



Algoritmos e Técnicas de Programação

Algoritmos e Técnicas de Programação

Vanessa Cadan Scheffer
Marcio Aparecido Artero

© 2018 por Editora e Distribuidora Educacional S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

Presidente

Rodrigo Galindo

Vice-Presidente Acadêmico de Graduação e de Educação Básica

Mário Ghio Júnior

Conselho Acadêmico

Ana Lucia Jankovic Barduchi

Camila Cardoso Rotella

Danielly Nunes Andrade Noé

Grasiele Aparecida Lourenço

Isabel Cristina Chagas Barbin

Lidiane Cristina Vivaldini Olo

Thatiane Cristina dos Santos de Carvalho Ribeiro

Revisão Técnica

Alessandra Cristina Santos Akkari

Ruy Flávio de Oliveira

Editorial

Camila Cardoso Rotella (Diretora)

Lidiane Cristina Vivaldini Olo (Gerente)

Elmir Carvalho da Silva (Coordenador)

Leticia Bento Pieroni (Coordenadora)

Renata Jéssica Galdino (Coordenadora)

Dados Internacionais de Catalogação na Publicação (CIP)

Scheffer, Vanessa Cadan
S316a Algoritmos e técnicas de programação / Vanessa Cadan
Scheffer, Marcio Aparecido Artero. – Londrina : Editora e
Distribuidora Educacional S.A., 2018.
248 p.

ISBN 978-85-522-1079-5

1. Programação. 2. Software. 3. Algoritmo. I. Scheffer,
Vanessa Cadan. II. Artero, Marcio Aparecido. III. Título.

CDD 005.7

Thamiris Mantovani CRB-8/9491

2018
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 – Londrina – PR
e-mail: editora.educacional@kroton.com.br
Homepage: <http://www.kroton.com.br/>

Sumário

Unidade 1 Fundamentos de algoritmos e das linguagens de programação	7
Seção 1.1 - Introdução aos algoritmos	9
Seção 1.2 - Conceitos Básicos de Linguagens de Programação	28
Seção 1.3 - Componentes e elementos de Linguagem de Programação	46
Unidade 2 Constantes, variáveis e operações	69
Seção 2.1 - Constantes e variáveis com tipos de dados primitivos	71
Seção 2.2 - Variáveis com tipos de dados compostos e ponteiros	88
Seção 2.3 - Operações e Expressões	107
Unidade 3 Estruturas de decisão e repetição	123
Seção 3.1 - Estruturas de decisão condicional	125
Seção 3.2 - Estruturas de repetição condicional	148
Seção 3.3 - Estruturas de repetição determinísticas	167
Unidade 4 Funções e recursividade	189
Seção 4.1 - Procedimentos e funções	191
Seção 4.2 - Escopo e passagem de parâmetros	207
Seção 4.3 - Recursividade	223

Palavras do autor

Caro aluno(a), bem-vindo(a) ao estudo dos algoritmos e das técnicas de programação. Esse livro representa um marco na sua caminhada, pois ao apropriar-se dos conhecimentos que aqui serão apresentados e discutidos, você dará um passo importante rumo a se tornar um profissional engajado em soluções tecnológicas.

A evolução da computação (hardware e software), desde sempre, teve como norte a necessidade de resolver problemas. Atualmente, vivemos em um cenário no qual o hardware alcançou altos níveis de desempenho, entretanto, para que ele seja de fato útil, será necessário software que realize tarefas de forma automatizada e precisa. Esses programas de computadores são construídos seguindo um conjunto de regras e técnicas bastante específicas, portanto, nessa era digital é imprescindível que profissionais de diversas áreas aprendam essas ferramentas, para que possam contribuir e evoluir em sua área de atuação.

Diante dessa demanda por profissionais capazes de solucionar problemas, na primeira unidade desse livro, você terá a oportunidade de conhecer e compreender o que são os algoritmos, as linguagens de programação e a estrutura de um programa de computador. Na segunda unidade, você aprenderá o que são, quais os tipos e para que servem as constantes e variáveis dentro de uma linguagem de programação. O conteúdo da terceira unidade abrirá para você uma gama de possibilidades, pois você conhecerá o que são, quais os tipos, para que servem e como utilizar as estruturas de decisão e repetição para criar códigos mais eficientes. Na última unidade, você aprenderá uma técnica que permitirá a você organizar e otimizar seu programa, estamos falando de funções e procedimentos.

Nessa jornada, o aprendizado e domínio dos algoritmos e técnicas de programação só terá sucesso com o seu empenho em estudar e implementar todos os exemplos e exercícios que serão propostos ao longo do livro. A cada unidade você aumentará seu repertório de técnicas, podendo se aventurar em problemas cada vez mais complexos. Para que possamos começar nosso trabalho, convido

você a se desafiar de forma que alcance excelentes resultados e seja um profissional diferenciado nessa área de grandes oportunidades.

Bons estudos!

Fundamentos de algoritmos e das linguagens de programação

Convite ao estudo

Iniciamos aqui a primeira unidade do livro Algoritmos e Técnicas de Programação, aproveitem ao máximo o conteúdo que nele será desenvolvido e com certeza proporcionará a você a oportunidade de ser um exímio programador.

Não tem como negar: a tecnologia é fascinante e com ela aprimoramos técnicas e elevamos o nível de conhecimento para solucionar os mais diversos tipos de problemas. Nada melhor que conhecer e compreender o que são os algoritmos, as linguagens de programação e a estrutura de um programa de computador e, assim, caminhar para execução das mais diversas tarefas computacionais.

Nesta unidade, você terá o prazer em conhecer uma empresa de tecnologia de informação cujo foco principal é o desenvolvimento de software para instituições de ensino. Com a grande demanda de negócios no setor educacional a empresa criou um projeto para contratação de estagiários para trabalhar com programação e então atender a demanda de mercado. Você, sendo exímio profissional na área de tecnologia da informação foi incumbido de treinar os estagiários.

A empresa não exigiu nenhuma experiência para os candidatos e, por este motivo, você deverá apresentar as definições e aplicações dos algoritmos, as inovações e os diferentes paradigmas para área de programação, além

dos componentes e estruturas utilizadas na linguagem de programação C.

Após esse treinamento inicial, o estagiário orientado por você terá a oportunidade de saber reconhecer os conceitos e parte da estrutura dos algoritmos e das linguagens de programação.

Para fazer valer a proposta desta unidade, na primeira seção, você terá a oportunidade de estudar os conceitos e introdução aos algoritmos, para que, dessa forma, aplique sua criatividade nas criações de algoritmos. Na segunda seção, será apresentado a você as famílias de linguagens de programação, assim como, sua usabilidade e as oportunidades que o mercado de trabalho reserva para o profissional de programação. Na terceira seção, será apresentado os componentes de um programa de computador, no qual será trabalhado a manipulação das informações e por fim, as estruturas dos algoritmos e programas.

Muito bem, essa é a proposta, você aceita o desafio?

Sigamos em frente.

Seção 1.1

Introdução aos algoritmos

Diálogo aberto

Caro aluno, você inicia agora o seu trabalho na empresa responsável por criar softwares educacionais. Como mencionado, você foi incumbido de treinar os estagiários contratados para trabalhar com programação dentro da empresa, lembrando que os mesmos não têm experiência alguma com programação ou até mesmo com algoritmos.

Antes de dar continuidade, vale saber que um algoritmo é uma sequência finita de passos que podem levar à criação e execução de uma determinada tarefa com a intenção de resolver uma problemática Forbellone e Eberspächer (2005). Sendo assim, você precisa entender as definições de um algoritmo, suas aplicações e tipos antes de avançar para os próximos níveis deste material.

Para provocar a curiosidade nos seus estagiários, construa um algoritmo em linguagem natural, diagrama de blocos (fluxograma) e pseudocódigo para cadastrar os dados pessoais do aluno (nome, endereço, cidade e estado) e mostrar na tela do computador o resultado do cadastro. Mostre o resultado a eles e faça a seguinte pergunta: os algoritmos podem contribuir de forma significativa na elaboração de códigos de programação?

Agora chegou o momento de iniciar essa jornada, muita atenção às aulas e um ótimo estudo!

Não pode faltar

Olá! A partir de agora você vai desmistificar como funciona os algoritmos e quais as suas aplicações dentro da programação, você conhecerá conceitos, aplicações e os tipos de algoritmos. Para tal, vamos resgatar alguns autores que descreveram as definições sobre algoritmos: segundo Szwarcfiter e Markenzon (1994), algoritmos são definidos como sendo o processo sistemático para a resolução

de um problema. Segundo Manzano (2015), um algoritmo é um conjunto lógico de operações predefinidas que resolva um determinado problema de forma intuitiva. Saliba (1993), Berg e Figueiró (1998) descrevem algoritmos como sendo uma sequência ordenada de passos que deve ser seguida para a realização de uma tarefa. Enfim, os algoritmos nortearão você a descobrir qual o melhor percurso para solucionar um problema computacional.

Partindo das definições citadas, veja a rotina para realização de um algoritmo para efetuar o cozimento de um arroz:

1. Acender o fogo.
2. Refogar os temperos.
3. Colocar o arroz na panela.
4. Colocar a água.
5. Cozinhar o arroz.
6. Abaixar o fogo.
7. Esperar o ponto.
8. Desligar o fogo.
9. Servir o arroz.

Podemos, ainda, criar um algoritmo um pouco mais detalhado para preparar o cozimento do arroz:

- 1- Comprar o arroz.
- 2- Analisar a qualidade.
- 3- Realizar a pré-seleção para o cozimento.
- 4- Preparar o tempero.
- 5- Pegar a panela.
- 6- Acender o fogo.
- 7- Colocar os temperos na panela para refogar.
- 8- Adicionar o arroz.
- 9- Colocar a água na medida considerada ideal par a quantidade.
- 10- Aguardar a água secar.
- 11- Baixar o fogo.
- 12- Fechar a panela com a tampa.
- 13- Aguardo o ponto.

14- Desligar o fogo.

15- Servir o arroz.

Perceba que não existe somente uma forma de realizar um algoritmo, você pode criar outras formas e sequências para obter o mesmo resultado, ou seja, eles são independentes, porém, com a mesma finalidade de execução.

Pois bem, você pode representar os algoritmos em três partes: **Entrada, Processamento e Saída**. Por exemplo:

- **Entrada:** ingredientes para o preparo do arroz.
- **Processamento:** o cozimento do arroz.
- **Saída:** finalização do arroz (Momento que será servido).

Neste livro, você vai entender o funcionamento dos algoritmos usando a **linguagem natural**, os **diagramas de blocos** (em algumas literaturas são conhecidos como **fluxograma**) e os **pseudocódigos**.

Linguagem natural

Segundo Santos (2001), a linguagem natural na definição geral é uma forma de comunicação entre as pessoas de diversas línguas, ela pode ser falada, escrita, gesticulada entre outras formas de comunicação. A linguagem natural tem uma grande contribuição quando vamos desenvolver uma aplicação computacional, pois pode direcionar de forma simples e eficiente as descrições dos problemas e suas soluções.

Para reforçar os conceitos de linguagem natural, você pode tomar como exemplo o cadastro de notas de alguns alunos do seu curso.

Vamos lá! O problema é o seguinte: o usuário deverá entrar com dois valores (as notas) e o computador retornar o resultado da média destes valores (média das notas).

Para realizar a solução desse problema, podemos fazer uso da seguinte estrutura:

1. Início.
2. Entrar com o primeiro valor.
3. Entrar com o segundo valor.
4. Realizar a soma do primeiro valor com o segundo.

5. Realizar a divisão do total dos valores por dois.
6. Armazenar o valor encontrado.
7. Mostrar na tela o resultado da média.
8. Fim.

Segundo Piva (2012), vale a pena citar o algoritmo Euclidiano, Euclides, usando de sua sabedoria, criou um algoritmo para calcular o máximo divisor comum, o famoso "mdc" no qual pode ser resumida da seguinte forma:

1. Dividir um número "a" por "b", onde o resto é representado por "r".
2. Substituir a por b.
3. Substituir b por r.
4. Continuar a divisão de a por b até que um não possa ser mais dividido, então "a" é considerado o mdc.

Veja na Tabela 1.1 a solução do "mdc" do algoritmo acima:

Tabela 1.1 | Calculo mdc (480, 130)

a	b	R
480	130	90
130	90	40
90	40	10
40	10	0
10	0	

Fonte: adaptada de Piva (2012).

Nesse caso, o resultado fica: $\text{mdc}(480,130) = 10$

Perceba que a linguagem natural é muito próxima da nossa linguagem.

Antes de iniciar a explicação sobre diagrama de blocos e pseudocódigo, vamos entender sucintamente o que são variáveis e atribuições. Vamos lá?

As **variáveis**, como o próprio nome sugere, é algo que pode sofrer variações, ou seja, estão relacionadas a identificação de uma

informação e **atribuição** (\leftarrow), que tem a função de indicar valores para as variáveis, ou seja, atribuir informação para variável. Por exemplo:

valor1 \leftarrow 12

nome \leftarrow marcio

Significa que a o número "12" está sendo atribuído para variável "valor1" e que o texto "marcio" está atribuído para variável "nome".

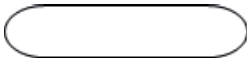

Muito bem, dando sequência ao seu estudo de algoritmos, veja agora o funcionamento dos diagramas de blocos que também pode ser descrito como fluxograma.



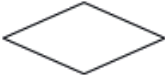
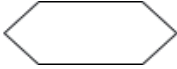



Diagrama de Blocos (Fluxograma)

Segundo Manzano (2015), podemos caracterizar diagrama de blocos como sendo um conjunto de símbolos gráficos, em que cada um desses símbolos representa ações específicas a serem executadas pelo computador. Vale lembrar que o diagrama de blocos determina a linha de raciocínio utilizada pelo programador para resolver problemas. Ao escrever um diagrama de blocos, o programador deve estar ciente que os símbolos utilizados devem estar em harmonia e de fácil entendimento. Para que os diagramas de blocos tenham certa coerência, os seus símbolos foram padronizados pela ANSI (Instituto Norte Americano de Padronização) conforme mostra o Quadro 1.1.

Veja agora a definição dos principais símbolos utilizados em um diagrama de blocos:

Quadro 1.1 | Descrição e significados de símbolos no diagrama de blocos

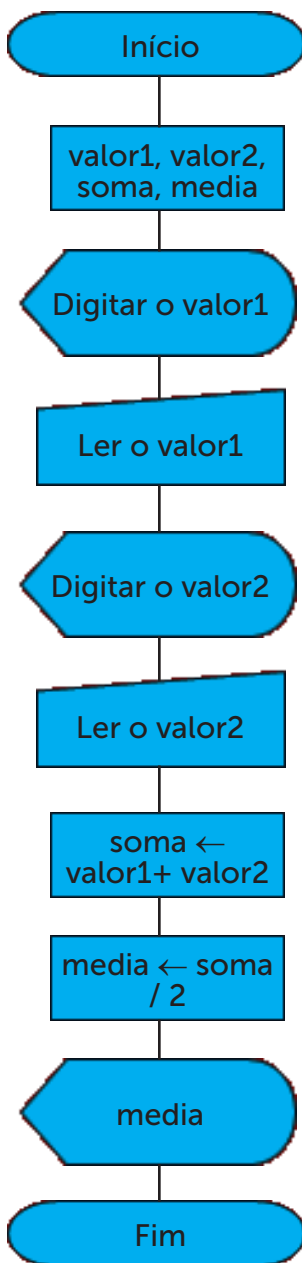
Símbolo	Significado	Descrição
	Terminal	Representa o início ou o fim de um fluxo lógico. Em alguns casos definem as sub-rotinas.
	Entrada Manual	Determina a entrada manual dos dados, geralmente através de um teclado.

Símbolo	Significado	Descrição
	Processamento	Representa a execução de ações de processamento.
	Exibição	Monstra o resultado de uma ação, geralmente através da tela de um computador.
	Decisão	Representa os desvios condicionais nas operações de tomada de decisão e laços condicionais para repetição de alguns trechos do programa.
	Preparação	Representa a execução de um laço incondicional que permite a modificação de instruções do laço.
	Processo Predefinido	Define um grupo de operações relacionadas a uma sub-rotina.
	Conector	Representa pontos de conexões entre trechos de programas, que podem ser apontados para outras partes do diagrama de bloco.
	Linha	Representa os vínculos existentes entre os símbolos de um diagrama de blocos.

Fonte: adaptado de Manzano (2015).

A partir do momento que você utilizar os símbolos com as suas instruções, você vai aprendendo e desenvolvendo cada vez mais a sua lógica em relação aos problemas. O exemplo do Diagrama 1.1 traz a solução de um algoritmo utilizando diagrama de blocos:

Diagrama 1.1 | Diagrama de blocos (Fluxograma)



Fonte: elaborado pelo autor.



Caro aluno, seguem algumas dicas para construir um diagrama de blocos (fluxograma):

- 1) Estar atento aos níveis.
- 2) O diagrama de blocos (fluxograma) deve começar de cima para baixo e da esquerda para direita.
- 3) Fique atento para não cruzar as linhas, principalmente as linhas de fluxos de dados.

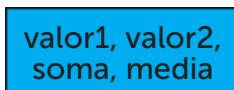
O diagrama de blocos apresentado acima mostra claramente a execução da média de dois valores.

Vejam os símbolos de cada passo do diagrama:

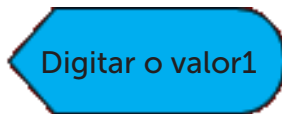
O símbolo terminal deu início ao diagrama de blocos.



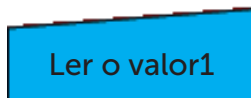
O símbolo de processamento definiu as variáveis.



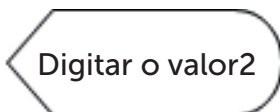
O símbolo exibição, mostra na tela o que o usuário deve fazer.



O símbolo de entrada manual, libera para o usuário entrar com o primeiro valor.



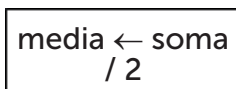
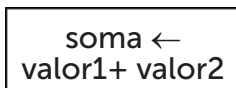
O símbolo exibição, mostra na tela o que o usuário deve fazer.



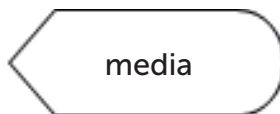
O símbolo de entrada manual, libera para o usuário entrar com o segundo valor.



O símbolo de processamento, é realizado as atribuições dos valores calculados para suas respectivas variáveis.



O símbolo de exibição, mostra na tela o resultado de cada valor calculado.



Finaliza o programa.



Pesquise mais

Hoje podemos contar com a ajuda de softwares específicos para construção de diagrama de blocos (fluxogramas), entre eles, você pode usar o Lucidchart que é um gerador de fluxograma online e

gratuito. Disponível em: <<https://www.lucidchart.com/pages/pt>>. Acesso em: 17 mar. 2018.

Outro software muito utilizado é o "Dia". Você pode fazer o download pelo link: disponível em <<http://dia-installer.de/>>. Acesso em: 17 mar. 2018.

Pseudocódigo

Segundo Aguilar (2011), o pseudocódigo é considerado uma ferramenta que pode auxiliar a programação, ela pode ser escrita em palavras similares ao inglês ou ao português para facilitar a interpretação e desenvolvimento de um programa.

Na programação, os algoritmos também podem ser caracterizados pelos pseudocódigos, a intenção do pseudocódigo é chegar na solução de um problema.



Refleta

É importante estar atento para algumas regras básicas quando utilizar pseudocódigos:

- Escolher um nome.
- Avaliar as variáveis, dar atenção aos seus tipos e características.
- Descrever de forma clara o que será armazenado e se as variáveis destinadas a essa informação estão corretas.
- Verificar se as instruções fazem sentido e se as mesmas têm uma sequência lógica.
- Avaliar o resultado e quando pertinente, mostre-o na tela.
- Finalizar o algoritmo.

O uso de pseudocódigo pode ser aplicado para qualquer linguagem de programação?

Recordando o exemplo acima que calcula a média dos alunos da sua turma, veja como fica em pseudocódigo:

- 1) calculo_media;
- 2) var
- 3) real: valor1, valor2, soma, media;
- 4) Início
- 5) escreva ("Digite o valor 1");
- 6) leia valor1;
- 7) escreva ("Digite valor 2");
- 8) leia valor2;
- 9) soma ← valor1 + valor2;
- 10) media ← soma/2;
- 11) escreva("A media do aluno e: "; media);
- 12) Fim.

Veja agora os comentários deste algoritmo:

Linha 1: "calculo_media" esse é o nome reservado para identificar o algoritmo.

Linha 2: "Var", indica a declaração das variáveis.

Linha 3: São os nomes dados para as variáveis (valor1, valor2, soma, media), nesta linha também é definida os tipos de variáveis ("real", veremos com maior detalhe na próxima unidade do livro).

Linha 4: inicia os procedimentos dos algoritmos(início).

Linha 5: "escreva" é um comando de saída, este comando indica o que vai sair na tela do computador, geralmente o conteúdo do texto a ser mostrado fica entre aspas ("Digite valor 1").

Linha 6: "leia" é comando de entrada, o valor digitado é armazenado na variável (valor1)

Linha 7: "escreva" é um comando de saída, este comando indica o que vai sair na tela do computador, geralmente o conteúdo do texto a ser mostrado fica entre aspas ("Digite valor 2").

Linha 8: "leia" é comando de entrada, o valor digitado é armazenado na variável (valor2)

Linha 9: A adição das variáveis valor1 e valor2 é atribuído para variável soma ($soma \leftarrow valor1 + valor2$);

Linha 10: Realiza o calcula da média e atribui o valor encontrado na variável media.

```
(media  $\leftarrow$  soma/2);
```

Linha 11: Escreve na tela o que está entre aspas. Escreva ("A media do aluno e: "; media). Perceba que a variável é colocada fora das aspas, para que a mesma seja representada pelo seu valor correspondente.

Linha 12: Encerra o algoritmo com a palavra "fim" e o ponto final.

Lembre-se, quando você escreve um algoritmo do tipo portugol, que é um pseudocódigo, é preciso escrever de forma clara para que as pessoas possam interpretar e futuramente, possam passar para uma linguagem de programação.



Exemplificando

Veja abaixo um algoritmo escrito em pseudocódigo e executado em Visualg:

```
algoritmo "media"
```

```
var
```

```
    valor1, valor2, soma, media: real
```

```
inicio
```

```
    Escreval("Digite o valor da nota 1: ")
```

```
    Leia (valor1)
```

```
Escreval("Digite o valor da nota 2: ")
```

```
Leia (valor2)
```

```
soma <- (valor1 + valor2)
```

```
media <- (soma / 2 )
```

```
Escreval("A media do aluno e:" media)
```

```
fimalgoritmo
```

Perceba que os parâmetros utilizados também são considerados um algoritmo do tipo português estruturado, ou seja, de fácil entendimento e interpretação.

O software do Visualg é gratuito e o seu download está disponível em: <http://visualg3.com.br/>. Acesso em: 18 mar. 2018.

Após os estudos de algoritmos e as suas formas de construções, Manzano (2015) coloca em destaque os paradigmas de programação, que são caracterizados pelos paradigmas de **programação estruturada**, em que o algoritmo é construído como sequência linear de funções ou módulo. O outro paradigma é **orientado a objetos** ou simplesmente Programação Orientada a Objetos, onde o programador abstrai um programa como uma coleção de objetos que interagem entre si.

Alternativas e inovações aos algoritmos

Transferir nossos conhecimentos para os algoritmos pode trazer uma série de benefícios para a humanidade, veja você, que os algoritmos são as bases para criação de um programa de computador, em que diversas aplicações poderão ocorrer.

Então, fica claro que um algoritmo bem estruturado vai gerar um programa para solução de um problema que antes, parecia complexo. Vamos agora citar algumas alternativas e inovações que podem ser criadas a partir de algoritmos:

- Algoritmos para tomada de decisão: podemos utilizar o exemplo de uma montadora de veículos que precisa decidir a quantidade de veículos em diferentes regiões do país, sabendo que cada região tem suas preferências por modelo, cor, motorização entre outras características. O algoritmo, neste caso, analisa os dados e toma a decisão mais acertada, quanto ao que produzir e onde produzir.
- Canal de empregabilidade: os algoritmos são realizados para analisar todas as informações dos candidatos de acordo com a vaga ofertada.
- Análise financeira: algoritmos responsáveis por analisar o mercado e tomar a melhor decisão de aplicações e investimentos.
- Algoritmos voltados para a medicina: grandes inovações na medicina já estão em execução devido aos algoritmos, rotinas padrões, até mesmo as mais complexas, como cirurgias, estudo de células, genéticas, entre outras tantas especializações.

Não tem como negar: todas as áreas estão voltadas para a tecnologia e são através de diversas formas de pensamentos que os algoritmos são realizados.

Muito bem, prepara-se, você com certeza poderá ser um grande entusiasta em algoritmos.

Boa sorte e até a próxima seção!

Sem medo de errar

Caro aluno, chegou o momento de colocar em prática todo o conhecimento adquirido nesta seção. Lembrando que você foi incumbido de treinar os estagiários contratados pela empresa de desenvolvimento de softwares educacionais. Cabe, ainda, lembrar que os estagiários não possuem nenhuma experiência em algoritmos.

Para tal, como primeira proposta, você deverá apresentar aos estagiários um algoritmo utilizando a linguagem natural, os diagramas de blocos (fluxogramas) e os pseudocódigos para cadastrar os dados pessoais do aluno (nome, endereço, cidade e estado) e mostrar na tela do computador o resultado do cadastro.

Seguem as dicas para realizar a solução da situação problema.

Quando falamos em linguagem natural, quer dizer que devemos escrever a situação o mais próximo possível da linguagem convencional. Não se preocupe com os passos a serem realizados, foque na solução do problema.

Para realização do diagrama de blocos (fluxograma), concentre-se na descrição e significados dos símbolos apresentados no Quadro 1.1.

Enfim, no pseudocódigo, faça uso das seguintes dicas:

- Escolher um nome.
- Avaliar as variáveis, dar atenção aos seus tipos e características.
- Descrever de forma clara o que será armazenado e se as variáveis destinadas a essa informação estão corretas.
- Verificar se as instruções fazem sentido e se as mesmas têm uma sequência lógica.
- Avaliar o resultado e mostrar na tela.
- Finalizar o algoritmo.

Muito bem, são várias as formas que você pode utilizar para a conclusão de um algoritmo, seguindo este pensamento, resolva o seu algoritmo e se possível elabore outras formas de solução.

Avançando na prática

A troca

Descrição da situação-problema

Um programador foi incumbido de realizar um algoritmo para coletar a quantidade de mulheres e de homens em um determinado evento, porém, algo deu errado. A variável que era para receber o

número de mulheres acabou recebendo o número de homens e vice-versa. Agora, você precisa ajustar rapidamente esses valores. Qual a forma mais adequada para realizar essa troca utilizando um algoritmo em pseudocódigo?

Resolução da situação-problema

- 1) inverter valores;
- 2) var
- 3) real: mulheres, homens, troca;
- 4) Início
- 5) escreva ("Acertar valores");
- 6) escreva("Digite a quantidade de mulheres");
- 7) leia homens; // Opa! Cometeu-se um erro: número de
mulheres foi para homens.
- 8) escreva ("Digite a quantidade de homens");
- 9) leia mulheres; // Opa cometeu-se um erro: número
de homens foi para mulheres.
- 10) troca ← mulheres;
- 11) mulheres ← homens;
- 12) homens ← troca;
- 13) Escreva("A quantidade correta de mulheres
é:", mulheres);
- 14) Escreva("A quantidade correta de homens é:", homens);
- 15) Fim.

Deixo como desafio realizar o diagrama de blocos e a linguagem natural do algoritmo acima.

Boa sorte!

Faça valer a pena

1. Existe um tipo de linguagem na qual a comunicação entre as pessoas de diversas línguas, pode ser falada, escrita ou gesticulada, possui uma grande contribuição quando vamos desenvolver uma aplicação computacional, pois pode direcionar de forma simples e eficiente as descrições dos problemas e suas soluções.

A alternativa correta para definição do texto acima é:

- a) Linguagem natural.
- b) Fluxograma.
- c) Pseudocódigo.
- d) Diagrama de blocos.
- e) Linguagem textual.

2. Segundo Aguilar (2011), o pseudocódigo é considerado uma ferramenta que pode auxiliar na programação, ela pode ser escrita em palavras similares ao inglês ou português para facilitar a interpretação e desenvolvimento de um programa.

Analise o algoritmo abaixo onde calcula o bônus de 30% para os funcionários cadastrados, complete a linha de programação que está faltando:

- 1. inicio
- 2. caracter: nome;
- 3. _____
- 4. escreva ("Entre com o nome do funcionário");
- 5. leia (nome);
- 6. escreva ("Entre com o salario Bruto do Funcionario");
- 7. leia (sal);
- 8. _____
- 9. totalsal sal+bonus;
- 10. escreva ("Seu salario com bonus será de:."; totalsal);
- 11. fim.

Assinale a alternativa correta:

- a) real: sal, bonus;
bonus ← sal*(30/100);

- b) real: sal, bonus, totsal;
sal ← bonus*(30/100);
- c) real: sal, bonus, totsal;
bonus ← sal*(30/100);
- d) real: sal, bonus, totsal;
- e) bonus ← sal*(30/100);)

3. Segundo Manzano (2015), podemos caracterizar diagrama de blocos como sendo um conjunto de símbolos gráficos, no qual cada um desses símbolos representa ações específicas a serem executadas pelo computador. Vale lembrar que o diagrama de blocos determina a linha de raciocínio utilizada pelo programador para resolver problemas.

Enunciado:

Para que os diagramas de blocos tenham certa coerência, os seus símbolos foram padronizados pela ANSI (Instituto Norte Americano de Padronização). Analise os símbolos abaixo:

I –



O símbolo significa um terminal, onde: representa o início ou o fim de um fluxo lógico. Em alguns casos definem as sub-rotinas.

II-



Quadro 1.2

O símbolo significa exibição, onde: Mostra o resultado de uma ação, geralmente através da tela de um computador.

III-



O símbolo significa entrada manual, em que: determina a entrada manual dos dados, geralmente através de um teclado.

Assinale a alternativa correta:

- a) Somente a afirmação I está correta.

- b) As afirmações I, II e III estão corretas.
- c) As afirmações I, II estão corretas.
- d) As afirmações II e III estão corretas.
- e) As afirmações I, III estão corretas.

Seção 1.2

Conceitos Básicos de Linguagens de Programação

Diálogo aberto

Caro aluno, na primeira seção desta unidade, você teve a oportunidade de compreender as definições de algoritmos, suas aplicações e tipos. Agora, vamos conhecer as definições de linguagem de programação, suas famílias, aplicações e mostrar as oportunidades dos profissionais de programação.

Vamos lá! Ninguém atinge excelência em alguma coisa se não for por meio do treinamento e persistência, certo?

Pois bem, agora chegou o momento de levar os seus estagiários contratados pela empresa de softwares educacionais para o próximo nível de conhecimento. Imagine que você precisa consertar um armário em sua casa (ou fazer um armário novo, você mesmo). Para isso, você vai a uma loja de casa e construção e passa por vários departamentos: materiais, ferramentas, máquinas para alvenaria, máquinas para bricolagem e tantos outros. Obviamente você vai procurar materiais, ferramentas e máquinas para marcenaria, não é verdade? Com as linguagens de programação também é assim: para cada tarefa, para cada objetivo a ser atingido, há uma linguagem adequada e um conjunto de ferramentas recomendado.

Sendo assim, você apresentará aos estagiários as principais famílias de linguagem de programação que podem ser utilizadas na área educacional e também, a oportunidade de crescimento profissional que eles poderão ter dentro da empresa.

Promova em seus estagiários a criatividade, apresente a eles um relatório dos principais programas que podem ser utilizados para criação de softwares educacionais.

Com o conhecimento adquirido, você acredita que os estagiários terão oportunidade de crescimento profissional conhecendo as linguagens de programação?

Supere-se e bons estudos!

Não pode faltar

Caro aluno, após compreendermos os conceitos, aplicações e tipos de algoritmos, chegou o momento de entender a importância da linguagem de programação e das suas famílias, assim como, as projeções profissionais que a carreira de programador pode proporcionar.

Segundo Marçula (2013, p. 170),

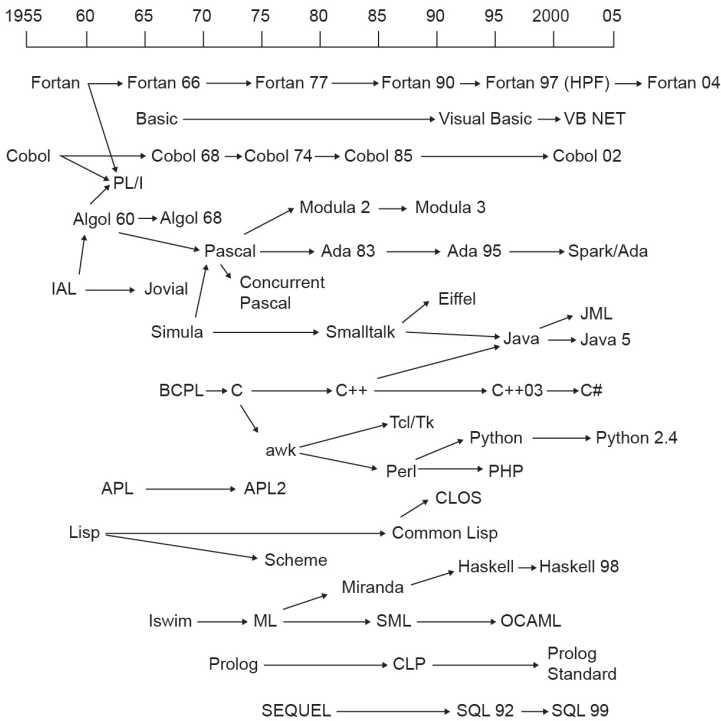
a linguagem de programação pode ser entendida como um conjunto de palavras (vocabulário) e um conjunto de regras gramaticais (para relacionar essas palavras) usados para instruir o sistema de computação a realizar tarefas específicas e com isso, criar os programas. Cada linguagem tem o seu conjunto de palavras-chave e sintaxes.



