em que é possível notar a condicional aninhada, no caso de o usuário informado ser "joão", em que será solicitada a senha e esta será verificada.

LINGUAGEM UAL

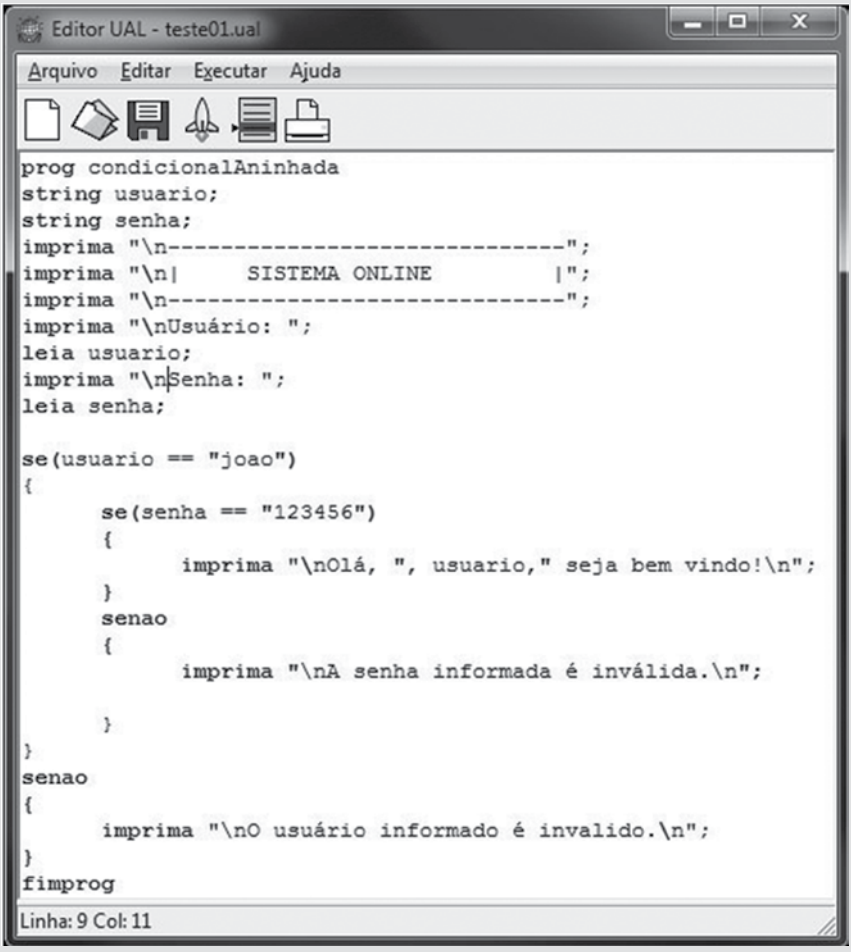


Figura 34 – Algoritmo com comando condicional aninhada que realiza login de um usuário.

Os possíveis resultados que podem ser encontrados com a execução da aplicação podem ser visualizados nas Figuras 35, 36 e 37.

Figura 35 – Resultado da execução da aplicação, para usuário e senha corretos.

Figura 36 – Resultado da execução da aplicação, para usuário correto e senha incorreta.

Figura 37 – Resultado da execução da aplicação, para usuário incorreto.

3.5 Comando condicional múltiplo

Um problema frequente que ocorre é quando temos uma única variável podendo assumir diversos valores diferentes, em que, para cada valor, teremos uma ação associada. Para resolver esse problema de imediato, pensaríamos em um conjunto de ses aninhados. A Figura 38 apresenta o código em UAL para um programa em que o usuário entra com um número de 1 a 12, que representa um mês do ano, e o programa exibe o nome do mês.

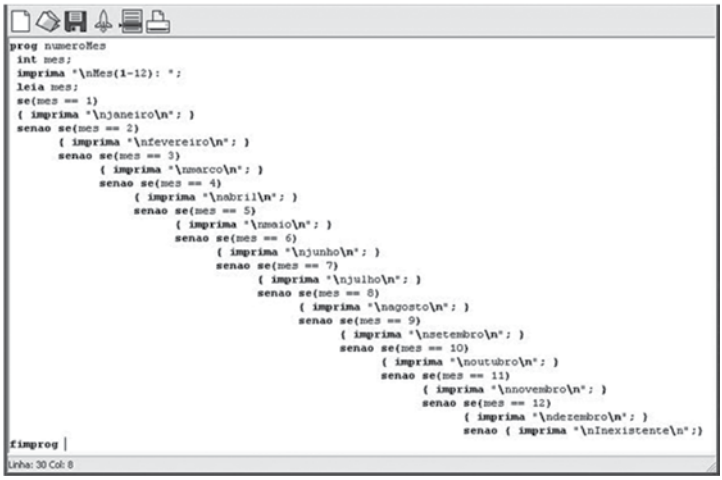


Figura 38 – Algoritmo UAL que imprime o nome do mês a partir no número.

É fácil perceber que no momento da escrita não é muito usual realizar o controle de quantos se ... senão se serão utilizados e principalmente dos abre e fecha chaves. O que pode levar facilmente a erros simples, porém difíceis de serem detectados.

A estrutura condicional múltipla é utilizada para estruturar de forma melhor e mais eficiente problemas como o apresentado anteriormente. Ela permite especificar no algoritmo que, dentre as condições possíveis, apenas uma condição poderá ser executada. Esse tipo de estrutura permite a elaboração de condicionais mutuamente exclusivas, ou seja, uma estrutura que seleciona e executa apenas uma condição por vez.

A codificação pode ser realizada utilizando a instrução *se ... senão se*, ou, ainda, uma estrutura específica denominada *escolha*. Para o uso da estrutura esco-

lha, precisamos estar atentos a exigências, como as listadas a seguir

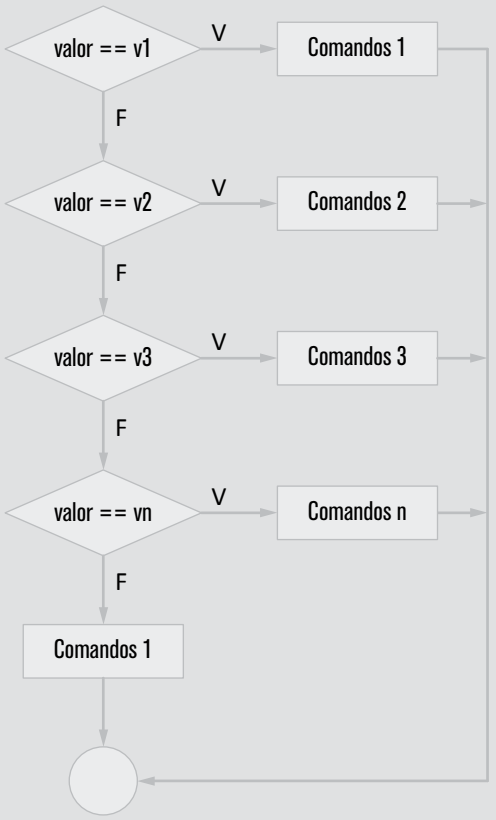
- A variável tem que ser a mesma em todos os testes.
- A variável tem que ser do tipo enumerável: inteira ou de um caractere.
- O operador relacional tem que ser o de igualdade.

A listagem código 8 demonstra a codificação de uma estrutura condicional múltipla utilizando a instrução **escolha**. Esse exemplo apresenta um algoritmo que realiza cálculos simples a partir de dois valores fornecidos pelo usuário e um símbolo da matemática que se refere à operação. O algoritmo foi nomeado **unid4_exemplo08.alg**.

Assim, faz-se necessário entender que, embora sejam mais simples, nem sempre poderemos substituir os ses encadeados e, às vezes, poderemos substituí-los em uma linguagem, mas em outra não. A seguir, é apresentado a estrutura do comando **escolha** em UAL, C++ e Fluxograma.

LINGUAGEM UAL (NÃO DISPONÍVEL NA VERSÃO)	LINGUAGEM C++
escolha (<variavel>) **{** **caso** <valor1>: <comandos> **pare**; **caso** <valor2>: <comandos> **pare**; **caso** <valor3>: <comandos> **pare**; **caso** <valor_n>: <comandos> **pare**; **senao** : <comandos> **}**	**switch** (<variavel>) **{** **case** <valor1>: <comandos> **break**; **case** <valor2>: <comandos> **break**; **case** <valor3>: <comandos> **break**; **case** <valor_n>: <comandos> **break**; **default** : <comandos> **}**

FLUXOGRAMA



No comando **escolha**, quando a variável é avaliada, seu valor é comparado com todos os valores informados de forma sequencial. Quando uma igualdade é encontrada, os comandos associados àquelas igualdades são executados. O comando **pare** (*break*) é utilizado para informar a estrutura de que as próximas comparações não devem ser executadas e que o controle deve sair da estrutura, pois sua omissão acarreta a execução das outras comparações. A opção **senão** (*default*) é opcional e será executada caso nenhuma das opções apresentadas anteriormente seja satisfeita.

Vejamos na Figura 39 o exemplo dos meses apresentados anteriormente utilizando a estrutura escolha.

LINGUAGEM C++

Programa1.cpp

```
using namespace std;
int main() {
    int mes;
    cout << "\nDigite um número de 1 a 12: ";
    cin >> mes;
    switch(mes)
    {
        case 1: cout << "\nJaneiro\n";
             break;
        case 2: cout << "\Fevereiro\n";
             break;
        case 3: cout << "\nMarco\n";
             break;
        case 4: cout << "\nAbril\n";
             break;
        case 5: cout << "\nMaio\n";
             break;
        case 6: cout << "\nJunho\n";
             break;
        case 7: cout << "\nJulho\n";
             break;
        case 8: cout << "\nAgosto\n";
             break;
        case 9: cout << "\nSetembro\n";
             break;
        case 10: cout << "\nOutubro\n";
             break;
        case 11: cout << "\nNovembro\n";
             break;
        case 12: cout << "\nDezembro\n";
             break;
        default: cout << "\nNumero invalido.\n";
    }
    system("pause");
}
```

Figura 39 – Algoritmo C++ utilizando estrutura escolha que imprime o nome do mês a partir no número.

O resultado da execução do algoritmo pode ser visualizado na Figura 40. A Figura 41 apresenta o resultado do algoritmo com um valor não tratado nas opções, o que culmina na execução da opção **senão**.

Figura 40 – Resultado da execução do algoritmo.

Figura 41 – Resultado da execução do algoritmo na opção senão.

Essa estrutura é ideal para algoritmos que funcionam através de um menu visto em que só podemos selecionar um item. Vejamos na Figura 42 a calculadora que fizemos na unidade anterior, porém com a opção de o usuário escolher a operação que deseja executar.

LINGUAGEM C++

Programa1.cpp

```
#include <iostream>
using namespace std;
int main() {
    float valor1, valor2;
    char operacao;
    cout << "\n--------CALCULADORA SIMPLES--------";
    cout << "\nDigite a operacao (+, -, *, /): ";
    cin >> operacao;
    cout << "\nDigite valor 1: ";
    cin >> valor1;
    cout << "\nDigite valor 2: ";
    cin >> valor2;

    switch(operacao)
    {
        case '+': cout << "\nSoma: " << valor1 + valor2;
             break;
        case '-': cout << "\nSubtracao: " << valor1 - valor2;
             break;
        case '*': cout << "\nMultiplicacao: " << valor1 * valor2;
             break;
        case '/': cout << "\nDivisao: " << valor1 / valor2;
             break;
        default: cout << "\nNumero invalido.\n";
    }
    cout << "\n";
    system("pause");
}
```

Figura 42 – Algoritmo C++ utilizando estrutura escolha como menu do programa de calculadora.

O resultado da execução do algoritmo pode ser visualizado na Figura 43.

Figura 43 – Resultado da execução do algoritmo.

ATIVIDADE

1. Escreva um algoritmo computacional que receba quatro valores do tipo inteiro e determine o menor elemento.

2. Entrar com 3 números para as variáveis v1, v2, v3. Trocar os conteúdos das variáveis de tal maneira que a variável v1 fique com o maior número, a variável v2 fique com o número do meio e a v3 com o menor número. Exibi-los de forma decrescente.

3. Ler três números correspondentes a lados de um triângulo e imprimir a classificação segundo seus lados.

4. No Brasil, o licenciamento de carros e caminhões é feito durante o ano de acordo com o final da placa. Dependendo do valor, o licenciamento deve ser feito até um determinado dia. De acordo com a tabela a seguir, solicite ao usuário que informe o tipo de automóvel e o número final de sua placa para você poder informá-lo da data final de seu licenciamento.

MÊS DO LICENCIAMENTO	AUTOMÓVEIS	CAMINHÕES
abril	1	-
maio	2	-
junho	3	-
julho	4	-
agosto	5 e 6	-

MÊS DO LICENCIAMENTO	AUTOMÓVEIS	CAMINHÕES
setembro	7	1 e 2
outubro	8	3, 4 e 5
novembro	9	6, 7 e 8
dezembro	0	9 e 0

REFLEXÃO

Com a inclusão da estrutura condicional, nossos códigos ficaram mais complexos, uma vez que os exercícios requerem um maior número de possibilidades de respostas. As estruturas estudadas até aqui compõem as estruturas básicas dos algoritmos e programação e nos permitem resolver um universo muito extenso de problemas.

Os comandos condicionais possuem diversas variações em suas aplicações, principalmente no que diz respeito aos condicionais aninhados, que podem ser organizados de inúmeras formas para resolver os mais complexos problemas. Sua estrutura apesar de simples pode ser de difícil estruturação. Por isso, pratique o máximo que puder para melhorar sua compreensão e entendimento.

LEITURA

Os programas e as estruturas em C++ estão ficando cada vez mais complexos, por isso já é hora de aprender um pouco mais sobre o ambiente DEV C++. Acesse o link do ICMC, da USP de São Carlos. <http://wiki.icmc.usp.br/images/e/e3/Manual_Dev_C.pdf>

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E.e A. V. Fundamentos da programação de computadores: Algoritmos, Pascal, C/C++ e Java. 2. ed. São Paulo: Pearson Education, 2008.

ASCENCIO, A. F. G.; EDILENE, A. V. de. Fundamentos da programação de computadores: Algoritmos, Pascal e C/C++. São Paulo: Prentice Hall, 2002.

FORBELLONE, A.L. V; EBERSPACHER, H. Lógica de programação. 3. ed. São Paulo: Makron Books, 2005.

PUGA, S.; RISSETTI, G. Lógica de programação e estrutura de dados: com aplicações em Java. 1. ed. São Paulo: Pearson Education, 2003.

SPALLANZANI, Adriana Sayuri; MEDEIROS, Andréa Teixeira de; FILHO, Juarez Muylaert. Linguagem UAL. Disponível em: <http://geocities.ws/ual_language/ual.html>. Acesso em: 25 abr. 2014.

NO PRÓXIMO CAPÍTULO

Veremos a seguir as estruturas que nos permitem economizar esforço, evitando repetição de código para execuções cíclicas, o que facilitará muito o desenvolvimento de algoritmos e programas mais longos, em razão da utilização de uma quantidade menor de códigos.

4

Estruturas de repetição

4 Estruturas de repetição

Neste capítulo, estudaremos as estruturas de repetição utilizadas nos algoritmos. Estudaremos três estruturas de repetição diferentes: para, enquanto e faça ... enquanto. Elas são muito importantes para facilitar a vida do programador e permitir a criação de programas maiores e mais complexos. Seu entendimento é necessário para que um grande volume de processamento sem a necessidade de grandes trechos de código seja possível, diminuindo muito o trabalho de codificação.

OBJETIVOS

- Entender a estrutura e o funcionamento das estruturas de repetição.
- Construir algoritmos usando a estrutura de repetição para.
- Conhecer as três estruturas de repetição do C++.
- Construir algoritmos usando a estrutura de repetição enquanto.
- Construir algoritmos usando a estrutura de repetição faça ... enquanto.
- Entender a diferença e as aplicações das estruturas de repetição.

REFLEXÃO

Estudamos no Capítulo 2 os operadores lógicos e de comparação e os utilizamos amplamente na Unidade 3 para realizar os controles condicionais de desvio. É muito importante que o uso e a aplicação destes operadores estejam bem entendidos por você, pois nesta unidade serão novamente utilizados como controle para as estruturas de repetição. Caso ainda tenha dúvidas ou dificuldade em sua aplicação, retorne e reforce os estudos para que eles não dificultem seu entendimento das estruturas de repetição.

4.1 Características da estrutura de repetição

No desenvolvimento de algoritmos computacionais, muitas vezes necessitamos que determinadas partes do código sejam executadas diversas vezes. Para que essas sucessivas execuções sejam possíveis, podemos utilizar um conjunto de estruturas denominadas estruturas de repetição.

Provavelmente você utiliza um despertador para acordar de manhã. Já pensou como faria um programa para criar um despertador? Pensando de forma simplificada, basta criarmos um programa que toca um som, em determinado horário, todos os dias, correto? Mas como faríamos isso? Quantos dias? Repetiríamos os comandos para tocar o som tantas vezes quantos forem os dias que quisermos que o despertador toque? Mas, nesse caso, isso deveria ser definido no momento de desenvolvimento do programa, e não poderia ser alterado pelo usuário. Criaríamos esse despertador para tocar apenas uma vez e o usuário precisaria iniciá-lo novamente todos os dias? É bem provavelmente que ao ler perguntas passou pela sua cabeça, enquanto o usuário deixar ativo. E esta é a resposta correta. Porém, com o que vimos até agora, não conseguimos fazer isso. Então, vamos aprender as estruturas de repetição.

Uma delas é o enquanto, que deve ter passado pela sua cabeça. Aprenderemos uma estrutura que nos permite dizer ao algoritmo ou programa "enquanto estiver ativo faça isso, ou "faça isso enquanto for diferente daquilo" ou ainda "para x de 10 até 100 realize tal tarefa". Perceba que todas as frases tratam de ações que se repetirão e nos permitem que, ao invés de repetir um mesmo trecho de código diversas vezes, nós o escrevamos uma única vez e a estrutura utilizada se encarrega de repetir.

Para exemplificar, foram apresentadas três frases, cada uma delas apresenta um tipo diferente de repetição: a estrutura pré-testada, a pós-testada e a com variável de controle. Na pré-testada, verificamos determinada condição e depois executamos a tarefa; na pós-testada, primeiro executamos determinada tarefa e depois verificamos se a condição é verdadeira para realizar as próximas iterações. Na opção com variável de controle, a execução das tarefas está associada diretamente ao valor de uma determinada variável de controle.

CONCEITO

Estruturas de repetição são também conhecidas como estruturas de iteração ou laços, as quais permitem que uma sequência de comandos seja executada repetidamente, até que determinada condição ou situação seja atendida.

Estudaremos detalhadamente as diferentes opções e descobriremos as vantagens e desvantagens de cada uma delas, para determinados tipos de aplicação. Assim você será capaz de selecionar a melhor estrutura para cada apli-

cação. É conhecido que alguns programadores gostam mais de determinados tipos de estruturas do que de outros, e são fiéis a esta, porém a utilização de uma determinada estrutura de forma incorreta pode levar a falhas no programa, ou aumento de complexidade de código deixando-o mais suscetível a erros.

4.2 Comando de repetição com variável de controle - PARA

A estrutura de repetição para é utilizada na construção de laços de repetição que são controlados por uma variável de controle ou variável contadora. Nesse tipo de estrutura, a variável contadora é necessária para determinar quantas vezes o bloco de repetição já foi repetido; além disso, essa variável particular é comparada, a cada iteração do laço, com uma condição de parada que define o término da execução do laço. A estrutura de repetição para possui ainda uma condição inicial que representa o valor da variável contadora e um critério que determina o que acontecerá com o contador a cada iteração do laço. Em geral, a cada iteração do laço, a variável contadora é incrementada ou decrementada, e essa operação ocorre até que a condição de parada da estrutura seja alcançada.

Geralmente começamos o estudo pela instrução para porque possui um número de repetições bem definido e conhecido, o que facilita o entendimento. Normalmente pensamos em algo do tipo para x de 1 até 10 faça isso. Você poderia então se perguntar: se conhecemos a quantidade previamente, por que não repetimos o código este tanto de vezes, ao invés de utilizar uma estrutura? Em um problema onde queremos apenas imprimir um valor conjunto de 10 valor, isso pode fazer sentido, repetimos o comando de imprimir 10 vezes e ponto, mas, e se quisermos imprimir 100 vezes, ou então 1000 vezes, será que faz sentido? E o que queremos fazer não é apenas imprimir, mas sim realizar um cálculo complexo que exige diversas etapas; então devemos repetir este cálculo complexo 10, 100, 1000 vezes? Será que isso faz sentido? Concordamos que não. Não é difícil imaginar a possibilidade de erro que surge quando temos de repetir esses conteúdos complexos ou quando temos de corrigir algo para alterar o mesmo ponto, 10, 100 ou 1000 vezes. Torna-se simplesmente inviável. Além do mais, as estruturas de repetição são muito fáceis, basta aprender sua estrutura. Vamos lá!

LINGUAGEM UAL

para(<valor inicial>; <expressão de teste>;<incremento>**){**

} bloco de comandos

<valor inicial>

nomeDaVariável <- valorInicial;

– deve ser uma variável enumerável, um inteiro (int)

– é atribuído um valor inicial, pode ser uma constante, uma variável ou uma expressão.