Para Tucker (2010), da mesma forma que entendemos as linguagens naturais, essas utilizadas por todos no dia a dia, a linguagem de programação é a comunicação de ideias entre o computador e as pessoas. Ainda segundo o autor, as primeiras linguagens de computadores utilizadas foram as linguagens de máquina e a linguagem *assembly*. Esse cenário foi criado a partir da década de 1940. Desde então, muitas linguagens foram aparecendo e, como é natural, muitas não sobreviveram, como o caso da linguagem Algol.

A Figura 1.1 mostra a linha do tempo das linguagens que tiveram um valor significativo na história da computação.

Figura 1.1 | Resumo da história das linguagens de programação



Fonte: Tucker (2010, p. 7).

Vejamos agora algumas linguagens de programação consideradas de baixo nível que fizeram parte da história da computação e que, com certeza, tiveram as suas contribuições para o desenvolvimento de novas linguagens:

- **Linguagem de máquina:** pode ser escrita dentro de um editor hexadecimal, tal linguagem é baseada no sistema binário, segundo Monteiro (2010), uma instrução de máquina é considerada conjunto de bits (0s e 1s), ou seja, um programa em linguagem de máquina é uma ampla sequência de algarismos binários.
- **Linguagem Assembly:** Segundo Tucker (2010), podemos dizer que essa linguagem é uma forma mais viável de conseguir ler a linguagem de máquina, lembrando que é uma linguagem voltada para máquina, ou seja, trabalha diretamente nas memórias e registros do computador, elas são executadas diretamente pelo processador.



Segundo Sebesta (2010), os primeiros computadores digitais apareceram nos anos 1940, naquele momento, os computadores eram utilizados para uso científico. Naquela ocasião utilizavam muitas aplicações matemáticas (vetores e matrizes) fazendo uso da linguagem Assembly, porém, a linguagem Fortran foi a primeira linguagem científica aplicada para suprir a demanda. A linguagem Fortran ainda se depara com muitos entusiastas da programação, sendo utilizada até hoje.

Segundo Tucker (2010), na década de 1950, as linguagens de programação tiveram seus marcos caracterizadas pelas "linguagens de ordem mais alta" (HOLs), nas quais se diferenciavam das linguagens de máquina ou assembly. A forma de programar era independente das arquiteturas de alguns computadores. Entre as linguagens de ordem mais alta podemos destacar:

- **Linguagem Fortran:** segundo Marçulo (2013), foi reconhecida como sendo a primeira linguagem de programação, ou seja, os seus comandos não são escritos em linguagem de máquina. Ideal para aplicações matemáticas, porém, não era muito usada para elaboração de softwares de tempo real e embarcado.
- **Linguagem Basic:** segundo Marçulo (2013), linguagem muito usada para criação de programas para computadores pessoais, foi muito utilizada para ensinar programação.
- **Linguagem COBOL:** segundo Marçulo (2013), foi a primeira linguagem para criar aplicações comerciais, a programação na linguagem COBOL ainda continua em atuação, principalmente em empresas que utilizam computadores de grande porte.
- **Linguagem Algol:** segundo Marçulo (2013), usada para aplicações científicas. Foi responsável por anunciar a linguagem de terceiras gerações, possui uma estrutura algorítmica.
- **Linguagem Lisp:** usada para fundamentos matemáticos, muito utilizada na área de inteligência artificial. Segundo Tucker (2010), foi projetada para ser uma linguagem de programação funcional,

tal linguagem dispensava o uso de laços, ou seja, utilizava de funções recursivas.

- **Linguagem Pascal:** segundo Sebesta (2011), a linguagem Pascal foi projetada como uma linguagem de ensino e aplicações, ele combinava uma linguagem simples e expressiva.
- **Linguagem C:** inicialmente foi utilizada para programação de sistemas operacionais, ela também era adequada para outras aplicações, coloca Sebesta (2011). O sistema operacional Unix é implementado em C.
- **Linguagem Prolog:** segundo Sebesta (2011), linguagem que usava notação lógica, na qual era realizada a comunicação de processos computacionais para o computador. Vale salientar que o Prolog não é procedural, ou seja, os programas não exprimem exatamente como um resultado deve ser computado, mas descrevem a forma necessária e/ou as características dele. Linguagem utilizada em Inteligência Artificial (IA).
- **Linguagem C++:** derivada da linguagem C, segundo Sebesta (2011), uma série de modificações foram realizadas para melhorar seus recursos imperativos e também teve o seu diferencial que era dar suporte à programação orientada a objetos.

Dando sequência às definições das famílias de linguagem de programação, seguem algumas contribuições da autora Dornelles (2017):

- **Linguagem Java:** teve seu início na empresa Sun Microsystems (Hoje, a Linguagem Java pertence à Oracle). No início, a ideia era criar programas para comunicação entre diferentes dispositivos, como era o caso dos eletrodomésticos (vídeo cassete, TV, conversores de televisão a cabo entre outros). Hoje, a linguagem Java é implementada em sistemas operacionais, computadores de grande porte e até dispositivos móveis. Java é uma linguagem orientada a objetos.
- **Linguagem C# (lê-se "c sharp"):** criada pela Microsoft em 2002, possui plataforma dot net (.NET) para melhorar a comunicação de tecnologias na empresa. Trata-se de uma linguagem orientada

a objeto e seu grande diferencial é de ser fácil a compreensão pelos programadores.

- **Linguagem Python:** criada pela Python Software Foundation na década de 90. Considerada uma linguagem de alto nível, suportando diversos paradigmas e por ter recursos muito poderosos. Além de tudo, a linguagem Python possui uma sintaxe de fácil entendimento, facilitando assim, a escrita dos códigos.
- **Linguagem JavaScript:** utilizada para programação em navegadores de internet. Só para deixar claro, JavaScript, não tem nada a ver com Java. O JavaScript trabalha com as estruturas HTML e modifica os estilos CSS, ou seja, traz grandes vantagens no desenvolvimento de páginas web.
- **Linguagem Perl:** usada para aplicações web, fácil de trabalhar e bem parecida com outras linguagens. Atualmente está caindo em desuso.
- **Linguagem PHP:** usada para criação de sistemas Web dinâmico, ou seja, seus códigos podem ser interpretados em um servidor. É uma linguagem interpretativa.
- **Linguagem Ruby:** simples e orientada a objetos, utilizada em programações para web. Vale a pena colocar o seu slogan "O melhor amigo do programador". Trata-se de uma linguagem de interpretação.
- **Linguagem Google GO:** como o próprio nome sugere, é uma linguagem criada pela Google, seu foco é o aumento de produtividade em projetos. Um dos grandes destaques da linguagem é ser multiplataforma, suportando Windows, MacOS, Linux entre outros. Detalhe importante: é código aberto.
- **Linguagem Swift:** criada pela Apple, desenvolve programas para as plataformas da marca, como: iOS, Apple Watch, Mac OS e Apple TV. Possui uma sintaxe bem simples e também é open source.
- **Linguagem Visual Basic (VB):** originalizada da Linguagem Basic, tornou-se uma linguagem orientada a objetos o que foi uma grande evolução, pois dessa forma poderia oferecer ao programador uma melhor interface gráfica. Importante salientar que em 2002 passou

a fazer parte da plataforma dot net (.NET) da Microsoft. No quesito produtividade se tornou uma linguagem bem poderosa.

- **Linguagem de Programação R:** utilizada para desenvolvimento de computação estatística, ou seja, sistemas para análise de dados, construções de gráficos entre outros.
- **Linguagem Objective-C:** mais uma linguagem que pertence a Apple, usada para aplicações do iOS. Uma linguagem com uma boa utilização no mercado por ser uma linguagem que possibilita a reutilização de códigos.
- **Linguagem Object Pascal:** vem da origem do Pascal, é uma linguagem orientada a objetos, suas principais ferramentas de programação estão ligadas a IDE Delphi da Embarcadero.

Veja abaixo o ranking das linguagens de programação divulgado por Vidal (2017). O ranking foi realizado pela empresa Tiobe Index, especializada em assessoria e qualidade de software.

1. Java – 13.774 %
2. C – 7.321 %
3. C++ – 5.576 %
4. Python – 3.543 %
5. C# – 3.518%
6. PHP -3,093%
7. Visual Basic .Net – 3,05%
8. JavaScript – 2,60%
9. Delphi/Object Pascal – 2,49%
10. Go – 2,36%

Vamos, agora, falar um pouco sobre os paradigmas das linguagens de programação, a proposta aqui não é aprofundar nas aplicações e sim nos seus conceitos. Vejamos algumas definições:

De acordo com Houaiss (2001, p. 329), "paradigma significa modelo, padrão. No contexto da programação de computadores, um paradigma é um jeito, uma maneira, um estilo de se programar". Segundo Tucker (2010), um paradigma de programação está relacionado a um padrão de soluções de problemas, no qual por sua vez estão relacionados a uma determinada linguagem de programação. Segundo Tucker (2010), quatro paradigmas de programação tiveram sua evolução reconhecida nas últimas três décadas:

Programação imperativa: considerada o paradigma mais antigo, pode armazenar o programa e suas variáveis juntamente, assim como a abstração procedural, as atribuições, as sequências, os laços, os comandos condicionais e a manipulação de exceções são seus componentes de programação.

Exemplo de programação imperativa: COBOL, Fortran, C, Ada e Perl.

Programação Orientada a Objeto: também conhecida na computação como (POO), como o próprio nome sugere, é considerado uma coleção de objetos onde se inter-relacionam, facilitando assim a programação. São exemplos de POO: vSmalltalk, C++, Java e C#.

Programação Funcional: caracterizada por possuir atuação matemática, cada uma com um espaço de entrada (domínio) e resultado (faixa). Exemplos de programação funcional: Lisp, Scheme, Haskell e ML.

Programação Lógica: considerada uma programação declarativa, na qual um programa pode modelar uma situação problema declarando qual resultado o programa deve obter, em vez de como ele deve ser obtido. Podemos citar como exemplo de programação lógica o Prolog.

Todas as linguagens de programação para criação de um programa possuem uma **sintaxe**, que nada mais é a forma de como o programa é escrito, podemos ainda, definir sintaxe de acordo com Tucker (2010), "[a] sintaxe de uma linguagem de

programação é uma descrição precisa de todos os seus programas gramaticalmente corretos”.



Refleta

Muitas linguagens de programação são orientadas a objetos, o que por sua vez agiliza o processo de criação e proporciona aos programadores uma certa flexibilidade para criação e modificações dos programas, podendo assim, atender os seus clientes com maior agilidade. Na sua visão a POO, substitui totalmente as programações estruturadas?

Pois bem, você já viu o quanto é importante conhecer as linguagens de programação e suas aplicações. Agora, vai conhecer como funciona a execução de um programa de computador.

Segundo Manzano (2015), os algoritmos criados por você deverão ser convertidos em linguagens de alto nível, como, por exemplo: a linguagem Java, Pascal, C, C++ entre outras conceituadas nesta seção. Você deverá escrever os códigos fontes na linguagem escolhida para que os mesmos se tornem executáveis, porém, cada linguagem adota um método particular para gerar o código executável.

Segundo Manzano (2015), existem três métodos para gerar um código executável:

1- Compiladores: é gerado um código executável sem a necessidade de interpretar comando por comando.

Tucker (2010) afirma que podemos definir em cinco passos o processo de compilação:

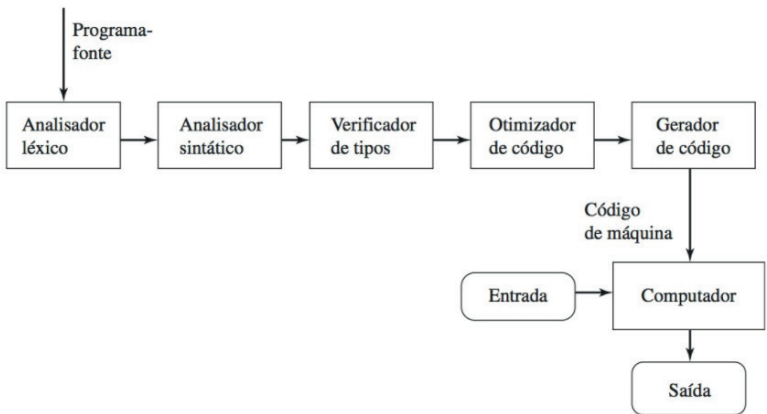
- **Análise léxica:** processo que analisa os caracteres de entrada do programa, ou seja, o código fonte.
- **Análise sintática:** processo que analisa os textos de um programa por meio do teclado, também é responsável por transformar o texto em uma estrutura de dados.
- **Verificação de tipos:** verifica se as instruções possuem lógica na linguagem, se possuem consistência das declarações, se o uso

dos identificadores está aplicado corretamente e, por fim, realiza as conversões onde dão sentido a uma sentença.

- Otimização de código: é feita a análise do código de forma intermediária, a fim de melhorar o código e deixá-lo mais rápido, entre suas atribuições, uma é verificar as repetições e redundâncias de um bloco de programa.
- Geração de código: é a fase final, onde depois de otimizado o código, é gerado um código definitivo.

Na Figura 1.2, podemos observar que a análise léxica, sintática e verificação de tipos estão relacionadas a identificar erros do programa e a otimização e geração de código estão relacionados para serem executados no computador.

Figura 1.2 | Processo de compilação e execução



Fonte: Tucker (2010, p. 18).

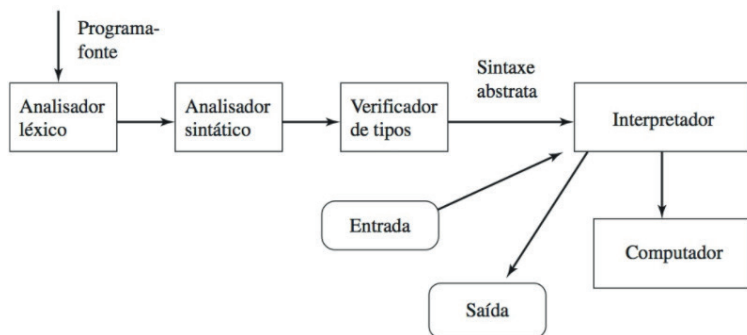
Sua vantagem é a execução mais rápida, porém, caso necessite de alterações, o código fonte deverá ser editado e executado em máquinas de mesma arquitetura.

Exemplo de linguagens que utilizam compiladores: Algol, C, C++, C#, Pascal, etc.

2- Interpretadores: lê uma instrução do código fonte, transforma em código binário e executa, repete esse processo até que todas as instruções do código fonte sejam executadas. O código é interpretado e executado em tempo real, sem a geração de código executável.

A Figura 1.3 mostra que como nos compiladores a análise léxica, sintática e verificação de tipos, também acontecem no interpretador. Segundo Tucker (2010), "a representação abstrata do programa que sai desses três passos se torna o objeto de execução de um interpretador".

Figura 1.3 | Interpretadores



Fonte: Tucker (2010, p. 19).

Uma das grandes vantagens desse método são as correções e alterações que podem ser realizadas rapidamente, proporcionando assim, um baixo consumo de memória.

Podemos dizer que a desvantagem está no fato de interpretar comando por comando deixando assim, a execução mais lenta.

Exemplos de linguagens que utiliza interpretadores: HTML, Javascript, Python, Basic, PHP, etc.

3- Tradutores: são gerados códigos intermediários, os quais não exigem tanto espaço de memórias quanto realizado pelo código fonte.

Sua vantagem é ter independência da arquitetura que realizará a execução final, porém necessita de um interpretador específico para sua geração. Exemplo de linguagem que utiliza tradutores: java.



Exemplificando

Podemos exemplificar interpretadores utilizando sistemas desenvolvidos para web, eles proporcionam uma melhor manipulação, facilitando assim, a manutenção. Um desconforto em relação aos interpretadores é quando tem a troca de navegadores ou atualizações, os programas podem sofrer alterações de layout.

Legal! Agora, você vai conhecer onde os profissionais da área de programação podem atuar.

Lembrando que o programador é responsável por desenvolver softwares e também realizar testes de códigos.

Como você pode notar, existem várias linguagens de programação, cada uma com um segmento em específico. Giovanelli (2017) caracteriza algumas categorias de profissionais voltado para área de programação:

- 1- Programador Desktop:** profissional que trabalha com sistemas internos, nos quais são relacionados a um computador ou a uma rede de uso específico de uma empresa. Podem ser desenvolvedores para ambientes: MAC, Linux ou Windows.
- 2- Programador Web:** profissional voltado para programação web.
- 3- Programador Mobile:** atua em uma das áreas que mais crescem no momento, no que concerne programação.
- 4- Programador de Jogos:** desenvolve criatividade na área de entretenimento, desenvolvendo as lógicas, programando sistemas para plataformas digitais, como dispositivos mobile, vídeos games e desktops.

Por se tratar de algumas das profissões do futuro, o profissional da área de programação não encontrará dificuldade de colocação

no mercado, porém um programador precisa ter em mente que as linguagens evoluem e que será preciso estar sempre atualizado, realizar certificações, estudar línguas e sempre buscar novos caminhos na sua área de atuação.



Pesquise mais

Para se tornar um(a) programador(a) reconhecido pelo mercado de trabalho o profissional precisa estar atento as inovações dos softwares de programação e, para isso, é preciso desenvolver pelos menos conhecimento em duas ou mais linguagens, ser organizado nas manipulações das informações, tranquilidade e agilidade para resolver problemas. Veja alguns sites especializados em recrutamento, que trazem a média salarial para os profissionais de programação de computadores.

CATHO. Guia de Profissões e Salários. 2018. Disponível em: <<https://www.catho.com.br/profissoes/programador/>>. Acesso em: 3 mar. 2018.

VAGAS.COM. Vagas de emprego para salário programador. Disponível em: <<https://www.vagas.com.br/vagas-de-salario-programador?v10m%5B%5D=2016-01-31>>. Acesso em: 3 mar. 2018.

Muito bem, uma seção recheada de informação. Agora, você terá a oportunidade de praticar os conhecimentos adquiridos. Boa sorte e ótimos estudos!

Sem medo de errar

Chegou o momento de apresentar para os seus estagiários as melhores linguagens de programação para o desenvolvimento dos softwares educacionais. Lembre-se de que os seus estagiários não possuem conhecimento suficiente no assunto, ao final, você deverá relatar como o mercado de trabalho vem se comportando diante dessa demanda de profissionais.

Para solucionar essa situação-problema coloque para os seus estagiários as seguintes questões:

O software funcionará via web ou somente dentro da empresa, em uma rede local?

Caro aluno, para solucionar essa questão, você poderá destacar as principais novidades de linguagem de programação, analise o custo e a plataforma que irá funcionar.

Veja algumas linguagens que estão em alta no mercado da programação web:

Java, no que diz respeito a programação web, é muito respeitada, pode ser utilizada tanto em ambientes de sites como em aplicativos para celulares.

Uma linguagem que vem sendo enfatizada é a Python, a nível de custo é uma linguagem de fácil acessibilidade e ótima para gerar mineração de dados, também se destaca em programação web.

JavaScript, utilizada para programação de sites, é possível realizar grande programações relacionadas para internet das coisas.

Agora, se você quiser se dedicar a uma linguagem que suporta a programação de jogos de computadores, aplicativos, firmware, entre outras tantas aplicações, você pode optar pela linguagem C++ e C.

Lembrando que PHP é uma linguagem que trabalha muito bem com HTML, no qual são utilizadas várias aplicações para web.

Se for local, analise as linguagens de melhor desempenho, todas as citadas acima são ótimas linguagem locais e com interface para internet. Vale lembrar que você dever estar atento às linguagens que fazem comunicação com banco de dados e com as redes de computadores.

Detalhe o máximo possível, compare as linguagens. Lembre-se, neste momento, de que você estará reconhecendo as principais linguagens de programação do mercado.

Após mostrar as famílias de linguagem de programação, faça visitas em sites especializados em empregabilidade e pesquise as melhores oportunidades na área de programação. Você verá que

são muitas, entre elas podemos destacar o box do *Pesquise Mais* do livro, no qual relata a média salarial para os profissionais de programação de computadores.

Estude bastante e sucesso na sua trajetória!

Avançando na prática

Alavancando as vendas

Descrição da situação-problema

Uma pequena empresa no segmento de joias percebeu que o mercado está reaquecido para este negócio, então o proprietário resolveu investir em tecnologia. Em um bate papo com um amigo ficou sabendo que poderia aumentar as suas vendas pela internet, e você foi apontado como o profissional ideal para indicar as melhores linguagens de programação. Pois bem, você já conhece muitas linguagens de programação, e também sabe que algumas delas são específicas em certos equipamentos.

Após sua análise, quais as melhores linguagens de programação para o desenvolvimento web?

Resolução da situação-problema

Existem várias soluções para o desenvolvimento web, uma delas é escolher as linguagens que poderão atender os tipos de equipamentos, isto é, trata-se de uma plataforma Windows, Linux ou iOS? Muito bem, veja abaixo algumas opções que você poderia estar indicando.

- **Linguagem JavaScript:** utilizada para programação em navegadores de internet. Só para deixar claro, JavaScript, não tem nada a ver com Java, trabalha com as estruturas HTML e modifica os estilos CSS, ou seja, traz grandes vantagens no desenvolvimento de páginas web.

- **Linguagem Perl:** usada para aplicações web, fácil de trabalhar e bem parecida com outras linguagens, atualmente está caindo em desuso.
- **Linguagem PHP:** usada para criação de sistemas Web dinâmico, ou seja, seus códigos podem ser interpretados em um servidor. É uma linguagem interpretativa.
- **Linguagem Ruby:** mais uma linguagem de programação simples e orientada a objetos, utilizada em programações para web. Vale a pena colocar o seu slogan "O melhor amigo do programador". Trata-se de uma linguagem de interpretação.
- **Linguagem Google GO:** criada pelo Google, seu foco é o aumento de produtividade em projetos. Um dos grandes destaques da linguagem é ser multiplataforma, suportando Windows, MacOS, Linux entre outros. Detalhe importante: é código aberto.
- **Linguagem Swift:** criada pela Apple, ou seja, desenvolve programas para as plataformas da marca, como: iOS, Apple Watch, Mac OS e Apple TV. Possui uma sintaxe bem simples e também é open source.
- **Linguagem Objective-C:** pertencente à Apple, usada para aplicações do iOS. Uma linguagem com uma boa utilização no mercado por ser uma linguagem que possibilita a reutilização de códigos.

Existem outras tantas linguagens para esse desenvolvimento, faça uma nova pesquisa e relate aos seus amigos, provocando um diálogo sobre o assunto.

Faça valer a pena

1. Segundo Marçula (2013, p. 170),

A linguagem de programação pode ser entendida como um conjunto de palavras (vocabulário) e um conjunto de regras gramaticais (para relacionar essas palavras) usados para instruir o sistema de computação a realizar tarefas específicas e com isso, criar os programas. Cada linguagem tem o seu conjunto de palavras-chave e sintaxes.



Assinale a alternativa que corresponde às linguagens de baixo nível:

- a) Linguagem Java, Linguagem Assembly.
- b) Linguagem Pascal, Linguagem de Máquina.
- c) Linguagem Máquina, Linguagem Assembly.
- d) Linguagem Máquina, Linguagem Fortran.
- e) Linguagem Java, Linguagem Fortran.

2. Segundo Manzano (2015), os algoritmos criados por você deverão ser convertidos em linguagens de alto nível. Você deverá escrever os códigos fontes na linguagem escolhida para que os mesmos se tornem executáveis, porém cada linguagem adota um método particular para gerar o código executável.

Manzano (2015) aponta existem três métodos para gerar um código executável, relacione-os de forma correta:

1- Compiladores

2- Interpretadores

3- Tradutores

A- É gerado um código executável sem a necessidade de interpretar comando por comando.

B- São gerados códigos intermediários em que não se exige tanto espaço de memórias, quanto realizado pelo código fonte.

C- Lê uma instrução do código fonte, transforma em código binário e executa, repete esse processo até que todas as instruções do código fonte sejam executadas.

Assinale a alternativa que corresponde corretamente os métodos para gerar um código executável:

- a) 1-A, 2-B, 3-C.
- b) 1-A, 2-C, 3-B.
- c) 1-B, 2-A, 3-C.
- d) 1-C, 2-B, 3-A.
- e) 1-B, 2-C, 3-A.

3. De acordo com Houaiss (2001, p. 329), "paradigma significa modelo, padrão. No contexto da programação de computadores, um paradigma é um jeito, uma maneira, um estilo de se programar". Para Tucker (2010), um paradigma de programação está relacionado a um padrão

de soluções de problemas, onde por sua vez estão relacionados a uma determinada linguagem de programação.

Analise as afirmações abaixo:

I- Programação imperativa: considerada o paradigma mais antigo, a programação imperativa pode armazenar o programa e suas variáveis juntamente, assim como a abstração procedural, as atribuições, as sequências, os laços, os comandos condicionais e a manipulação de exceções são seus componentes de programação.

II- Programação orientada a objeto: também conhecida na computação como (POO), como o próprio nome sugere, é considerado uma programação somente por objetos, não podendo ter manipulações por código.

III- Programação funcional: são caracterizadas por possuírem atuação matemática, cada uma com um espaço de entrada (domínio) e resultado (faixa).

IV- Programação lógica: considerada uma programação declarativa, na qual um programa pode modelar uma situação problema declarando qual resultado o programa deve obter, em vez de como ele deve ser obtido.

Assinale a alternativa correta:

- a) Apenas as afirmações I, II e III estão corretas.
- b) Apenas a afirmação IV está correta.
- c) Apenas as afirmações II, III e IV estão corretas.
- d) Apenas as afirmações I, II, III e IV estão corretas.
- e) As afirmações I, III e IV estão corretas.

Seção 1.3

Componentes e elementos de Linguagem de Programação

Diálogo aberto

Maravilha! Agora que você já conhece os algoritmos, as famílias das linguagens de programação e as possibilidades profissionais que um programador pode atingir dentro e fora da empresa de softwares educacionais, nesta última seção da unidade, você terá a oportunidade de estudar os componentes de um programa de computador, as variáveis e constantes utilizadas em uma linguagem de programação, as operações e atribuições, assim como as estruturas dos algoritmos e programas.

Para avançar nas premissas que permeiam as linguagens de programação, vamos utilizar a analogia de uma cadeia de produção, na qual cada um dos envolvidos possuem uma função específica e que de forma alguma possam interferir na demanda um dos outros para entrega de um produto final. Assim funciona a estrutura de uma linguagem de programação, em que todos os seus componentes precisam estar de forma organizada para serem executados.

Passe tranquilidade aos seus estagiários e promova a criatividade, mostrando a eles um programa em linguagem C para calcular suas notas semestrais e a média final. Lembre-se da importância da utilização dos componentes para composição do código do programa, das aplicações e das ferramentas para essa tarefa.

Existe apenas uma forma para criar a estrutura de um programa de computador?

Tenha sempre o olhar para o futuro e bons estudos!

Não pode faltar

Daremos início à terceira e última seção da primeira unidade do livro de algoritmos e técnicas de programação. Nesta seção, você trabalhará os componentes de um programa de computador, as variáveis, as constantes, os operadores, as atribuições utilizadas nas

estruturas dos algoritmos e programas, em específico, a linguagem de programação C.

Vamos começar esta unidade falando das diversidades que existem nas formas de comunicação. Cada país e região tem sua língua nativa, certo? Na linguagem de programação, as pessoas precisam entender e se fazerem entender. Para cada linguagem, existe uma sintaxe a ser seguida (os comandos usados na criação dos programas), por este motivo, vamos iniciar a seção falando dos componentes de linguagem de programação e, para um melhor entendimento das demais unidades deste livro, você conhecerá os componentes envolvendo algoritmos e linguagem de programação C.

Segundo Damas (2016), um programa é uma sequência de código organizada de tal forma que permita resolver um determinado problema. Um programa pode ser desenvolvido em módulos distintos e/ou em subprogramas.

De acordo com Joyanes (2011), define-se um programa de computador como sendo um conjunto de instruções. Seria mais o menos assim: você direciona uma ordem para o computador e o mesmo executa uma determinada tarefa.

Lembrando que, para criar um programa de computador, será necessário seguir as seguintes regras (JOYANES, 2011):

- Definir e analisar o problema a ser solucionado.
- Criar um algoritmo ou um diagrama de fluxo.
- Realizar o pseudocódigo.

Para executar um programa é importante estar atento à sequência que os dados percorrem:

- Entrada de dados: realiza as coletas de dados.
- Processamento: os dados são transformados em informação.
- Saída: todas as informações geradas pelo processamento de dados são apresentadas em um periférico.

Para criação de um programa, você precisará definir as instruções a serem utilizadas na solução de um problema.

Segundo Joyanes (2011), as sintaxes (instrução) deverão ser escritas e armazenada na memória do computador na mesma ordem que se espera ser executada, ou seja, ela pode ser linear

(executada sequencialmente) e não linear (executada de forma a serem redirecionadas, isto é, uma instrução de bifurcação).

Veja um exemplo de instrução linear:

Instrução 1

Instrução 2

Instrução n

No caso de um programa não linear, ele pode se comportar da seguinte maneira.

Instrução 1

Instrução x

Instrução y

Instrução 2

Instrução n

Muito bem, a partir de agora você vai conhecer as sequências de instruções para criação de um programa de computador. Vamos lá!

No contexto geral ficaria assim:

- 1- Início do programa
- 2- Definição das variáveis e de possíveis atribuições
- 3- Instrução de leitura dos dados
- 4- Instrução do formato de escrita
- 5- Demais instruções e funções
- 6- Fim do programa.

Ou seja,

```
Início  
variáveis;  
comando1;  
    comando2;
```

```
comandoN;  
Fim.
```

Vale lembrar que, em algoritmos, a forma de escrever (maiúsculas e minúsculas) não acarretará em erros. Porém, em linguagem de programação C, é preciso diferenciar as palavras em letras maiúsculas e minúsculas.

Veja no Quadro 1.2 a estrutura básica de um algoritmo e de um programa em linguagem C.

Quadro 1.2 | Estrutura de um algoritmo e da linguagem C

Algoritmo	Linguagem C
Nome_Programa Declaração das variáveis início Ações Fim.	Definição das bibliotecas Início do programa (main) Início das funções Declaração das variáveis Instrução Termino das funções.

Fonte: elaborado pelo autor.

Quando iniciamos um programa na linguagem de programação C, segundo Manzano (2015), as primeiras linhas de programação são definidas pelas bibliotecas, também conhecida como arquivos de cabeçalho.

Para inserir as bibliotecas no programa, é necessário colocar `#include` (inclusão de um arquivo no programa fonte) e, em seguida, entre os símbolos de menor "<" e maior ">" (quando se usa < e > o arquivo é procurado na pasta include) o nome da biblioteca.

Veja alguns exemplos de biblioteca em linguagem de programação C:

- **stdio** – essa biblioteca é responsável pelas funções de entradas e saídas, como é o caso da função `printf` e `scanf` que vamos aprender mais à frente.

Exemplo: `#include <stdio.h>`

- **stdlib** – essa biblioteca transforma as strings (vetores de caracteres) em números.

Exemplo: `#include <stdlib.h>`

- **string** - biblioteca responsável pela manipulação de strings.

Exemplo: `#include <string.h>`

- **time** – biblioteca utilizada para manipulação de horas e datas.

Exemplo: `#include <time.h>`

- **math** – biblioteca utilizada para operações matemáticas.

Exemplo: `#include <math.h>`

- **ctype** – biblioteca utilizada para classificação e transformação de caracteres.

Exemplo: `#include <ctype.h>`.



Refleta

Para cada tipo de aplicação dentro do programa, será necessário inserir bibliotecas que interpretam as variáveis e os resultados que poderão surgir.

Diante desta situação é possível determinar a quantidade de bibliotecas e suas aplicações dentro da programação em linguagem C?

Após a definição das bibliotecas, o programa é inicializado pela função (`main`). Veja o seguinte exemplo:

```
main()
{
}
```

Quando utilizamos a “{” (chave aberta) indica o início de uma função em C e quando usamos a “}” (chave fechada) indica o

término das funções e do programa, porém, elas podem indicar o início e o término de alguns contextos na programação.

Quando usamos a `int` antes de `main ()` significa que retornará um número do tipo inteiro.

```
int main ()  
{  
}
```

Também pode ser utilizada a função `void`, esta é uma função sem retorno, ou seja, não recebe nenhum argumento.

```
void main ()  
{  
}
```

Para iniciar um algoritmo, como mostra no Quadro 1.3, é necessário colocar um nome que identifique o algoritmo. Logo em seguida, a definição das variáveis e somente depois a função "início" é liberada para criação das ações.

Vamos entender o que são variáveis:

As variáveis são locais reservados na memória para armazenamento dos dados, cada uma possui um nome próprio para sua identificação.

Os tipos de variáveis mais usadas são:

- **inteiro**: armazena os números inteiros (negativos ou positivos). Em linguagem C é definida por "int", em algoritmos é definida por "inteiro". Veja as especificações no Quadro 1.3:

Quadro 1.3 | Exemplo de aplicação de variável

Algoritmos	Linguagem C
Nome_Programa; var inteiro: valor1, valor2,soma; início Fim.	#include <stdlib.h> void main() { int valor1, valor2, soma; }

Fonte: elaborado pelo autor.

- **real**: permite armazenar valores de pontos flutuantes e com frações. Em linguagem C é definido por "float" e, quando precisa do dobro de dados numéricos, é utilizado o tipo "double" ou "long double". Em algoritmo, pode ser usado simplesmente a palavra "real". Veja um exemplo no Quadro 1.4.

Quadro 1.4 | Exemplo de aplicação de variável

Algoritmo	Linguagem C
Nome_Programa; var valor1, valor2,soma: real; Inicio Fim.	<pre>#include <stdlib.h> void main() { float valor1, valor2, soma; } </pre>

Fonte: elaborado pelo autor.

- **Caractere**: caracteriza os caracteres, números e símbolos especiais. São delimitadas por aspas simples ('). Em linguagem C é definida por "char". Em algoritmo é utilizado a palavra "caractere"

Veja o exemplo no Quadro 1.5:

Quadro 1.5 | Exemplo de aplicação de variável

Algoritmo	Linguagem C
Nome_Programa; var nome1, nome2: caractere; inicio Fim.	<pre>#include <stdlib.h> void main() { char nome1, nome2; } </pre>

Fonte: elaborado pelo autor.