– Exemplos: i<-0; i<- strtam(nome); i <- x+1;

<expressão de teste>

nomeDaVariável <, >, <=, >=, == valor;

– Estrutura relacional ou condição que será utilizada como termo final do laço

– O valor poder ser constante, variável ou expressão

– Exemplos: i >= 10; i>strtam(nome); i <= x+10;

<incremento>

nomeDaVariável <- nomeDaVariável **operador** valor

– é um comando de atribuição que incrementa a variável no laço

– o operador pode ser qualquer operador aritmético ou expressão

– Exemplos: i <- i+1; i<-i+2; i++; i--;

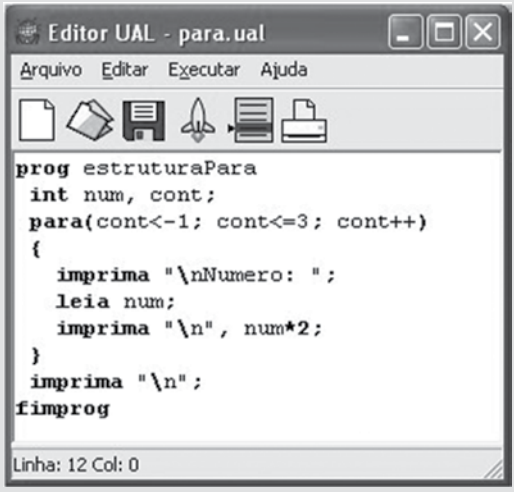


Figura 44 – Exemplo estrutura para em UAL

for(<valor inicial>; <expressão de teste>;<incremento>**){**

bloco de comandos

}

<valor inicial>

nomeDaVariável = valorInicial;

– deve ser uma variável enumerável, um inteiro (int)

– é atribuído um valor inicial, pode ser uma constante, uma variável ou uma expressão.

– Exemplos: i=0; i=strtam(nome); i=x+1;

<expressão de teste>

nomeDaVariável <, >, <=, >=, == valor;

– Estrutura relacional ou condição que será utilizada como termo final do laço

– O valor poder ser constante, variável ou expressão

– Exemplos: i >= 10; i>strtam(nome); i <= x+10;

<incremento>

nomeDaVariável <- nomeDaVariável **operador** valor

– é um comando de atribuição que incrementa a variável no laço

– o operador pode ser qualquer operador aritmético ou expressão

– Exemplos: i =i+1; i=i+2; i++; i--;

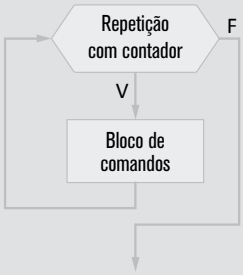
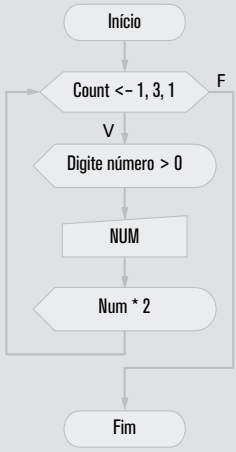
O comando < ; ; > pode ser utilizado sem parâmetro algum para criar um loop ou for infinito, porém os ; são obrigatórios

As chaves do bloco de comandos são opcionais no caso de haver um único comando.

para.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int num, cont;
6    for(cont=1; cont<=3; cont++)
7    {
8     cout<<"\nNumero: ";
9     cin>>num;
10    cout<< "\n"<<num*2;
11   }
12  cout<<"\n";
13  system("pause");
14 }
```

Figura 45 – Exemplo estrutura para em C++

FLUXOGRAMA	EXEMPLO
	

Segue abaixo alguns exemplos de uso da estrutura para:

	LINGUAGEM UAL	LINGUAGEM C++
Repetir 100 vezes	para(c<-1; c<=100; c++)	for(c=1; c<=100; c++)
Repetir 30 vezes	para(c<-1; c<=30; c++)	for(c=1; c<=30; c++)
Contar de 10 até 1 decrementando de 1	para(c<-10; c>=1; c--)	for(c=10; c>=1; c--)
Contar de 10 até 1 decrementando de 2	para(c<-10; c>=1; c<-c-2)	for(c=10; c>=1; c=c-2) for(c=10; c>=1; c-=2)
Contar de 0 até 100 de 10 em 10	para(c<-0; c<=100; c<-c+10)	for(c=0;c<=100; c=c+5) for(c=0; c<=100; c+=5)

A listagem código 9 demonstra a utilização da estrutura de repetição para. Nesse exemplo, a estrutura de repetição é empregada na leitura de um conjunto de 10 valores numéricos do tipo real que são informados pelo usuário. Além disso, uma variável especial denominada acumulador realiza a soma dos valores que têm entrada pelo teclado e, ao final, apresenta a soma total destes valores. Um detalhe importante a respeito de acumuladores está relacionado à necessidade de inicialização da variável acumuladora. Para evitar inconsistências na soma dos valores, é fundamental que a variável acumuladores seja iniciada com o valor zero.

LINGUAGEM UAL	LINGUAGEM C++

prog exemplo
int cont;
real valor, acumulador;
acumulador <- 0.0;
para(cont<-1; cont<=10; cont++)
{
imprima "\nValor ", cont, ": ";
leia valor;
acumulador <- acumulador + valor;
}
imprima "\nSoma dos valores: ", acumulador;
fimprog
	#include <iostream>
	using namespace std;
	int main()
	{
	int cont;
	float valor, acumulador;
	acumulador = 0;
	for(cont=1; cont<=10; cont++)
	{
	cout << "\nNumero: " << cont;
	cin >> valor;
	acumulador = acumulador + valor;
	}
	cout <<"\nSoma dos valores: " << acumulador;
	}

O resultado da execução do algoritmo que demonstra a utilização da estrutura de repetição para é apresentado na Figura 46.

Figura 46 – Resultado da execução do algoritmo

Na estrutura de repetição para, os passos iterativos do laço também podem ocorrer de maneira decremental. Nesse próximo exemplo, é apresentada a codificação de um algoritmo que exibe na tela os números pares no intervalo de 10 até 1. Note que a variável contadora é decrementada até atingir a condição de parada.

LINGUAGEM UAL	LINGUAGEM C++

prog exemplo
int cont;
para(cont<-10; cont>1; cont--)
{
se (cont % 2 == 0)
{
imprima "\n", cont;
}
}
fimprog
	#include <iostream>
	using namespace std;
	int main()
	{
	int cont;
	for(cont=10; cont>1; cont--)
	{
	if((cont%2) == 0)
	cout<<"\n" <<cont;
	}
	}

A Figura 47 apresenta o resultado da execução do algoritmo em que é possível visualizar os números pares no intervalo de 10 até 1.

Figura 47 – Resultado da execução do algoritmo

4.3 Comando de repetição com teste lógico no início - ENQUANTO

Em programação, é comum sabermos quantas vezes vamos precisar repetir determinado comando ou trecho de código. Por exemplo, quando vamos registrar uma compra, não sabemos quantos produtos existem nesta compra, mas sabemos que teremos de repetir o registro dos produtos tantas vezes quantas necessárias, ou seja, até que todos os produtos sejam registrados. Em situações como essa, precisamos de uma estrutura de controle que nos permita verificar uma condição ou realizar um teste lógico como controle de seu laço de repetição. Para isso, utilizamos os comandos de repetição com teste lógico. Existem dois tipos de comandos com teste lógico, com teste no início, ou seja, antes da execução do trecho de código, ou com teste ao final, depois da execução do trecho de código. Nesse tópico, vamos aprender a estrutura enquanto, que possui teste lógico no início.

Na estrutura de repetição enquanto, um determinado bloco de instruções é executado sucessivamente enquanto o resultado da expressão condicional da condição de parada permanecer verdadeiro. Assim, enquanto a condição de parada resultar em verdadeiro, o bloco será executado. Nesse tipo de estrutura, o bloco de instruções pode ser executado um número predeterminado de vezes (como na estrutura de repetição para) ou executar um número indeterminado de vezes até que uma condição seja satisfeita. Nas linguagens de programação, essa estrutura de repetição é conhecida como while. Veja sua estrutura.

LINGUAGEM UAL

enquanto(<condição>)
{
bloco de comandos
}

<condição>
variável **operador** valor;
– Estrutura relacional ou condição que será utilizada como validação para a execução do bloco de comandos.
– Pode ser uma única expressão simples ou um conjunto de expressões relacionais e lógicas, como as utilizadas nos comandos condicionais, vistas anteriormente.
– Exemplos: i >= 10; i>strtam(nome); i <= x+10;

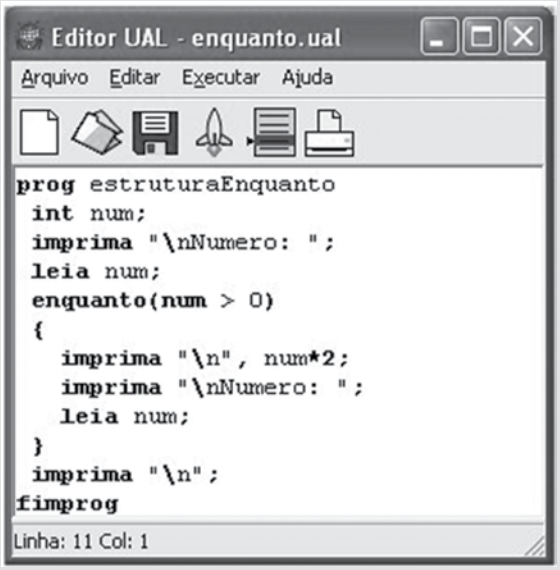


Figura 48 – Exemplo estrutura enaquanto em UAL

LINGUAGEM C++

while (<condição>)

{

bloco de comandos

}

<condição>

variável **operador** valor;

– Estrutura relacional ou condição que será utilizada como validação para execução do bloco de comandos.

– Pode ser uma única expressão simples ou um conjunto de expressões relacionais e lógicas, como as utilizadas nos comandos condicionais, vistas anteriormente.

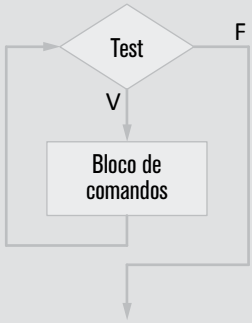
– Exemplos: i >= 10; i>strtam(nome); i <= x+10;

enquanto.cpp

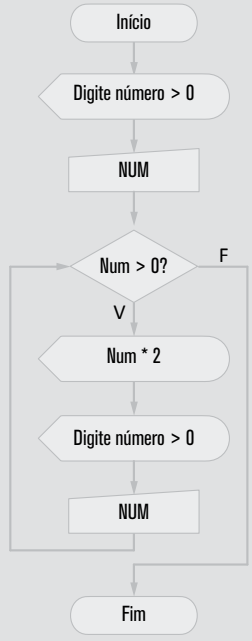
```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int num;
6    cout<<"\nNumero: ";
7    cin>>num;
8    while(num > 0)
9    {
10    cout<<"\n"<<num*2;
11    cout<<"\nNumero: ";
12    cin>>num;
13   }
14   cout<<"\n";
15   system("pause");
16 }
```

Figura 49 – Exemplo estrutura enaquanto em C++

FLUXOGRAMA



EXEMPLO



A listagem código a seguir apresenta na codificação a estrutura de repetição enquanto para um laço de repetição controlado por uma variável contadora. Esse tipo de codificação representa a adaptação da estrutura de repetição para na sintaxe da estrutura enquanto. Nesse exemplo, o algoritmo computacional tem como objetivo apresentar os valores pares no intervalo de 1 até 10, com isso você poderá comparar com a codificação da listagem 10 que realiza uma tarefa semelhante.

LINGUAGEM UAL	LINGUAGEM C++

prog exemplo
int cont;
cont <- 1;
enquanto(cont<=10)
{
se (contador % 2 = 0)
{
imprima "\n", contador;
}
cont <- cont + 1;
}
fimprog
	#include <iostream>
	using namespace std;
	int main()
	{
	int cont = 1;
	while(cont <= 10)
	{
	if((cont%2) == 0)
	cout<<"\n" <<cont;
	cont++;
	}
	}

O resultado da execução do algoritmo pode ser visualizado na Figura 50.

Figura 50 – Resultado da execução do algoritmo

Como apresentado anteriormente, a aplicação mais comum e indicada da estrutura enquanto se dá em problemas computacionais em que não se sabe previamente o número de vezes que o laço deverá ser repetido. Nesses casos,

uma expressão condicional determina se o laço será ou não executado novamente. A listagem código 11 demonstra a codificação de uma estrutura de repetição enquanto com um laço controlado por uma expressão condicional. O laço será executado enquanto o valor informado pelo usuário for diferente de zero. Além disso, o algoritmo realiza a contagem do total de números positivos e do total de números negativos que foram informados.

LINGUAGEM UAL	LINGUAGEM C++

prog exemplo
int numero, pos, neg;
numero <- 1;
pos <- 0;
neg <- 0;
enquanto(numero <> 0)
{
imprima "\n Escreva o número: ";
leia numero;
se (numero > 0)
{
pos<- pos +1;
} senão
{
se (numero < 0)
{
neg<- neg +1;
}
}
}
imprima "\nTotal de números positivo: ", pos;
imprima "\nTotal de números negativos", neg;
fimprog
	#include <iostream>
	using namespace std;
	int main()
	{
	int numero = 1;
	int pos, nerg;
	pos = neg = 0;
	while(cont != 0)
	{
	cout<<"\nEscreva o número";
	cin>>numero;
	if((numero > 0)
	pos++;
	else
	{
	if((numero < 0)
	neg++;
	}
	}
	cout<<"\nTotal de números positos: "<<pos;
	cout<<"\nTotal de números negativos: "<<neg;
	}

O resultado da execução do algoritmo pode ser visualizado na Figura 51.

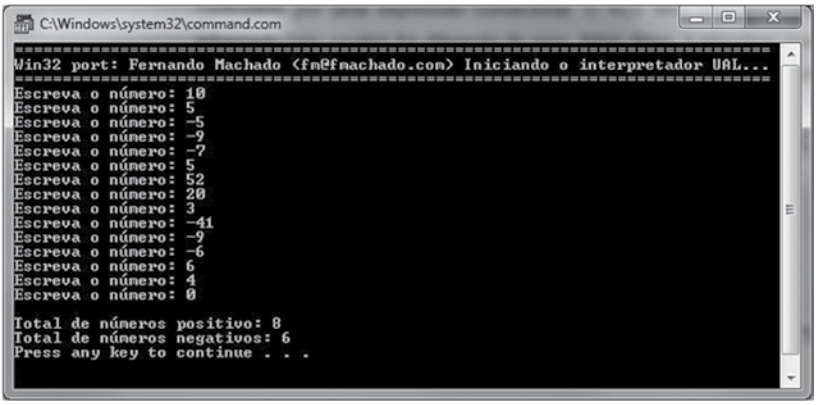


Figura 51 – Resultado da execução do algoritmo

É importante percebermos que no uso do enquanto é necessário que haja uma atribuição ou entrada do valor que será utilizado na verificação condicional antes do início do laço e uma atribuição ou leitura de dados ao final do laço, com o objetivo de atualizar o valor da condição de verificação. Se o primeiro valor passar pelo teste, então a execução do algoritmo entra no bloco de repetição e fica nesse bloco até que um valor não satisfaça mais a condição. Caso o valor de verificação não seja atualizado dentro do bloco de repetição, o programa irá entrar em "*lopping* infinito" e nunca saíra do laço.

Vimos anteriormente que a estrutura do enquanto pode simular a estrutura do para, mas será que o inverso é verdadeiro? As linguagens de programação mais antigas possuem uma estrutura limitada para o para (ou *for*), então isso não era possível, mas a linguagem C apresentou uma estrutura com mais recursos e, dessa forma, todas as linguagens posteriores ao C adotaram este modelo, tornando isso possível.

Veja o exemplo a seguir.

ENQUANTO	PARA

using namespace std;
int main()
{
int num;
cout<<"\nNumero: ";
cin>>num;
while(num <= 0)
{
cout<<"\n"<<num*2;
cout<<"\nNumero: ";
cin>>num;
}
cout<<"\n;
system("pause");
}
	using namespace std;
	int main()
	{
	int num;
	for(; ;);
	{
	cout<<"\nNumero: ";
	cin>>num;
	if(num <= 0) break;
	cout<<"\n"<<num*2;
	}
	cout<<"\n;
	system("pause");
	}

Neste exemplo, a estrutura *for* foi utilizada sem nenhum parâmetro, combinada com um *if* e *break*, simulando o funcionamento do *while*.

4.4 Comando de repetição com teste lógico no fim - FAÇA... ENQUANTO

A estrutura de repetição *faca...enquanto*, que nas linguagens de programação é conhecida como do...*while*, tem um funcionamento semelhante à estrutura enquanto. A principal diferença entre as duas estruturas está no fato de a estrutura enquanto realizar o teste condicional para determinar o término da repetição no início da instrução; por outro lado, a estrutura *faca...enquanto* realiza o teste de parada no final. O *faca...enquanto* primeiro executa o bloco de código, depois realiza o teste lógico de controle do laço. Veja sua estrutura.

LINGUAGEM UAL

faça
{
bloco de comandos
}
enquanto(<condição>)

<condição>
variável **operador** valor;
– Estrutura relacional ou condição que será utilizada como validação para execução do bloco de comandos novamente.
– Pode ser uma única expressão simples ou um conjunto de expressões relacionais e lógicas, como as utilizadas nos comandos condicionais, vistas anteriormente.
– Exemplos: i >= 10; i>strtam(nome); i <= x+10;

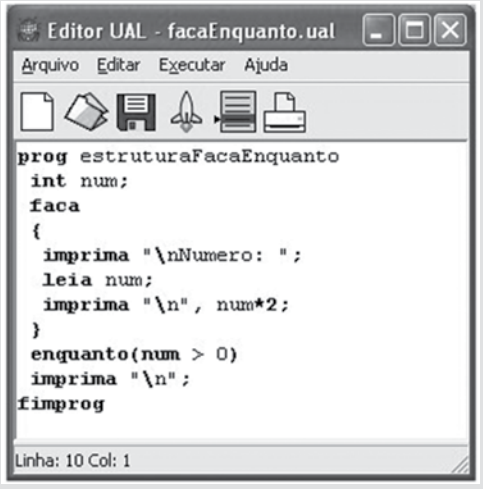


Figura 52 – Exemplo estrutura faça enquanto em UAL

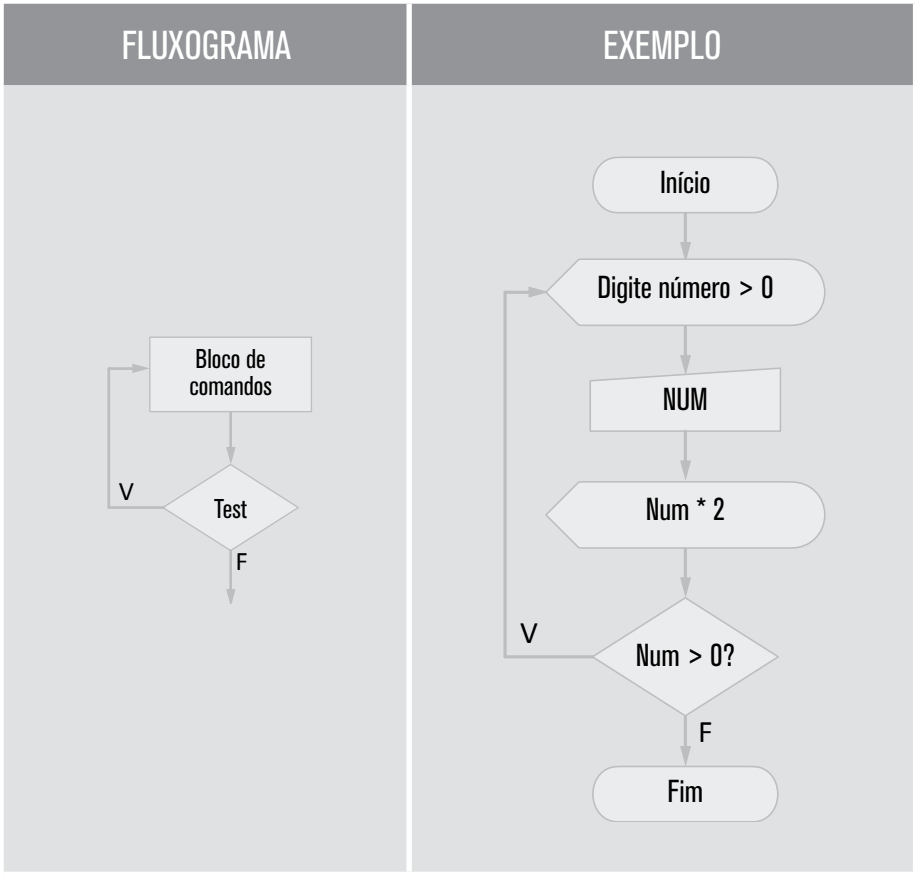
do
{
bloco de comandos
}
while(<condição> **);**

<condição>
variável **operador** valor;
– Estrutura relacional ou condição que será utilizada como validação para execução do bloco de comandos novamente.
– Pode ser uma única expressão simples ou um conjunto de expressões relacionais e lógicas, como as utilizadas nos comandos condicionais, vistas anteriormente.
– Exemplos: i >= 10; i>strtam(nome); i <= x+10;
Atente-se para o ; após o comando while, ele é obrigatório.

FacaEnquanto.cpp

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5    int num;
6    do
7    {
8     cout<<"\nNumero: ";
9     cin>>num;
10    cout<<"\n"<<num*2;
11   }
12   while(num > 0);
13   cout<<"\n";
14   system("pause");
```

Figura 53 – Exemplo estrutura faça enquanto em C++



Com a estrutura de repetição faca...enquanto, é possível construir laços de repetição tanto controlados por contador, quanto controlados por condição lógica. A listagem denominada código 12 apresenta a utilização da estrutura de repetição faca...enquanto para a codificação de um problema que envolve um laço controlado por condição lógica. Nesse exemplo, a estrutura de repetição é executada sempre que o que usuário informar a opção 'N' (Não).