Agora você vai conhecer como funciona as constantes. Vamos lá! O próprio nome já sugere, constante é algo que não se altera. Para Schildt (2005), as constantes são consideradas modificadores de tipo de acesso, ou seja, não podem ser alteradas, elas podem ser representadas pelo comando "const".

Exemplo:

```
const int fixo=500;
```

As constantes também podem ser caracterizadas por qualquer tipo de dados básicos, por exemplo: as constantes do tipo texto são envolvidas por aspas simples (') ou aspas duplas, as aspas simples representam um único caractere, por exemplo 'a' ou '100' e as aspas duplas caracterizam um conjunto de caracteres, por exemplo "A conversão da temperatura de graus centígrados para graus Fahrenheit".

As constantes inteiras são representadas por números inteiros negativos ou positivos, exemplo: -30 e 30 são constantes inteiras (int).

Elas seguem as mesmas regras para as variáveis do tipo flutuante, você poderá utilizar os comandos float e double, por exemplo: 30.30 é um número em ponto flutuante.



Pesquise mais

Sugiro a você, caro aluno, reforçar o seu conhecimento quanto aos tipos de variáveis e constantes, fazendo a leitura do artigo abaixo. Disponível em: <<http://www.ic.unicamp.br/~wainer/cursos/2s2011/Cap03-TiposBasicos-texto.pdf>>. Acesso em: 16 abr. 2018.

Para dar sequência a seção, você conhecerá os principais operadores utilizados nos algoritmos e na linguagem de programação C.

Vamos iniciar pelos operadores aritméticos que são representados por **binários** e **unários**.

- **Operadores Binários**

+ soma

- subtração

* multiplicação

/ divisão

% resto de divisão

- **Operador Unário**

Segundo Mizrahi (2008), o operador unário também pode ser usado para representar a troca de sinais de uma determinada variável, por exemplo:

a = -5;

a = -a;

Após essa operação o valor de "a" assume o valor de 5 positivo.



Assimile

Quando você deve usar os operadores de **incremento** e **decremento**? Quando você precisar adicionar um "1" à variável, você fará uso do "++" (incremento) e quando quiser tirar um "1" da variável você utiliza "--" (decremento), essas operações são caracterizadas unárias. Veja os exemplos abaixo:

a = a + 1; utilizando incremento ficaria desta maneira: ++a

a = a - 1; utilizando o decremento ficaria desta maneira: --a

Os **operadores relacionais** são utilizados para realizar a comparação de valores. Eles podem ser classificados de acordo com o Quadro 1.6.

Quadro 1.6 | Operadores relacionais

Operador em Linguagem C	Operador em algoritmos	Descrição
>	>	Maior
<	<	Menor
>=	>=	Maior ou igual
<=	<=	Menor ou igual
==	=	Igual
!=	<>	Diferente

Fonte: elaborado pelo autor.

Os operadores lógicos são utilizados quando nos deparamos com situações de conectividade lógica. Veja suas atribuições no quadro 1.7:

Quadro 1.7 | Operadores lógicos

Operador em Linguagem C	Operador em algoritmos	Operador em algoritmos
&&	E	Lógico E - conjunção
	Ou	Lógico OU - disjunção
!	não	Lógico NÃO - negação

Fonte: elaborado pelo autor.

Segundo Mizrahi (2008), o operador lógico "NÃO" é considerado um operador unário e os operadores lógicos "E" e "OU" são binários.

Joyanes (2011) afirma que todas as linguagens de programação possuem elementos básicos que são utilizados como blocos construtivos, os quais formam regras para que esses elementos se combinem. Essas regras são chamadas de sintaxes de linguagem ou comandos para programação.

Em linguagem de programação C, segundo Soffner (2013), existem um total de 32 palavras reservadas para programação, conforme definido pelo padrão ANSI. Veja na Figura 1.4 as palavras reservadas em linguagem de programação C.

Figura 1.4 | Palavras reservadas em linguagem C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Fonte: Soffner (2013, p. 36).

Veja também, algumas palavras reservadas que são utilizadas em algoritmos, no Quadro 1.8.

Quadro 1.8 | Palavras reservadas para uso de algoritmos.

Leia	para	faca	então	cos
Escreva	de	enquanto	senão	maiusc
Algoritmo	ate	fim_ enquanto	fim_se	minusc
Constantes	passo	repita, pare	sem	raiz
Var	fim_para	se	compr	div
Tan	trunc	abs	resto	e
Ou	não	inteiro	real	logico
Literal	vetor	matriz	verdadeiro	falso

Fonte: elaborado pelo autor.

Para atribuir um valor a uma variável em linguagem de programação C, utilizamos o sinal de igual "=", e utilizando algoritmos usamos o símbolo (←).

Exemplos com algoritmo:

```
soma ← valor1 + valor2;
```

```
valor ← 5;
```

Exemplos com linguagem C:

```
soma = valor1 + valor2;
```

```
valor=5;
```

Muito bem! Depois de declarar as variáveis e entender os operadores, vamos finalizar a seção estudando os comandos de entrada e saída de dados dentro da programação.

A função printf () é um **comando de saída**, o qual possui um vínculo com a biblioteca stdio.h. É utilizada quando se pretende obter uma resposta na tela do computador.

A sua síntese é definida por:

```
printf ("expressão de controle", listas de argumentos);
```

Existem algumas formatações na utilização da função printf(), conforme mostra a Figura 1.5, estas formatações podem mudar de acordo com os tipos de variáveis declaradas.

Figura 1.5 | código de formatação para função printf()

Código	Função
%c	Permite a escrita de apenas um caractere.
%d	Permite a escrita de números inteiros decimais.
%e	Realiza-se a escrita de números em notação científica.
%f	É feita a escrita de números reais (ponto flutuante).
%g	Permite a escrita de %e ou %f no formato mais curto.
%o	Permite que números octais sejam escritos.
%s	Efetua-se a escrita de uma série de caracteres.
%u	Escreve-se um número decimal sem sinal.
%x	Permite a escrita de um número hexadecimal.
%n	Permite determinar entre colchetes quais caracteres podem ser ou não aceitos na entrada de uma sequência de caracteres, sendo "n" um valor opcional que determina o tamanho da sequência de caracteres.

Fonte: Manzano (2013, p. 38).

Veja no exemplo abaixo a aplicação de formatação da função printf():

```
printf("O valor encontrado foi %d", valor1);
```

Perceba que o valor da variável "valor1" foi posicionado no local do "%d", lembrando que "%d" é uma formatação para um dado do tipo inteiro.

Outras particularidades:

Na função printf, você pode pular linhas com o comando "\n" e pode obter um resultado numérico determinando a quantidade de casas decimais. Veja os exemplos abaixo:

```
Printf (" \n Resposta: a = %.2f e b = %.2f \n", a,b);
```

Neste exemplo, antes de apresentar a frase o programa pulou uma linha "\n", o "%f", é utilizado quando os dados numéricos são flutuantes, ou seja, valores fracionados e quando usamos %.2f significa que o valor será arredondado em duas casas decimais, ex: 2,45.

Em algoritmos, a função utilizada para saída de dados é a palavra "escreva", veja o exemplo abaixo:

```

Valor_program;
var
real: valor;
inicio
    escreva("Digite um valor"); // saída da informação na tela
do computador

```

Fim.

A função **scanf()** é um comando de entrada, isto é, são informações que possibilitam a entrada de dados pelo teclado, assim, a informação será armazenada em um determinado espaço da memória, como o nome e tipo específico da variável. A sintaxe é definida por uma expressão de controle (sempre entre aspas duplas) e pela lista de argumento.

A sintaxe da função scanf() é definida por:

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

A função scanf() faz uso de alguns códigos de formação, veja na Figura 1.6:

Figura 1.6 | código de formatação para função scanf()

Código	Função
%c	Permite que seja efetuada a leitura de apenas um caractere.
%d	Permite fazer a leitura de números inteiros decimais.
%e	Permite a leitura de números em notação científica.
%f	É feita a leitura de números reais (ponto flutuante).
%l	Realiza-se a leitura de um número inteiro longo.
%o	Permite a leitura de números octais.
%s	Permite a leitura de uma série de caracteres.
%u	Efetua-se a leitura de um número decimal sem sinal.
%x	Permite que seja feita a leitura de um número hexadecimal.
%[código]	Permite que seja feita uma entrada formatada pelo código.

Fonte: Manzano (2013, p. 38).

Veja a sintaxe abaixo:

```
scanf ("%d", &valor);
```

Neste exemplo, o computador entrará com um valor decimal e retornará o valor da variável "valor".

O "&" é utilizado na função scanf() na lista de argumentos, sua função é retornar o conteúdo da variável, ou seja, retorna o endereço do operando.

```
main()
{
int valor;
printf("Digite um número: ");
scanf("%d",&valor);
printf("\n o número é %d",valor);
printf("\no endereço e %u",&valor);
}
```

Para realizar a entrada de dados em algoritmos, você pode utilizar a palavra "leia", veja o exemplo abaixo:

```
valor_program;
var
real: valor;
inicio
escreva("Digite um valor");
leia(valor); // valor de entrada, será armazenado na memória do computador
escreva("O valor digitado foi:", valor);
fim.
```

Cabe ressaltar que você pode fazer comentários em qualquer lugar do seu programa, basta utilizar barras duplas "//".

Exemplo:

```
#include <stdio.h> // biblioteca para entrada e saída de dados
int main() // comando de início e o mais importante do programa
{ // início de uma função
```

```
printf("Meu primeiro programa"); // comando para saída de
dados na tela
return 0; // indica que o processo está voltando para o Sistema
Operacional
} // fim de uma função ou de um programa
```



Exemplificando

Para um programa retornar ao sistema operacional, segundo Manzano (2015), será necessário utilizar a instrução retorna zero "return 0", assim como a instrução system("pause") tem a função de pausar a execução do programa, para que o resultado seja visualizado.

Veja o exemplo abaixo:

```
# include <stdio .h>

int main ( )
{
    int idade;

    printf ( "Digite a idade do candidato" ) ;
    scanf ( "%d" , &id ) ;
    printf ( "O candidato tem %d anos !\n" , id);
    system ( "pause");
    return 0 ;
}
```

Muito bem! Chagamos ao final da última seção da primeira unidade do livro de *Algoritmos e técnicas de programação*, dedique-se e estude bastante.

Boa sorte e bons estudos!

Sem medo de errar

Agora é o momento de resolver a situação-problema proposta no início desta seção. Lembre-se de que você tem a missão de apresentar aos seus estagiários os componentes de um programa de computador, as variáveis e constantes utilizadas em uma linguagem de programação, as operações e atribuições, assim como as estruturas dos algoritmos e programas.

Com muita tranquilidade, resolva a situação proposta para os seus estagiários, mostrando a eles como resolver um programa na linguagem de programação C, em que será preciso apresentar as notas bimestrais e, no final, apresentar a média aritmética. Recorde a importância da utilização dos componentes para a composição do código do programa, das aplicações e ferramentas para essa tarefa.

Para resolver o cálculo das notas bimestrais de um aluno e no final apresentar a média, você deverá apresentar aos estagiários os componentes para programação da linguagem C, usar as bibliotecas pertinentes, as variáveis, assim como, os seus tipos e a estrutura de entrada e saída de dados.

Pois bem, uma das possíveis soluções para essa problemática é:

```
1- #include <stdio.h>
2- #include <stdlib.h>
3- #include <math.h>
4- main(void)
5- {
6- float nota1,nota2,media;
7- printf("\n \n Digite a primeira nota: ");
8- scanf("%f",&nota1);
9- printf("Digite a segunda nota: ");
10- scanf("%d",&nota2);
11- media = (nota1 + nota2)/2;
12- printf("Media do aluno = %d\n",media);
13- return 0;
14- }
```

Na linha 1, 2 e 3 foram definidas as bibliotecas do programa.

Na linha 4 foi utilizado o main para iniciar o programa.

Na linha 5 teve início a função do programa.

Na linha 6 foram declaradas as variáveis, assim como, os seus tipos.

Na linha 7 foi realizado a saída na tela para atribuir uma entrega de informação.

Na linha 8 foi direcionado a função para armazenar o valor digitado.

Na linha 9, idem a linha 7.

Na linha 10, idem a linha 8.

Na linha 11 foi atribuído a variável "media" à soma das notas, seguido da divisão para encontrar a média.

Na linha 12, será mostrado na tela o texto e o valor da media encontrada.

Na linha 13 apresenta o retorno "0".

Na linha 14 será realizado o término da função.

Lembre-se: existem várias formas de criar uma estrutura para um programa de computador, assim sendo, existem mais de uma opção para chegar a uma solução de problema.

Pratique, crie novas situações e bons estudos.

Avançando na prática

Construção de um campo de futebol

Descrição da situação-problema

O brasileiro é um grande apaixonado por futebol. Sua universidade decidiu, junto ao grêmio estudantil, construir um campo de futebol dentro do campus. Você, fazendo parte interino do grêmio, ficou responsável por calcular a área total do círculo central do campo para que fosse instalada uma grama sintética com a cor da universidade. Para essa tarefa você deverá usar a fórmula da área do círculo e criar um programa de computador em linguagem

C. Ao final do programa, existem outros meios de chegar a mesma solução deste problema?

Fórmula da área do círculo:

$$A = \pi * r^2$$

A = área do círculo

$$\pi = 3.141592$$

r: raio

Resolução da situação-problema

Para resolver esse programa em linguagem C, você deverá interpretar a fórmula da área de círculo e utilizar os componentes de programação da linguagem.

1. `#include <stdio.h>`
2. `#include <stdlib.h>`
3. `#include <math.h>`
4. `void main()`
5. `{`
6. `float area, pi, raio;`
7. `printf("Digite o raio do circulo\n");`
8. `scanf("%f", &raio);`
9. `pi=3.141592;`
10. `area=pi*(pow(raio,2));`
11. `printf("A area do circulo e: %.2f\n", area);`
12. `return 0;`
13. `}`

Faça valer a pena

1. Segundo Joyanes (2011), define-se um programa de computador como sendo um conjunto de instruções, seria mais o menos assim: você direciona uma ordem para o computador e o mesmo executa uma determinada tarefa.

Análise se as afirmativas abaixo fazem parte das regras de criação de um programa de computador

- I- Definir e analisar o problema a ser solucionado.
- II- Criar um algoritmo ou um diagrama de fluxo.
- III- Realizar o pseudocódigo.

De acordo com a sua análise responda a alternativa correta:

- a) Somente a afirmação I está correta.
- b) As afirmações I, II e III estão corretas.
- c) Somente as afirmações I, II estão corretas.
- d) Somente as afirmações II e III estão corretas.
- e) Somente as afirmações I, III estão corretas.

2. As variáveis são locais reservados na memória para armazenamento dos dados, cada uma possui um nome próprio para sua identificação. Existe um tipo variável para cada representação da informação, não podendo assim, um determinado tipo de variável, ser usada em situações controversas.

De acordo com o texto acima, responda a alternativa correta:

- a) Variável do tipo inteiro armazena os números inteiros somente positivos e as variáveis do tipo real permite armazenar valores de pontos flutuantes e com frações.
- b) Variável do tipo inteiro armazena os números inteiros (negativos ou positivos) e as variáveis do tipo real permite armazenar somente valores de pontos flutuantes.
- c) Variável do tipo inteiro armazena os números inteiros (negativos ou positivos) e as variáveis do tipo real permite armazenar valores de pontos flutuantes e com frações.
- d) Variável do tipo inteiro armazena os números flutuante e as variáveis do tipo real permite armazenar valores inteiros.
- e) Variável do tipo inteiro armazena os números inteiros (negativos ou positivos) e as variáveis do tipo real permite armazenar valores de pontos flutuantes com frações e caracteres.

3. Analise a aplicação abaixo da formatação da função printf():
printf("O resultado final das aplicações foram %f", result);
Perceba que o valor da variável "result" será posicionado no local do "%f"
Porque
"%f" é uma formatação para um dado do tipo flutuante.

Analise a asserção acima e responda a alternativa correta:

- a) A primeira afirmação está correta e a segunda não justifica com a primeira.
- b) A primeira afirmação está correta e complementa a segunda.
- c) A primeira afirmação está incorreta e a segunda está correta.
- d) A primeira e a segunda afirmação estão corretas, porém não se justificam.
- e) A primeira e a segunda afirmação estão incorretas.

Referências

- AGUILAR, L. J. **Fundamentos de programação: algoritmos, estruturas de dados e objetos**. 3. ed. Porto Alegre: AMGH, 2011.
- DAMAS, L. **Linguagem C**. Tradução João Araújo Ribeiro, Orlando Bernardo Filho. - 10. ed. Rio de Janeiro: LTC, 2016.
- DONELLES, N. **As 15 principais linguagens de programação do mundo!** Disponível em: <<https://becode.com.br/principais-linguagens-de-programacao/>>. Acesso em: 31 mar. 2018.
- FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de programação: a construção de Algoritmos e Estruturas de Dados**. São Paulo: Makron, 2000.
- GIOVANELLI, L. **Conheça a carreira de Programador**. 2017. Disponível em: <<https://carreiras.empregos.com.br/profissao/conheca-a-carreira-de-programador/>>. Acesso em: 2 abr. 2018.
- GRASEL, G. F. **A história da programação**. 2014. Disponível em: <<https://www.oficinadnet.com.br/post/12895-a-historia-da-programacao>>. Acesso em: 31 mar. 2018.
- HOUAISS, A.; FRANCO, F. M. M.; VILLAR, M. S. **Dicionário Houaiss da Língua Portuguesa**. São Paulo: Objetiva, 2001.
- JOYANES AGUILAR, L. **Fundamentos de programação: algoritmos, estruturas de dados e objetos**. 3. ed. – Porto Alegre: AMGH, 2011.
- LOPES, A.; GARCIA, G. **Introdução a programação**. 8ª reimpressão. Rio de Janeiro: Elsevier, 2002.
- MANZANO, J. A. N. G. **Algoritmos: técnicas de programação**. 2. ed. São Paulo: Érica, 2015.
- MANZANO, J. A. N. G. **Estudo Dirigido de Linguagem C**. 17. ed. rev. -- São Paulo: Érica, 2013.
- MARÇULA, M. **Informática: conceitos e aplicações**. 4. ed. rev. São Paulo: Érica, 2013
- MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.
- MONTEIRO, M. A. **Introdução à organização de computadores**. 5.ed. Rio de Janeiro: LTC, 2010.
- MOTA, L. **Aprendendo a Programar com a Linguagem C**. 2010. Disponível em <<https://www.youtube.com/watch?v=HGVLKS2VmH4>>. Acesso em: 16 abr. 2018.
- NODE, Studio Treinamentos. **Lógica de Programação - Aula 03 - Tipos de dados, Variáveis, Constantes e Instruções**. 2016. Disponível em <<https://www.youtube.com/watch?v=-ny7Kqm0V68>>. Acesso em: 16 de abr. 2018.
- PEREIRA, A. P. **O que é algoritmo**. 2009. Disponível em <<https://www.tecmundo.com.br/programacao/2082-o-que-e-algoritmo-.htm>>. Acesso em: 19 mar. 2018.

- PIVA JUNIOR, D. et al. **Algoritmos e programação de computadores**. Rio de Janeiro: Elsevier, 2012
- PROCOPIO, R. **Fazer programa vale a pena? A Profissão de Programador**. Disponível em: <<https://www.youtube.com/watch?v=9ztYo4HH570>>. Acesso em: 31 mar. 2018.
- SALIBA, W. L. C. **Técnica de programação: uma abordagem estruturada**. São Paulo: Makron, 1993.
- SANTOS, D. **Processamento da linguagem natural: uma apresentação através das aplicações**. Ranchhod (org.). Lisboa: Caminho, 2001.
- SEBESTA, R. W. **Conceitos de linguagens de programação**. 9. ed. Porto Alegre: Bookman, 2011.
- SOFFNER, R. **Algoritmos e Programação em Linguagem C**. 1. Ed. – São Paulo: Saraiva, 2013.
- SZWARCFTER, J. L.; MARKENZON, L. **Estruturas de dados e seus algoritmos**. Rio de Janeiro: Editora LTC, 1994.
- TUCKER, A. B. **Linguagens de programação: princípios e paradigmas**. Porto Alegre: AMGH, 2010.
- VIDAL, V. **As linguagens de programação mais usadas de 2017 (até julho)**. 2017. <<https://www.showmetech.com.br/as-linguagens-de-programacao-mais-usadas-de-2017-ate-julho/>>. Acesso em: 1 abr. 2018.

Constantes, variáveis e operações

Convite ao estudo

Caro estudante, bem-vindo a mais uma unidade de estudo sobre os algoritmos e as técnicas de programação. Desde os primórdios da computação, sua concepção foi embasada pela necessidade de realizar cálculos, manipular dados e obter respostas para os mais diversos problemas da humanidade. Portanto, um computador tem, por essência, receber dados, processá-los e transmitir resultados, o que nos leva a intuir que os dados são a matéria prima de um sistema computacional.

Sabendo dessa premissa, conhecer e compreender o que são, quais os tipos e para que servem as constantes e variáveis dentro de uma linguagem de programação é essencial para seu futuro profissional. Nesta unidade, iremos entender como essa matéria prima é usada em um algoritmo. Será que existe diferença na utilização dos dados em diferentes linguagens de programação? Ou, ainda, em como um sistema operacional lida com esse recurso? Após a inserção dos dados, o que podemos fazer com eles?

Dando continuidade ao seu trabalho em uma empresa de desenvolvimento de software, você tem como objetivo treinar uma equipe de jovens programadores. Para isso, você deve explorar o uso dos dados em algoritmos implementando as soluções na linguagem de programação C. Com esse estudo, algumas questões poderão ser esclarecidas, tais como: quais os tipos de dados que podem ser tratados em um sistema computacional? O que é uma variável? O que é o escopo de uma variável? Quais operações matemáticas ou lógicas podemos usar para manipulação dos dados?

Para cumprir sua missão, nesta unidade você verá os conceitos de constante e variável e como classificar esses elementos em uma linguagem de programação, entenderá a diferença de dados primitivos e compostos, além de conhecer o tipo ponteiro. Após conhecer todos os tipos de dados, você aprenderá como manipulá-los com expressões matemáticas e lógicas, transformando-os em resultados.

Levando em consideração que estamos na era da informação, na qual, todos os dias, milhões de dados trafegam pelas redes, o profissional que sabe trabalhar com esse recurso tem sido muito valorizado pelo mercado. Então, mãos à obra!

Seção 2.1

Constantes e variáveis com tipos de dados primitivos

Diálogo aberto

Caro estudante, pegue um papel e uma caneta ou abra o bloco de notas. Olhe no relógio e anote a hora, o minuto e o segundo. Anotou? Novamente, olhe no relógio e faça a mesma anotação. Pronto? O valor foi o mesmo? E se anotasse novamente, seria igual? Certamente que não. Agora considere um computador que possui uma massa de 3 quilogramas, se você mudar ele de mesa, sua massa altera? E se colocá-lo no chão? Se esses elementos (tempo e massa) fizessem parte de uma solução computacional, eles seriam representados da mesma forma em um algoritmo?

Nesta seção, veremos como os dados podem ser classificados em algoritmos implementados na linguagem C. Se existem diferentes tipos de dados, é natural que existam diferentes formas de representá-los em uma linguagem de programação.

A empresa em que você atua contratou, recentemente, jovens programadores e ficou sob sua responsabilidade treiná-los. Para que a equipe se aproprie adequadamente do conhecimento, o primeiro passo é apresentar-lhes os conceitos de variáveis e constantes, bem como o que é um dado primitivo e quais tipos existem na linguagem C. Esses tipos são os mesmos de outra linguagem, por exemplo, Java? Como é possível saber quanto espaço uma variável ocupa na memória? Todos as variáveis de tipo primitivo ocupam o mesmo espaço na memória?

Você deverá criar um documento digital, em forma de slides, contendo esses conceitos, bem como uma lista com os tipos de dados primitivos que podem ser utilizados. Não se esqueça de apresentar exemplos de uso para cada tipo de dado.

Para completar essa etapa no trabalho, você aprenderá a diferença entre constantes e variáveis, bem como o que são dados primitivos e quais os tipos disponíveis na linguagem C.

Bons estudos!

Não pode faltar

É comum ouvirmos que vivemos na era da informação digital. O site <<http://www.internetlivestats.com/>> (acesso em: 17 abr. 2018) é um projeto de estatística em tempo real que mapeia parte do tráfego na Internet, por exemplo, quantidade de *tweets* e de fotos postadas no *Instagram* no dia, etc. São enviados mais de 200 bilhões de e-mails por dia, são vistos mais de 5 bilhões de vídeos no *Youtube*, entre vários outros dados.



Refleta

Um termo que vem sendo amplamente usado por empresários e profissionais de TI é o *Big Data*. Segundo Tarifa (2014), o termo nasceu por volta de 2005 e está relacionado ao volume, à velocidade e à variedade de dados que trafegam todos os dias pelas redes do mundo. Esses dados podem, também, ser considerados informações? Existe diferença entre esses termos?

Durante nossa formação escolar, aprendemos a realizar diversos cálculos, por exemplo, a área de uma circunferência $A = \pi r^2$, em que π (*pi*) equivale a, aproximadamente, 3,14 e r é o raio. Pensando ainda nesse problema da área da circunferência, vamos analisar o valor da área para alguns raios. Observe Tabela 2.1.

Tabela 2.1 | Alguns valores para área da circunferência

π (<i>pi</i>)	raio (r)	Área
3,14	2 cm	12,56 cm^2
3,14	3 cm	28,26 cm^2
3,14	4 cm	50,24 cm^2

Fonte: elaborada pela autora.

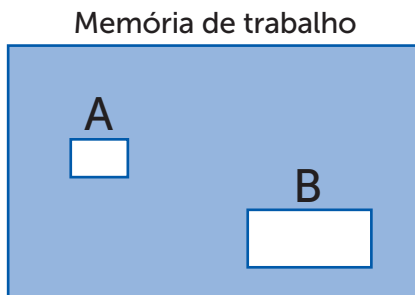
Após aplicar a fórmula para alguns valores, percebemos que, conforme o valor do raio varia, a área também varia, mas o *pi* permanece constante. São inúmeros os exemplos que poderiam ser dados, nos quais se observam valores que variam ou que são fixos, isso porque nosso mundo é composto por situações que podem ou

não variar, de acordo com certas condições. Sabendo que os sistemas computacionais são construídos para solucionar os mais diversos problemas, eles devem ser capazes de lidar com essa característica, ou seja, efetuar cálculo com valores que podem ou não variar. No mundo da programação, esses recursos são chamados de **variáveis** e **constantes**. A principal função desses elementos é armazenar temporariamente dados na memória de trabalho.

Deitel e Deitel (2011, p. 43) nos trazem a seguinte definição: "Uma variável é uma posição na memória onde um valor pode ser armazenado para ser utilizado por um programa." Soffner (2013, p. 33) incrementa dizendo que "variáveis são endereços de memória de trabalho que guardam, temporariamente, um valor utilizado pelo programa." A associação que os autores fazem entre as variáveis e os endereços de memória nos ajuda a compreender que esse elemento, quando criado, existe de fato na memória de trabalho e, portanto, ocupa um espaço físico. O mesmo se aplica às constantes, porém, nesse caso, o valor armazenado nunca será alterado.

Na Figura 2.1 foi criada uma representação simbólica para a memória de trabalho, e dentro dessa memória foram alocadas duas variáveis, A e B. Como você pode perceber, o espaço alocado para B é maior que o alocado para A, mas como é feita essa especificação? A quantidade de espaço que será alocada para uma variável pode ser especificada de duas maneiras: a primeira refere-se ao tipo de dado que será armazenado no espaço reservado, caso em que o programador não consegue manipular o tamanho alocado; a segunda é feita de maneira manual pelo programador, por meio de funções específicas, que conheceremos no decorrer do livro.

Figura 2.1 | Alocação de variáveis na memória de trabalho



Fonte: elaborada pela autora.



Independente da linguagem de programação que se esteja usando, os dados serão armazenados em variáveis, para cada tipo de dado será especificado um tipo de variável e, conseqüentemente, um determinado espaço físico será reservado na memória de trabalho.

Todas as linguagens de programação possuem tipos primitivos (ou básicos) e compostos. No grupo dos primitivos estão os tipos:

- **Numérico inteiro:** usando linguagem matemática, diríamos que é o conjunto dos números naturais, ou seja, valores inteiros que podem ser positivos, negativos ou zero. Como exemplo, podemos citar variáveis que armazenam idade, quantidade de produtos, código de identificação, dentre inúmeros outros.
- **Numérico de ponto flutuante:** esse tipo é utilizado para armazenar valores pertencentes ao conjunto dos números reais, ou seja, valores com casas decimais. São exemplos de variáveis desse grupo as que armazenam peso, altura, dinheiro, etc.
- **Caractere:** é o tipo usado para armazenar letras. Como exemplo de uso, podemos citar o armazenamento do gênero de uma pessoa, caso seja feminino, armazena *F*, caso masculino, armazena *M*.
- **Booleano:** variáveis desse tipo só podem armazenar um de dois valores: **verdadeiro** ou **falso**. Costuma ser usado para validações, por exemplo, para verificar se o usuário digitou um certo valor, ou se ele selecionou uma determinada opção em uma lista, etc.

O tipo composto será estudado na próxima seção, mas, por dedução lógica, podemos adiantar que esse tipo é formado a partir dos básicos.

Uso de variáveis em linguagens de programação

Para se usar uma variável em uma linguagem de programação é preciso criá-la, e, para isso, usa-se a seguinte sintaxe:

```
<tipo> <nome_da_variavel>;
```

Esse padrão é aceito por todas linguagens de programação, embora algumas permitam certas variações, tais como não definir o

tipo explicitamente. Na linguagem de programação C, esse padrão é obrigatório e podemos usar os seguintes tipos primitivos: **int** (inteiro), **float** ou **double** (ponto flutuante), **char** (caractere) e **void** (sem valor). O tipo booleano é representado pelo comando **bool**, entretanto, para seu uso, é necessário incluir a biblioteca `<stdbool.h>`. Veja no Quadro 2.1 a criação de algumas variáveis na linguagem C.

Quadro 2.1 | Criação de variáveis na linguagem C

```
1. #include <stdbool.h>
2. void main() {
3.     int idade;
4.     float salario = 7500;
5.     double qtd_atomos;
6.     bool confirma = false;
7.     char genero = 'M';
8. }
```

Fonte: elaborado pela autora.

Como você pode observar, ao criar uma variável, o programador pode optar por já atribuir ou não um valor. Mesmo existindo essa opção, é uma boa prática de programação sempre inicializar as variáveis para evitar que recebam dados que estejam na memória. Portanto, quando a variável for numérica, sempre iremos inicializar com zero, quando booleana, com falso, e quando do tipo caractere, iremos usar ' ' para atribuir vazio.

Outro ponto importante é o nome da variável, que deve ser escolhido de modo a identificar o dado a qual se refere. Por exemplo, em vez de usar "ida" para criar uma variável que irá armazenar a idade, opte pelo termo completo. A maioria das linguagens de programação são case *sensitive*, o que significa que letras maiúsculas e minúsculas são tratadas com distinção. Então, a variável "valor" é diferente da variável "Valor". No nome, não podem ser usados acentos nem caracteres especiais, como a interrogação.

Como já dissemos, a quantidade de espaço que será alocada depende do tipo de variável, por exemplo, para uma variável *int* serão alocados 4 bytes na memória de trabalho (MANZANO, 2010). O tamanho alocado na memória pelo tipo de variável limita o valor que pode ser guardado naquele espaço. Por exemplo, não seria

possível guardar o valor 10 trilhões dentro da variável do tipo *int*. Vejamos, 4 bytes são 32 bits. Cada bit só pode armazenar zero ou um. Portanto, nós temos a seguinte equação: valor da variável inteira = $2^{32} = 4.294.967.296$, porém esse valor precisa ser dividido por dois, pois um inteiro pode armazenar números negativos e positivos. Logo, uma variável *int* poderá ter um valor entre -2.147.423.648 e 2.147.423.648. (CASAVELLA, 2017).

Para suprir parte da limitação dos valores que uma variável pode assumir pelo seu tipo, foram criados modificadores de tipos, comandos usados na declaração da variável que modificam sua capacidade padrão. Os três principais modificadores são: **unsigned**, usado para especificar que a variável irá armazenar somente a parte positiva do número; **short**, que reduz o espaço reservado pela memória; e **long**, que aumenta a capacidade padrão. A Tabela 2.2 mostra o tamanho e os possíveis valores de alguns tipos, inclusive com os modificadores.

Tabela 2.2 | Tipos de variáveis e sua capacidade

Tipo	Tamanho (byte)	Valores
<i>int</i>	4	-2.147.423.648 até 2.147.423.648
<i>float</i>	4	-3,4 ³⁸ até 3,4 ³⁸
<i>double</i>	8	-1,7 ³⁰⁸ até 1,7 ³⁰⁸
<i>char</i>	1	-128 até 127
<i>unsigned int</i>	4	4.294.967.296
<i>short int</i>	2	-32.768 até 32.767
<i>long double</i>	16	-3,4 ⁴⁹³² até 1,1 ⁴⁹³²

Fonte: adaptada de Manzano (2017, p. 35).

Na linguagem C de programação, cada tipo de variável usa um **especificador de formato** para fazer a impressão do valor que está guardado naquele espaço da memória (PEREIRA, 2017). Veja o código no Quadro 2.2, em que foram criadas cinco variáveis (linhas 3 a 7) e, para cada tipo, usou-se um especificador na hora de fazer a impressão com a função `printf()`. Na linha 8, para o tipo inteiro, foi usado `%d`. Nas linhas 9 e 10, para os pontos flutuantes, foi usado

`%f`. Na linha 11, para o caractere, foi usado o `%c`, e, na linha 12, o especificador de ponto flutuante ganhou o adicional `.3`, o que significa que o resultado será impresso com três casas decimais.

Quadro 2.2 | Impressão de variáveis

```
1. #include<stdio.h>
2.
3. void main(){
4.     short int idade = 18;
5.     float salario = 7500;
6.     double qtd_atomos = 123456789123;
7.     char genero = 'F';
8.     float altura = 1.63;
9.
10.    printf("\n idade: %d",idade);
11.    printf("\n salario: %f",salario);
12.    printf("\n qtd_atomos: %f",qtd_atomos);
13.    printf("\n genero: %c",genero);
14.    printf("\n altura: %.3f",altura);
15. }
```

Fonte: elaborado pela autora.