LINGUAGEM UAL	LINGUAGEM C++

```
prog exemplo
int total;
real nota, soma, maior, menor, media;
string resposta;
soma <- 0.0;
total <- 0;
faca
{
total <- total +1;
imprima "\nNota ", total, ": ";
leia nota;
soma <- soma + nota;
se (total == 1)
{
maior <- nota;
menor <- nota;
} senao
{
se (nota > maior)
{
maior <- nota;
}
se (nota < menor)
{
menor <- nota;
}
}
imprima "Deseja continuar [s] ou [n]? ";
leia resposta;
} enquanto(resposta=="s" || resposta=="S")
media <- soma / total;
escreva "\nMédia das notas: ", media;
escreva "\nMaior nota.....: ", maior;
escreva "\nMenor nota.....: ", menor;
fimprog
```

```
#include <iostream>
using namespace std;
int main()
{
int total;
float nota, soma, maior, menor, media;
char resposta;
soma = 0;
total = 0;
do{
total++;
cout<<"\nNota "<<total<<": ";
cin>>nota;
if( (total == 1)
{
maior = nota;
menor = nota;
} else
{
if (nota > maior)
maior = nota;
if (nota < menor)
menor = nota;
}
cout"\nDeseja continuar [s] ou [n]? ";
cin>>resposta;
} while(resposta=='n' || resposta=='N');
media = soma / total;
cout << "\nMédia das notas: " << media;
cout << "\nMaior nota.....: " << maior;
cout << "\nMenor nota.....: << menor;
}
```

O programa acima lê notas até que o usuário informe que não deseja continuar respondendo “n“ ou “N“ à pergunta e posteriormente imprime a média das notas, a maior e a menor nota. A Figura 10 apresenta o resultado da execução do algoritmo. Note que a execução do bloco ocorreu algumas vezes até que o usuário optou por finalizar o algoritmo.

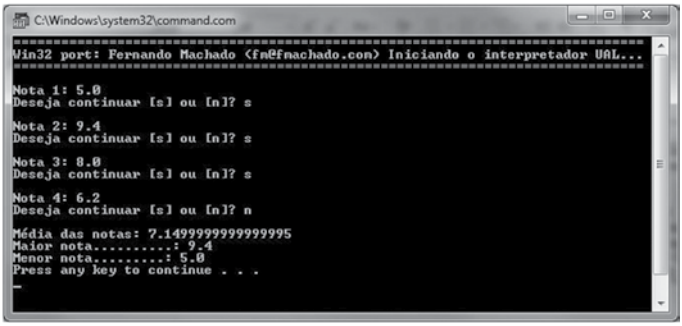


Figura 54 – Resultado da execução do algoritmo

Uma aplicação muito comum do faca...enquanto é para construção de menus e programas interativos, em que um bloco de código será executado repetidamente enquanto o usuário não escolher a opção de saída, como no exemplo anterior.

CONEXÃO

Você pode utilizar assugestão de endereço apresentado abaixo para complementar seus estudos a respeito de estruturas de repetição:
<http://www.ufpa.br/sampaio/curso_de_icc/icc/aula%2011/repita_ate.htm>

4.5 Quando utilizar cada estrutura de repetição

Acredito que neste momento você deva estar se perguntando: quando devo usar cada uma das estruturas? É normal que haja preferência por uma ou outra estrutura de repetição, porém existem algumas indicações ou melhores aplicações, conforme segue.

4.5.1 Indicação do para

Estrutura ideal quando o número de repetições for conhecido durante a elaboração do algoritmo ou quando o usuário puder fornecê-lo durante a execução.

Na linguagem C++, essa estrutura recebe o nome de for e, diferentemente de outras linguagens, simula com facilidade as estruturas do enquanto e do faca... enquanto como veremos mais adiante.

4.5.2 Indicação do enquanto

Estrutura que testa no início e é usada quando o número de repetições for desconhecido.

Simula com facilidade a estrutura do *faca...enquanto* e a estrutura do para (desde que criemos uma variável que terá seu valor incrementado/decrementado dentro da estrutura de repetição).

Na linguagem C++, essa estrutura recebe o nome de while.

Sua lógica é: repete enquanto a condição for verdadeira. Como ela testa antes de executar o bloco, pode ser que nem execute o bloco se a condição de início for falsa.

4.5.3 Indicação do faça...enquanto

A estrutura é indicada para as situações em que o número de repetições é desconhecido. Sua diferença em relação à estrutura do enquanto é o teste ao final, após a execução do bloco de código, que executa o bloco de comandos pelo menos uma vez. Essa estrutura também precisa de um teste para interromper a repetição.

A estrutura da repetição é mais parecida com a estrutura do para, não precisando de leitura/atribuição antes do faca para entrar na estrutura, pois testa, como já foi dito, ao final. É muito usada em algoritmos com menus.

Na linguagem C++, essa estrutura recebe o nome de *do...while*.

A estrutura da repetição é mais parecida com a estrutura do para, não precisando de leitura/atribuição antes do faca para entrar na estrutura, pois seu teste é realizado após a execução do bloco de código.

Com isso, finalizamos o estudo sobre as estruturas de controle, tanto para a elaboração de condicionais quanto para a especificação de repetições.

ATIVIDADE

1. Construa um algoritmo que escreva 100 vezes a frase: Algoritmo só se aprende praticando.

2. Construa um algoritmo que permita entrar com vários números enquanto forem diferentes de –999999999. Exibir, ao final, a quantidade de números digitados, a média de todos os números e o percentual de números negativos.

3. Escreva um algoritmo que, utilizando a estrutura de repetição repita.. até seja capaz de determinar quantos valores pares foram digitados pelo usuário em um conjunto de 20 elementos.

4. Elabore um algoritmo que receba a altura de 30 pessoas, calcule e apresente na tela: a maior altura, a menor altura, a média de altura, o número de pessoas com menos de 18 anos, o número de pessoas com mais de 60 anos. Para isso, utilize a estrutura enquanto.

5. Elabore um algoritmo que calcule o total de números primos entre 1 e 1000. Para isso, utilize a estrutura de repetição para.

REFLEXÃO

Quando as estruturas de repetição são introduzidas, deparamo-nos com a possibilidade de resolver muitos exercícios interessantes, dada as possibilidades e complexidades que elas permitem. Além disso, começamos a construir programa com aparência mais profissional.

Os conteúdos apresentados nesta unidade precisam de muita prática e dedicação. As aplicações e formas de utilização das estruturas de repetição são muito variadas e serão dominadas apenas com a prática.

Com esse tipo de estrutura de repetição, contemplamos as funções ou estruturas básicas dos principais paradigmas de programação. É muito importante que você, como futuro programador, tenha completo domínio sobre todos, pois o uso combinado permitirá o desenvolvimento de praticamente qualquer tipo de aplicação.

LEITURA

Com o estudo dos conceitos de Estrutura Condicional e Estrutura de Repetição, você poderá aprofundar a leitura da obra Fundamentos da programação de computadores. No livro, há capítulos específicos que descrevem cada uma das estruturas, além de uma infinidade de exemplos de codificação. A referência completa é descrita a seguir:

ASCENCIO, Ana Fernanda Gomes; EDILENE, Aparecida Veneruchi De. Fundamentos da programação de computadores: algoritmos, Pascal e C/C++. São Paulo: Prentice Hall, 2002.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E.e A. V. Fundamentos da programação de computadores: Algoritmos, Pascal, C/C++ e Java. 2. ed. São Paulo: Pearson Education, 2008.

FORBELLONE, A.L. V; EBERSPACHER, H. Lógica de programação. 3. ed. São Paulo: Makron Books, 2005.

PUGA, S.; RISSETTI, G. Lógica de programação e estrutura de dados: com aplicações em Java. 1. ed. São Paulo: Pearson Education, 2003.

SPALLANZANI, Adriana Sayuri; MEDEIROS, Andréa Teixeira de; FILHO, Juarez Muylaert. Linguagem UAL. Disponível em: <http://geocities.ws/ual_language/ual.html>. Acesso em: 25 abr. 2014.

NO PRÓXIMO CAPÍTULO

Vimos neste e no decorrer dos capítulos todos os comandos e estruturas básicas de algoritmos e das linguagens de programação em geral e já conseguimos resolver praticamente qualquer problema. No próximo capítulo veremos algumas estruturas de dados que nos permitirão realizar computação de um volume maior de dados tratando de problemas mais complexos, com menor quantidade de código.

5

Estrutura de dados homogêneas

5 Estrutura de dados homogêneas

Neste capítulo vamos iniciar o estudo estruturas de dados, mais especificamente estrutura de dados homogêneos unidimensionais, também conhecidos como vetores e estrutura bidimensionais ou como matrizes.
Para tal, faremos muito uso das estruturas de repetição estudadas nas unidades anteriores. Por isso, é muito importante que você tenha compreendido e praticado bastante para que não haja dificuldades nestas estruturas, dificultando seu aprendizado na utilização de vetores e matrizes. Caso ainda tenha dúvidas, sugiro que volte às unidades anteriores e pratique mais exercícios, até que fique mais fácil utilizar as estruturas de repetição.
Este conteúdo completa os conceitos básicos para a construção de programas com uma ampla gama de aplicabilidade.

OBJETIVOS

- Diferenciar estruturas homogêneas de estruturas heterogêneas.
- Construir programas usando matrizes unidimensionais (vetores).
- Construir programas utilizando matrizes bidimensionais.
- Usar matrizes de char em seus comandos.
- Compreender as vantagens no uso de matrizes bidimensionais.

REFLEXÃO

Você se lembra da teoria de conjuntos aprendida no ensino básico? É importante lembrar esses conceitos, pois eles estão diretamente relacionados com o conteúdo desta unidade.
Podemos definir um conjunto como um agrupamento de elementos do mesmo tipo. Na matemática, geralmente um conjunto é nomeado utilizando uma letra maiúscula do alfabeto. Um conjunto é descrito através da enumeração de seus elementos entre um par de chaves com uma vírgula separando os elementos. Veja exemplo:

M:{Janeiro, Fevereiro, Março, Abril, Maio, Junho}

Esse é o conjunto dos meses do primeiro semestre do ano. Como sempre há uma relação entre os elementos de um conjunto, é possível representá--lo por esta relação também.
Matriz é um arranjo de elementos dispostos em linhas e colunas e representada por

uma letra maiúscula do alfabeto. Seus elementos são representados entre um par de colchetes ou parênteses. Ou seja, uma matriz é um conjunto específico de elementos, representado de acordo com certa estrutura.

5.1 Estruturas homogêneas e heterogêneas de programação

Em programação, encontramos situações em que a utilização de uma simples variável não é o suficiente para representar situações ou objetos do mundo real. Se quisermos representar um ponto em um plano cartesiano, não conseguimos, em linguagens de programação, utilizando apenas uma variável. Se a análise de um ponto é composta por dois valores, o valor do ponto no eixo X e o valor do ponto no eixo Y, logo precisamos de duas variáveis distintas.

Para permitir que agrupemos variáveis que possuem certa relação, de forma a representar de maneira mais realista os objetos do mundo real, as linguagens de programação utilizam a técnica chamada de estrutura. Para melhor representar a situação descrita, podemos utilizar uma estrutura, com os dois valores existentes, como, por exemplo:

```
estrutura {
        inteiro X;
        inteiro Y;
} Ponto;
```

onde definimos uma estrutura chamada de ponto, composta de dois valores inteiros, X e Y, que irão representar o valor do ponto nos eixos X e Y, respectivamente. Não se preocupe neste momento com a sintaxe necessária para criação, cada linguagem possui a sua.

Basicamente, essas estruturas podem ser de dois tipos: homogêneas e heterogêneas.

Estruturas homogêneas são estruturas como o ponto definido anteriormente, em que todas suas subpartes ou elementos são de um único tipo. No caso do ponto, os dois elementos são do tipo inteiro.

Os exemplos mais comuns em que são utilizadas estruturas homegêneas em programação são vetores e matrizes.

Estruturas heterogêneas podem ser formadas por subpartes ou elementos de diferentes tipos, o que auxilia e simplifica muitos a estruturação de programas e representação de elementos do mundo real em um programa. Como, por exemplo, a representação dos dados de uma pessoa. Geralmente uma pessoa

possui um conjunto de dados que podem ser de diferentes tipos, como: nome, idade, sexo, endereço, telefone, CPF, RG.Podemos notar que entre os dados temos diferentes tipos; alguns são literais, como: nome, endereço, sexo, enquanto outros não numerais: idade, telefone, CPF e RG. Para representar uma pessoa, agrupando seus dados, utilizamos uma estrutura heterogênea, como a proposta abaixo:

```
estrutura {
        literal nome;
        inteiro idade;
        literal sexo;
        literal endereço;
        inteiro telefone;
        inteiro cpf;
        inteiro rg;
} Pessoa;
```

Este tipo de estrutura é muito importante para representar elementos complexos e heterogêneos do mundo real. Novamente, não se preocupe com a sintaxe de estrutura; mais adiante, no curso, você aprenderá outras estruturas para representar elementos heterogêneos. Apenas a título de curiosidade, na linguagem C elas são definidas utilizando-se o comando *struct*.

5.2 Tipo String

O tipo *String*, muito utilizado em todas as linguagens para armazenar e manipular conteúdos literais, na verdade é uma estrutura homogênea do tipo *char* ou um vetor de caracteres. A representação literal de uma string é sempre feita por um conjunto de caracteres da tabela ASCII, apresentados entre aspas duplas.

"Este é um exemplo de uma String."

A representação física de uma variável do tipo String é um conjunto de caracteres finalizado pelo caractere nulo, cujo valor inteiro é igual a zero e é representado pelo símbolo '\0', ou barra invertida zero, conhecido também apenas como barra zero. Logo, a representação da variável nome com o conteúdo "Maria", é dada da seguinte forma:

NOME					
M	a	r	i	a	\0
0	1	2	3	4	5

Um vetor de tamanho 6, ou seja de 6 posições, é iniciado na posição zero e terminado na posição 5 pelo símbolo '\0'. Logo, sempre que quisermos guardar uma palavra ou frase, precisamos de uma variável do tamanho dos dados mais um. Geralmente, essa complexidade é transparente e fica encapsulada no tipo *String*. Caso opte por trabalhar com a variação nativa ou básica do tipo, que é o vetor de caracteres, é preciso ter o cuidado na manipulação e definição dos tamanhos, bem como realizar os tratamentos necessários.

Veremos a seguir como funcionam os vetores; assim, poderemos também entender melhor o funcionamento do tipo *String*.

5.3 Matriz unidimensional (vetor)

Você deve se lembrar do ensino médio o conceito de matriz, que consiste de uma tabela de linha e colunas formando um conjunto de números ou elementos, a qual está representada na Figura 55. Na representação, temos uma matriz de dimensões m por n, ou seja, ela possui um número m de linha e um número n de colunas.

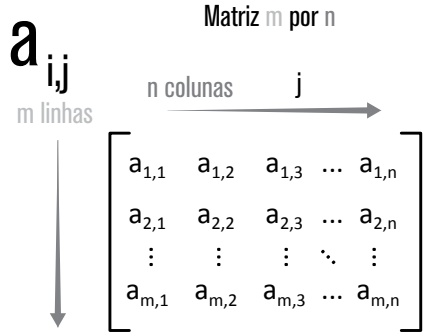


Figura 55 – Representação gráfica de uma matriz (retirado de: <http://pt.wikipedia.org/wiki/Ficheiro:Matriz_organizacao.png>).

Quando temos uma matriz em que m e n são diferentes de zero, essa matriz é bidimensional, ou seja, possui duas dimensões. Esse tipo de matriz será estudada um pouco mais adiante. Quando uma de suas dimensões é igual a zero, temos uma matriz unidimensional, também conhecida como vetor, que estudaremos em detalhes a seguir.

5.3.1 Conceito de vetor

Matematicamente, um vetor é a matriz de uma dimensão; em termos computacionais, é um arranjo homogêneo de dados. São duas definições semelhantes, porém sua representação no mundo real é um pouco diferente.

Computacionalmente, todo vetor pode ser composto apenas de um tipo e possui um tamanho finito, definido no momento de sua criação ou declaração.

Em grande parte das linguagens, um vetor de n posições é numerado de 0 a n-1. Sua primeira posição é a posição 0 e sua última posição é a posição n-1, como vimos no exemplo da String com o conteúdo "Maria", em que um vetor de char, de 6 posições, possui suas posições numeradas de 0 a 5.

Essa característica é devida à estruturação dos dados na memória do computador. Eles são estruturados de forma sequencial, onde o endereço base, ou endereço do primeiro elemento, é o próprio endereço do vetor, e todos os outros elementos são encontrados e acessados, através do endereço base mais seu deslocamento em relação a este endereço, o que depende de sua posição e do tamanho dos elementos.

Veja aa forma para encontrar qualquer endereço de um vetor:

Endereço Base + Posição * Tamanho do Tipo

Voltando ao nosso exemplo da String, vamos considerar que o endereço base seja 1200 e encontrar o endereço da letra 'r'.

NOME					
M	a	r	i	a	\0
0	1	2	3	4	5

A letra ‘r’ é o terceiro elemento do vetor, ou seja, está na posição 2, logo seu endereço será:

1200 + 2*(tamanho do char)

Considerando o tamanho do char como 4, temos:

1200 + 2*4 = 1208

Ou seja, a posição da letra ‘r’ na memória é 1208. Assim, considerando o primeiro elemento, essa é a forma mais simples para o computador calcular e encontrar a posição dos demais elementos de um vetor.

5.3.2 Declaração de um vetor

Para utilizarmos um vetor ou matriz unidimensional, temos que declará-lo, ou seja, precisamos criar uma variável do tipo vetorial. Para declarar uma variável vetorial, utilizamos a seguinte sintaxe:

LINGUAGEM UAL	LINGUAGEM C++
tipo nomeDoVetor**[**tamanahoDoVetor**]**;	**tipo** nomeDoVetor**[**tamanahoDoVetor**]**;

Exemplos:

LINGUAGEM UAL	LINGUAGEM C++
/* Vetor para guardar 10 notas */	
Int notas[10];	**Int** notas**[**10**]**;
/* Vetor para guardar o sexo de 50 pessoas */	
string sexo[50];	**char** sexo**[**50**]**;
/* Vetor para armazenar o valor do salário de 100 funcionários */	

LINGUAGEM UAL	LINGUAGEM C++
real salario [100];	**float** salario**[**100**]**;
/* Vetor para armazenar 1 nome com 30 caracteres */	
string nome;	char nome**[**30**]**;

Vendo os exemplos, podemos notar que o tamanho do vetor é sempre definido no momento de sua declaração, entre colchetes, e não se altera durante toda a sua existência. O tamanho deve ser sempre um número inteiro.

5.3.3 Inclusão de dados em um vetor

Após declarar um vetor, temos que popular este vetor, ou seja, incluir os dados desejados. Podemos fazer a inicialização juntamente da declaração ou posteriormente a qualquer momento.

Para inicializar um ver com um determinado valor, ou seja, declará--lo e em sequência já inserir um conjunto de dados, utilizamos os comandos a seguir:

LINGUAGEM UAL
tipo nomeDoVetor**[**tamanahoDoVetor**]** = **{** dado1, dado2, ..., dadotamanhoDoVetor **}**
LINGUAGEM C++
tipo nomeDoVetor**[**tamanahoDoVetor**]** = **{** dado1, dado2, ..., dadotamanhoDoVetor **}**

Exemplos:

LINGUAGEM UAL	LINGUAGEM C++
/* Inicializar um vetor notas de tamanho 3 com os valores 5, 8 e 10 */	
Int notas**[]** = **{**5, 8, 10**}**;	**Int** notas**[]** = **{**5, 8, 10**}**;
/* Inicializar um vetor com sexo de 5 pessoas com os valores M,M,F,M,F */	
string sexo**[]** = **{**'M', 'M', 'F', 'M', 'F' **}**;	**string** sexo**[]** = **{**'M', 'M', 'F', 'M', 'F' **}**;

LINGUAGEM UAL	LINGUAGEM C++
/* Inicializar um vetor nota de tamanho 3 com os valores reais 5.3, 8.5 e 10.1 */	
real vreais [3] = **{**5.3, 8.5, 10.1**}**;	**float** vreais**[**3**]** = **{**5.3, 8.5, 10.1**}**;
/* Inicializar uma variável para armazenar 1 nome o valor "Joao" */	
string nome = "Joao";	**char** nome**[**5**]** = "Joao";

Perceba que a inicialização é feita utilizando-se os valores entre chaves, divididos por vírgula, e que a indicação do tamanho do vetor é opcional. Quando declaramos um vetor e já o inicializamos, o compilador utiliza a quantidade de dados usada para inicializar o vetor como tamanho, caso tenhamos definido um tamanho, será usado o tamanho definido com limite. Porém, temos um caso em especial, o vetor de *char*, que pode ser inicializado pela *string* entre parênteses. Nesse caso, é importante lembrar da estrutura do vetor de *char*, que ele é finalizado pelo símbolo '\0', portanto deverá sempre ter o tamanho da *string* somado de um, como no exemplo anterior: o nome João possui 4 letras, mas o vetor terá o tamanho 5.

Para atribuir valores após a inicialização, utilizamos a estrutura a seguir:

LINGUAGEM UAL	LINGUAGEM C++
nomeDoVetor**[**posição**]** = valor;	nomeDoVetor**[**posição**]** = valor;

Exemplos:

LINGUAGEM UAL	LINGUAGEM C++
/* Atribuir o valor 20 para a posição 2 do vetor notas */	
notas**[**1**]** = 20;	notas**[**2**]** = 20;
/* Atribuir o valor F na primeira posição do vetor */	
sexo**[**0**]** = 'F';	sexo**[**0**]** = 'F';

/* Atribuir o valor 12,23 na posição 3 do vetor vreais */	
vreais **[**2**]** = 12.12;	vreais **[**2**]** = 12.12;
/* Trocar o a por ã do vetor nome nome que possui o valor "Joao" */	
nome = "João";	nome**[**2**]** = 'ã';

Para atribuir um valor a uma posição de um vetor, indicamos a posição desejada entre colchetes na frente do nome da variável seguido pela atribuição do valor desejado. Temos sempre de lembrar que os vetores começam na posição zero, portanto, se queremos atribuir um valor na posição n, devemos utilizar o índice n-1.

Se queremos trocar o valor da quinta posição do vetor, devemos utilizar o índice 4.

Imagine agora que você não tenha os valores na hora da inicialização, mas precisará popular posteriormente com valores informados pelo usuário, como você realizaria esta tarefa? Para um vetor de 8 posições, usaria 8 comandos de leitura, conforme o exemplo a seguir?