Pesquise mais

Existem vários outros especificadores, por exemplo, `%x` permite apresentar um valor numérico inteiro positivo em hexadecimal.

Nas páginas 60 e 61 do livro *Linguagem C: acompanhada de uma xícara de café* você encontrará uma lista com vários especificadores de formatos. Esse livro encontra-se disponível na biblioteca virtual.

O endereço de memória de uma variável

A memória de um computador é dividida em blocos de *bytes*, e cada bloco possui um endereço que o identifica. Podemos fazer uma analogia com os endereços das casas, já que cada casa possui uma localização, e se existissem dois endereços iguais, certamente

seria um grande problema. Já sabemos que as variáveis são usadas para reservar um espaço temporário na memória, que possui um endereço único que o identifica. Será que conseguimos saber o endereço de alocação de uma variável? A resposta é sim! Para sabermos o endereço de uma variável basta utilizarmos o operador `&` na hora de imprimir a variável.



Exemplificando

Vamos criar um programa que armazena as coordenadas `x`, `y` com os seguintes valores (5,10). Em seguida, vamos pedir para imprimir o endereço dessas variáveis. Veja que nas linhas 7 e 8, a impressão foi feita usando `&x`, `&y`, o que resulta na impressão dos endereços de memória de `x` e `y` em hexadecimal, pois usamos o especificador `%x`.

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

2. void main(){
3.     int x = 5;
4.     int y = 10;

5.     printf("\n Valor guardado em x:
6.     %d", x);
7.     printf("\n Valor guardado em y:
8.     %d", y);

9.     printf("\n Endereco de x: %x", &x);
    printf("\n Endereco de y: %x", &y);
}
```

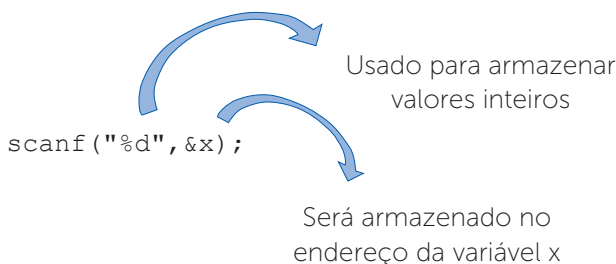
O operador `&` é muito importante na linguagem C, justamente por permitir acessar diretamente os endereços de memória das variáveis. Conforme avançarmos em nossos estudos, usaremos esse operador para alocarmos nosso próprio espaço de memória.

Para armazenar valores digitados pelo usuário em uma variável, podemos usar a função `scanf()`, com a seguinte estrutura:

```
scanf("especificador", &variavel);
```


A Figura 2.2 apresenta um exemplo no qual se utilizou o especificador “%d” para indicar ao compilador que o valor que será digitado deve ser um inteiro, e esse valor será guardado no endereço de memória da variável x.

Figura 2.2 | Armazenamento em variáveis



Fonte: elaborada pela autora.

Agora que já sabemos como ter acesso ao endereço de memória de uma variável e como armazenar valores digitados pelo usuário, vamos criar um programa em C que armazena dois valores, um em cada variável. Para isso o usuário terá que informar as entradas, que deverão ser armazenadas nas variáveis *valor1* e *valor2*. O Quadro 2.3 apresenta a solução. Veja que, na linha 3, como todas as variáveis eram do mesmo tipo, foram declaradas na mesma linha, separadas por vírgula, e todas foram inicializadas com zero. Nas linhas 5 e 7, os valores digitados pelo usuário serão armazenados nos endereços das variáveis *valor1* e *valor2*, respectivamente.

Quadro 2.3 | Impressão de valores armazenados

1.	<code>#include<stdio.h></code>
2.	<code>void main(){</code>
3.	<code>float valor1=0, valor2=0;</code>
4.	<code>printf("\n Digite o primeiro valor:");</code>
5.	<code>scanf("%f",&valor1);</code>
6.	<code>printf("\n Digite o segundo valor:");</code>
7.	<code>scanf("%f",&valor2);</code>
8.	<code>printf("Variavel 1 = %.2f",valor1);</code>

9.

```
printf("Variavel 2 = %.2f", valor3);  
}
```

Fonte: elaborado pela autora.

Constantes

Entende-se por constante um valor que nunca irá se alterar. Na linguagem C, existem duas formas de criar valores constantes. A primeira delas é usar a diretiva `#define`, logo após a inclusão das bibliotecas. Nesse caso a sintaxe será da seguinte forma:

```
#define <nome> <valor>
```

Veja que não possui ponto e vírgula no final. Outro ponto interessante é que a diretiva não utiliza espaço em memória, ela apenas cria um rótulo associado a um valor (MANZANO, 2015). Se não é reservado um espaço na memória, logo o operador `&` não pode ser usado nesse caso.

A outra forma de se criar valores constantes é similar à criação de variáveis, porém, antes do tipo, usa-se o comando `const`, portanto a sintaxe ficará:

```
const <tipo> <nome>;
```

Quando se utiliza a segunda forma de declaração, a alocação de espaço na memória segue o mesmo princípio das variáveis, ou seja, *int* alocará 4 bytes, *char* 1 byte, etc. A principal diferença entre as constantes e as variáveis é que o valor de uma constante nunca poderá ser alterado. Caso você crie uma constante, por exemplo, `const int x = 10;`, e tente alterar o valor no decorrer do código, o compilador acusará um erro e não será gerado o arquivo executável.

No Quadro 2.4, apresentamos um exemplo das duas formas de sintaxes para constantes. Na linha 2 definimos uma constante (rótulo) chamada *pi* com valor 3.14. Na linha 4 criamos uma constante usando o comando `const`. Nas linhas 5 e 6 imprimimos o valor de cada constante, veja que nada difere da impressão de variáveis.

Quadro 2.4 | Sintaxe para criação de constantes em C

1.

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

2.

```
#define pi 3.14
```

```

3. void main() {
4.     const float g = 9.80;

5.     printf("\n pi = %f",pi);
6.     printf("\n g = %f",g);
    }

```

Fonte: elaborado pela autora.

A linguagem C possui uma biblioteca matemática, que pode ser incluída por meio do comando `<math.h>`. Além de diversas funções, essa biblioteca possui algumas constantes matemáticas que podem ser acessadas seguindo os passos: (i) inclua a biblioteca `<math.h>`; (ii) inclua a diretriz `#define _USE_MATH_DEFINES`; (iii) use a notação `M_XXX` para acessar as constantes. O `XXX` deve ser substituído por um dos valores válidos. Para saber quais são os valores possíveis, acesse <https://msdn.microsoft.com/pt-br/library/4hwaceh6.aspx> (acesso em: 19 abr. 2018).



Exemplificando

Uma das opções de constantes da biblioteca é o `M_PI`, que armazena o valor da constante π .

```

1. #include<stdio.h>
2. #include<math.h>

3. #define _USE_MATH_DEFINES

4. void main() {
5.     M_PI;

6.     printf("\n pi = %f",M_PI);
7. }

```

Comando `sizeof`

Podemos usar o comando `sizeof(variavel)` para saber o espaço que a variável está ocupando na memória, mas é importante ter em mente que esse comando retorna o valor em *bytes*. Veja no código do Quadro 2.5 alguns exemplos do uso desse comando. Ele pode ser aplicado a constantes da biblioteca `math.h` (linha 8), a variáveis (linhas 9 e 10) e também a tipos (linha 11).

Quadro 2.5 | Comando `sizeof`

```
1.  #include<stdio.h>
2.  #include<math.h>

3.  #define _USE_MATH_DEFINES

4.  void main(){
5.      M_PI;
6.      int x = 10;
7.      float altura = 1.70;

8.      printf("\n Espaço alocado para pi =
9.      %d", sizeof(M_PI));
10.     printf("\n Espaço alocado para x =
10.     %d", sizeof(x));
11.     printf("\n Espaço alocado para altura =
12.     %d", sizeof(altura));
12.     printf("\n Espaço alocado para um char =
12.     %d", sizeof(char));
    }
```

Fonte: elaborado pela autora.

A compreensão do uso das variáveis é imprescindível para o avanço no estudo das técnicas de programação, pois iremos, cada vez mais, conhecer métodos que podem ser usados para resolverem problemas nos quais os dados são a matéria prima. Continue seus estudos!

Sem medo de errar

Após estudar as variáveis e constantes nessa seção, é hora de você estruturar a solução para o treinamento da sua equipe. Lembre-se, jovens programadores certamente já usaram variáveis em suas soluções, mas você deve garantir que eles compreendam todos os aspectos que envolvem esse importante elemento na programação. Seu treinamento deverá ser feito usando um documento digital em forma de slides, portanto a ordem lógica de apresentação do conteúdo é importante.

O primeiro item que você deve apresentar é a definição formal de variáveis e constantes. Essa informação ajuda a entender a relação que existe entre a criação de uma variável e a memória de trabalho de um computador. Para ajudar na solução, você pode usar imagens, como as das Figuras 2.1 e 2.2, assim ficará claro que os dados em um programa ocupam espaço na memória e que esse espaço irá variar conforme o tipo da variável.

O segundo passo é apresentar uma lista dos tipos de dados primitivos na linguagem C. Faça uma pesquisa e veja se os dados primitivos na linguagem C são os mesmos que a linguagem Java, já que essas duas linguagens pertencem a diferentes paradigmas de programação. Para essa etapa, é interessante construir um quadro com os tipos primitivos em ambas linguagens, por exemplo, conforme o Quadro 2.6.

Quadro 2.6 | Tipos primitivos em C e Java

Tipos primitivos em C	Tipos primitivos em Java	Exemplo
int	int	int idade = 18;
float	float	float salario = 2500;
...

Fonte: elaborado pela autora.

Outro ponto importante a ser abordado em seu treinamento é a questão do espaço que cada tipo de variável ocupa na memória. Faça um programa em C que utilize o comando `sizeof` para imprimir o tamanho de cada tipo primitivo, conforme exemplo no Quadro 2.7.

Inclua no código do programa em C exemplos de cada tipo de variável, bem como exemplos de como guardar dados inteiros e de ponto flutuante em variáveis para posterior impressão.

Quadro 2.7 | Espaço ocupado por variáveis

1.	<code>#include<stdio.h></code>
2.	<code>void main(){</code>
3.	<code>int x = 10, y = 5;</code>
4.	<code>float peso = 75.5;</code>
	<code>char genero = 'M' ;</code>
	<code>//crie outras variaveis</code>
5.	
6.	<code>printf("\n Espaco alocado para x =</code>
	<code>%d", sizeof(x));</code>
7.	<code>printf("\n Espaco alocado para y =</code>
	<code>%d", sizeof(y));</code>
8.	<code>printf("\n Espaco alocado para peso</code>
9.	<code>%d", sizeof(peso));</code>
10.	<code>printf("\n Espaco alocado para genero =</code>
	<code>%d", sizeof(genero));</code>
	<code>}</code>

Fonte: elaborado pela autora.

Todas essas informações alinharão a equipe para o uso consciente das variáveis para o armazenamento de dados.

Avançando na prática

Troca de valores

Descrição da situação-problema

Uma empresa desenvolvedora de games lhe contratou para implementar um novo jogo de tabuleiro com dados (tudo de forma digital). Esse jogo será executado em um ambiente online com dois jogadores. O avanço de cada participante depende do valor

tirado pelo outro, por exemplo, após lançar o dado, o jogador A tira 2 no dado, então será o jogador B quem andará duas casas no tabuleiro. Crie um programa em C que armazene o valor tirado por cada jogador e faça a troca dos valores, imprimindo na tela quantas casas cada um terá que avançar. Existem duas regras para a implementação:

- O valor tirado pelo jogador A, deverá ser guardado na variável **jogadorA** e a quantidade de casas que ele andará deverá ser impressa usando a mesma variável.
- O valor tirado pelo jogador B, deverá ser guardado na variável **jogadorB** e a quantidade de casas que ele andará deverá ser impressa usando a mesma variável.

Resolução da situação-problema

Devido às regras impostas para a implementação do programa, é necessário usar uma terceira variável para que possamos efetuar a troca dos valores. Por isso, na linha 3, além das variáveis *jogadorA* e *jogadorB*, também foi criada a *aux*. Na linha 5 é guardado o primeiro valor no endereço da variável *jogadorA* e, na linha 7, guarda-se o segundo valor no endereço de *jogadorB*. Na linha 8 usamos a variável auxiliar para guardar o primeiro valor, em seguida, na linha 9, colocamos o valor de *jogadorB* em *jogadorA* e por fim, na linha 10 colocamos o valor da variável auxiliar em *jogadorB*, concluindo assim a troca dos valores. As linhas 11 e 12 fazem a impressão da quantidade que cada jogador poderá avançar, seguindo as regras da implementação.

Quadro 2.8 | Programa para troca de valores

```
1. #include<stdio.h>
2. void main(){
3.     int jogadorA = 0, jogadorB = 0, aux = 0;
4.     printf("\n Digite o valor tirado pelo jogador
   A: ");
5.     scanf("%d",&jogadorA);
6.     printf("\n Digite o valor tirado pelo jogador
   B: ");
```

```

7.     scanf("%d",&jogadorB);
8.     aux = jogadorA;
9.     jogadorA = jogadorB;
10.    jogadorB = aux;

11.    printf("\n O jogador A deveria andar %d casas",
12.    jogadorA);
        printf("\n O jogador B deveria andar %d casas",
jogadorB);
    }

```

Fonte: elaborado pela autora.

Faça valer a pena

1. Variáveis são usadas para guardar valores temporariamente na memória de trabalho. A linguagem C oferece recursos para que seja possível conhecer o endereço de memória que foi alocado. Durante a execução de um programa, uma variável pode assumir qualquer valor desde que esteja de acordo com o tipo que foi especificado na sua criação.

A respeito dos tipos primitivos de variáveis, escolha a opção correta.

- Todas as linguagens de programação possuem os mesmos tipos primitivos de dados.
- Para todos os tipos primitivos na linguagem C são alocados os mesmos espaços na memória.
- Os valores numéricos podem ser armazenados em tipos primitivos inteiros ou de ponto flutuante.
- O número 10 é inteiro e por isso não pode ser guardado em uma variável primitiva do tipo float.
- O número 12.50 é decimal e por isso não pode ser guardado em uma variável primitiva do tipo int, pois gera um erro de compilação.

2. Constantes são usadas para guardar temporariamente valores fixos na memória de trabalho. O valor armazenado em uma constante não pode ser alterado em tempo de execução e essa é a principal diferença

com relação às variáveis. Na linguagem C, existem algumas constantes já definidas na biblioteca `math.h`.

A respeito das constantes na linguagem C, escolha a opção correta.

- a) Valores constantes podem ser declarados usando o comando `#define <nome> <valor>`.
- b) O comando `const int a = 10`, alocará um espaço de 1 byte para a constante `a`.
- c) O comando `const int a = 13.4`, resultará em um erro de compilação.
- d) A constante `M_PI` é usada para referenciar o valor do `pi`.
- e) A constante definida pelo comando `#define g 9.8` ocupará 4 bytes na memória.

3. A linguagem C de programação utiliza especificadores de formato para identificar o tipo de valor guardado nas variáveis e constantes. Eles devem ser usados tanto para leitura de um valor, como para a impressão. Quando um programa é executado, o compilador usa esses elementos para fazer as devidas referências e conexões, por isso o uso correto é fundamental para os resultados.

Considerando o código apresentado, analise as asserções e escolha a opção correta.

```
1.     #include<stdio.h>
2.     void main() {
3.         int idade = 0;
4.         float salario = 0;
5.         char a_letra = 'a';
6.         char A_letra = 'A';
7.     }
```

I- O comando `scanf("%f",&idade);` guardará o valor digitado na variável **idade**.

II- O comando `printf("%d",a_letra);` imprimirá **a** letra a na tela.

III- O comando `printf("%c",A_letra);` imprimirá a letra **A** na tela.

- a) Somente a alternativa I está correta.
- b) Somente a alternativa II está correta.
- c) Somente as alternativas I e II estão corretas.
- d) Somente a alternativa III está correta.
- e) Somente as alternativas II e III estão corretas.

Seção 2.2

Variáveis com tipos de dados compostos e ponteiros

Diálogo aberto

Caro estudante, reflita sobre o seguinte problema: você precisa implementar um programa que, ao final de um dia, recebe os dados sobre a cotação de uma determinada empresa na bolsa de valores e calcula a cotação média com base nos dados recebidos. Os dados são coletados das 8h às 17h, de hora em hora, logo, você possui 9 valores de um mesmo “assunto”. Não existe problema técnico em você criar nove variáveis para armazenar os valores, mas, do ponto de vista, das boas práticas de programação, essa não é uma boa opção. A fim de criar algoritmos bem estruturados, de forma a facilitar a manutenção e até mesmo o seu aperfeiçoamento, o programador deve armazenar temporariamente valores de um mesmo contexto, de forma otimizada, e esse será o assunto central dessa seção.

Continuando o trabalho com o treinamento dos jovens programadores, agora que você apresentou para a equipe quais tipos de dados primitivos podem ser utilizados em um sistema computacional, você deve apresentar os dados compostos na linguagem C. Nessa fase, você deve começar a tornar o conteúdo mais prático, a fim de possibilitar que a equipe comece a se familiarizar com uma das linguagens de programação mais utilizadas no mercado. Para que os jovens programadores se apropriem desse conhecimento, crie uma variável composta heterogênea que armazene dados de um cliente. Os dados a serem armazenados são: nome, idade, email, cpf e renda mensal. Com a estrutura criada, como os dados podem ser acessados para leitura e escrita? É possível armazenar dados de quantos clientes? Qual recurso poderia ser utilizado para armazenar os dados de 100 clientes?

Para cumprir essa fase, você aprenderá nesta seção o que são e como utilizar variáveis compostas, bem como para que servem e como utilizar as variáveis do tipo ponteiro. Vamos lá?

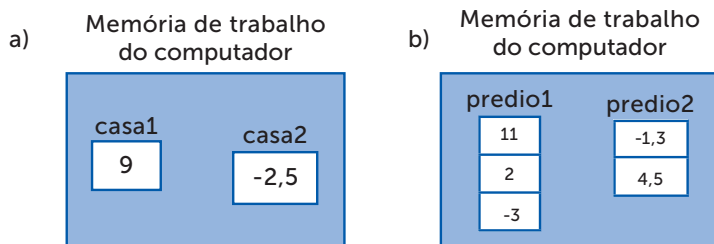
Não pode faltar

Já sabemos que as variáveis são usadas para armazenar dados na memória de trabalho e que esses dados podem ser de diferentes tipos (inteiro, decimal, caractere ou booleano), chamados de tipos primitivos. Vimos que podemos armazenar a idade de uma pessoa em uma variável do tipo *int*, a altura em um tipo *float*, etc, mas e se fosse necessário armazenar quinze medidas da temperatura de um dispositivo, usaríamos quinze variáveis? Com nosso conhecimento até o momento teríamos que criar as quinze, porém muitas variáveis podem deixar o código maior, além de não ser uma boa prática de programação. A melhor solução para armazenar diversos valores dentro de um mesmo contexto é utilizar variáveis compostas. Esse recurso permite armazenar diversos valores em um endereço de memória (MANZANO, 2015).

Quando alocamos uma variável primitiva, por exemplo um *int*, um espaço de 4 *bytes* é reservado na memória, ou seja, um bloco é reservado e seu endereço é usado para armazenamento e leitura dos dados. Quando alocamos uma variável composta do tipo *int*, um conjunto de blocos de 4 *bytes* será reservado. O tamanho desse conjunto (**1,2,3...N** blocos) é especificado pelo programador, conforme veremos adiante.

Para entendermos as variáveis compostas, vamos fazer uma analogia entre casas, prédios e as variáveis. Uma casa possui um endereço único que a identifica, composto por rua, número, bairro, cidade, estado e CEP. Considerando que uma casa é construída para uma família morar, podemos compará-la a uma variável primitiva que armazena um único valor em seu endereço. Por outro lado, um prédio possui um endereço composto pelos mesmos parâmetros que a casa, porém, nesse mesmo endereço moram várias famílias. Assim são as variáveis compostas, em um mesmo endereço são armazenados diversos valores. Esses conceitos estão ilustrados na Figura 2.3: veja que do lado esquerdo, a variável *casa1* armazena o valor 9 e a variável *casa2* armazena o valor -2,5; já no lado direito, a variável *predio1* armazena 3 valores inteiros e a variável *predio2* armazena dois valores decimais.

Figura 2.3 | a) Variáveis primitivas b) variáveis compostas



Fonte: elaborada pela autora.

Mas, se os vários valores têm o mesmo endereço, como diferenciar um do outro? Assim como os apartamentos em um prédio possuem números para diferenciá-los, as variáveis compostas possuem **índices** que as diferenciam. Portanto, uma variável composta possui um endereço na memória e índices para identificar seus subespaços. Existem autores que usam a nomenclatura “variável indexada” para se referir às variáveis compostas, pois sempre existirão índices (*index*) para os dados armazenados em cada espaço da variável composta (PIVA JUNIOR et al, 2012).



Assimile

Variáveis compostas são utilizadas para armazenar diversos dados em uma única estrutura na memória, por isso também é chamada de estrutura de dados. Existem vários tipos de estruturas de dados, por exemplo, filas, pilhas, etc. e, entre todas elas, a variável composta se destaca em termos de performance, pois cada dado é alocado sequencialmente na memória, tornando seu acesso extremamente rápido.

As variáveis compostas são formadas a partir dos tipos primitivos e podem ser classificadas em homogêneas e heterogêneas. Além disso, podem ser unidimensionais ou multidimensionais, sendo a bidimensional mais comumente usada (MANZANO, 2015). Quando armazenam valores do mesmo tipo primitivo, são homogêneas, mas quando armazenam valores de diferentes tipos são heterogêneas.

Variáveis compostas homogêneas unidimensionais (vetores)

As variáveis compostas *predio1* e *predio2*, ilustradas na Figura 2.3, são homogêneas, pois a primeira armazena apenas valores

do tipo inteiro e a segunda somente do tipo decimal. Além disso, elas também são unidimensionais, isso quer dizer que elas possuem a estrutura de uma tabela contendo apenas 1 coluna e N linhas (o resultado não se altera se pensarmos em uma estrutura como uma tabela de 1 linha e N colunas). Esse tipo de estrutura de dados é chamado de vetor ou matriz unidimensional (MANZANO; MATOS; LOURENÇO, 2015). Como o nome vetor é o nome mais utilizado entre os profissionais, adotaremos essa forma no restante do livro.

A criação de um vetor é similar a uma variável primitiva, tendo que acrescentar apenas um número entre colchetes indicando qual será o tamanho desse vetor (quantidade de blocos). Portanto, a sintaxe ficará da seguinte forma:

```
<tipo> <nome_do_vetor>[tamanho];
```



Exemplificando

Vamos criar dois vetores em C para armazenar a idade e a altura (em metros) de 3 pessoas. Veja no código do Quadro 2.9 que, na linha 3, foi criado o vetor *idade*, que é do tipo inteiro e com capacidade para armazenar três valores. Já na linha 4, o vetor *altura* foi criado e foi inicializado com valores. Para armazenar valores no vetor no momento da criação, colocamos os elementos entre chaves separados por vírgula. Da linha 5 a 7 é feita a impressão dos valores guardados no vetor *altura*.

Quadro 2.9 | Vetor em C

```
1. #include<stdio.h>
2. main() {
3.     int idade[3];
4.     float altura[3] = {1,1.5,1.7}
5.     printf("\n Vetor altura[0] = %d",altura[0]);
6.     printf("\n Vetor altura[1] = %d",altura[1]);
7.     printf("\n Vetor altura[2] = %d",altura[2]);
8. }
```

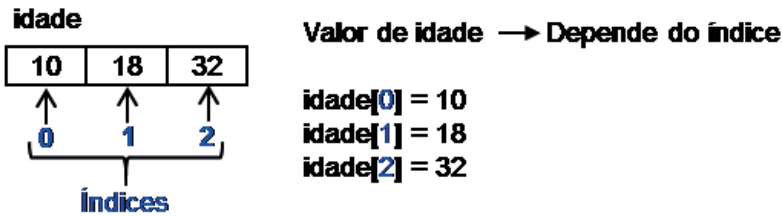
Fonte: elaborado pela autora.

Cada elemento no vetor é acessado por meio do seu índice, que sempre começará pelo valor zero, independentemente da linguagem de programação. Por isso, no código do Quadro 2.9, na linha 5, usou-se `altura[0]` para imprimir o primeiro valor, `altura[1]` para imprimir o segundo e `altura[2]` para imprimir o terceiro. O índice é usado tanto para leitura, como para escrita, por exemplo, podemos atribuir valores ao vetor *idade* seguinte forma:

```
idade[0] = 10;
idade[1] = 18;
idade[2] = 32;
```

Para ajudar a compreensão, observe a Figura 2.4, que representa um esquema para o vetor *idade* na memória. O valor do vetor depende da posição, ou seja, do índice. Outro ponto importante é que um vetor com *N* posições terá seus índices variando de 0 até *N*-1. Veja que o vetor *idade* tem capacidade de 3, portanto seu índice varia de 0 até 2.

Figura 2.4 | Vetor idade



Fonte: elaborada pela autora.

Na maioria dos programas, os dados a serem utilizados são digitados pelo usuário ou lidos de alguma base de dados. Para guardar um valor digitado pelo usuário em um vetor, usamos a função `scanf()`, porém, na variável, deverá ser especificado em qual posição do vetor se deseja guardar. Então, se quiséssemos guardar uma idade digitada pelo usuário no vetor, fariamos da seguinte forma:

```
printf("Digite uma idade: ");
scanf("%d", &idade[1]);
```

Nesse caso, o valor seria armazenado no índice 1 do vetor, ou seja, no segundo bloco, pois o primeiro é o bloco zero.



O vetor é uma estrutura de dados estática, ou seja, seu tamanho deve ser informado no momento da criação, pelo programador, e não é alterado por nenhum recurso em tempo de execução. Além disso, ao se criar um vetor com N posições, nem todas precisam ser utilizadas, mas lembre-se, quanto maior, mais espaço será reservado na memória de trabalho.

Vetor de caracteres (*string*)

Como é formada uma palavra? Por uma sequência de caracteres, correto? Vimos que o caractere é um tipo primitivo nas linguagens de programação. Sabendo que uma palavra é uma cadeia de caracteres, podemos concluir que esta é, na verdade, um vetor de caracteres (MANZANO, 2015). No mundo da programação, um vetor de caracteres é chamado de *string*, portanto adotaremos essa nomenclatura.

A declaração de uma *string* em C é feita da seguinte forma:

```
char <variavel>[tamanho];
```

Exemplos:—

```
char nome[16];  
char sobrenome[31];  
char frase[101];
```

Ao criarmos uma *string* em C, temos que nos atentar ao seu tamanho, pois a última posição da *string* é reservada pelo compilador, que atribui o valor “\0” para indicar o final da sequência. Portanto, a *string* nome[16] possui 15 “espaços” disponíveis para serem preenchidos.

A atribuição de valores à *string* pode ser feita no momento da declaração de três formas: (i) entre chaves informando cada caractere, separados por vírgula (igual aos outros tipos de vetores); (ii) atribuindo a palavra (ou frase) entre aspas; e (iii) atribuindo a palavra (ou frase) entre aspas e entre chaves.

Exemplos:

```
char nome[16]={'J','o','a','o'};  
char sobrenome[31] = "Alberto Gomes";  
char frase[101] = {"Disciplina de Algoritmos"};
```

Existem várias funções que fazem a leitura e impressão de *strings*

em C. Iremos estudar duas delas. A primeira já é conhecida, a função `scanf()`, mas agora usando o especificador de formato `%s` para indicar que será armazenada uma *string*. Portanto, para armazenar o nome digitado por um usuário usamos:

```
char nome[16];
printf("\n Digite um nome:");
scanf("%s", nome);
printf("\n Nome digitado: %s", nome);
```

Uma observação importante é que nesse caso o operador `&` não é obrigatório na função `scanf()`.

Essa forma de atribuição possui uma limitação: só é possível armazenar palavras simples, compostas não. Isso acontece porque a função `scanf()` interrompe a atribuição quando encontra um espaço em branco. Para contornar essa limitação, uma opção é usar a função `fgets()`, que também faz parte do pacote padrão `<stdio.h>`. Essa função possui a seguinte sintaxe:

```
fgets(destino, tamanho, fluxo);
```

O destino especifica o nome da *string* que será usada para armazenar. O tamanho deve ser o mesmo da declaração da *string*. O fluxo indica de onde está vindo a *string*, no nosso caso, sempre virá do teclado, portanto usaremos `stdin` (standard input).

Exemplo:

1. `char frase[101];`
2. `printf("\n Digite uma frase:");`
3. `fflush(stdin);`
4. `fgets(frase, 101, stdin);`
5. `printf("\n Frase digitada: %s", frase);`

Veja que antes de usar o `fgets()`, usamos a função `fflush(stdin)`, não é obrigatório, mas garante que a entrada padrão (`stdin`) seja limpa antes de armazenar.



Pesquise mais

Existem várias funções que fazem a leitura e impressão de *strings* em C. O capítulo 11 do livro *Linguagem C: acompanhada de uma xícara de*

café é dedicado a explicação de *strings*. Recomendamos a leitura da página 391, que apresenta uma tabela com várias funções de entrada e saídas de *strings*. Esse livro encontra-se disponível na biblioteca virtual.

Variáveis compostas homogêneas bidimensionais (matrizes)

Observe a Tabela 2.3, que apresenta alguns dados fictícios sobre a temperatura da cidade de São Paulo em uma determinada semana. Essa tabela representa uma estrutura de dados matricial com 7 linhas e 2 colunas. Para implementarmos um programa que calcula a média dessas temperaturas, precisaríamos armazenar esses valores em uma variável composta bidimensional, ou, como é mais comumente conhecida, uma matriz bidimensional.

Tabela 2.3 | Temperatura máxima de São Paulo na última semana

Dia	Temperatura (°C)
1	26,1
2	27,7
3	30,0
4	32,3
5	27,6
6	29,5
7	29,9

Fonte: elaborada pela autora.

Para criarmos uma matriz em C usamos a seguinte sintaxe:

```
<tipo> <nome_da_matriz>[linhas][colunas];
```

Exemplos:

1. `int coordenadas[3][2];`
2. `float temperaturas[7][2];`

Na linha 1 é criada uma estrutura com 3 linhas e 2 colunas. Na linha 2 é criada a estrutura da Tabela 2.3 (7 linhas e 2 colunas).

A criação e manipulação de matrizes bidimensionais exige a especificação de dois índices, portanto a atribuição de valores deve ser feita da seguinte forma:

```
matriz[M][N] = valor;
```

M representa a linha que se pretende armazenar e N a coluna. Assim como nos vetores, aqui os índices sempre iniciarão em zero.



Exemplificando

Vamos criar uma matriz em C para armazenar as notas do primeiro e segundo bimestre de três alunos. Veja na linha 3 do Quadro 2.10 que criamos uma matriz chamada `notas`, com 3 linhas e 2 colunas, o que significa que serão armazenados 6 valores (linhas x colunas). Nas linhas 4 e 5 armazenamos as notas do primeiro aluno: veja que a linha não se altera (primeiro índice), já a coluna sim (segundo índice). O mesmo acontece para o segundo e terceiro aluno, que são armazenados respectivamente na segunda e terceira linha.

Quadro 2.10 | Matriz em C

```
1. #include<stdio.h>
2. main(){
3.     float notas[3][2];
4.
5.     //aluno 1
6.     notas[0][0] = 10;
7.     notas[0][1] = 8.5;
8.
9.     //aluno 2
10.    notas[1][0] = 5.5;
11.    notas[1][1] = 2.7;
12.
13.    //aluno 3
14.    notas[2][0] = 4;
15.    notas[2][1] = 10;
16. }
```

Fonte: elaborado pela autora.

A Figura 2.5 ilustra a estrutura de dados que é criada na memória para o código do Quadro 2.10. Veja que as linhas foram usadas para representar os alunos e as colunas para as notas.

Figura 2.5 | Esquema de atribuição de notas

		Nota 1	Nota 2
		coluna 0	coluna 1
Aluno 1	linha 0	10,0	8,5
Aluno 2	linha 1	5,5	2,7
Aluno 3	linha 2	4,0	10,0

Fonte: elaborada pela autora.

Para armazenar valores digitados pelo usuário em uma matriz, usamos a função `scanf()`, indicando os dois índices para selecionar a posição que se deseja guardar. Para impressão de valores, também devemos selecionar a posição por meio dos dois índices.

Exemplo:

1. `printf("Digite uma nota: ");`
2. `scanf("%f", ¬a[1][0]);`
3. `printf("Nota digitada: %.2f", nota[1][0]);`

Nesse exemplo, a nota digitada será guardada na linha 1, coluna 0, e será exibida na tela usando duas casas decimais (comando linha 3).

Variáveis compostas heterogêneas (*structs*)

Já sabemos como otimizar o uso de variáveis usando as estruturas compostas (vetor e matriz), porém, só podemos armazenar valores de um mesmo tipo. Além das estruturas homogêneas, as linguagens de programação oferecem variáveis compostas heterogêneas chamadas de **estruturas** (*structs*) ou, ainda, de registros por alguns autores (SOFFNER, 2013).

Assim como associamos os vetores e matrizes a tabelas, podemos associar uma estrutura a uma ficha de cadastro com diversos campos. Por exemplo, o cadastro de um cliente poderia ser efetuado a partir da inserção do nome, idade, CPF e endereço em uma *struct*.

Na linguagem C, a criação de uma estrutura deve ser feita antes da função `main()` e deve possuir a seguinte sintaxe:

```

struct <nome>{
    <tipo> <nome_da_variavel1>;
    <tipo> <nome_da_variavel2>;
    ...
};

```

Nela, <nome> é o nome da estrutura, por exemplo, cliente, carro, fornecedor, etc., e as variáveis internas são os campos que se deseja guardar dessa estrutura. Na prática, uma estrutura funciona como um "tipo de dado" e seu uso sempre será atribuído à uma ou mais variáveis.



Exemplificando

Vamos criar uma estrutura para armazenar o modelo, o ano e o valor de um automóvel. No Quadro 2.11 a estrutura "automovel" foi criada entre as linhas 2 e 6. Mas, para utilizar essa estrutura, na linha 8 foi criada a variável "dadosAutomovel1", que é do tipo *struct automovel*. Somente após essas especificações é que podemos atribuir algum dado.

Quadro 2.11 | Struct em C

```

1.  #include<stdio.h>
2.
3.  struct automovel{
4.      char modelo[20];
5.      int ano;
6.      float valor;
7.  };
8.
9.  main(){
10.     struct automovel dadosAutomovel1;
11. }

```

Fonte: elaborado pela autora.

O acesso aos dados da *struct* (leitura e escrita) é feito usando o nome da variável que recebeu como tipo a estrutura com um ponto (.) e o nome do campo: *variavel.campo*. Por exemplo, para acessar o campo *ano* da *struct automovel* do Quadro 2.11, devemos escrever: *dadosAutomovel1.ano*. Veja no Quadro 2.12 a implementação completa para guardar valores digitados pelo usuário na estrutura "automovel" e depois imprimi-los.