LINGUAGEM UAL	LINGUAGEM C++
int vetor[8]; imprima "\nDigite 1o elemento: "; leia vetor[0]; imprima "\nDigite 2o elemento: "; leia vetor[1]; imprima "\nDigite 3o elemento: "; leia vetor[2]; imprima "\nDigite 4o elemento: "; leia vetor[3]; imprima "\nDigite 5o elemento: "; leia vetor[4]; imprima "\nDigite 6o elemento: "; leia vetor[5]; imprima "\nDigite 7o elemento: "; leia vetor[6]; imprima "\nDigite 8o elemento: "; leia vetor[7];	Int vetor[8] cout<<"\nDigite 1o elemento: "; cin>>vetor[0]; cout<<"\nDigite 2o elemento: "; cin>>vetor[1]; cout<<"\nDigite 3o elemento: "; cin>>vetor[2]; cout<<"\nDigite 4o elemento: "; cin>>vetor[3]; cout<<"\nDigite 5o elemento: "; cin>>vetor[4]; cout<<"\nDigite 6o elemento: "; cin>>vetor[5]; cout<<"\nDigite 7o elemento: "; cin>>vetor[6]; cout<<"\nDigite 8o elemento: "; cin>>vetor[7];

Essa é a forma mais simples e direta de resolver este problema, porém, se tivéssemos que entrar com um vetor de 100 posições e com um de 1000 posições, teríamos de repetir 1000 vezes as duas linhas de código? Você se lembra das estruturas de repetição que aprendemos anteriormente? Elas não seriam úteis nesse caso? Vamos ver como poderíamos usá-las.

Primeiramente temos de perceber que há algumas informações que não se repetem, como a posição que estamos lendo. Para isso, devemos quebrar a mensagem mostrada.

LINGUAGEM UAL	LINGUAGEM C++
imprima "\nDigite ", ..., o elemento: ";	**cout**<<"\nDigite " <<... <<"o elemento: ";

Para controlar a posição de leitura, temos de utilizar uma variável.

LINGUAGEM UAL	LINGUAGEM C++
leia vetor**[**n**]**;	**cin**>>vetor**[**n**]**;

kPor fim, utilizamos a estrutura de repetição para percorrer todo o vetor. Vamos ver como ficaria para ler o mesmo vetor de 8 posições utilizado no exemplo anterior.

LINGUAGEM UAL	LINGUAGEM C++
para (n<-0; n<7;n++**)** **{** **imprima** "\nDigite ", n+1, "o elemento: "; **leia** vetor**[**n**]**; **}**	**for(int** n = 0; n<7; n++**)** **{** **cout**<<"\nDigite " <<n+1 <<"o elemento: "; **cin**>>vetor**[**n**]**; }

Vejamos o Teste de Mesa para a execução deste trecho de código:

MEMÓRIA PRINCIPAL			DISPLAY
N	NÚMEROS		
~~0~~	0	10	Digite 1º elemento: 10
~~1~~	1	16	Digite 2º elemento: 16
N	NÚMEROS		
~~2~~	2	21	Digite 3º elemento: 21
~~3~~	3	24	Digite 4º elemento: 24
~~4~~	4	27	Digite 5º elemento: 27
~~5~~	5	30	Digite 6º elemento: 30
~~6~~	6	31	Digite 7º elemento: 31
7	7	32	Digite 8º elemento: 32
8			

Agora que já sabemos como ler um vetor de qualquer tamanho, vamos estruturar um código genérico, o qual podemos utilizar sempre que precisarmos, pois serve de base para qualquer vetor.

LINGUAGEM UAL	LINGUAGEM C++
para (n<-0; n<tamanho;n++**)** **{** imprima "\nDigite ...", n+1, "...."; **leia** nomeDoVetor**[**n**]**; **}**	**for(int** n = 0; n<tamanho; n++**)** **{** **cout**<<"\nDigite " <<n+1 <<"... "; **cin**>>nomeDoVetor**[**n**]**; }

Outra forma de se inserir dados em um vetor é por meio da atribuição. O

processo é exatamente o mesmo do anterior, porém, no lugar do comando de leitura da fonte de entrada do computador, atribuímos um valor diretamente com o comando de atribuição. Podemos fazer isso para um único valor, ou utilizando uma estrutura de repetição, conforme segue.

LINGUAGEM UAL	LINGUAGEM C++
nomeDoVetor**[**posição**]** <- expressão com o conteúdo ou valor a ser atribuído no vetor;	nomeDoVetor**[**posição**]** = expressão com o conteúdo ou valor a ser atribuído no vetor;
para (n<-0; n<tamanho;n++**)** **{** nomeDoVetor**[**n**]** <- expressão com o conteúdo ou valor a ser atribuído no vetor; **}**	**for(int** n = 0; n<tamanho; n++) **{** nomeDoVetor**[**n**]** = expressão com o conteúdo ou valor a ser atribuído no vetor; }

5.3.4 Leitura de dados de um vetor

A leitura ou obtenção de dados de um vetor é muito semelhante à inclusão de dados em um vetor, como vimos anteriormente. A principal diferença é com o comando e a localização do vetor nos comandos, conforme veremos a seguir.

Para realizar a impressão dos valores da tela, a saída sempre se inicia com um trecho identificando o seu conteúdo, para, em seguida, fazermos a impressão utilizando o comando imprima ou cout.

Como já sabemos, podemos fazer isso de forma individual, ou para todo o vetor, utilizando ou não uma estrutura de repetição. Vamos ver diretamente a forma utilizando estrutura de repetição.

Quando vamos imprimir um vetor, podemos imprimi-lo em uma linha, ou seja, com todos os valores na mesma linha, ou em uma coluna com um valor em cada linha. Veremos as duas formas.

LINGUAGEM UAL	LINGUAGEM C++
/* IMPRESSÃO COMO COLUNA */	
imprima "\nTítulo\n"; **para (**n<-0; n<tamanho;n++**)** **{** **imprima** "\n", nomeDoVetor[n]; **}**	**cout** << "\nTítulo\n"; **for(int** n = 0; n<tamanho; n++**)** **{** **cout**<<"\n..."<<nomeDoVetor**[**n**]**; }
/* IMPRESSÃO COMO LINHA */	
imprima "\nTítulo\n"; **para** (n<-0; n<tamanho;n++) **{** **imprima** nomeDoVetor**[**n**]**, "\t: "; **}**	**cout** << "\nTítulo\n"; **for(int** n = 0; n<tamanho; n++**)** **{** **cout**<<nomeDoVetor**[**n**]** << "\t: ";

5.4 Matriz bidimensional (matriz)

Agora que já temos total conhecimento sobre vetores ou matrizes unidimensionais, vamos um pouco mais além e aprender sobre as matrizes bidimensionais. Sabemos uma *string* é um vetor; se quisermos armazenar um conjunto de nomes, ou um conjunto de *string*, precisamos de uma estrutura bidimensional, uma matriz bidimensional. Antes de iniciar o estudo desta etapa, sugiro que vá até o item de atividades sobre vetores para consolidar seu conhecimento, pois isso facilitará muito a compreensão sobre matrizes bidimensionais.

5.4.1 Conceito de matriz

Vamos começar o estudo de matrizes bidimensionais analisando algumas situações.

O professor ou tutor de sua turma precisa utilizar uma ferramenta para controlar a nota e o desempenho de todos os alunos. Vamos imaginar que sua turma possua 60 alunos, e que cada aluno realize 4 avaliações durante o semestre.

Logo, o professor ou tutor deverá guardar as notas das 4 avaliações mais a média de cada aluno. Geralmente, eles ainda guardam mais uma informação relevante do desempenho do aluno que é a frequência, ou a quantidade de faltas que o aluno possui. Se você for fazer um sistema para ele controlar essas notas, como faria? Utilizaria um vetor para cada tipo de dado, conforme a seguir?

Vetor Notas 1

A1	A2	A3	A4	A4	A6	.....	A60

Vetor Notas 2

A1	A2	A3	A4	A4	A6	.....	A60

Vetor Notas 3

A1	A2	A3	A4	A4	A6	.....	A60

Vetor Notas 4

A1	A2	A3	A4	A4	A6	.....	A60

Vetor Faltas

A1	A2	A3	A4	A4	A6	.....	A60

É uma opção possível e viável, porém temos 5 vetores independentes, sem qualquer relação, os quais temos de manipulá-los, gerenciá-los e controlá-los de forma independente, tomando sempre o cuidado para que nunca invertamos um índice e causar a mistura das notas de alunos diferentes ou cadastramos uma informação em local indevido.

Se fôssemos controlar essas informações, provavelmente a solução mais lógica seria montar uma tabela com esses dados. Provavelmente, não utilizaríamos uma lista para cada avaliação separadamente. Então, por que não fazemos da mesma forma em um sistema computacional?

	ALUNO 1	ALUNO 2	ALUNO 3	ALUNO 4		ALUNO 60
NOTA 1	V1	V2	V3	V4	.....	V60
NOTA 2	V1	V2	V3	V4	.....	V60
NOTA 3	V1	V2	V3	V4	.....	V60
NOTA 4	V1	V2	V3	V4	.....	V60
FALTAS	V1	V2	V3	V4	.....	V60

Para controlarmos nossas despesas do ano, geralmente utilizamos uma tabela ou planilha, como no exemplo a seguir.

	JAN.	FEV.	MAR.	ABR.	MAIO	JUN.	...
ÁGUA							
LUZ							
ALUGUEL							
TRANSPORTE							
SUPERMERCADO							
RESTAURANTE							

Todos os exemplos do nosso dia a dia em que utilizamos uma tabela para organizar os dados podem ser diretamente mapeados em uma matriz, de forma simples e direta. Vamos ver agora como fazer isso utilizando linguagem de programação. Nessa etapa, vamos utilizar apenas a linguagem C++.

5.4.2 Declaração de uma matriz

A declaração e manipulação de matrizes bidimensionais é bem semelhante a de vetores. Porém, agora temos que tratar duas dimensões, por isso, ao invés de utilizarmos um par de colchetes, usamos dois pares, um para cada dimensão da matriz. Apesar de não ser foco de nosso estudo neste momento, na linguagem C++ é possível declarar matrizes multidimensionais, ou seja, com mais de duas dimensões. Nesse caso, é necessário usar um par de colchetes para cada dimensão.

Veja o padrão para a declaração de matrizes bidimensionais.

LINGUAGEM C++

```
tipo nomeDaMatriz[tamanahoD1] [tamanahoD2] ;
```

Exemplos:

LINGUAGEM C++

```
/* declara  uma matriz para guardar 3 informações diferentes de 300 pacientes */

float pacientes[300] [3];

/* declara uma matriz de 4 notas para 100 alunos */

float notas[100] [4];

/* Declara uma matriz para guardar 1000 nomes de até 30 caracteres */

char nome[1000] [31];
```

Note que, na declaração da matriz, para guardar 100 nomes com até 30 caracteres, utilizamos uma matriz bidimensional de 1000x31, e não 1000x30. Você deve se lembrar de que vimos no início deste capítulo que todo vetor de char termina com o símbolo '\0', por isso precisamos de uma coluna a mais. É necessário estar sempre atento a este tipo de detalhe.