```

1.  #include<stdio.h>
2.
3.  struct automovel{
4.      char modelo[20];
5.      int ano;
6.      float valor;
7.  };
8.
9.  main() {
10.
11.     struct automovel dadosAutomovell;
12.
13.     printf("\n Digite o modelo do automovel: ");
14.     scanf("%s", dadosAutomovell.modelo);
15.     printf("\n Digite o ano do automovel: ");
16.     scanf("%d", &dadosAutomovell.ano);
17.     printf("\n Digite o valor do automovel: ");
18.     scanf("%f", &dadosAutomovell.valor);
19.
20.     printf("\n Dados atribuidos");
21.     printf("\n %s", dadosAutomovell.modelo);
22.     printf("\n %d", dadosAutomovell.ano);
23.     printf("\n %f", dadosAutomovell.valor);
24. }

```

Fonte: elaborado pela autora.



Refleta

As estruturas são muito úteis para o armazenamento de diversos tipos de dados. Vimos um exemplo no qual a estrutura armazena os dados de um automóvel, mas e se fosse necessário armazenar os dados de 10, 20, 100 automóveis, o que poderia ser alterado para atender a essa demanda?

Variável do tipo ponteiro

Além das variáveis primitivas e compostas, existe um tipo de variável muito especial na linguagem C, chamada de **ponteiro**. Soffner (2013, p. 121) afirma que



Ponteiros são fundamentais para a Linguagem C; são parte do poder que essa linguagem tem em relação as demais. É por meio dos ponteiros que podemos manipular variáveis e outros recursos pelo endereço de memória, o que gera amplas capacidades e possibilidades.

Pela afirmação de Soffner, podemos perceber que existe uma relação direta entre um ponteiro e endereços de memória, e é exatamente por esse motivo que esse tipo de variável é utilizado, principalmente para manipulação de memória, dando suporte às rotinas de alocação dinâmica (MANZANO, 2015).

Variáveis do tipo ponteiro são usadas exclusivamente para armazenar endereços de memória. O acesso à memória é feito usando dois operadores, o asterisco (*), usado para criação do ponteiro e o “&”, que, como já vimos, é usado para acessar o endereço da memória, por isso é chamado de operador de referência. A sintaxe para criar um ponteiro tem como padrão:

```
<tipo> *<nome_do_ponteiro>;
```

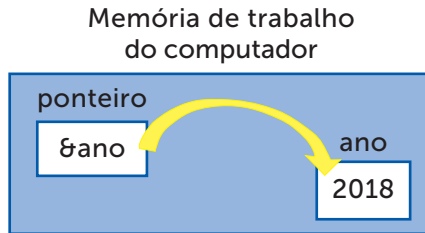
Exemplo: `int *idade;`

Nesse exemplo, é criado um ponteiro do tipo inteiro, isso significa que ele deverá “apontar” para o endereço de uma variável desse tipo. A criação de um ponteiro só faz sentido se for associada a algum endereço de memória. Para isso, usa-se a seguinte sintaxe:

1. `int ano = 2018;`
2. `int *ponteiro_para_ano = &ano;`

Na linha 1 criamos uma variável primitiva inteira com valor 2018, e na linha 2 associamos um ponteiro chamado *ponteiro_para_ano* ao endereço da variável primitiva *ano*. Agora tudo que estiver atribuído à variável *ano* também estará ao ponteiro. A Figura 2.6 ilustra, de forma simplificada, o esquema de uma variável com um ponteiro na memória. Veja que o conteúdo do ponteiro é o endereço da variável a que ele aponta e que ele também ocupa espaço na memória.

Figura 2.6 | Ponteiro e variável primitiva na memória



Fonte: elaborada pela autora.

Para finalizar esta seção, vamos ver como imprimir as informações de um ponteiro. Podemos imprimir o conteúdo do ponteiro, que será o endereço da variável a que ele aponta. Utilizando o ponteiro criado anteriormente (*ponteiro_para_ano*) temos a seguinte sintaxe:

```
printf("\n Conteudo do ponteiro: %p",ponteiro_
      para_ano);
```

O especificador de formato `%p` é usado para imprimir o endereço de memória armazenado em um ponteiro, em hexadecimal (também poderia usar o `%x`).

Também podemos acessar o conteúdo da variável que o ponteiro aponta, com a seguinte sintaxe:

```
printf("\n Conteudo da variavel pelo ponteiro:
      %d",*ponteiro_para_ano);
```

A diferença do comando anterior é o asterisco antes do nome do ponteiro.

Podemos, também, imprimir o endereço do próprio ponteiro, por meio da seguinte sintaxe:

```
printf("\n Endereco do ponteiro: %p",&ponteiro_
      para_ano);
```

Agora que você conhece vários tipos de variáveis, podemos avançar para o processamento dos dados armazenados nesses elementos, assunto das próximas seções!

Sem medo de errar

Dando continuidade ao seu treinamento para os jovens programadores, chegou o momento de desafiá-los a uma

atividade prática: criar uma variável composta heterogênea para armazenar o nome, a idade, o e-mail, o CPF e a renda mensal de um cliente. Além de criar a estrutura, o programa deverá receber os dados digitados pelo usuário e armazenar nos respectivos campos e, por fim, imprimir os valores que foram atribuídos. O primeiro passo para resolução consiste em criar de fato a estrutura, com o comando a seguir:

```
struct nome {
    <tipo> <campo1>;
    <tipo> <campo2>;
    ...
}
```

Com a estrutura definida, é preciso criar uma variável e a atribuir a ela o tipo da estrutura criada, portanto será um comando similar a:

```
struct nome <variavel>;
```

Agora é preciso criar uma sequência de funções `printf()` e `scanf()` para ir armazenando os dados que o usuário informar em cada campo, por exemplo:

```
printf("Digite o nome do cliente: ");
scanf("%s",variavel.nome);
```

Por fim, é necessário uma sequência de funções `printf()` para imprimir os dados recebidos e armazenados na estrutura, por exemplo:

```
printf("Nome digitado: %s",variavel.nome);
```

Esse programa permitirá armazenar os dados de um cliente. Para armazenar os dados de mais clientes, uma opção é criar mais de uma variável do tipo da estrutura, por exemplo:

```
struct nome <variavel1>;
struct nome <variavel2>;
```

Outra opção é criar um vetor de estruturas, por exemplo:

```
struct nome <variavel>[3];
```

A partir das instruções, complete a implementação na linguagem C para que o programa possa ser executado e testado.

Estruturas multidimensionais

Descrição da situação-problema

Você foi designado para implementar um programa para o setor de produção de uma empresa. O setor precisa de um programa em linguagem C que armazene a temperatura de 3 máquinas. O operador das máquinas fará a leitura da temperatura, duas vezes ao dia e digitará no programa o código da máquina e as temperaturas. Como você implementará a solução?

Resolução da situação-problema

Para resolver esse problema, podemos usar uma *struct* combinada com um vetor. Veja o resultado no Quadro 2.13. Foi criada uma estrutura monitoramento para armazenar um código e duas temperaturas para cada estrutura. Na linha 7, criamos uma variável para alocar espaço para 3 estruturas do tipo monitoramento. Depois foram usadas funções `printf()` e `scanf()` para armazenar os valores em suas respectivas posições.

Quadro 2.13 | Programa que combina *struct* e vetor

```
1.     #include<stdio.h>
2.
3.     struct monitoramento{
4.         int codigo;
5.         float temperatura[2];
6.     };
7.
8.     main(){
9.
10.        struct monitoramento dadosMaquina[3];
11.
12.        printf("\n Digite o codigo da primeira maquina: ");
13.        scanf("%d",&dadosMaquina[0].codigo);
14.
15.        printf("\n Digite as temperaturas da primeira
16. maquina: ");
17.        scanf("%f %f",&dadosMaquina[0].temperatura[0],
18. &dadosMaquina[0].temperatura[1]);
```

```

12.
13.     printf("\n Digite o codigo da segunda maquina: ");
14.     scanf("%d",&dadosMaquina[1].codigo);
15.
16.     printf("\n Digite as temperaturas da segunda
maquina: ");
17.     scanf("%f %f",&dadosMaquina[1].temperatura[0],
18. &dadosMaquina[1].temperatura[1]);
19.
20.     printf("\n Digite o codigo da terceira maquina: ");
21.     scanf("%d",&dadosMaquina[2].codigo);
22.
23.     printf("\n Digite as temperaturas da primeira
maquina: ");
24.     scanf("%f %f",&dadosMaquina[2].temperatura[0],
&dadosMaquina[2].temperatura[1]);
25.
26.     printf("\n Dados atribuidos");
27.     printf("\n Primeira maquina: \n Codigo:%d Temp1:
%f Temp2: %f",dadosMaquina[0].codigo,
28. dadosMaquina[0].temperatura[0],
dadosMaquina[0].temperatura[1]);
29.     printf("\n Segunda maquina: \n Codigo:%d Temp1:
30. %f Temp2: %f",dadosMaquina[1].codigo,
dadosMaquina[1].temperatura[0],
dadosMaquina[1].temperatura[1]);
31.     printf("\n Terceira maquina: \n Codigo:%d Temp1:
32. %f Temp2: %f",dadosMaquina[2].codigo,
dadosMaquina[2].temperatura[0],
dadosMaquina[2].temperatura[1]);
33. }

```

Fonte: elaborado pela autora.

Faça valer a pena

1. “Variáveis indexadas, ao contrário das variáveis tradicionais, armazenam mais de um valor de mesmo tipo, e são úteis para a manipulação de séries de valores semelhantes, sejam elas uni ou multidimensionais” (SOFFNER, 2013, p. 87).

A respeito das variáveis indexadas, analise as asserções abaixo e escolha a opção correta.

I – Para armazenar a idade de 15 pessoas em uma variável indexada, precisamos alocar 16 posições, pois o compilador reserva o último espaço para o caractere “\0”.

II – Para armazenar um valor na última posição de uma variável indexada unidimensional com tamanho 10, devemos atribuir o valor ao índice 9 da variável.

III - Para armazenar um valor na primeira posição de uma variável indexada unidimensional com tamanho 8, devemos atribuir o valor ao índice 1 da variável.

- a) Somente a afirmação I está correta.
- b) Somente a afirmação II está correta.
- c) Somente a afirmação III está correta.
- d) Somente as afirmações I e II estão corretas.
- e) Somente as afirmações II e III estão corretas.

2. A formação de uma palavra ou frase é feita a partir de uma cadeia de caracteres. No mundo da programação, essa estrutura é conhecida por *string* e pode ser usada em todas as linguagens. Na linguagem C, uma *string* é criada a partir de um vetor de caracteres, por meio do comando `char <variavel>[tamanho];`

A respeito dos vetores de caracteres em C, escolha a opção correta.

- a) A atribuição do valor pode ser feita no momento da criação usando a sintaxe `char nome[20] = {"Joao", "da Silva"};`
- b) A atribuição do valor pode ser feita no momento da criação usando a sintaxe `char nome[20] = 'J', 'o', 'a', 'o';`
- c) Ao criar uma *string*, não pode ser atribuído um valor.
- d) A função `scanf("%s", variavel)` pode ser usada somente para palavras simples.
- e) A função `fgets (variavel, tamanho, fluxo)` pode ser usada somente para palavras compostas ou frases.

3. Ponteiro é um tipo especial de variável usado para armazenar endereços de memória. Esse recurso deu origem ao que se chama de “passagem por valor” e “passagem por referência” em linguagens como Java e C#. Em C, os ponteiros são usados para dar suporte a funções de alocação de memória, proporcionando a criação de programas complexos.

Considere o seguinte código em C:

```
1. #include<stdio.h>
2. main(){
3.     int x = 10;
4.     int *pX= &x;
5.     printf("%x", &x);
6.     printf("%d", x);
7.     printf("%p", pX);
8.     printf("%d", *pX);
9.     printf("%p", &pX);
10. }
```

Considerando o código apresentado, escolha a opção correta.

- a) A linha 5 imprimirá o endereço do ponteiro.
- b) A linha 9 imprimirá o endereço a que o ponteiro aponta.
- c) A linha 8 imprimirá o conteúdo armazenado na variável x.
- d) A linha 6 imprimirá o conteúdo do ponteiro.
- e) A linha 7 imprimirá o endereço do ponteiro.

Seção 2.3

Operações e Expressões

Diálogo aberto

Desde o momento em que você liga um computador, *tablet* ou *smartphone*, centenas de processos são inicializados e passam a competir o processador para que possam ser executados e a “mágica” do mundo digital possa acontecer. Todos os resultados desses sistemas são obtidos por meio do processamento de dados e, nesta seção, começaremos a estudar os recursos que lhe permitirão implementar soluções com processamento.

Parabéns pelo trabalho! Sua equipe já está dominando os tipos de dados que podem ser utilizados em um sistema computacional, bem como já sabem como declará-los e guardá-los temporariamente em variáveis. Entretanto, a essência de um computador consiste em receber os dados, processá-los e exibir os resultados. Logo, chegou o momento de ensinar sua equipe a processar os dados por meio de operações e expressões matemáticas, e a combiná-las usando operadores lógicos.

Para que o treinamento continue contribuindo com o aperfeiçoamento teórico e prático, crie um programa em C que calcule a quantidade média de veículos registrados no estado de São Paulo em cada ano, usando os dados da Tabela 2.4. Qual a média de veículos registrados em 2014? E em 2015 e 2016? Conseguimos saber qual ano obteve a maior média? Existe outra forma de comparar os dados? O programa criado deve ser mostrado a toda equipe.

Tabela 2.4 | Quantidade de veículos no estado de São Paulo

Tipo de veículo	Quantidade de veículos por ano		
	2014	2015	2016
Carro	16.319.979	16.834.629	17.247.123
Moto	4.133.366	4.268.872	4.378.772
Caminhão	658.713	664.617	669.056

Fonte: IBGE (2018).

Para cumprir com êxito a missão, você verá nesta seção como utilizar os operadores aritméticos, relacionais e lógicos, além de conhecer algumas funções pré-definidas.

Bons estudos!

Não pode faltar

Sistemas computacionais são construídos para resolver os mais diversos problemas. Todos esses sistemas, independentemente da sua aplicação, são construídos em três partes: entrada, processamento e saída. Na entrada, os valores que serão utilizados pelo processamento são lidos basicamente a partir de três fontes: (i) digitados pelo usuário, nesse caso, a partir de uma *interface* textual ou gráfica pela qual o usuário alimenta o sistema com dados; (ii) leitura de arquivos, pois é possível implementar sistemas que fazem a leitura de dados a partir de arquivos de texto, planilhas, arquivos pdf, entre outros; e (iii) acesso a banco de dados, caso em que são usados programas que fazem o gerenciamento da base de dados ao qual o sistema computacional possui acesso. Nos três casos, a leitura dos dados é feita apenas para processar e gerar informações, e essa etapa é construída a partir da combinação de operações aritméticas, relacionais, lógicas e outras técnicas de programação que você conhecerá no decorrer do livro.

Operadores aritméticos em linguagens de programação

Vamos começar a aprimorar nossos algoritmos com as operações aritméticas. Veja no Quadro 2.14 algumas operações disponíveis nas linguagens de programação e seus respectivos exemplos.

Quadro 2.14 | Operações aritméticas básicas em linguagens de programação

Operador	Descrição	Exemplo	Resultado
+	Soma	$4 + 2$	6
-	Subtração	$4 - 2$	2
*	Multiplicação	$4 * 2$	8
/	Divisão	$4 / 2$	2
=	Atribuição	$x = 4$	$x = 4$
%	Módulo	$4 \% 2$	0

Fonte: adaptado de Manzano (2015, p. 43).



Vamos criar um programa em C que soma a idade de duas pessoas e imprime na tela o resultado. No Quadro 2.15 está o código que realiza tal processo. Veja que primeiramente as idades foram solicitadas e armazenadas em duas variáveis (linhas 6 a 9 – entrada pelo usuário), para depois ser feito o processamento (linha 10) e, por fim, a exibição do resultado na tela.

Quadro 2.15 | Soma da idade de duas pessoas

```
1.  #include<stdio.h>
2.  main() {
3.      int idade1=0;
4.      int idade2=0;
5.      int resultado=0;
6.      printf("Digite a primeira idade: ");
7.      scanf("%d",&idade1);
8.      printf("Digite a segunda idade: ");
9.      scanf("%d",&idade2);
10.     resultado = idade1 + idade2;
11.     printf("Resultado = %d",resultado);
12. }
```

Fonte: elaborado pela autora.

Quando trabalhamos com operadores, a ordem de precedência é muito importante. Segundo Soffner (2013), Os operadores aritméticos possuem a seguinte ordem de execução:

- 1° parênteses;
- 2° potenciação e radiciação;
- 3° multiplicação, divisão e módulo;
- 4° soma e subtração;



Ao implementar uma solução em software, um dos maiores desafios é garantir que a lógica esteja correta e, tratando-se da parte de processamento, ao escrever uma expressão matemática, é preciso se atentar à ordem de precedência dos operadores. Se ao implementar uma solução que calcula a média aritmética, você usar a expressão `resultado = a + b / c`, você terá o resultado correto? Se for um cálculo para o setor financeiro de uma empresa, seu cálculo mostraria um lucro ou prejuízo?

Das operações aritméticas apresentadas no Quadro 2.14, a operação módulo (%) talvez seja a que você não tenha familiaridade. Essa operação faz a divisão de um número, considerando somente a parte inteira do quociente, e retorna o resto da divisão.



Exemplificando

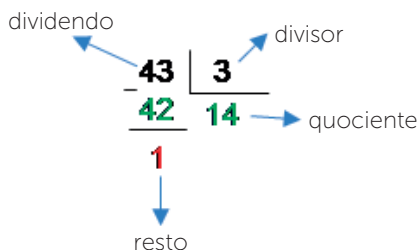
Vamos aplicar o operador módulo para efetuar o processamento de $43 \% 3$.

```
#include<stdio.h>

main(){
    int resultado = 43%3;
    printf("Operacao modulo 43%3 = %d",resultado);
}
```

Ao executar o código, obtém-se como resultado o resto da divisão, ou seja, nesse caso, o valor 1. Veja na Figura 2.7 o cálculo matemático que é efetuado e como o resultado é obtido.

Figura 2.7 | Operação aritmética módulo



Fonte: elaborada pela autora.

Os operadores aritméticos podem ser classificados em unário ou binário (MANZANO, 2015). Os binários, que nós já conhecemos no Quadro 2.14, são operadores que usam dois componentes, já os operadores unários usam apenas um componente. É o caso dos operadores aritméticos de incremento (++) e decremento (--). Esses operadores acrescentam ou diminuem “um” ao valor de uma variável e podem ser usados de duas formas:

- Após a variável:
 - o Pós-incremento: $x++$; nesse caso, é adicionado um após a primeira execução.
 - o Pós-decremento: $x--$; nesse caso, é decrementado um após a primeira execução.
- Antes da variável:
 - o Pré-incremento $++x$; nesse caso, é adicionado um antes da primeira execução.
 - o Pré-decremento $--x$; nesse caso, é decrementado um antes da primeira execução.

O Quadro 2.16 apresenta um resumo dos operadores unários.

Quadro 2.16 | Operadores aritméticos unário

Operador	Descrição	Exemplo	Resultado
++	Pós-incremento	$x++$	$x + 1$
++	Pré-incremento	$++x$	$x + 1$
--	Pós-decremento	$y--$	$y - 1$
--	Pré-decremento	$--y$	$y - 1$

Fonte: elaborado pela autora.

Operadores relacionais em linguagens de programação

Faz parte do processamento fazer comparações entre valores, para, a partir do resultado, realizar novas ações. Por exemplo, podemos criar um programa que soma a nota de dois bimestres de um aluno e efetua a média aritmética. A partir do resultado, se o aluno obteve média superior a seis, ele está aprovado, caso contrário, está reprovado. Veja que é necessário fazer uma comparação da média obtida pelo aluno com a nota estabelecida como critério.

Em programação, para compararmos valores usamos operadores relacionais. O Quadro 2.17 apresenta os operadores usados nas linguagens de programação (DEITEL; DEITEL, 2011).

Quadro 2.17 | Operadores relacionais em linguagens de programação

Operador	Descrição	Exemplo
==	igual a	$x == y$
!=	diferente de	$x != y$
>	maior que	$x > y$
<	menor que	$x < y$
>=	maior ou igual que	$x >= y$
<=	menor ou igual que	$x <= y$

Fonte: adaptado de Manzano (2015, p. 82).

Os operadores relacionais são usados para construir expressões booleanas, ou seja, expressões que terão como resultado **verdadeiro** ou **falso**. Quando fazemos uma comparação na linguagem C, o resultado será um ou zero, sendo que o primeiro representa um resultado verdadeiro e o segundo um falso.

Vamos criar um programa que solicita ao usuário dois números inteiros e faz algumas comparações com esses valores. Veja no Quadro 2.18 que na linha 9 comparamos se os números são iguais, na linha 10 se o primeiro é maior que o segundo e na linha 11 se o primeiro é menor ou igual ao segundo.

Quadro 2.18 | Comparações entre dois números

```

1. #include<stdio.h>
2. main(){
3.     int n1=0;
4.     int n2=0;
5.     printf("Digite o primeiro numero: ");
6.     scanf("%d",&n1);
7.     printf("Digite o segundo numero: ");
8.     scanf("%d",&n2);
9.     printf("\n n1 e n2 sao iguais? %d",n1==n2);
10.    printf("\n n1 e maior que n2? %d",n1>n2);
11.    printf("\n n1 e menor ou igual a n2? %d",n1<=n2);
12. }
```

Fonte: elaborado pela autora.

Operadores lógicos em linguagens de programação

Além dos operadores relacionais, outro importante recurso para o processamento é a utilização de operadores lógicos, que possuem

como fundamento a lógica matemática clássica e a lógica Booleana (GERSTING, 2017). O Quadro 2.19 apresenta os operadores lógicos que podem ser usados nas linguagens de programação.

Quadro 2.19 | Operadores lógicos em linguagens de programação

Operador	Descrição	Exemplo
!	negação (not)	$!(x == y)$
&&	conjunção (and)	$(x > y) \&\&(a == b)$
	disjunção (or)	$(x > y) (a == b)$

Fonte: adaptado de Soffner (2013, p. 35).

Os operadores lógicos são utilizados juntamente com os relacionais, criando comparações mais complexas.



Assimile

O operador de negação é usado para inverter o resultado da expressão. O operador de conjunção é usado para criar condições em que todas as alternativas sejam verdadeiras. O operador de disjunção é usado para criar condições em que basta uma condição ser verdadeira para que o resultado também seja.

Veja no Quadro 2.20 o uso dos operadores relacionais e lógicos aplicados à comparação dos valores de três variáveis. Na linha 4, a condição criada será verdadeira caso o valor de "a" seja igual ao de "b" **E** o valor de "a" também seja igual a "c", nesse caso a primeira condição não é verdadeira, logo o resultado da expressão será 0. Na linha 5, a condição criada será verdadeira caso uma das condições seja satisfeita, logo o resultado será 1. Por fim, na linha 6, invertemos esse resultado com o operador de negação.

Quadro 2.20 | Operadores relacionais e lógicos

```
1. #include<stdio.h>
2. main(){
3.     int a=5, b=10, c=5;
4.     printf("\n (a==b) &&(a==c) = %d", ((a==b) &&(a==c)));
5.     printf("\n (a==b) || (a==c) = %d", ((a==b) || (a==c)));
6.     printf("\n!(a==b) || (a==c) =%d", !((a==b) || (a==c)));
7. }
```

Fonte: elaborado pela autora.

Funções predefinidas para linguagem de programação

Para facilitar o desenvolvimento de soluções em software, cada linguagem de programação oferece um conjunto de funções pré-definidas que ficam à disposição dos programadores. Entende-se por função “um conjunto de instruções que efetuam uma tarefa específica” (MANZANO, 2015, p. 153).



Pesquise mais

Existe uma série de bibliotecas e funções disponíveis na linguagem C que podem facilitar o desenvolvimento de soluções. No endereço <https://www.tutorialspoint.com/c_standard_library/index.htm> você encontrará uma vasta referência à esses elementos (acesso em: 23 jul. 2018).

Neste livro, você já usou algumas das funções da linguagem C, por exemplo, para imprimir uma mensagem na tela, usa-se a função `printf()`, que pertence à biblioteca `stdio.h`. Uma biblioteca é caracterizada por um conjunto de funções divididas por contexto (MANZANO, 2015). Vamos apresentar algumas funções, que costumam aparecer com frequência nos programas implementados na linguagem C (Quadro 2.21).

Quadro 2.21 | Algumas bibliotecas e funções na linguagem C

Biblioteca	Função	Descrição
<stdio.h>	<code>printf()</code> <code>scanf()</code> <code>fgets(variavel, tamanho, fluxo)</code>	Imprime na tela Faz leitura de um dado digitado Faz a leitura de uma linha
<math.h>	<code>pow(base, potencia)</code> <code>sqrt(numero)</code> <code>sin(angulo)</code> <code>cos(angulo)</code>	Operação de potenciação Calcula a raiz quadrada Calcula o seno de um ângulo Calcula o cosseno de um ângulo
<string.h>	<code>strcmp(string1, string2)</code> <code>strcpy(destino, origem)</code>	Verifica se duas <i>strings</i> são iguais Copia uma <i>string</i> da origem para o destino
<stdlib.h>	<code>malloc(tamanho)</code> <code>realloc(local, tamanho)</code> <code>free(local)</code>	Aloca dinamicamente espaço na memória Modifica um espaço já alocado dinamicamente Libera um espaço alocado dinamicamente

Fonte: adaptado de tutorialspoint, [s.d.].

A função `strcmp(string1, string2)` compara o conteúdo de duas *strings* e pode retornar três resultados, o valor nulo (zero), positivo ou negativo, conforme as seguintes regras:

- Quando as *strings* forem iguais, a função retorna 0.
- Quando as *strings* forem diferentes, o primeiro caractere não coincidir entre elas, sendo "maior" na primeira, a função retorna um valor positivo. Entende-se por "maior" o caractere com maior código ASCII, que é atribuído em ordem alfabética, ou seja, o caractere "b" é maior que "a".
- Quando as *strings* forem diferentes e a primeira possuir o caractere, não coincidente e "menor" que a segunda, então o valor resultante é negativo. Por exemplo, o caractere "d" é menor que o "h".

A função `malloc(tamanho)` é muito utilizada em estruturas de dados para alocar espaços dinâmicos na memória de trabalho. O parâmetro "tamanho" se refere à quantidade de bytes se deseja alocar. Algumas sintaxes possíveis para essa função são:

1. `int x=0;`
2. `malloc(4);`
3. `malloc(sizeof(float));`
4. `malloc(sizeof(x));`

Na linha 2 serão alocados 4 *bytes* na memória. Na linha 3, será alocado o tamanho de uma variável do tipo *float*. Na linha 4 será alocado o tamanho da variável *x*, que é do tipo *int*, ou seja, 4 *bytes*.



Assimile

Cada função, independentemente da linguagem de programação, precisa ter um tipo de retorno, por exemplo, retornar um inteiro, um real, um booleano, um *void* que significa tipo genérico, dentre outros.

As funções de alocação dinâmica de memória, por exemplo, a `malloc()` possui como tipo de retorno um endereço de memória, por isso seu uso é comumente vinculado a ponteiros (lembrando que são variáveis especiais que armazenam endereços).

Com esta seção, finalizamos a segunda unidade, na qual exploramos as formas de armazenamento temporário de dados em diversos tipos de variáveis e como podemos utilizar os operadores para realizar o processamento dos dados. Nas próximas seções você aprenderá novas técnicas de programação, que lhe permitirão criar estratégias de processamento ainda mais complexas.

Sem medo de errar

Chegou o momento de realizar uma etapa importante no treinamento dos jovens programadores, pois você deverá ensiná-los a implementar o processamento de alguns dados estatísticos. Como entrada, você tem acesso aos dados de veículos registrados no estado de São Paulo nos anos de 2014, 2015 e 2016.

Foi solicitado o cálculo da média de veículos cadastrados em cada ano e, para isso, o primeiro passo consiste em armazenar os dados em variáveis. Nesse ponto, vale a pena ressaltar que não existe uma única maneira de implementar uma solução, já que cada programador tem seu estilo e suas técnicas de implementação.

Vamos realizar o armazenamento dos dados usando vetores, cada ano será um vetor com espaço para armazenar 3 valores, na posição zero serão armazenados os carros, na posição um as motos e na posição dois os caminhões. Portanto, crie três variáveis compostas, conforme exemplo:

```
int ano_2014[3]={0};
```

Vamos precisar de mais três variáveis para guardar a média de cada ano.