Olhando um pouco mais atentamente aos exemplos apresentados, podemos notar que todos trazem o nome de entidades (pacientes, alunos, nomes) nas linhas (primeiro índice da matriz) e suas propriedades nas colunas. Você poderia se perguntar se isso é uma regra ou se poderíamos utilizar matrizes transpostas ou invertidas para representar as mesmas informações. Esta pergunta é muito comum e sempre surge. Você poderia , sem problemas, utilizar qualquer configuração para sua matriz, conforme o exemplo abaixo, em uma matriz com 4 notas para 8 alunos.

float notas[8] [4];

	NOTA 1	NOTA 2	NOTA 3	NOTA 4
ALUNO 1				
ALUNO 2				
ALUNO 3				
ALUNO 4				
ALUNO 5				
ALUNO 6				
ALUNO 7				
ALUNO 8				

float notas[4] [8];

	ALUNO 1	ALUNO 2	ALUNO 3	ALUNO 4	ALUNO 5	ALUNO 6	ALUNO 7	ALUNO 8
NOTA 1								
NOTA 2								
NOTA 3								
NOTA 4								

Os dois formatos anteriores são corretos, porém, de uma maneira geral, a maioria dos programadores utiliza a primeira forma, com linhas para identificar as entidades e colunas para as propriedades ou atributos das entidades. Como veremos a seguir, essa forma facilita as estruturas de programação e o entendimento dos códigos. Além disso, a matriz bidimensional é armazenada na memória do computador como se fosse uma única linha contígua, com todas as linhas em sequência, uma após a outra. A Figura 2 abaixo mostra a organização em memória. Logo, estruturando da primeira forma, temos os dados mais organizados na memória.

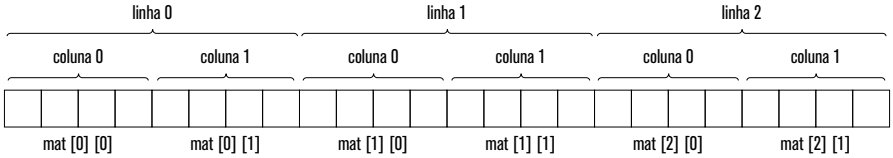


Figura 2 – Organização de uma matriz bidimensional em memória

5.4.3 Inclusão de dados em uma matriz

O armazenamento ou inclusão de dados em uma matriz bidimensional é feito de forma semelhante à utilizada nos vetores. Podemos realizar a inicialização em sua declaração ou fazer uma atribuição das informações desejadas.

LINGUAGEM C++

```
tipo nomeDaMatriz[tamanahoD1] [tamanahoD2] = { dado1, dado2, ..., dadotamanhoD1xD2};
```

Exemplos:

LINGUAGEM C++

```
/* Inicializa uma matriz de 3 linhas e 4 colunas */
int matrx [3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
/* Inicializa uma matriz com 3 nomes de até 10 caracteres */
```

```
char str_vect [3][11] = {"Joao", "Maria", "Jose"};

                /* Inicializa uma matriz de 5 linhas e 2 colunas */

int matrx [][2] = {1,2,2,4,3,6,4,8,5,10};
```

Nos exemplos apresentados, podemos notar algumas características e peculiaridades da inicialização de matrizes bidimensionais. Como já dito, a estrutura é muito semelhante à de vetores, porém precisamos ter em mente que estamos trabalhando com duas dimensões, e o nome da variável deve ser seguido de dois pares de colchetes, como vimos anteriormente. Os dados que queremos popular são apresentados entre chaves de forma sequencial. Não precisamos nos preocupar com a divisão entre linhas, a linguagem se incumbirá de gerenciar isso; precisamos apenas fornecer a quantidade correta de dados. Podemos ver pelo primeiro exemplo que, para inicializar uma matriz bidimensional de três linhas e quatro colunas, basta informar 12 valores entre chaves, divididos por vírgulas. O C++ irá distribuir de forma sequencial os dados nas linhas e nas colunas, sendo assim, os quatro primeiros valores serão colocados na primeira linha, os quatro seguintes na segunda e os quatro últimos na terceira linha.

No segundo exemplo, podemos notar que, para inicializar uma matriz bidimensional com 3 nomes, basta indicarmos o tamanho desejado e informarmos os nomes entre chaves, separados por vírgulas. Novamente o C++ irá distribuir os nomes, alocando um em cada linha, de forma sequencial. No terceiro e último exemplos, podemos ver que, quando estamos inicializando uma matriz bidimensional, não precisamos obrigatoriamente informar os dois índices, podemos informar somente o número de colunas; sendo assim, o C++ irá criar quantas linhas forem necessárias para guardar os dados. Fique atento, pois isso pode ocorrer apenas no momento da inicialização. Em todos os acessos às matrizes, será necessário informar os dois índices, de linha e coluna.

Após a declaração ou inicialização para atribuirmos um valor a uma matriz bidimensional, basta informar os índices de linha e coluna desejados e o valor a ser atribuído, de forma semelhante ao feito com os vetores. Segue a estrutura e alguns exemplos.

LINGUAGEM C++

```
tipo nomeDaMatriz[posiçãoLinha] [posiçãoColuna]  = ValorDesejado;
```

Exemplos:

LINGUAGEM C++

```
/* atribuir o valor 13 à primeira posição da primeira linha */
matrx [0][0] = 13;
/* Atribuir o caracter 'P' a primeira posição da linha terceira linha */
char str_vect [2][0] = 'P';
/* Atribuir o valor 11 a segunda coluna da segunda linha */
int matrx [1][1] = 11;
```

Podemos notar pelo exemplo que basta termos o cuidado de lembrar sempre que os índices se iniciam em 0 e que vão até tamanho –1.

Vejamos agora como é inserir todos os valores em uma matriz bidimensional após sua declaração, como, por exemplo, recebendo as entradas através do dispositivo de entrada padrão (teclado). De maneira semelhante aos vetores, teremos de utilizar estruturas de repetição, porém, como estamos tratando de estruturas com duas dimensões, precisaremos percorrer as duas estruturas. Veja a seguir como fica um trecho para receber a entrada para você usar como base para qualquer matriz bidimensional, do tipo numérica.

LINGUAGEM C++

```
for(int L = 0; L<tamanhoLinha; L++)
{
for(int C = 0; C<tamanhoColuna; C++)
{
cout<<"\nLinha " <<L+1 <<" coluna "<<C+1<<":;
cin>>nomeMatriz[L] [C];
}
}
```

Para percorrer as duas dimensões da matriz bidimensional, utilizamos duas estruturas de repetição. A estrutura externa percorre as linhas e a estrutura interna percorre as colunas de cada linha. É muito importante que você analise e entenda o funcionamento desta estrutura, pois será muito utilizada sempre que precisar percorrer estruturas multidimensionais.

Vejamos agora a estrutura para a leitura de matriz bidimensional de char.

LINGUAGEM C++

```
for(int L = 0; L<tamanhoLinha; L++)
{
cout<<"\nLinha número " <<L+1 <<": ";
cin.getLina(nomeMatriz[L], tamanhoDeclardoNaColuna);
}
```

Perceba que utilizamos apenas uma estrutura de repetição, porém a associamos ao comando de entrada cin.getLine que lê os caracteres de um arranjo de char, inclusive o caractere limitador '\0'. Sendo assim, ele obtém todas as colunas da linha indicada e as coloca no local correto.

5.4.4 Leitura de dados de uma matriz

A leitura ou obtenção de dados de uma matriz bidimensional, ao exemplo do que ocorre com o vetor, é muito semelhante à inclusão que vimos anteriormente. Para realizar a impressão dos valores da tela, a saída sempre se inicia com um trecho identificando o seu conteúdo, em seguida fazemos a impressão utilizando o comando *imprima* ou *cout*.

Como já vimos, podemos fazer isso de forma individual, ou para toda a matriz bidimensional, utilizando ou não uma estrutura de repetição. Vamos ver diretamente a forma utilizando as estruturas de repetição.

Veremos a seguir como imprimir uma matriz bidimensional numérica e como exibir uma matriz de char.

LINGUAGEM C++

/* IMPRESSÃO COMO COLUNA */

```
cout << "\nTítulo\n";
for(int L = 0; L<tamanhoLinha; L++)
{
for(int C = 0; C<tamanhoColuna; C++)
{
cout<< nomeMatriz[L] [C];
}
cout<<"\n";
}
```

/* IMPRESSÃO COMO LINHA */

```
cout << "\nTítulo\n";
for(int L = 0; L<tamanhoLinha; L++)
{
cout<< nomeMatrizChar[L] [C]<<"\n";
}
```

ATIVIDADE

1. Construa um programa que armazene valores reais em um vetor de 10 posições. Imprima na tela o vetor com o dobro dos valores.

2. Faça um programa que leia matrículas e duas notas de 5 alunos. Calcule e armazene a média aritmética. Exiba a média e a situação final: AP (aprovado) ou RP (reprovado). A média é 6,0.

3. Construa um programa que leia e armazene 5 produtos e seus valores de compra e venda. Imprimir todos os dados em tela.

4. Construa um programa que leia e armazene números em uma matriz de ordem 5. Exiba todos os elementos da matriz e, em seguida, apresente apenas os elementos de sua diagonal principal (onde o índice da linha é igual ao índice da coluna).

REFLEXÃO

Neste capítulo, vimos como manipular estruturas mais complexas, como vetores e matrizes. No seu dia a dia de programador, as estruturas nem sempre serão simples e de fácil visualização e manipulação, por isso é muito importante conhecer bem e ter facilidade com a manipulação destas estruturas.
Pesquise e pratique até que tenha o total controle delas. Você não irá se arrepender.

LEITURA

Reveja o conteúdo e mais alguns detalhes sob outras perspectivas no link abaixo: <http://pt.wikibooks.org/wiki/Programar_em_C++/Vetores>.

Veja algumas operações avançadas que utilizam vetores e matriz. Muitas dessas operações são estruturas de dados que serão estudadas mais adiante em nosso curso. Então, não se preocupe se tiver alguma dificuldade.

REFERÊNCIAS BIBLIOGRÁFICAS

ASCENCIO, A. F. G.; CAMPOS, E.e A. V. Fundamentos da programação de computadores: Algoritmos, Pascal, C/C++ e Java. 2. ed. São Paulo: Pearson Education, 2008.

FORBELLONE, A.L. V; EBERSPACHER, H. Lógica de programação. 3. ed. São Paulo: Makron Books, 2005.

PUGA, S.; RISSETTI, G. Lógica de programação e estrutura de dados: com aplicações em Java. 1. ed. São Paulo: Pearson Education, 2003.

SPALLANZANI, Adriana Sayuri; MEDEIROS, Andréa Teixeira de; FILHO, Juarez Muylaert. Linguagem UAL. Disponível em <http://geocities.ws/ual_language/ual.html>. Acesso em: 25 abr. 2014.

DASGUPTA, Sanjoy; PAPADIMITRIOU, Christos; VAZIRANI, Umesh. Algoritmos. 1. ed. São Paulo: McGraw-Hill Brasil, 2009.

FEOFILOFF, Paulo. Algoritmos em linguagem C. 1. ed. Rio de Janeiro: Campus, 2008.

MANZANO, José Augusto N. G.; OLIVEIRA, Jayr Figueiredo. Algoritmos: lógica para desenvolvimento de programação de computadores. 22. ed. São Paulo: Érica, 2009.

MIZRAHI, V. V. Treinamento em linguagem C / Algoritmos. São Paulo: Prentice Hall, 2008 (Biblioteca Virtual) DEITEL, P. J.; DEITEL, H.. C: como programar. 6.ed. São Paulo: Pearson Prentice Hall, 2011. (Biblioteca Virtual).