```
float media_2014 = 0;
```

```
float media_2015 = 0;
```

```
float media_2016 = 0;
```

Agora vamos inserir manualmente os valores, nos três vetores, conforme o comando:

```
ano_2014[0] = 16319979
```

```
ano_2014[1] = 4133366
```

```
ano_2014[2] = 658713
```

Fique à vontade para alterar o código para que o usuário digite os valores, em vez de fornecê-los manualmente. Caso opte por esse modo, use o comando:

```
scanf("%d", &ano_2014[0]);
```

Feita a inserção dos dados, agora podemos passar ao processamento das informações que, nesse caso, consiste em calcular a média aritmética. Portanto, vamos somar os dados de cada ano, dividir pelo número de ocorrências (três) e guardar dentro da variável média do respectivo ano, conforme código abaixo:

```
media_2014 = (ano_2014[0] + ano_2014[1] + ano_2014[2]) / 3;
```

Veja que o somatório está entre parênteses, e isso é necessário por que a divisão tem precedência sobre a soma.

Após ter efetuado o cálculo das médias, você pode usar operadores relacionais e lógicos para descobrir qual ano obteve a maior média. Por exemplo, para verificar se 2014 foi o ano que o obteve a maior média você pode usar o comando:

```
printf("2014 obteve a maior media? %d", ((media_2014 > media_2015) && (media_2014 > media_2016)));
```

Faça a impressão das médias e dos testes lógicos. Complete o código com os cálculos necessários e com as comparações necessárias, conforme os exemplos, execute e veja se todos os alunos estão com o programa funcionando.

Os operadores relacionais e lógicos podem ser trabalhados dentro de estruturas condicionais, na próxima unidade você aprenderá esse recurso e poderá aperfeiçoar suas implementações.

Avançando na prática

Calculo de desconto

Descrição da situação-problema

Uma pizzaria lhe procurou, pois gostaria de automatizar seu caixa. A princípio, foi lhe solicitado apenas implementar um cálculo simples, em que dado o valor total da conta de uma mesa, o programa divide esse valor pela quantidade de integrantes na mesa e calcula o desconto concedido. O programa deve receber como dados o valor da conta, a quantidade de pessoas e o percentual de desconto (%). Com os dados no programa, como deverá ser feito

o cálculo do valor total da conta com o desconto e valor que cada pessoa deverá pagar?

Resolução da situação-problema

O código no Quadro 2.22 apresenta o resultado do problema. Um ponto importante é o cálculo do desconto feito na linha 13, utilizamos uma regra de três simples para montar o desconto. Outro ponto é o cálculo do valor por pessoa, feito na linha 15 diretamente dentro do comando de impressão. Esse recurso pode ser usado quando não é preciso armazenar o valor.

Quadro 2.22 | Resolução do problema

```
1. #include<stdio.h>
2. main(){
3.     float valor_bruto=0;
4.     float valor_liquido=0;
5.     float desconto=0;
6.     int qtd_pessoas=0;

7.     printf("\n Digite o valor total da conta: ");
8.     scanf("%f",&valor_bruto);
9.     printf("\n Digite a quantidade de pessoas: ");
10.    scanf("%d",&qtd_pessoas);
11.    printf("\n Digite o desconto (em porcentagem): ");
12.    scanf("%f",&desconto);

13.    valor_liquido = valor_bruto - (valor_bruto *
14.        desconto/100);
15.    printf("\n Valor da conta com desconto =
16.    %f",valor_liquido);
17.    printf("\n Valor a ser pago por pessoa =
18.    %f",valor_liquido/qtd_pessoas);
19. }
```

Fonte: elaborado pela autora.

Faça valer a pena

1. Todo sistema computacional é construído para se obter alguma solução automatizada. Uma das áreas promissoras da computação é a mineração de dados, que, como o nome sugere, se refere a um determinado montante de dados e o modo como eles podem ser minerados para

gerar informações de valor. Dentro do processamento de informações, os operadores matemáticos, relacionais e lógicos são essenciais, pois são a base do processo.

Considerando o comando `resultado = a + b * (c - b) / a`, e os valores **a = 2, b = 3 e c = 5**. Escolha a opção correta.

- a) O valor em resultado será 5.
- b) O valor em resultado será 10.
- c) O valor em resultado será 6.
- d) O valor em resultado será 7.
- e) O valor em resultado será 8.

2. Considerando o comando `printf("%d", ((a > b) || (b < c) && (c < b)))`; , é correto afirmar que:

I – O resultado será um para a = 30, b = 20, c = 10

PORQUE

II – Para a expressão lógica proposta, basta que uma das condições seja verdadeira.

Assinale a alternativa correta

- a) As afirmações I e II são verdadeiras, e a segunda é uma justifica válida da primeira.
- b) As afirmações I e II são verdadeiras, mas a segunda não é uma justifica válida da primeira.
- c) Somente a afirmação I é verdadeira.
- d) Somente a afirmação II é verdadeira.
- e) As afirmações I e II não são verdadeiras.

3. Algumas soluções em *software* necessitam de alocação dinâmica de memória. Nesses casos, é comum utilizar estruturas de dados chamadas de listas, filas, pilhas, árvores e grafos (EDELWEISS; GALANTE, 2009). Para a alocação dinâmica, usam-se funções da biblioteca `<stdlib.h>`, tais como a função `malloc(tamanho)`.

1. `#include<stdlib.h>`
2. `struct automovel{`
3. `char modelo[20];`

```
4.     int ano;
5. };
6. main(){
7.     struct automovel auto1;
8.     malloc(sizeof(auto1));
9. }
```

Considerando o código apresentado, escolha a alternativa correta:

- a) Esse código não pode ser executado.
- b) A função `malloc()` alocará 4 bytes na memória.
- c) A função `malloc()` alocará 20 bytes na memória.
- d) A função `malloc()` alocará 24 bytes na memória.
- e) A função `malloc()` alocará 0 bytes na memória.

Referências

- DEITEL, P.; DEITEL, H. C **Como Programar**. 6. ed. São Paulo: Pearson, 2011.
- GERSTING, J. L. **Fundamentos matemáticos para a ciência da computação: matemática discreta e suas aplicações**. Rio de Janeiro: LTC, 2017.
- MANZANO, J. A. N. G. **Linguagem C: acompanhada de uma xícara de café**. São Paulo: Érica, 2015. 480 p.
- MANZANO, J. A. N. G.; MATOS, E.; LOURENÇO, A. E. **Algoritmos: Técnicas de Programação**. 2. ed. São Paulo: Érica, 2015.
- PEREIRA, S. L. **Linguagem C**. [S.l.; s.d.]. Disponível em: <<https://www.ime.usp.br/~slago/slago-C.pdf>>. Acesso em: 23 jul. 2018.
- PIVA JUNIOR, D. et al. **Algoritmos e Programação de Computadores**. Rio de Janeiro: Elsevier, 2012.
- SOFFNER, R. **Algoritmos e programação em linguagem C**. São Paulo: Saraiva, 2013.
- TARIFA, A. **O que é Big Data?** [S.l.; s.d.]. Disponível em: <<https://endeavor.org.br/big-data-descubra-o-que-e-e-como-usar-na-sua-empresa/>>. Acesso em: 23 jul. 2018.

Estruturas de decisão e repetição

Convite ao estudo

Podemos observar que tudo que realizamos em nossa vida envolve uma condição, certo! **Se** você decidiu estudar programação, **então** é porque, de alguma forma, isso vai contribuir para sua formação profissional, **se não** estudaria outro curso ou, até mesmo, seguiria outros rumos. Compreende? As condicionais estão por toda parte, seja nos fornecendo uma solução ou várias possibilidades de solução.

Nesta unidade, vamos nos apegar a uma famosa instituição de ensino, que é grande referência no mercado educacional, para a qual você presta serviço por meio da empresa de software onde você foi eleito o funcionário do mês, tendo como diferencial o treinamento que realizou com os estagiários. Em reconhecimento ao seu trabalho lhe foi concedido o direito de efetivar um dos estagiários. Agora, é o momento de dar sequência ao treinamento e se empenhar ao máximo para garantir o aprendizado do seu novo funcionário que, até então, era estagiário. Ele já conhece os conceitos de programação, o trabalho com algoritmos, variáveis e tipos de dados.

O próximo passo será começar a trabalhar com a linguagem de programação C, e, para isso, nada melhor que começar pelas estruturas de decisões condicionais (Seção 3.1), avançando para as estruturas de repetição condicional (Seção 3.2) e finalizando com as estruturas de repetições determinísticas (Seção 3.3).

É certo que a programação condicionada passará a ser uma oportunidade de otimização e melhoramento das rotinas da instituição de ensino para a qual você e seu estagiário

prestarão serviço. Assim como um grande maratonista se prepara para uma corrida, você deverá realizar vários exercícios de programação para chegar à frente.

Força e um ótimo estudo!

Seção 3.1

Estruturas de decisão condicional

Diálogo aberto

Caro aluno, certamente, você está preparado para o próximo passo na programação de computadores e já estudou os conceitos de algoritmos, linguagem de programação, tipos de dados, constantes, variáveis, operadores e expressões, portanto, agora é o momento de avançar. Nesta seção, você irá estudar as estruturas de decisão condicional e de seleção, e nada melhor que começar com uma analogia, certo?

Um grande evento será realizado na sua cidade e seu cantor favorito fará uma apresentação. Você, sortudo que é, ganhou um convite para uma visita ao camarote do cantor e, em seguida, assistir ao show junto à banda do cantor. Porém, nem tudo são flores! Você foi escalado pela empresa em que trabalha para a implantação de um sistema de computador em um dos seus clientes. Para que o tempo seja suficiente e você possa usufruir do seu mimo, foram lhe oferecidas as seguintes condições: se você instalar o sistema até as 12h, poderá realizar o treinamento no período da tarde e, então, ir ao show; se não, terá que agendar o treinamento para o final do dia e comprometer a sua ida ao show.

Veja que é uma analogia simples de estrutura de decisão condicional, poderíamos ainda criar alguns casos que proporcionariam a sua ida ao show. Diante de tal situação, voltaremos as nossas atenções à instituição de ensino em que você e seu funcionário (ex-estagiário) prestarão serviço por meio da empresa de software onde vocês trabalham. A instituição de ensino está passando por um processo de otimização nas suas atividades e colocou o seguinte desafio para vocês: realizarem um programa em linguagem C que calculasse o valor do salário bruto, levando em consideração os descontos de INSS e Imposto de Renda (Tabelas 3.1 e 3.2). Para efeito de documentação, ao final da compilação do programa, devem realizar um relatório com o código fonte.

Tabela 3.1 | Descontos INSS

SALÁRIO DE CONTRIBUIÇÃO (R\$)	ALÍQUOTA / INSS
até 1.693,72	8%
de 1.693,73 até 2.822,90	9%
de 1.693,73 até 2.822,90	9%
de 2.822,91 até 5.646,80	11%
Acima de 5.646,80	R\$ 621.04 (invariavelmente)

Fonte: elaborada pelo autor.

Tabela 3.2 | Descontos IR

SALÁRIO (R\$)	ALÍQUOTA / IR
Até 1.903,98	–
De 1.903,99 até 2.826,65	7,5%
De 2.826,66 até 3.751,05	15,0%
De 3.751,06 até 4.664,68	22,5%
Acima de 4.664,68	27,5%

Fonte: elaborada pelo autor.

Diante dessas informações, cabe uma pergunta: a estrutura de decisão nos garante a viabilidade nas mais diferentes condicionais de programação? Ou, ainda, existem formas mais rápidas para programar condicionais?

Muito bem, aproveite ao máximo seus estudos e boa sorte!

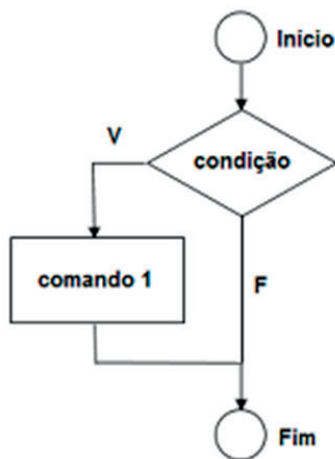
Não pode faltar

Caro aluno, agora que você já estudou os tipos de variáveis, os tipos de dados, as operações e as expressões para composição de um programa de computador, chegou o momento de trabalhar as estruturas de decisão e seleção. Pois bem, vamos iniciar a seção com a estrutura de decisão condicional: if/else (se/então).

Segundo Manzano (2013), para a solução de um problema envolvendo situações, podemos utilizar a instrução "if", em português "se", cuja função é tomar uma decisão e criar um desvio dentro do programa, para que possamos chegar a uma condição verdadeira ou falsa. Lembrando que a instrução pode receber valores em ambos os casos.

Na linguagem de programação C, utilizamos chaves ("{" e "}") para determinar o início e fim de uma instrução. Veja na Figura 3.1 como fica a estrutura condicional simples utilizando fluxograma:

Figura 3.1 | Fluxograma representando a função "if".



Fonte: elaborada pelo autor.

Na sequência, veja a sintaxe da instrução "if" (se) utilizada na linguagem C:

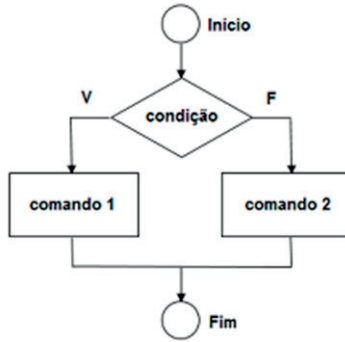
```
if <(condição)>
{
<conjunto de comandos>;
}
```

Muito bem, nada melhor do que visualizar uma aplicação condicional na prática. No exemplo abaixo, usaremos uma aplicação de condicional simples, lembrando que será executado um teste lógico, que, se o resultado for verdadeiro, então ela trará uma resposta, caso contrário não retornará nada. Veja no exemplo abaixo, a situação de um jovem que verifica se poderá ou não tirar a carteira de habilitação:

```
1. int main()
2. {
3.     float idade;
4.     printf("Digite sua idade: \n");
5.     scanf("%f", &idade);
6.     if (idade>=18)
7.     {
8.         printf("Voce ja pode tirar sua carteira de
9.     Habilidade, voce e maior de 18");
10.    }
11.    return 0;
12. }
```

Nesse exemplo, não é considerado o "se não" (*else*). Simplesmente, se a condição não for verdadeira, ela não exibirá nada como resposta. Agora, veremos abaixo a **estrutura condicional composta**, que completa a nossa condição inicial com o comando "else", que significa "se não". Vejamos como fica a estrutura no fluxograma da Figura 3.2:

Figura 3.2 | Fluxograma representando as funções "if" e "else".



Fonte: elaborada pelo autor.

Agora, veja a representação da sintaxe:

```
if <(condição)>
{
<primeiro conjunto de comandos>;
}
else
{
<segundo conjunto de comandos>;
}
```

Vamos, agora, criar outra situação para a estrutura condicional composta em linguagem C: Maria e João estão se preparando para uma viagem, porém, se o orçamento final deles for igual ou maior que R\$ 10.000,00 eles farão uma viagem internacional, se não deverão fazer uma viagem nacional.

```
1. #include <stdio.h>
2. int main() {
3. float orcamento;
```

```

4. printf("Digite o valor do orcamento para viagem \n");
5. scanf("%f", &orcamento);
6. if (orcamento >=10000)
7. {
8.     printf("\n Joao e maria possuem orçamento para uma
9. viagem internacional, pois seu orcamento e de %f",
10. orcamento);
11. }
12. else
13. {
14.     printf("\n Joao e Maria irão optar por uma viagem
15. nacional, seu orçamento ficou em %f", orcamento);
16. }
17. return 0;
18. }

```

Melhorou o pensamento? Ficou bem mais estruturado, e o comando "else" possibilitou um retorno à condicional "if".



Exemplificando

Para reforçar o seu conhecimento, vamos ver o exemplo abaixo em linguagem de programação C, que retorna se o valor de um número digitado é positivo ou negativo, representando uma estrutura condicional composta:

```

#include <stdio.h>
#include <stdlib.h>

int main() {

```

```

int num;

printf ("Digite um numero: ");

scanf ("%d",&num);

if (num>0)
    {
        printf ("\n\n0 numero e positivo\n");
    }

else
    {
        printf ("0 numero e negativo");
    }

return 0;
}

```

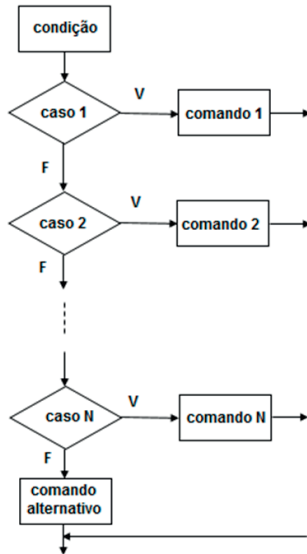
Dando sequência aos estudos, vamos conhecer a **Estrutura Condicional de Seleção de Casos**, “*switch-case*”, que, segundo Schildt (1997, p. 35), “testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere”. Quando os valores são avaliados o comando é executado.

Devemos estar atentos a algumas particularidades para o comando *switch-case*:

- Caso nenhum dos valores seja encontrado, o comando default será executado.
- Os comandos são executados até o ponto que o comando break for localizado.

Veja na Figura 3.3 o fluxograma representando a estrutura condicional de seleção de casos:

Figura 3.3 | Fluxograma de estrutura condicional de seleção de casos



Fonte: elaborada pelo autor.

Vamos ver como fica a sintaxe em linguagem C:

```
switch (variável)
{
case constante1:
<comandos>
break;
case constante2:
<comandos>
break;
default: <comandos>
}
```

Veja que o comando `break` é utilizado para forçar a saída do laço de repetição, ou seja, ele sai do comando sem executar as próximas instruções. Caso não seja colocado o comando `break`, o programa continua e averigua o próximo caso até o fim do `switch` ou até encontrar um `break`. Para fixar o que está sendo estudado, vamos aplicar um exemplo, que tem a finalidade de descobrir o desconto que um cliente terá, de acordo com a escolha de uma cor específica:

```
1.  #include <stdlib.h>
2.  int main() {
3.      char x;
4.      float valor,desc, total;
5.      printf("\n Digite o valor da compra \n");
6.      scanf("%f", &valor);
7.      printf("\n Digite a letra que representa o seu
8.  desconto de acordo com a cor\n");
9.      printf("a. azul\n");
10.     printf("v. vermelho\n");
11.     printf("b. branco\n");
12.     printf(" Digite sua opcao:");
13.     scanf("%s", &x);
14.     switch(x)
15.     {
16.     case 'a':
17.         printf("Voce escolheu azul, seu desconto sera
18.     de 30 por cento \n");
19.         desc=valor*0.30;
```

```

20.         total=valor-desc;
21.         printf("O valor da sua compra e %.2f\n",
22. total);
23.         break;
24.     case 'v':
25.         printf("Voce escolheu vermelho, seu desconto
26. sera de 20 por cento \n");
27.         desc=valor*0.20;
28.         total=valor-desc;
29.         printf("O valor da sua compra e %.2f\n",
30. total);
31.         break;
32.     case 'b':
33.         printf("Voce escolheu branco, seu desconto
34. sera de 10 por cento \n");
35.         desc=valor*0.10;
36.         total=valor-desc;
37.         printf("O valor da sua compra e %.2f\n",
total);
        break;
        default:
            printf("opcao invalida\n");
        }
return 0;
}

```




Para não perder o ritmo dos estudos, vamos relembrar os operadores lógicos e a tabela verdade:

Quadro 3.1 | Operadores lógicos

Operadores	Função
!	Negação – NOT
$\&$	Conjunção – AND
	Disjunção Inclusiva – OR

Fonte: elaborado pelo autor.

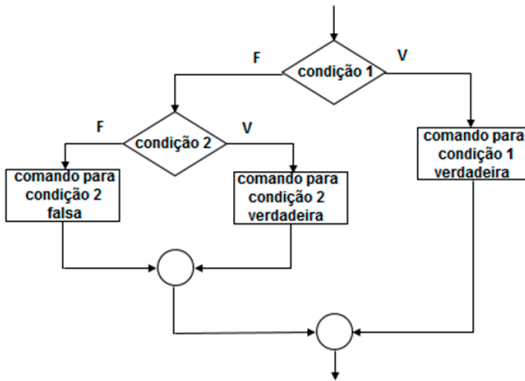
Quadro 3.2 | Tabela Verdade

A	B	$A \& B$	$A B$! A	! B
Verdade	Verdade	Verdade	Verdade	Falso	Falso
Verdade	Falso	Falso	Verdade	Falso	Verdade
Falso	Verdade	Falso	Verdade	Verdade	Falso
Falso	Falso	Falso	Falso	Verdade	Verdade

Fonte: elaborada pelo autor.

Para finalizar a seção que trata das estruturas de decisão e seleção, iremos entender o funcionamento da **estrutura condicional encadeada**, também conhecida como **ifs aninhados**. Segundo Schildt (1997), essa estrutura é um comando if que é o objeto de outros if e else. Em resumo, um comando else sempre estará ligado ao comando if de seu nível de aninhamento. Veja na Figura 3.4 um dos tipos de fluxogramas que representa uma estrutura condicional encadeada.

Figura 3.4 | Fluxograma estrutura condicional encadeada



Fonte: elaborada pelo autor.

Podemos caracterizar a sintaxe de uma estrutura condicional encadeada da seguinte forma:

```
if (condição) comando;  
  
else  
    if (condição) comando;  
    else(condição) comando;  
    .  
    .  
    .  
  
else comando;
```



Pesquise mais

Assista à videoaula que fala sobre as estruturas condicionais encadeadas, ou seja, como montar um programa utilizando o desvio condicional aninhado.

BÓSON TREINAMENTOS. **13** – Programação em Linguagem C – Desvio Condicional Aninhado – if / else if. 27 fev. 2015. Disponível em: <<https://www.youtube.com/watch?v=7ZL8tHLTffs>>. Acesso em: 26 jul. 2018.

Agora, veja no programa abaixo uma estrutura condicional encadeada, em que serão analisados os tipos de triângulo, partindo da premissa que ele deverá ser testado antes para verificar se forma ou não um triângulo:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main( ) {
4.  int a, b, c;
5.  printf("Classificacao do triangulo: informe a medida dos
6.  lados apertando a Tecla ENTER para cada medidia:\n");
7.  scanf("%d %d %d", &a, &b, &c);
8.  if (a< b + c && b< a + c && c < a + b)
9.      {
10.         printf("\n\n Dadas as medidas: %d, %d, %d,
11. temos um triangulo", a, b, c);
12.         if( a == b && a == c)
13.             {
14.                 printf("Este e um triangulo EQUILATERO! \n");
15.             }
16.         else
17.             if ( a==b || a == c || b ==c)
18.                 {
19.                     printf("Este e um triangulo
20. ISOSCELES!\n");
21.                 }
```

```

22.                                     else
23.                                     printf("Este e um triangulo
24. ESCALENO! \n");
25.     }
26.     else
27.         printf("\n\n As medidas fornecidas, %d,%d,%d
28. nao formam um triangulo", a, b, c);
        return 0;
    }

```



Refleta

Pense nas possibilidades que você pode ter usando as estruturas de tomadas de decisão. "if-else", if-else-if e "switch-case". Lembre-se que para cada caso poderá haver uma particularidade diferente em desenvolver um programa. Imagine se você tivesse que criar um programa que calculasse o desempenho dos professores, de acordo com as suas produtividades, como você faria?

Quanta informação e quantas possibilidades foram criadas nesta seção, porém, não pense que acabou. Estamos apenas começando. Bons estudos e até a próxima seção!

Sem medo de errar

Agora que você já conhece as estruturas de decisão condicional e de seleção, chegou o momento de verificar se realmente o seu funcionário (ex-estagiário) conseguiu resolver o programa em linguagem C, que calcula o valor do salário líquido, levando em consideração os descontos de INSS e Imposto de renda.

Veja as Tabelas 3.1 e 3.2 novamente e resolva o problema utilizando a estrutura de decisão condicional. Vamos lá!

Tabela 3.1 | Descontos INSS

SALÁRIO DE CONTRIBUIÇÃO (R\$)	ALÍQUOTA / INSS
até 1.693,72	8%
de 1.693,73 até 2.822,90	9%
de 2.822,91 até 5.646,80	11%
Acima de 5.646,80	R\$ 621.04 (invariavelmente)

Fonte: elaborada pelo autor.

Tabela 3.2 | Descontos IR

SALÁRIO (R\$)	ALÍQUOTA / IR
Até 1.903,98	-
De 1.903,99 até 2.826,65	7,5%
De 2.826,66 até 3.751,05	15,0%
De 3.751,06 até 4.664,68	22,5%
Acima de 4.664,68	27,5%

Fonte: elaborada pelo autor.

```
1. #include "stdio.h"
2. #include "stdlib.h"
3. int main()
4. {
```

```
5.     float salario, inss, ir, sal_liquido;
6.
7.     printf("Calculo de Salario Liquido Com desconto do IR
8. e INSS\n\n");
9.     printf("\nDigite seu salario Bruto\n");
10.    scanf_ s("%f", &salario);
11.        //Calcular o INSS
12.    if ( salario <= 1693.72)
13.    {
14.        inss = salario * 0.08;
15.    }
16.    else
17.    if ( salario >= 1693.73 && salario <= 2822.90)
18.    {
19.        inss = salario * 0.09;
20.    }
21.    else
22.    if ( salario >= 2822.91 && salario <= 5645.80)
23.    {
24.        inss = salario * 0.11;
25.    }
26.    else
27.    {
28.        inss = 621.04;
29.    }
```

```
30.     //Calcular i IR
31.     if ( salario <= 1903.98 )
32.     {
33.         ir = salario*0;
34.     }
35.     else
36.         if ( salario >= 1903.99 && salario <= 2826.65)
37.         {
38.             ir = salario * 0.075;
39.         }
40.     else
41.         if ( salario >= 2826.66 && salario <= 3751.05 )
42.         {
43.             ir = salario * 0.15;
44.         }
45.     else
46.         if ( salario >= 3751.06 && salario <= 4664.68 )
47.         {
48.             ir = salario * 0.225;
49.         }
50.     else
51.         if ( salario > 4664.69 )
52.         {
53.             ir = salario * 0.275;
54.         }
```

```

55. //Calculo do Salario liquido
56. sal_liquido = (salario - inss) - ir;
57. //Resultados
58.         printf( "\nDesconto do INSS e: %.2f\n\n",
59. inss);
60.         printf( "Desconto do imposto de renda e:
61. %.2f\n\n", ir);
           printf( "Salario liquido: %.2f\n\n",
sal_liquido);
return 0;
}

```

Essa é uma das formas de se chegar ao resultado, mas você ainda pode acrescentar mais variáveis ao programa e, assim, calcular os dependentes, convênios e outros descontos, e até mesmo os benefícios na sua folha de pagamento. Treine bastante, otimize ao máximo e ótimos estudos!

Avançando na prática

Semana do desconto

Descrição da situação-problema

Na dinâmica do dia a dia de uma pizzaria, você resolveu realizar um programa em linguagem C para que em cada dia da semana fosse ofertado um desconto aos seus clientes. Seria mais o menos assim: na segunda-feira, o desconto seria de 30% no valor da pizza; na terça, 40%; na quarta, a pizza é em dobro; na quinta, 20% de desconto; na sexta, 10%; no sábado não haverá desconto; e no domingo, ganha-se o refrigerante. Existem várias formas de

criar esse programa em linguagem C, certo? Qual maneira você escolheria par criar esse programa?

Resolução da situação-problema

Para resolver o caso dos descontos da pizzeria, veja abaixo uma das opções que você pode adotar. É claro que existem outras formas de chegar no mesmo resultado e até mesmo com uma melhor otimização.

```
1.  #include <stdio.h>
2.  int main ( )
3.  {
4.      int compra;
5.      printf ("Digite um numero que corresponde o dia semana
6.  e seu respectivo desconto \n");
7.      printf("Digite 1 para Domingo\n");
8.      printf("Digite 2 para Segunda\n");
9.      printf("Digite 3 para Terca\n");
10.     printf("Digite 4 para Quarta\n");
11.     printf("Digite 5 para Quinta\n");
12.     printf("Digite 6 para Sexta\n");
13.     printf("Digite 7 para Sabado\n");
14.     scanf("%d", &compra);
15.     switch ( compra )
16.     {
17.         case 1 :
18.             printf ("Domingo e dia de refri gratis\n");
19.             break;
```

```
20.         case 2 :
21.             printf ("Segunda o desconto sera de 40 por cento
22. no valor da pizza\n");
23.         break;
24.         case 3 :
25.             printf ("Terca o desconto sera de 30 por cento
26. no valor da pizza\n");
27.         break;
28.         case 4 :
29.             printf ("Quarta e dia de pizza em dobro \n");
30.         break;
31.         case 5 :
32.             printf ("Quinta o desconto sera de 20 por cento
33. no valor da pizza \n");
34.         break;
35.         case 6 :
36.             printf ("Sexta o desconto sera de 10 por cento no
37. valor da pizza\n");
38.         break;
39.         case 7 :
40.             printf ("Sabado nao tem desconto\n");
41.         break;
42.         default :
43.             printf ("\n O valor digitado nao corresponde a
nenhum dia da semana\n");
```

```
    }  
  
    getch();  
  
    return 0;  
  
}
```

Pois bem, agora é sua vez de treinar um pouco. Boa sorte e ótimo estudo.

Faça valer a pena

1. Podemos dizer que o comando “else” é uma forma de negar o que foi colocado em uma situação do comando “if”. Sendo assim, “else” é o caso contrário do comando “if”.

Funciona da seguinte forma:

```
if <(condição)>  
{  
  
<conjunto de comandos>;  
  
}  
  
else  
  
{  
  
<conjunto de comandos>;  
  
}
```

Assinale a alternativa que melhor se compõe à contextualização acima:

- a) Para cada “else” é necessário um “if” anterior, no entanto, nem todos os “ifs” precisam de um “else”.
- b) Para cada “else” é necessário um “if” anterior, sendo assim, todos os “ifs” precisam de um “else”.
- c) Vários “ifs” precisam de um único “else” dentro de uma condição.

- d) Para cada "if" é necessário um "else" para completar uma condição.
e) Podemos dizer que o comando "else" é a afirmação de um comando "if".

2. A estrutura condicional encadeada, também é conhecida como ifs aninhados, segundo Schildt (1997), é um comando if que é o objeto de outros if e else. Em resumo, um comando else sempre estará ligado ao comando if de seu nível de aninhamento.

Assinale a alternativa que corresponde à sintaxe da estrutura condicional encadeada:

- a) `if (condição) comando;
 if (condição) comando;
 else(condição) comando;
 else comando;`
- b) `if (condição) comando;
 else
 if (condição) comando;
 else(condição) comando;
 else comando;`
- c) `if (condição) comando;
 else
 if (condição) comando;`
- d) `else (condição) comando;
 if (condição) comando;
 else(condição) comando;`
- e) `if (condição) comando;
 else
 else (condição) comando;
 else(condição) comando;
 else comando;`

3. A Estrutura Condicional de Seleção de Casos, "*switch-case*", segundo Schildt (1997, p. 35) "testa sucessivamente o valor de uma expressão contra uma lista de constantes inteiras ou de caractere", ou seja, quando os valores são avaliados o comando é executado.

Levando em consideração a estrutura condicional de seleção utilizando casos, qual a principal função dos comandos *default* e *break*? Assinale a alternativa correta:

- a) O comando *default* é executado quando nenhum dos valores é executado, porém, não é necessariamente obrigatório, já o comando *break* determina o fim de uma das opções de comando.
- b) O comando *default* é executado quando nenhum dos valores é executado, já o comando *break* determina o início de uma das opções de comando.
- c) O comando *default* é executado para iniciar um conjunto de comandos, já o comando *break* determina o fim de uma das opções de comando.
- d) O comando *default* é executado no início das condições de valores, já o comando *break* determina o início de uma das opções de comando.
- e) O comando *default* é executado quando nenhum dos valores é executado, já o comando *break* determina o fim de uma das opções de comando.

Seção 3.2

Estruturas de repetição condicional

Diálogo aberto

Caro aluno, chegamos a mais um desafio do nosso curso, em que você terá a oportunidade de estudar as estruturas de repetição condicional `While` e `Do/While`, e seus comparativos e aplicações.

Assim como as estruturas decisão, as estruturas de repetição têm a função de otimizar as soluções de problemas. Imaginemos que você decidiu distribuir cinco livros de computação ao final de um evento; a estrutura de repetição, por exemplo, ficaria assim: enquanto o número de pessoas for menor que cinco, você entregará um livro, depois a distribuição será encerrada. Veja que a expressão “enquanto” foi utilizada no início da nossa frase. Fácil! Vamos criar uma outra situação: no final do evento, você irá distribuir cinco livros de computação para os primeiros que chegarem com a solução de um problema proposto. Veja que várias pessoas terão a oportunidade de resgatar o livro, porém, a quantidade de livros ficou condicionada ao final. Tanto a primeira situação como a segunda estão condicionadas à estrutura de repetição, mas uma realiza o teste no início e a outro no final.

Pois bem, agora você deverá auxiliar o seu atual funcionário (ex-estagiário) a criar um programa em linguagem C para ajudar a instituição de ensino para a qual vocês prestam serviço. Foi solicitada a elaboração de um programa que receba as notas de um semestre, de uma determinada disciplina. O professor poderá realizar quantas avaliações achar necessário para a composição da nota do aluno e, por fim, deverá apresentar a média final deste aluno.

Pense nas soluções e execute o código em um compilador de linguagem C. Apresente o código livre de erros em um documento de texto. Nesse caso específico, qual a melhor solução: usar um teste de repetição no início ou no final? Faça o teste e otimize ao máximo as possíveis soluções para o problema.

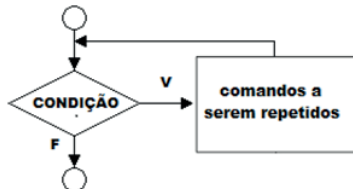
Boa sorte e uma ótima aula!

Não pode faltar

Após trabalhar as estruturas condicionais, chegou o momento de encarar o desafio de estudar as estruturas de repetição. Segundo Manzano (2013), para a solução de um problema, é possível utilizar a instrução “if” para tomada de decisão e para criar desvios dentro de um programa para uma condição verdadeira ou falsa. Seguindo essa premissa, vamos iniciar nossos estudos com as **repetições com teste no início – while**. É preciso estar ciente que algo será repetidamente executado enquanto uma condição verdadeira for verificada, somente após a sua negativa essa condição será interrompida.

Segundo Soffner (2013 p. 64), o programa “não executará nenhuma repetição (e as ações que ali dentro estiverem programadas) sem antes testar uma condição”. Na realização dessa condição, vamos fazer uso do comando iterativo “while”, que significa “enquanto” em português. Veja na Figura 3.4 a forma simplificada do fluxograma do comando *while* direcionado para o teste no início.

Figura 3.4 | Fluxograma do comando *while*



Fonte: elaborada pelo autor.

Como o programa será elaborado em linguagem C, veja a seguir a sintaxe com a repetição com teste no início:

```
while
(<condição>)
{
Comando 1;
Comando 2;
Comando n;
}
```

Em alguns casos, quando utilizamos um teste no início, pode ocorrer o famoso *loop* (laço) infinito (quando um processo é executado repetidamente). Para que isso não aconteça, você poderá utilizar os seguintes recursos:

- *Contador* – é utilizado para controlar as repetições, quando são determinadas.
- *Incremento e decremento* – trabalham o número do contador, seja aumentando ou diminuindo.
- *Acumulador* – segundo Soffner (2013), irá somar as entradas de dados de cada iteração da repetição, gerando um somatório a ser utilizado quando da saída da repetição.
- *Condição de parada* – utilizada para determinar o momento de parar quando não se tem um valor exato desta repetição.



Refleta

Perceba que, quando aplicamos um laço, várias instruções podem ser aplicadas, até mesmo um laço dentro de outro laço. Que nome damos a esse laço? Lembre-se também que podem ocorrer os laços infinitos, ou seja, as condições são sempre verdadeiras.

Veja no exemplo abaixo uma aplicação do comando *while* em um teste no início, que deverá mostrar a palavra "PROGRAMA" dez vezes:

```
#include <stdio.h>

#include <stdlib.h>

main()

{

    int cont=0; // foi definido na declaração da variável um
valor inicial de "0"

    while (cont < 10) // Será executado enquanto a cont for
menor que 10
```



```

    {
        printf("PROGRAMA \n");

        cont++; // será necessário incrementar um valor, para
dar sequência no programa

    }

    system("PAUSE");

    return 0;
}

```

O próximo exemplo é para checar se um número está entre um número escolhido e outro.

```

#include <stdio.h>

int main(void)
{
    char PARAR;

    int NUM;

    printf("\nDigite um Numero: ");

    scanf("%d", &NUM);

    while ((getchar() != '\n') && (!EOF)); // getchar() != '\n'
está comparando esse caractere de leitura com o caractere de
nova linha.

// EOF é um valor
especial, para garantir que ele não continue tentando ler quando
chegar ao final do arquivo.

    if (NUM >= 10 && NUM <= 50)

```

```

printf("O numero esta entre 10 e 50.\n");

else

printf("O numero nao esta entre 10 e 50.\n");

printf("\n");

printf("aperte <Enter> para parar... ");

PARAR = getchar();

return 0;

}

```

Vale ressaltar que a instrução `while ((getchar() != '\n') && (!EOF))` está sendo apresentada, sendo o operador relacional diferente de `(!=)`.

Agora, vamos aplicar as repetições com testes no final (*do-while*), Segundo Schildt (1997), o laço "*do-while*" analisa a condição ao final do laço, ou seja, os comandos são executados antes do teste de condição. Nesse caso específico, o usuário tem a possibilidade de digitar novamente uma nova informação.



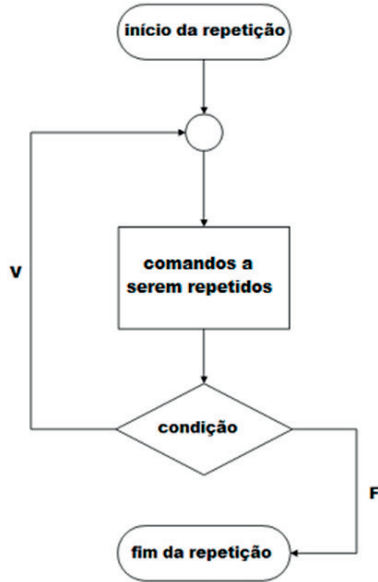
Pesquise mais

O comando *do-while* pode ter várias aplicações. Veja o vídeo no youtube "feito de aluno para aluno" sobre esse tema:

DE ALUNO PARA ALUNO. **Programar em C** - Como Utilizar "do while" - Aula 13. 24 out. 2012. Disponível em: <<https://www.youtube.com/watch?v=bjMD1QSVV-s>>. Acesso em: 26 jul. 2018.

Vamos ao fluxograma utilizando o teste de repetição no final:

Figura 3.5 | Fluxograma com teste de repetição no final



Fonte: elaborada pelo autor.

Veja como fica a sintaxe para realização da repetição com teste no final:

```
Do
{
comandos;
}
while (condição);
```

O exemplo abaixo realiza um programa que calcula a metragem quadrada de um terreno, usando o teste no final para criar a opção de digitar novos valores sem sair do programa:

```
#include <stdio.h>

main() {

float metragem1,metragem2,resultado;
```

```

int resp;

metragem1 = 0;

metragem2 = 0;

resultado = 0;

do

{

    printf("C A L C U L O   D E   M E T R O S   Q U A D R A D
O S");

    printf("\n \n Digite a primeira metragem do terreno: \n");

    scanf("%f",&metragem1);

    printf("\n Digite a segunda metragem do terreno: \n");

    scanf("%f",&metragem2);

    resultado = (metragem1 * metragem2);

    printf("\n \n O Terreno tem = %.2f M2 \n",resultado);

    printf("Digite 1 para continuar ou 2 para sair\n");

    scanf("%d", &resp);

}while (resp==1);

return 0;

}

```



Exemplificando

Segundo Soffner (2013), quando aplicado o comando do-while, as ações poderão ser executadas pelo menos uma vez antes do teste condicional. Nesse caso, é possível ter uma sequência de repetição de processos, sendo assim, é possível ter várias opções para a solução

de um problema em um programa. No exemplo abaixo, você terá a opção de calcular a força digitando a massa e aceleração e poderá, também, calcular a área e o perímetro de um círculo, digitando o raio.

```
#include <stdio.h>

#include <stdlib.h>

main()

{

    char material[60];

    float n, m, a;

    float raio, area, perimetro, pi;

    int opcao;

    do

    {

        printf("\t\t\n OPCOES DE CALCULOS

\n");

        printf("\n 0. SAIR DO MENU \n");

        printf("\n 1. CALCULAR A FORCA \n");

        printf("\n 2. CALCULAR , AREA E

PERIMETRO DO CIRCULO \n");

        printf("\n 3. RETORNAR AO MENU \n");

        printf("\n Opcao: ");

        scanf("%d", &opcao);

        switch( opcao )

        {

            case 0:
```

```

        printf("SAIR...\n");

        break;

    case 1:

        printf ("\n Digite a
massa do objeto: ");

        scanf("%f", &m);

        printf ("\n Digite a aceleracao: ");

        scanf("%f", &a);

        n=(m*a);

        printf("\n O calculo da forca e: %.2f
\n", n);

        break;

    case 2:

        printf("Digite o raio:
");

        scanf("%f", &raio);

        pi = 3.141592;

        area = pi*(raio * raio);

        perimetro = 2.0 * pi * raio;

        printf(" \n Raio: %.2f \n", raio);

        printf(" \n Area: %.2f \n", area);

        printf(" \n Perimetro: %.2f \n",
perimetro);

        break;

```

```

        case 3: system("cls");

                break;

        default:

                printf("OPÇÃO INVALIDA

\n");

        }

    } while(opcao);

    return 0;

}

```

Para dar continuidade aos nossos estudos, vamos considerar um problema utilizando conjectura de Collatz: tomando um número natural "n", se n for par, será dividido por 2; se n for ímpar, será multiplicado por 3 e ao resultado será somado 1. Repete-se o processo indefinidamente. A Conjectura de Collatz estabelece que, com essas duas regras simples, todos os números naturais chegam até 1 (e a partir daí o laço 1,4,2,1,4,2,1,4,2,1... se repete indefinidamente). Matematicamente, as regras são:

$(n) = n/2$ se n é par

$(n) = 3n + 1n$ se n é ímpar

Vamos, então, fazer um programa que calcula todos os números da sequência de Collatz para uma entrada qualquer. A sequência de comandos é:

- Entrar com um número inteiro positivo superior a 1.
- Se o número for par, dividir por dois,
- Se o número for ímpar, multiplicar por três e somar 1
- Pela conjectura de Collatz, a sequência sempre termina em 1, como já foi explanado.

Solicitar o número cuja sequência de Colatz será calculada.

Pois bem, agora veja a sua execução em linguagem C:

```

#include <stdio.h>

#include <stdlib.h>

int main()

{

    int num,i;

    printf("\n\nDIGITE UM NUMERO PARA O PROBLEMA DE L.
COLLATZ:\n");

    scanf("%d",&num);

    i=0;

    while(num>1)

    {   if(num%2==0)

        num=num/2;

        else

        num=3*num+1;

    printf("\n%d\n", num);

    i++;

    }

    return 0;

}

```



Assimile

Algumas variáveis podem sofrer alterações baseadas nos seus valores anteriores. Para facilitar, você pode utilizar o que chamamos de atribuição composta, que indica qual operação será realizada. Nesse caso, coloca-se o operador à esquerda do sinal de atribuição. Ex: $y*=x+1$, que tem o mesmo efeito que $y=y*(x+1)$, neste caso evitando colocar a variável a direita da atribuição.

Na sequência dos nossos estudos, vamos trabalhar algumas aplicações das estruturas de repetição condicional, realizando um programa que simula uma conta bancária (tela de opções das transações), adaptado do livro do Soffner (2013). Ele escreve um programa que repete uma entrada de dados até que determinada condição de saída seja atingida e, em seguida, acumule os valores digitados.

```
#include <stdio.h>

#include <stdlib.h>

main()

{

float soma=0;

float valor;

int opcao;

do {

    printf("\n Digite uma Operacao");

    printf("\n 1. Deposito");

    printf("\n 2. Saque");

    printf("\n 3. Saldo");

    printf("\n 4. Sair");

    printf("\n Opcao? ");

    scanf("%d", &opcao);

    switch(opcao) {

        case 1: printf("\n Valor do deposito? ");
```

```

scanf("%f", &valor);

soma=soma+valor;

break;

case 2: printf("\n Valor do saque? ");

scanf("%f", &valor);

soma=soma-valor;

break;

case 3: printf("\n Saldo atual = R$ %.2f \n", soma);

break;

default: if(opcao!=4)

printf("\n Opcao Invalida! \n");

}

}

while (opcao!=4);

printf("Fim das operacoes. \n\n");

system("pause");

return 0;

}

```

Observe que foi utilizado o laço *do-while* para implementar o menu do programa, em que sua função desejada é executada pelo menos uma vez dentro do laço, isto é, foi aplicada uma estrutura de repetição usando comparativo.

Pois bem, chegamos ao final de mais seção do nosso livro, agora é o momento de aplicar o conhecimento adquirido. Sucesso e ótimos estudos!

Sem medo de errar

Chegou o momento de resolver o problema solicitado pela instituição de ensino, para a qual você deverá criar um programa em linguagem C que calcule a média de um aluno de acordo com as avaliações realizadas.

Para resolver essa situação, é sugerida uma das possíveis soluções:

- Criar uma variável para entrada das notas.
- Criar uma condição de entrada das notas, até que um comando de saída seja executado.
- Após o lançamento das notas, digitar uma letra para sair e calcular a média do aluno.

```
#include <stdlib.h>

#include <string.h>

int main()

{

int avalia;

char letra;

float media;

int cont=0;

int soma=0;

do

{

printf("Digite uma nota para avaliacao: \n");

scanf("%d", &avalia);

fflush(stdin);

cont++;

soma = soma + avalia;
```

```

    printf("Digite qualquer letra pra continuar ou \'s\' para
encerrar: \n");
}

while (letra=getchar() != 's');

    printf("\n \nQuantidade de avaliacao = %d e soma das notas
= %d. \n", cont, soma);

    media = soma/cont;

    printf("Media = %f. \n", media);

    system("PAUSE");

    return 0;
}

```

Realize outros testes e crie situações para a solução desse problema.

Avançando na prática

Valor nutricional do sorvete

Descrição da situação-problema

Muitos gostam de sorvete! Acreditamos que a maioria das pessoas apreciam um bom sorvete, porém, não tem o conhecimento do seu valor nutricional. Para resolver tal situação, você foi contratado por uma sorveteria para elaborar um programa em que os clientes consigam ver os valores nutricionais de cada sorvete que gostariam de consumir. Para solucionar esse problema, existe somente uma forma de programação ou há diversas formas permitidas?

Resolução da situação-problema

Para resolver essa situação, vamos trabalhar a estrutura de repetição com teste no final, porém, você deverá analisar

o programa sugerido e aperfeiçoar com uma pesquisa das informações nutricionais de cada sorvete, e até mesmo melhorar as rotinas do programa.

```
#include <stdio.h>

int main ()

{

    int i;

    do

        {

            printf ("\n\n INFORMACAO NUTRICIONAL DO

SORVETE\n\n");

            printf ("\n\n Digite um numero que corresponde

ao saber desejado\n\n");

            printf ("\t(1)...flocos\n");

            printf ("\t(2)...morango\n");

            printf ("\t(3)...leite condensado\n\n");

            scanf("%d", &i);

        } while ((i<1)||i>3);

    switch (i)

        {

            case 1:

                printf ("\t\tVoce escolheu flocos.\n");

                break;

            case 2:
```

```

        printf ("\t\tVoce escolheu morango.\n");

        break;

        case 3:

            printf ("\t\tVoce escolheu leite
condensado.\n");

            break;

        }

        return(0);

    }

```

Lembre-se, o treinamento deixará você craque na programação.

Faça valer a pena

- 1.** O comando *while* executa a rotina de um programa enquanto uma sintaxe do programa for correta. Neste caso, podemos afirmar que:
- I - O programa não executará nenhuma repetição (e as ações que ali dentro estiverem programadas) sem antes testar uma condição.
 - II - Em alguns casos, quando utilizamos teste no início, pode ocorrer o famoso loop infinito.
 - III - Geralmente usamos o comando *while* quando não sabemos quantas vezes o laço da condição deve ser repetido.

Assinale a alternativa correta de acordo com as afirmações acima:

- a) Somente a afirmação I está correta.
- b) As afirmações I e II estão corretas.
- c) Somente a afirmação II está correta.
- d) As afirmações I, II e III estão corretas.
- e) Somente a afirmação III está correta.

2. Levando em consideração que precisamos estar atentos para que não ocorra um loop infinito, analise as afirmações abaixo:

I. *Contador* – é utilizado para controlar as repetições, quando são determinadas.

II. *Incremento e decremento* – trabalham o número do contador, seja aumentando ou diminuindo.

III. *Acumulador* – segundo Soffner (2013), irá somar as entradas de dados de cada iteração da repetição, gerando um somatório a ser utilizado quando da saída da repetição.

IV. *Condição de parada* – utilizada para determinar o momento de parar quando não se tem um valor exato desta repetição.

De acordo com as afirmações apresentadas, assinale a alternativa correta:

- a) Somente a afirmação I está correta;
- b) As afirmações I, III e IV estão corretas;
- c) As afirmações II e III estão corretas;
- d) Somente a afirmação IV está correta;
- e) As afirmações I, II, III e IV estão corretas.

3. Segundo Soffner (2013), quando aplicado o comando do/while as ações poderão ser executadas pelo menos uma vez antes do teste condicional. Nesse caso, é possível ter uma sequência de repetição de processos, sendo assim, é possível ter várias opções para a solução de um problema em um programa.

Analise o programa abaixo, que realiza a soma dos números positivos usando repetição com teste no final, e complete as partes que estão faltando no programa.

```
#include<stdio.h>

int main()
{

int n;

int soma = 0;

-----

{

printf("Digite um número positivo para ser somado ou negativo para
sair:
```

```
");  
  
scanf("%d", &n);  
  
____( n >= 0 )  
  
soma = soma + n;  
  
}while( _____ );  
  
printf("A soma eh %d\n", soma);  
  
return 0;  
  
}
```

Assinale a alternativa que completa o programa acima:

- a) "do", "if", "n >= 0"
- b) "if", "n=", "n=0"
- c) "if", "else", "n >= 0"
- d) "if", "else", "n <= 0"
- e) "do", "if", "n = 0"

Seção 3.3

Estruturas de repetição determinísticas

Diálogo aberto

Muito bem! Chegamos ao final de mais uma unidade. Nesta seção, você terá a oportunidade de estudar a estrutura de repetição usando “for” (para), o histórico e as aplicações da repetição determinística, assim como o comparativo com estruturas condicionais.

Pense na seguinte situação: uma grande empresa no ramo de tecnologia lança um novo produto e, para popularizar esse produto, resolve premiar os dez primeiros clientes que se interessarem por ele, ou seja, **para** os clientes de um **até** dez, essa ação será executada um a um, até o último cliente. Bem tranquilo, certo?

Você e o seu funcionário (ex-estagiário) receberam uma nova missão da instituição de ensino à qual prestam serviços. Eles solicitaram um programa em linguagem C para transformar o sobrenome digitado dos alunos em letras maiúsculas, assim, caso uma pessoa digite o sobrenome do aluno em letras minúsculas, o programa transformará automaticamente em maiúsculas. Após a criação do programa, entregue o código no formato TXT para documentação.

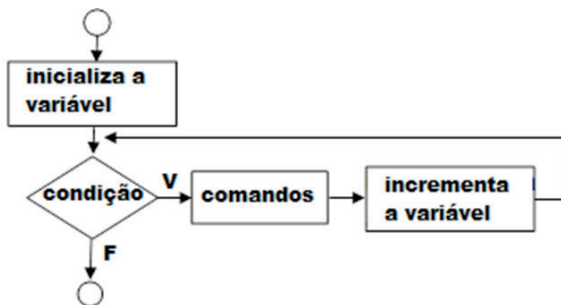
E agora? Qual função em linguagem C devemos usar para converter maiúsculas em minúsculas e vice-versa? Caro aluno, são tantas as possibilidades de aplicação de estrutura de repetição que não podemos deixar passar esse novo desafio. Então, vamos buscá-lo? Pratique bastante, e boa sorte!

Não pode faltar

Chegou o momento de avançar em nossos estudos. Nesta seção, iremos aprender como a estrutura de repetição usando “for”, os históricos e aplicações de estruturas de repetição determinísticas e os comparativos com estruturas condicionais serão aplicados dentro da Linguagem de programação.

Para tal, vamos dar início falando da repetição com variáveis de controle, ou seja, como aplicaremos o laço “for”. Esse comando, que em português significa “para”, segundo Mizrahi (2008), é geralmente usado para repetir uma informação por um número fixo de vezes, isto é, podemos determinar quantas vezes acontecerá a repetição. Veja na Figura 3.6 como é representada a estrutura de repetição usando o comando “for”.

Figura 3.6 | Fluxograma de repetição com variáveis de controle.



Fonte: elaborada pelo autor.

A sintaxe usando a linguagem de programação em C fica da seguinte forma:

```
for(inicialização; condição final;
incremento)
{
comandos;
}
```

Na aplicação do comando “for”, você encontra três expressões separadas por ponto e vírgula. Veja abaixo o que significa cada uma delas:

- *Inicialização*: neste momento, coloca-se a instrução de atribuição. A inicialização é executada uma única vez antes de começar o laço.
- *Condição final*: realiza um teste que determina se a condição é verdadeira ou falsa; se for verdadeira, permanece no laço e, se for falsa, encerra o laço e passa para a próxima instrução.

- *Incremento*: parte das nossas explicações anteriores, em que é possível incrementar uma repetição de acordo com um contador específico, lembrando que o incremento é executado depois dos comandos.



Assimile

Para assimilar, vamos usar o comando iterativo "for" em várias situações, podendo-se citar:

(I) Realizar testes em mais de uma variável:

```
for (i=1, x=0; (i + x) < 10; i++, x++);
```

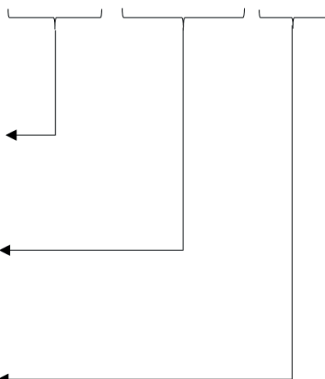
(II) Utilizar o comando "for" para representações de caracteres, por exemplo:

```
for(ch='d'; ch < 'm'; ch++).
```

Para facilitar ainda mais, veja a seguinte representação:

```
for(x = 10,y = 0; x >= 0, y <= 10; x--,y++)
```

Na primeira expressão, "x" tem o seu valor iniciado em "10" e "y" iniciado em "0".
Na segunda expressão, o laço se repetirá enquanto x for maior ou igual a zero e enquanto y for menor ou igual a 10".
Ao final da execução dos comandos do laço de repetição, x será decrementado de 1 e y será incrementado em 1.



Agora, vamos aplicar essa representação em um programa que mostra uma sequência de números, em que x vai de 10 a 0 e y vai de 0 a 10.

```

1. #include <stdio.h>
2. int main()
3. {
4.     int x,y;
5.     for(x = 10,y = 0; x >= 0, y <= 10; x--,y++)
6.     {
7.         printf("x=%2d, y=%2d\n",x,y);
8.     }
9.     return 0;
10. }

```

Para o nosso próximo programa em linguagem C, iremos criar uma contagem regressiva de um número qualquer, digitado pelo usuário:

Vejamos:

```

1. #include <stdio.h>
2. int main(void)
3. {
4.     int contador;
5.     printf("\nDigite um numero para contagem
6. regressiva\n\n");
7.     scanf("%d", &contador);
8.     for (contador; contador >= 1; contador--)
9.     {
10.         printf("%d ", contador);
11.     }
12.     getch();
13.     return(0);
14. }

```



Você pode usar o comando "break" dentro de um laço "for" para uma determinada condição, forçando, assim, o término do laço. Veja o exemplo abaixo:

```
#include <stdio.h>

main()
{
    int w;

    for ( w = 1; w <= 15; w++ )
    {
        if ( w == 8 )
        {
            break;
        }

        printf ( "%d ", w );

        printf( "\n \n Parar a condicao de repeticao w
= %d \n", w );

        return 0;
    }
}
```

Agora, vamos trabalhar algumas aplicações utilizando vetores. Segundo Manzano (2010), vetor (*array*) é um tipo especial de variável, capaz de armazenar diversos valores "ao mesmo tempo", usando um mesmo endereço na memória. Por armazenar diversos valores, também é chamado de variável composta, ou, ainda, de estrutura matricial de dados. Veja a sintaxe abaixo para utilização de vetores homogêneos:

```
tipo variavel [n]
```

Na sintaxe acima, “[n]” representa a quantidade de colunas ou linhas. Vamos ao nosso exemplo, que armazena valores em um vetor:

```
1. #include <stdio.h>
2. void main ()
3. {
4.     int num[5];
5.     printf("Entre com um numero\n");
6.     scanf("%d", &num[0]);
7.     printf("O valor digitado foi: %d", num [0]*2);
8.     getchar();
9. }
```

Que tal mesclar o comando “for” com “while”? O programa abaixo encontra a primeira posição para um determinado número inserido pelo usuário.

Vejamos então como fica a programação:

```
1. #include <stdio.h>
2. int main()
3. {
4.     int numero;
5.     int i;
6.     int posicao=0;
7.     int vetor[10];
8.     printf("Entre com o numero de ate 3 casas, diferente
9. de zero, a ser procurado em um vetor de 10 posicoes: ");
10.     scanf("%d", &numero);
```

```

11.     //Preenche o vetor com numeros
12.     for(i=0;i<10;i++)
13.     {
14.         printf("\nEntre com o numero para a posicao %02d:
15.     ", i+1);
16.         scanf("%d", &vetor[i]);
17.     }
18.     //identifica a posicao do numero lido no vetor de
19.     entrada
20.     while(vetor[posicao] != numero)
21.     {
22.         posicao++;
23.     }
24.     // Imprime vetor
25.     for(i=0;i<10;i++)
26.     {
27.         printf("%03d ", vetor[i]);
28.     }
29.     // Imprime espaços até a posição do numero, e em
30.     seguida um "*" sob o numero
31.     printf("\n ");
32.     for(i=0;i<posicao;i++)
33.     {
34.         printf("    ");
35.     }

```

```
    printf("**");  
  
    return 0;  
  
}
```

Segundo Damas (2016), uma instrução **continue** dentro de um laço possibilita que a execução de comandos corrente seja terminada, passando à próxima iteração do laço, ou seja, quando usamos o **continue** dentro de um laço, passa o laço para a próxima iteração.

Vejamos, agora, um programa que percorrerá os números de 1 a 30 e que, neste percurso, irá testar se foi digitado algum número ímpar, caso seja ímpar o programa continua o teste até o fim do laço.

```
1. #include <stdio.h>  
  
2. main()  
  
3. {  
  
4.     int i;  
  
5.     for (i=1; i<=100;i=i+1)  
  
6.         if (i==30)  
  
7.             break;  
  
8.         else  
  
9.             if (i%2==1)  
  
10.                continue;  
  
11.            else  
  
12.                printf("%2d\n",i);  
  
13.                printf("Termino do laco\n");  
  
14. }
```


Figura 3.7 | Execução do laço com comando continue

```
2
4
6
8
10
12
14
16
18
20
22
24
26
28
Termino do laço
-----
Process exited after 0.4944 seconds with return value 0
Pressione qualquer tecla para continuar. . . .
```

Fonte: captura de tela do DEV C++, elaborada pelo autor.

Perceba que os valores apresentados foram os números de 2 a 28, ou seja, todos os números pares.



Refleta

Segundo Damas (2016), a instrução **continue** poderá apenas ser utilizada dentro de laços. No entanto, o comando **break** pode ser utilizado em laços ou nas instruções utilizando *switch*. Existem outras formas de continuar uma instrução dentro do laço?

Dando sequência aos nossos estudos, vamos entender como são aplicadas as matrizes. “Matrizes são arranjos de duas ou mais dimensões. Assim como nos vetores, todos os elementos de uma matriz são do mesmo tipo, armazenando informações semanticamente semelhantes” (EDELWEISS, 2014, p.196).

Veja como fica a sintaxe de matrizes:

tipo variável [M] [N]

M representa a quantidade de linhas e N a quantidade de colunas.

Importante lembrar que:

- Em qualquer variável composta, o índice começa por zero, então, em uma matriz, o primeiro espaço para armazenamento é sempre (0,0), ou seja, índice 0 tanto para linha como para coluna.

- Não é obrigatório que todas as posições sejam ocupadas, sendo possível declarar uma matriz com 10 linhas (ou colunas) e usar somente uma.

Veja no programa abaixo uma matriz 3x3, em que os valores são lançados de acordo com a linha e coluna, e a mesma é montada no formato da matriz. Veja o resultado da programação na Figura 3.8.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. main()
4. {
5.     int linha,coluna;
6.     int matriz[3][3];
7.     for (linha=0; linha<3; linha++)
8.     {
9.         for (coluna=0; coluna<3;coluna++)
10.        {
11.            printf("Digitar os valores da matriz para: linha %d,
12.        coluna %d: ",linha+1,coluna+1);
13.            scanf("%d", &matriz[linha][coluna]);
14.        }
15.    }
16.    printf("Veja a sua Matriz\n");
17.    for (linha=0;linha<=2;linha++)
18.    {
19.        for (coluna=0;coluna<3;coluna++)
20.            printf("%d\t",matriz[linha][coluna]);
```

```

21.         printf("\n\n");
22.     }
23.     system("pause");
24.     return 0;
    }

```

Figura 3.8 | Resultado da programação matriz 3x3

```

Digitar os valores da matriz para: linha 1, coluna 1: 1
Digitar os valores da matriz para: linha 1, coluna 2: 2
Digitar os valores da matriz para: linha 1, coluna 3: 3
Digitar os valores da matriz para: linha 2, coluna 1: 4
Digitar os valores da matriz para: linha 2, coluna 2: 5
Digitar os valores da matriz para: linha 2, coluna 3: 6
Digitar os valores da matriz para: linha 3, coluna 1: 7
Digitar os valores da matriz para: linha 3, coluna 2: 8
Digitar os valores da matriz para: linha 3, coluna 3: 9
Veja a sua Matriz
1      2      3
4      5      6
7      8      9
Pressione qualquer tecla para continuar. . .

```

Fonte: captura de tela do DEV C++, elaborada pelo autor.



Pesquise mais

O vídeo sugerido traz uma dinâmica muito interessante na aplicação de vetores e matrizes. Realizado de "aluno para aluno", apresenta uma revisão bem minuciosa da programação em linguagem C:

DE ALUNO PARA ALUNO. **Programar em C - Revisão Vetores/Matrizes** - Aula 27. 21 nov. 2012. Disponível em: <<https://www.youtube.com/watch?v=7oA8SBAAOoA>>. Acesso em: 26 jul. 2018.

Para encerrar a seção, vamos trabalhar um clássico da programação, o quadrado mágico. Ele tem vários elementos que utilizam as estruturas de decisão e repetição. O Quadrado Mágico é caracterizado por uma tabela quadrada em que a soma de cada coluna, de cada linha e das duas diagonais são iguais.

Uma das possíveis soluções para o problema do quadrado mágico para quadrados de ordem $n \times n$, quando n é ímpar, é:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <ctype.h>
4. int main()
5. {
6.     int Matriz[31][31], i, j, ordem=0, quadrado_da_ordem=0,
7.     linha=0, coluna=0, soma=0;
8.         printf("Qual a ordem do quadrado magico? (deve
9.     ser um numero impar positivo) \n");
10.     scanf("%d", &ordem);
11.         //encerra o programa se a ordem for par (pois so
12.     funciona para numeros impares)
13.     if(ordem%2==0 || ordem <=0)
14.     {
15.         printf("Tem que ser numero impar positivo \n");
16.         return(0);
17.     }
18.         quadrado_da_ordem = ordem * ordem;
19.     printf("Quantidade de numero no Quadrado Magico:
20.     %d\n", quadrado_da_ordem);
21.         //calculo da posicao inicial (onde vai o numero "1").
22.         // Quando a ordem e diferente de 1, sera sempre na
```

```

23. penultima coluna e na ultima linha.
24.     if (ordem == 1)
25.     {
26.         linha = 0;
27.         coluna = 0;
28.     }
29.     else
30.     {
31.         linha = ((ordem+1)/2) - 1;
32.         coluna = ordem - 1;
33.     }
34. //Loop de loop para zerar a Matriz
35.     for (i=0; i<ordem; i++)
36.     {
37.         for (j=0; j<ordem; j++)
38.         {
39.             Matriz[i][j] = 0;
40.         }
41.     }
42. //Preenchendo a Matriz com os valores do Quadrado
43. Magico
44.     for(i=1; i<=quadrado_da_ordem; i++)
45.     {
46.         Matriz[linha][coluna] = i;

```

```

47.
48.             //se estamos na ultima coluna, voltamos
49. a  coluna "0", do contrario, vamos uma coluna a frente.
50.             if (coluna == ordem - 1) { coluna = 0; } else {
51.  coluna++; }
52.             // se estamos na primeira linha, vamos a ultima
53.  coluna (ordem-1), do contrario, vamos uma linha atras
54.             if (linha == 0) { linha = ordem - 1; } else {
55.  linha--; }
56.
57.             //Se a proxima posição ja esta; ocupada (e
58.  diferente de "0"), devemos ocupar a posicao
59.             //do lado esquerdo da ultima posicao preenchida.
60.             //Caso contrario, devemos preencher a posição
61.  acima e a direita
62.             //da ultima posição preenchida.
63.             if (Matriz[linha][coluna] != 0)
64.             {
65.                 if (coluna == 0) { coluna = ordem - 2; } else
66.  { coluna = coluna - 2; }
67.                 if(linha == ordem - 1) { linha = 0; } else {
68.  linha++; }
69.             }
70.         }

```

```

71.         printf("\nO quadrado magico de %dx%d e:\n", ordem,
72. ordem);
73.         //Loop de loop para imprimir a Matriz
74.         for (i=0; i<ordem; i++)
75.         {
76.             for (j=0; j<ordem; j++)
77.             {
78.                 printf("%03d ", Matriz[i][j]);
79.             }
80.             printf("\n");
81.         }
82.         //Loop para calcular a soma de linhas, colunas e da
83. diagonal principal do Quadrado magico.
84.         for(i=0; i<ordem; i++)
85.         {
86.             soma = soma + Matriz[i][i];
87.         }
88.         printf("A soma de cada linha, de cada coluna ou da
89. diagonal principal e: %d", soma);
90.         return(0);
91.     }

```

Excelente! Chegamos ao final desta seção. Procure praticar e testar os programas aqui apresentados. Lembre-se, sempre existe uma forma diferente para resolver problemas computacionais. Sucesso!

Sem medo de errar

Acreditamos que você já esteja preparado para solucionar o desafio dado pela instituição de ensino em que você e seu funcionário prestam serviços. Lembrando que foi solicitado um programa em linguagem C para transformar o sobrenome digitado dos alunos em letras maiúsculas e, se o usuário digitar o sobrenome do aluno em minúsculas, o programa as transformará automaticamente em maiúsculas.

Para resolver essa questão, é importante que você esteja atento às seguintes condições antes de iniciar a programação:

Utilizar a biblioteca `ctype.h` dentro da linguagem C, que irá proporcionar o uso de funções e macros para trabalhar com caracteres.

No nosso desafio, especificamente, podemos utilizar a função "toupper", que converte os caracteres minúsculos em maiúsculos.

Agora sim, vamos à programação:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include < ctype.h>
4. int main( )
5. {
6.     char nome[30];
7.     int i;
8.     printf("Digite o sobrenome do aluno ou aluna:\n");
9.     gets(nome);
10.    for(i=0; nome[i]!= ' '; i++)
11.        nome[i] = toupper(nome[i]);
12.    printf("\n\nSobrenome convertido: %s\n\n",nome);
13.    getch();
14.    return 0;
15. }
```


Fácil! Agora tente fazer o contrário, utilizando a função "tolower", que converte os caracteres maiúsculos em minúsculos.

Boa sorte e ótimos estudos.

Avançando na prática

Formatar CPF

Descrição da situação-problema

Você foi contratado por uma empresa de comunicação para resolver um problema na digitação dos CPFs dos clientes. A questão é que, quando o usuário digita o CPF do cliente com pontos e traço, a indexação e busca são dificultadas, ou seja, pode acontecer um erro de autenticidade. Para resolver esse impasse, você deverá desenvolver um programa para padronizar o formato do CPF, eliminando os pontos e traços digitados pelos usuários. E agora, como resolver essa situação?

Resolução da situação-problema

Para resolver essa situação, você poderá usar vetores com laços de repetições para eliminar os pontos e traço e o comando "continue" dentro da estrutura de repetição. Veja no código uma das possíveis soluções:

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. int main(int argc, char *argv[]) {
4.     char cpf1[14];
5.     char cpf2[11];
6.     int i=0,n=0;
7.     printf("Digite seu cpf na forma NNN.NNN.NNN-NN:
8.     \n");
9.     scanf("%s",cpf1);
```

```

10.     for(i=0;i<14;i++){
11.         if(cpf1[i] == '.' || cpf1[i]=='-'){
12.             continue;
13.         }
14.         else{
15.             cpf2[n] = cpf1[i];
16.             n++;
17.         }
18.     }
19.     printf("\n\n CPF formatado = %s",cpf2);
20.     return 0;
    }

```

Muito bem, agora é com você! Modifique e tente otimizar ao máximo os seus programas.

Faça valer a pena

1. Segundo Manzano (2013), podemos dizer que o programa abaixo executa as seguintes instruções:

I- Inicia o contador de índice, variável *I*, como 0 (zero), em um contador até 5.

II- Lê os cinco valores, um a um.

III- Verificar se o elemento é par; se sim, efetuar a soma dos elementos.

Apresentar o total de todos os elementos pares da matriz.

```

#include <stdio.h>

int main(void) {

}

int A[5];

int I, SOMA = 0;

```

```

printf("\nSomatorio de elementos\n\n");

for (I = 0; I <= 4; I ++)

{

}

printf("Informe um valor para o elemento nr. %2d: ", I);

scanf("%d", &A[I]);

for (I = 0; I <= 4; I ++)

if (A[I] % 2 != 0)

SOMA += A[I];

printf("\nA soma dos elementos equivale a: %4d\n\n", SOMA);

return 0;

}

```

Após a análise do código, selecione a alternativa correta:

- a) Somente a afirmação I está correta.
- b) As afirmações I e II estão corretas.
- c) As afirmações I, II e III estão corretas.
- d) Somente a afirmação II está correta.
- e) Somente a afirmação III está correta.

2. Quando trabalhamos com o comando "for", podemos encontrar três expressões separadas por ponto e vírgula. A primeira expressão é a *Inicialização*, que é executada uma única vez, antes de começar o laço. A segunda é a *condição final*, em que é realizado um teste que determina se a condição é verdadeira ou falsa e, caso seja verdadeira, permanece no laço, caso falsa, encerra o laço e passa para a próxima instrução. A última expressão é executada depois dos comandos. Qual é o nome dado para esta última expressão?

Assinale a alternativa correta:

- a) Somatório.
- b) Finalização.
- c) Processamento.
- d) Incremento.
- e) Substituição.

3. Analise o código do programa abaixo, em que foi utilizada a estrutura de repetição com variável de controle:

```
1. #include <stdio.h>
2. main()
3. {
4.     int contador; //variável de controle do loop
5.     for(contador = 1; contador <= 10; contador++)
6.     {
7.         printf("%d ", contador);
8.     }
9.     return(0);
10. }
```

Analisando o programa acima, qual é a leitura que podemos fazer da linha 5:

- a) Na primeira expressão, "contador" tem o seu valor iniciado em "1". Na segunda expressão, "contador" está condicionado a ser igual a "10". Na terceira expressão, "contador" será realizado o incrementado de 1 para ao seu valor.
- b) Na primeira expressão, "contador" tem o seu valor iniciado em "1". Na segunda expressão, "contador" está condicionado a ser menor a "10". Na terceira expressão, "contador" será realizado o incrementado de 1 para ao seu valor.
- c) Na primeira expressão, "contador" tem o seu valor iniciado em "1". Na segunda expressão, "contador" está condicionado a ser maior ou igual a "10". Na terceira expressão, "contador" será realizado o decremento de 1 para ao seu valor.
- d) Na primeira expressão, "contador" tem o seu valor iniciado em "0". Na segunda expressão, "contador" está condicionado a ser menor ou igual a

"10". Na terceira expressão, "contador" será realizado o incrementado de 2 para ao seu valor.

e) Na primeira expressão, "contador" tem o seu valor iniciado em "1". Na segunda expressão, "contador" está condicionado a ser menor ou igual a "10". Na terceira expressão, "contador" será realizado o incrementado de 1 para ao seu valor.

Referências

BÓSON TREINAMENTOS. **13** - Programação em Linguagem C - Desvio Condicional Aninhado - if / else if. 27 fev. 2015. Disponível em: <<https://www.youtube.com/watch?v=7ZL8tHLTTfs>>. Acesso em: 26 jul. 2018.

DAMAS, L. **Linguagem C**. 10. ed. Rio de Janeiro: LTC, 2016.

EDELWEISS, N. **Algoritmos e programação com exemplos em Pascal e C**. Porto Alegre: Bookman, 2014.

HÉLIO ESPERIDIÃO. **02** - Exercício - Estruturas de repetição em C. 10 mar. 2017. Disponível em <<https://www.youtube.com/watch?v=PNeYeq4-Tt0>>. Acesso em: 26 jul. 2018.

PETERSON LOBATO. **Programação C** - Aula 07 - while, do-while, for - Estruturas de repetição. Disponível em: <<https://www.youtube.com/watch?v=rCFh-tvoXlc>>. Acesso em: 26 jul. 2018.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo : Érica, 2015.

MANZANO, J. A. N. G. **Estudo Dirigido de Linguagem C**. 17. ed. rev. São Paulo: Érica, 2013.

MIZRAHI, V. V. **Treinamento em linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.

SCHILD, H. C. **Completo e total**. 3. Ed. São Paulo: Pearson Prentice Hall, 2005.

SOFFNER, R. **Algoritmos e Programação em Linguagem C**. 1. ed. São Paulo: Saraiva, 2013.

Funções e recursividade

Convite ao estudo

Caro estudante, bem-vindo a última unidade do livro de algoritmos e técnicas de programação. Ao longo do livro você teve a oportunidade de conhecer diversas técnicas que possibilitam resolver os mais variados problemas. Esse conhecimento pode lhe abrir muitas oportunidades profissionais, uma vez que você passa a olhar e pensar sobre a solução de um problema de maneira mais estruturada.

Todos os computadores (e agora os smartphones e tablets) precisam de um sistema que faça o gerenciamento dos recursos, o qual é chamado de sistema operacional (SO). Existem três SO que se destacam no mercado, o Android, o Windows e o Linux. Segundo Counter (2018), somente o núcleo (*Kernel*) do código fonte do Linux possui mais de 20 milhões de linhas de comandos, você consegue imaginar como os programadores conseguem manter tudo isso organizado? Como conseguem encontrar certas funcionalidades? Certamente não é procurando linha por linha. Tanto o desenvolvimento quanto a manutenção de um sistema só é possível de ser realizado, porque as funcionalidades são divididas em “blocos”, que são chamados de funções ou procedimentos, assunto central desta última unidade de estudo, o qual lhe habilitará a criar soluções computacionais com tais recursos.

A carreira de desenvolvedor não se limita a criar soluções comerciais, como sistemas de vendas, sistemas de controle de estoques, etc. O universo desse profissional é amplo e o que não falta são áreas com possibilidades de atuação. Você foi contratado por um laboratório de pesquisa que conta com a atuação de engenheiros, físicos, químicos e matemáticos os quais trabalham pesquisando e desenvolvendo soluções para empresas que os contratam em regime terceirizado. Muitas vezes, as

soluções desenvolvidas precisam ser implementadas em um software e você foi o escolhido para essa missão.

Nesta primeira seção você aprenderá a criar funções que retornam valores, na segunda seção avançaremos com as funções que, além de retornar, também recebem valores e, por fim, na terceira seção, veremos uma classe especial de funções, chamadas de recursivas.

Prontos para mais um universo de conhecimento? Vamos lá!

Seção 4.1

Procedimentos e funções

Diálogo aberto

Olá!

Em uma empresa, quando você precisa resolver um problema na sua folha de pagamento você se dirige até o setor financeiro. Quando você vai sair de férias, você precisa ir até o RH para assinar o recibo de férias. Quando você precisa de um determinado suprimento, o setor responsável o atende. Mas e se não houvesse essa divisão, como você saberia a quem recorrer em determinada situação? Seria mais difícil e desorganizado, não acha? Um software deve ser construído seguindo o mesmo princípio de organização, cada funcionalidade deve ser colocada em um "local" com uma respectiva identificação, para que o requisitante possa encontrar. Uma das técnicas de programação utilizada para construir programas dessa forma é a construção de funções, que você aprenderá nessa seção.

Você foi contratado por um laboratório de pesquisa, que presta serviço terceirizado, para atuar juntamente com diversos profissionais. Seu primeiro trabalho será com o núcleo de engenheiros civis. Eles receberam uma demanda da construtora local para calcular e escolher qual guindaste deve ser usado em uma determinada construção para levantar as colunas de concreto armado. Eles possuem três tipos de guindastes: G1, G2 e G3. A escolha de qual utilizar depende do peso da peça que será alçado pelo equipamento, que é calculado pela fórmula $P = VR$, onde P é o peso da coluna, V é o volume e R é a constante utilizada que vale $25\text{kN} / \text{m}^3$. A regra para escolher o guindaste a ser usado está especificada na Quadro 4.1.

Quadro 4.1 | Regras para escolha do guindaste

Regra	Guindaste
Peso \leq 500 kg	G1
500 < Peso \leq 1500 Kg	G2
Peso > 1500 Kg	G3

Fonte: elaborado pela autora.

Como você poderá automatizar o processo, a partir da entrada das medidas da coluna (base, largura e altura)? Como o programa escolherá e informará qual guindaste deverá ser usado?

Para que você possa cumprir sua missão, você aprenderá, nesta seção, a diferença entre procedimento e função, bem como criá-los e utilizá-los.

Bons estudos!

Não pode faltar

Até esse momento você já implementou diversos algoritmos na linguagem C, não é mesmo? Nesses programas, mesmo sem ainda conhecer formalmente, você já utilizou funções que fazem parte das bibliotecas da linguagem C, como *printf()* e *scanf()*. Na verdade, você utilizou uma função desde a primeira implementação, mesmo sem conhecê-la, pois é uma exigência da linguagem. Observe o programa no Quadro 4.2 que imprime uma mensagem na tela. Veja o comando na linha 2, ***int main()***. Esse comando especifica uma função que chama “*main*” e que irá devolver para quem a requisitou um valor inteiro, nesse caso, zero. Complicado? Calma! Vamos entender direito como construir e como funciona esse importante recurso.

Quadro 4.2 | Programa *Hello World*

```
1.  #include<stdio.h>
2.  int main() {
3.      printf("Hello World!");
4.      return 0;
5.  }
```

Fonte: elaborado pela autora.

A ideia de criar programas com blocos de funcionalidades vem de uma técnica de projeto de algoritmos chamada *dividir para conquistar* (MANZANO; MATOS; LOURENÇO, 2015). A ideia é simples, dado um problema, este deve ser dividido em problemas menores, que facilitem a resolução e organização. A técnica consiste em três passos (Figura 4.1):

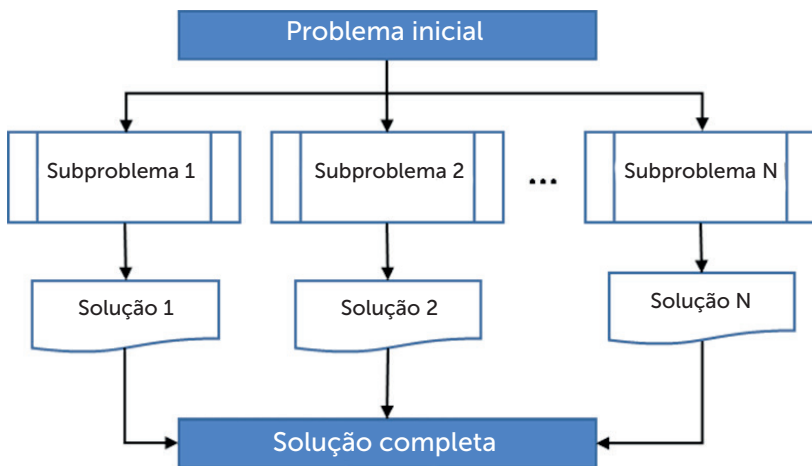
(i) **Dividir:** quebrar um problema em outros subproblemas menores. “*Solucionar pequenos problemas, em vez de um grande*

problema, é, do ponto de vista computacional, supostamente mais fácil" (MANZANO; MATOS; LOURENÇO, 2015, p. 102).

(ii) **Conquistar**: usar uma sequência de instruções separada, para resolver cada subproblema.

(iii) **Combinar**: juntar a solução de cada subproblema para alcançar a solução completa do problema original.

Figura 4.1 | Esquema da técnica *dividir para conquistar*



Fonte: adaptada de Manzano, Matos e Lourenço (2015, p. 102).

Agora que foi apresentada a ideia de subdividir um problema, podemos avançar com os conceitos sobre as funções. Uma função é definida como um trecho de código escrito para solucionar um subproblema (SOFFNER, 2013). Esses blocos são escritos tanto para dividir a complexidade de um problema maior, quanto para evitar a repetição de códigos. Essa técnica também pode ser chamada de modularização, ou seja, um problema será resolvido em diferentes módulos (MANZANO, 2015).



Assimile

A modularização é uma técnica de programação que permite a divisão da solução de um problema, afim de diminuir a complexidade, tornar o código mais organizado e evitar a repetição de códigos.

Sintaxe para criar funções

Para criar uma função utiliza-se a seguinte sintaxe:

```
<tipo de retorno> <nome> (<parâmetros>)  
{  
    <Comandos da função>  
    <Retorno> (não obrigatório)  
}
```

Em cada declaração da função alguns parâmetros são obrigatórios e outros opcionais, veja cada parâmetro.

- **<tipo de retorno>** – Obrigatório. Esse parâmetro indica qual o tipo de valor a função irá retornar. Pode ser um valor inteiro (*int*), decimal (*float* ou *double*), caractere (*char*), etc. Quando a subrotina faz um processamento e não retorna nenhum valor, usa-se o parâmetro *void* e, nesse caso, é chamado de **procedimento** (MANZANO, 2015).
- **<nome>** – Obrigatório. Parâmetro que especifica o nome que identificará a função. É como o nome de uma pessoa, para você convidá-la para sair você precisa “chamá-la pelo nome”. O nome não pode ter acento, nem caractere especial e nem ser nome composto (mesmas regras para nomes de variáveis).
- **<parênteses depois do nome>** – Obrigatório. Toda função ou procedimento, sempre terá o nome acompanhado de parênteses. Por exemplo, *main()*, *printf()*, *somar()*, etc.
- **<parâmetros>** – Opcional. Veremos na próxima seção.
- **<comandos da função>** – Obrigatório. Só faz sentido criar uma função se ela tiver um conjunto de comandos para realizar.
- **<retorno>** – Quando o tipo de retorno for *void* esse parâmetro não precisa ser usado, porém, quando não for *void* é obrigatório. O valor a ser retornado tem que ser compatível com o tipo de retorno, senão o problema dará um erro de compilação em algumas linguagens, em outras retornará um valor errôneo. Na linguagem C, irá ser retornado um valor de acordo com o tipo.

Em qual parte do código a função deve ser programada?

A função *main()*, que traduzindo significa principal, é uma função de uso obrigatório em várias linguagens de programação, por exemplo, em C, em Java, em C#, etc. Ela é usada para identificar qual é a rotina principal do programa e por onde a execução deve começar.

Na linguagem C, vamos adotar sempre a criação das funções (subrotinas) antes da função *main()*, por uma questão de praticidade e conveniência.



Exemplificando

Veja no Quadro 4.3 um programa que utiliza uma função para calcular a soma entre dois números e a seguir sua explicação detalhada.

Quadro 4.3 | Programa com a função *somar()*.

```
1.  #include<stdio.h>
2.  int somar(){
3.      return 2 + 3;
4.  }
5.  int main(){
6.      int resultado = 0;
7.      resultado = somar();
8.      printf("O resultado da funcao e = %d",re-
9.  sultado);
10.     return 0;
11. }
```

Fonte: elaborado pela autora.

A função *somar()* no Quadro 4.3, inicia-se na linha 2 e termina na linha 4 com o fechamento das chaves. Vamos analisar os parâmetros para esse caso.

<tipo de retorno> – Foi especificado que a função irá retornar um valor inteiro (*int*).

<nome> – *somar*.

<comandos da função> – Essa função foi criada de modo a retornar a soma de dois valores, tudo em um único comando: ***return 2+3 ;*** vale ressaltar que cada programador criará suas

funções da maneira que julgar mais adequada. Por exemplo, se os comandos tivessem sido escritos na forma:

```
int somar() {
    int x = 0;
    x = 2 + 3;
    return x;
}
```

A função teria o mesmo efeito. Mas veja, da forma que está no Quadro 4.3, escrevemos duas linhas a menos (imagina essa diferença sendo acumulada nas mais de 20 milhões linhas do Linux). Veja que no código do Quadro 4.3 foi usado uma variável a menos, o que significa economia de memória de trabalho e do processamento para a alocação.



Assimile

Os comandos que serão usados em cada função dependem da escolha de cada desenvolvedor. Tenha em mente que dado um problema podem haver 1, 2 ... N maneiras diferentes de resolver, o importante é tentar fazer escolhas que otimizem o uso dos recursos computacionais.

Outra característica da utilização de funções é que elas “quebram” a linearidade de execução, pois a execução pode “dar saltos” quando uma função é invocada (SOFFNER, 2013). Para entender melhor como funciona esse mecanismo, vamos criar uma função que solicita um número para o usuário, calcula o quadrado desse número e retorna o resultado. Veja no Quadro 4.4 o código para essa solução. Todo programa sempre começa pela função *main()*, ou seja, esse programa começará a ser executado na linha 8. Na linha 10 a função *calcular()* é chamada, ou seja, a execução “pula” para a linha 2. Na função *calcular()* é solicitado um valor ao usuário (linha 4), armazenado em uma variável (linha 5) e retornado o valor multiplicado por ele mesmo (linha 6). Após retornar o valor, a execução do programa “volta” para a linha 10, pois foi nesse ponto que a função foi chamada. O resultado da função é armazenado na variável *resultado* e impresso na linha 11.

Quadro 4.4 | Função para calcular o quadro de um número

```
1.  #include<stdio.h>
2.  float calcular(){
3.      float num;
4.      printf("Digite um numero: ");
5.      scanf("%f", &num);
6.      return num*num;
7.  }
8.  int main(){
9.      float resultado = 0;
10.     resultado = calcular();
11.     printf("A potencia do numero digitado = %.2f
12.     ",resultado);
13.     return 0;
    }
```

Fonte: elaborado pela autora.



Reflita

A utilização de funções permite que a execução de um programa “pule” entre as linhas, ou seja, que a execução não aconteça de modo sequencial da primeira até a última linha. Essa característica pode deixar a execução mais lenta? Esses “saltos” podem, em algum momento, causar um erro de execução? Os valores das variáveis podem se perder ou se sobrepor?

O uso de funções com ponteiros na linguagem C

Você viu que uma função pode retornar um número inteiro, um real e um caractere, mas e um vetor? Será possível retornar? A resposta é sim! Para isso devemos utilizar ponteiros (ou apontador como alguns programadores o chamam). Não é possível criar funções como `int[10] calcular()`, onde `int[10]` quer dizer que a função retorna um vetor com 10 posições. A única forma de retornar um vetor é por meio de um ponteiro (MANZANO, 2015). Lembrando que um ponteiro é um tipo especial de variável, que armazena um endereço de memória, podemos utilizar esse recurso para retornar o endereço de um vetor, assim, a função que “chamou” terá acesso ao vetor que foi calculado dentro da função (MANZANO; MATOS; LOURENÇO, 2015). Portanto, a sintaxe dessa função ficará da seguinte forma:

```

tipo* nome() {
    tipo vetor[tamanho];
    return vetor;
}

```

Veja que ainda haverá um tipo primitivo no retorno da função, mas que estará acompanhado do asterisco, indicando o retorno de um ponteiro (endereço). Outra observação é que basta retornar o nome do vetor.

Para exemplificar o uso desse recurso vamos implementar uma função, que cria um vetor de dez posições e os preenche com valores aleatórios, imprime os valores, e posteriormente passa esse vetor para “quem” chamar a função. Veja no Quadro 4.5 a implementação dessa função. O programa começa sua execução pela linha 11, na função *main()*. Na linha 12 é criado um ponteiro do tipo inteiro, ou seja, este deverá apontar para um local que tenha número inteiro. Na linha 13 é criada uma variável para controle do laço de repetição. Na linha 14 a função *gerarRandomico()* é invocada, nesse momento a execução “pula” para a linha 2, a qual indica que a função irá retornar um endereço para valores inteiros (*int**). Na linha 3 é criado um vetor de números inteiros com 10 posições e este é estático (*static*), ou seja, o valor desse vetor não será alterado entre diferentes chamadas dessa função. Na linha 5 é criada uma estrutura de repetição para percorrer as 10 posições do vetor. Na linha 6, para cada posição do vetor é gerado um valor aleatório por meio da função *rand()* e na linha 7 o valor gerado é impresso para que possamos comparar posteriormente. Na linha 9 a função retorna o vetor, agora preenchido, por meio do comando *return r;* com isso, a execução volta para a linha 14, armazenando o endereço obtido pela função no ponteiro *p*. Na linha 15 é criada uma estrutura de repetição para percorrer o vetor que será acessado pelo seu endereço. Veja que na linha 16, para acessar o conteúdo do vetor foi usado o seguinte comando **(p + i)*. É importante ter em mente que o ponteiro retorna o endereço da primeira posição do vetor, por isso é usado *p + i*, para acessar o primeiro (*p*) mais os próximos, conforme *i* incrementa, imprimindo assim todos os valores do vetor.

Quadro 4.5 | Função que retorna vetor

```
1.  #include<stdio.h>
2.  int* gerarRandomico(){
3.      static int r[10];
4.      int a;
5.      for(a = 0; a < 10; ++a) {
6.          r[a] = rand();
7.          printf("r[%d] = %d\n", a, r[a]);
8.      }
9.      return r;
10. }
11. int main(){
12.     int *p;
13.     int i;
14.     p = gerarRandomico();
15.     for ( i = 0; i < 10; i++ ) {
16.         printf("\n p[%d] = %d", i, *(p + i));
17.     }
18.     return 0;
19. }
```

Fonte: elaborado pela autora.

O resultado do código do Quadro 4.5 pode ser conferido na Figura 4.2. Veja que a impressão de ambos será igual devido ao atributo *static*, usado na declaração do vetor.

Figura 4.2 | Resultado do código do Quadro 4.5

r[0] = 41	p[0] = 41
r[1] = 18467	p[1] = 18467
r[2] = 6334	p[2] = 6334
r[3] = 26500	p[3] = 26500
r[4] = 19169	p[4] = 19169
r[5] = 15724	p[5] = 15724
r[6] = 11478	p[6] = 11478
r[7] = 29358	p[7] = 29358
r[8] = 26962	p[8] = 26962
r[9] = 24464	p[9] = 24464

Fonte: elaborada pela autora.

Outro caso importante de ponteiros com funções é na alocação de memória dinâmica. A função *malloc()* pertencente a biblioteca `<stdlib.h>` é usada para alocar memória dinamicamente. Entender o tipo de retorno dessa função é muito importante, principalmente para seu avanço, quando você começar a estudar estruturas de dados. Veja esse comando:

```
int* memoria = malloc (100);
```

A função *malloc()* pode retornar dois valores: NULL ou um ponteiro genérico (ponteiro genérico é do tipo *void**) (MANZANO, 2015). Quando houver um problema na tentativa de alocar memória a função retornará NULL e quando tudo der certo, a função retornará o ponteiro genérico para a primeira posição de memória alocada (SOFFNER, 2013). Nesse caso é preciso converter esse ponteiro genérico para um tipo específico, de acordo com o tipo de dado que será guardado nesse espaço reservado. Para isso é preciso fazer uma conversão, chamada de *cast*, que consiste em colocar entre parênteses, antes da função, o tipo de valor para o qual se deseja converter. Portanto, a forma totalmente correta de se utilizar a função é:

```
int* memoria = (int*)malloc (100);
```

Agora que vimos o tipo de retorno da função *malloc()*, vamos entender como usar essa função dentro de uma função criada por nós. Veja o código no Quadro 4.6, a função *alocar()* foi criada da linha 3 até 5, tendo seu tipo de retorno especificado como *int**, isso significa que o espaço será alocado para guardar valores inteiros. Veja que na linha 7 foi criado um ponteiro inteiro e na linha 8, a função *alocar()* foi chamada, tendo seu resultado armazenado no ponteiro chamado *memoria*. Em seguida, usamos uma estrutura condicional para saber se alocação foi feita com êxito. Caso o valor do ponteiro *memoria* seja diferente de NULL (linha 9), então sabemos que a alocação foi feita com sucesso e imprimimos o endereço da primeira posição, caso contrário, o usuário é informado que a memória não foi alocada.

Quadro 4.6 | Função para alocar memória dinamicamente

```
1. #include<stdio.h>
2. #include<stdlib.h>

3. int* alocar(){
4.     return malloc(200);
5. }
6. int main(){
7.     int *memoria;
8.     memoria = alocar();
9.     if(memoria != NULL){
10.         printf("Endereco de memoria alocada = %x", -
11. memoria);
12.     }
13.     else{
14.         printf("Memoria nao alocada");
15.     }
16.     Return 0;
    }
```

Fonte: elaborado pela autora.



Pesquise mais

A utilização de funções para alocação dinâmica de memória é extremamente usada em estruturas de dados. Estude com atenção esse recurso, pois lhe será muito útil nas próximas etapas do seu desenvolvimento profissional. Assista a uma série de seis vídeos sobre esse recurso, disponível em:

LINGUAGEM C Programação Descomplicada. [C] **Aula 60 - Alocação Dinâmica - Parte 1 – Introdução**. 5 nov. 2012. <<https://goo.gl/ps7VPa>> Acesso em: 30 jun. 2016.

Nesta seção você aprendeu a criar funções que, após um determinado conjunto de instruções, retorna um valor para “quem” chamou a subrotina. Esse conhecimento lhe permitirá criar programas mais organizados e também evitar repetição de códigos.

Sem medo de errar

Recentemente você foi contratado por um laboratório de pesquisas multidisciplinar que atende a várias empresas. Seu

primeiro trabalho é desenvolver uma solução para a equipe de engenharia civil. Um cliente deseja saber qual guindaste usar para cada tipo de construção. Ele possui três modelos, que devem ser usados de acordo com o peso da coluna de concreto armado que irá levantar. Os engenheiros lhe passaram a fórmula para calcular o peso da coluna, bem como uma tabela com as regras que devem ser seguidas e lhe solicitaram uma solução automatizada para o processo, baseada nos parâmetros de altura, largura e comprimento da coluna.

Para resolver esse problema você deve ter a fórmula para o cálculo do peso anotada: $P = VR$ e lembrar que o volume é calculado por meio da multiplicação entre os três parâmetros e o parâmetro R possui valor 25. Também será preciso criar condicionais para verificar qual guindaste deve ser usado.

Primeiro construa a função principal do programa, seguindo as seguintes dicas:

1. Crie uma variável para armazenar um valor real, pois o peso certamente não será inteiro.
2. Guarde o resultado da função na variável criada.
3. Compare o valor armazenado com a primeira condição, para verificar se o guindaste a ser usado é o G1, caso seja, imprima uma mensagem para o usuário.
4. Crie uma condicional encadeada com a primeira, comparando o valor armazenado com a terceira condição, para verificar se o guindaste a ser usado é o G3, caso seja, imprima uma mensagem para o usuário.
5. Crie o último "senão", pois caso não seja nenhum dos anteriores o guindaste a ser utilizado será o G2, e imprima uma mensagem para o usuário.
6. Agora crie uma função para calcular o peso (antes da função *main()*).
7. Crie três variáveis para armazenar os valores da base, da altura e do comprimento.
8. Peça para o usuário digitar os valores e armazene nas variáveis criadas.
9. Calcule o peso da coluna usando a fórmula passada pelos engenheiros.



Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link <https://cm-kl-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/algoritmos-e-tecnicas-de-programacao/u4/s1/resolucaoSP_atp.pdf> ou QR Code.

Avançando na prática

Função para gerar senhas

Descrição da situação-problema

Uma empresa está ingressando no mercado de segurança de dados e você foi contratado para implementar um algoritmo que cria senhas automaticamente para os clientes. A ideia é pedir para o usuário digitar uma frase e, a partir da quantidade de letras "a" e "e" encontradas na frase, o algoritmo irá gerar uma senha aleatória.

Resolução da situação-problema

O programa para geração de senhas pode ser construído utilizando uma função, além disso, a função *rand()* também pode ser usada para gerar as senhas aleatórias. Veja no Quadro 4.7 uma possível solução para o problema.

Quadro 4.7 | Função para gerar senhas aleatoriamente

```
#include<stdio.h>

int gerarSenha(){
    char frase[40]="";
    int i, cont = 0;
    printf("\n Digite uma frase para gerar sua senha:
");
    fflush(stdin);
    fgets(frase,40,stdin);
    for(i=0; i < 40; i++){
        if(frase[i]=='a' || frase[i]=='b'){
            cont++;
        }
    }
}
```

```

    }
}
    srand(cont); //passando uma "semente" para a geração
aleatória
    return rand() * rand();
}

void main(){
    int senha;
    senha = gerarSenha();
    printf("Sua senha e = %d",senha);
}

```

Fonte: elaborado pela autora.

Faça valer a pena

1. Dado um certo problema para ser resolvido por meio de um programa, a solução pode ser implementada em blocos de funcionalidades, técnica essa conhecida como *dividir para conquistar*. A aplicação dessa técnica em uma linguagem de programação pode ser feita por meio de funções ou procedimentos.

A respeito de funções e procedimentos, analise as afirmações a seguir:

- I. Funções e procedimentos possuem o mesmo objetivo, ou seja, resolver parte de um problema maior. Ambas técnicas farão o processamento de uma funcionalidade e retornarão um valor para "quem" chamou a subrotina.
- II. Em uma função criada para retornar um valor inteiro, o comando *return* não pode retornar outro tipo de valor.
- III. Uma função pode ser invocada quantas vezes for necessário em um programa.

Escolha a alternativa que representa a resposta correta.

- a) Somente a afirmação I está correta.
- b) Somente a afirmação II está correta.

- c) Somente a afirmação III está correta.
- d) Somente a afirmação I e II estão corretas.
- e) Somente a afirmação II e III estão corretas.

2. Funções são usadas para organizar o código, evitando a repetição de linhas de comandos. Uma boa prática de programação é avaliar se um determinado trecho precisa ser escrito mais de uma vez. Se a resposta for sim, então esse trecho deve ser transformado em uma funcionalidade.

Avalie o código a seguir e escolha a opção correta.

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

```
int somar() {  
    return 2 + 3.23;  
}  
int main() {  
    int resultado = 0;  
    resultado = somar();  
    printf("O resultado da funcao e = %d", resultado);  
    return 0;  
}
```

- a) Será impresso na tela *"O resultado da funcao e = 5.23"*.
- b) Será impresso na tela *"O resultado da funcao e = 2"*.
- c) Será impresso na tela *"O resultado da funcao e = 3"*.
- d) Será impresso na tela *"O resultado da funcao e = 5"*.
- e) Será dado um erro de execução, pois a função espera retornar um *int*, e está sendo retornado um número real.

3. Vetores são estruturas de dados estáticas, ou seja, não são redimensionadas em tempo de execução. Uma vez criadas com tamanho *N*, esse tamanho se mantém fixo. Para criar uma função que retorna um vetor é preciso recorrer ao uso de ponteiros.

Avalie o código a seguir e escolha a opção correta.

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

```
int* retornarVetor() {  
    static int v[10];  
    int a;  
    for(a = 0; a < 10; ++a) {  
        v[a] = 2 * a;  
    }  
}
```

```
    }  
    return v;  
}  
  
int main(){  
    int *p;  
    p = retornarVetor();  
    printf("Valor = %d", *(p + 2));  
    return 0;  
}
```

- a) Será impresso na tela "Valor = 0".
- b) Será impresso na tela "Valor = 2".
- c) Será impresso na tela "Valor = 4".
- d) Será impresso na tela "Valor = 6".
- e) Será impresso na tela "Valor = 8".

Seção 4.2

Escopo e passagem de parâmetros

Diálogo aberto

Caro estudante, você sabe quantos números de linhas telefônicas ativas existem no mundo hoje? Segundo Almeida e Zanlorenssi (2018), o número de linhas ativas ultrapassa a marca de 8 bilhões. No Brasil, um número de telefone fixo é dividido em três partes: código da região, código da cidade e o número. Imagine um usuário desatento que digita 3441-1111, essa ligação iria para qual cidade? Pois existem várias cidades com o código 3441 e o que as diferencia é o código da região, delimitando o escopo daquele número. O mesmo acontece com variáveis em um programa, o seu escopo definirá o limite de utilização desse recurso.

Dando continuidade ao seu trabalho no laboratório de pesquisa multidisciplinar, após construir com êxito o programa para a equipe de engenharia civil, agora você trabalhará com o núcleo de química. Em uma reação química, a proporção entre os reagentes define o resultado. Nesse contexto, dados dois componentes químicos, A e B , a sua mistura resultará em um terceiro ($A + B \rightarrow C$). O núcleo foi procurado para otimizar as condições em uma reação chamada de proteção. Para obter a otimização, é preciso variar as condições do experimento, ou seja, usar diferentes massas dos compostos. Um mol do composto A possui massa 321,43 g e um mol de B possui massa 150,72 g. Você precisa criar uma função que facilite o cálculo dos químicos, ou seja, a partir de diferentes combinações de massa, o programa calculará a massa final do composto. Além do cálculo, seu programa deverá exibir os resultados das combinações da Quadro 4.8, pois estes valores são usados como referência pelos químicos.

Quadro 4.8 | Condições do experimento

Quantidade de mol de A	Quantidade de mol de B	Quantidade de mol de C
1,2	1,0	?
1,4	1,0	?
1,0	1,6	?

Fonte: elaborada pela autora.

Para cumprir essa missão, nesta seção, você aprenderá sobre o escopo das variáveis, bem como a passagem de parâmetros em uma função. Com esse conhecimento seus programas ficarão mais organizados e otimizados. Bom estudo!

Não pode faltar

Escopo de variáveis

Já sabemos que variáveis são usadas para armazenar dados temporariamente na memória, porém o local onde esse recurso é definido no código de um programa determina seu escopo e sua visibilidade. Observe o código no Quadro 4.9.

Quadro 4.9 | Exemplo de variáveis em funções

```
1.  #include<stdio.h>
2.  int testar(){
3.      int x = 10;
4.      return x;
5.  }
6.  int main(){
7.      int x = 20;
8.      printf("\n Valor de x na funcao main() =
9.  %d",x);
      printf("\n Valor de x na funcao testar() =
10. %d",testar());
11.
      return 0;
    }
```

Fonte: elaborado pela autora.

Na implementação do Quadro 4.9, temos duas variáveis chamadas "x", isso acarretará em erro? A resposta, nesse caso, é não, pois, mesmo as variáveis tendo o mesmo nome, elas são definidas em lugares diferentes, uma está dentro da função *main()* e outra dentro da *testar()* e cada função terá seu espaço na memória de forma independente. Na memória, as variáveis são localizadas pelo seu endereço, portanto, mesmo sendo declaradas com o mesmo nome, seus endereços são distintos.

Com esse exemplo em mente, podemos definir como sendo o escopo de uma variável "a relação de alcance que se tem com o

local onde certo recurso se encontra definido, de modo que possa ser “enxergado” pelas várias partes do código de um programa” (MANZANO, 2015, p. 288). O escopo é dividido em duas categorias, **local** ou **global** (MANZANO; MATOS; LOURENÇO, 2015). No exemplo do Quadro 4.9, ambas variáveis são locais, ou seja, elas existem e são “enxergadas” somente dentro do corpo da função onde foram definidas. Para definir uma variável global, é preciso criá-la fora da função, assim ela será visível por todas as funções do programa. Nesse livro, vamos adotar criá-las logo após a inclusão das bibliotecas.

Veja no Quadro 4.10 um exemplo de declaração de uma variável global, na linha 2, logo após a inclusão da biblioteca de entrada e saída padrão. Veja que na função principal não foi definida nenhuma variável com o nome de “x” e mesmo assim pode ser impresso seu valor na linha 7, pois é acessado o valor da variável global. Já na linha 8 é impresso o valor da variável global modificado pela função `testar()`, que retorna o dobro do valor.

Quadro 4.10 | Exemplo de variável global

```
1.  #include<stdio.h>
2.  int x = 10;
3.  int testar(){
4.      return 2*x;
5.  }
6.  int main(){
7.      printf("\n Valor de x global = %d",x);
8.      printf("\n Valor de x global alterado na funcao
   testar() = %d",testar());
9.      return 0;
10. }
```

Fonte: elaborado pela autora.



Assimile

A utilização de variáveis globais permite otimizar a alocação de memória, pois em vários casos o desenvolvedor não precisará criar variáveis locais. Por outro lado, essa técnica de programação deve ser usada com cautela, pois variáveis locais são criadas e destruídas ao fim da função, enquanto as globais permanecem na memória durante todo o tempo de execução.

Vejamos um exemplo prático da utilização do escopo global de uma variável. Vamos criar um programa que calcula a média entre duas temperaturas distintas. Veja o código no Quadro 4.11, na linha 2, foram declaradas duas variáveis. Lembrando que o programa sempre começa pela função principal, a execução inicia na linha 6. Na 7, é solicitado ao usuário digitar duas temperaturas, as quais são armazenadas dentro das variáveis globais criadas. Na linha 9, a função *calcularMedia()* é invocada para fazer o cálculo da média usando os valores das variáveis globais. Nesse exemplo, fica claro a utilidade dessa técnica de programação, pois as variáveis são usadas em diferentes funções, otimizando o uso da memória, pois não foi preciso criar mais variáveis locais.

Quadro 4.11 | Calcular média de temperatura com variável global

```
1.  #include<stdio.h>
2.
3.  float t1, t2;
4.
5.  float calcularMedia(){
6.      return (t1 + t2)/2;
7.  }
8.  int main(){
9.      printf("\n Digite as duas temperaturas: ");
10.     scanf("%f %f",&t1,&t2);
11.     printf("\n A temperatura media = %.2f",calcu-
12.         larMedia());
13.
14.     return 0;
15. }
```

Fonte: elaborado pela autora.



Exemplificando

Vimos que é possível criar variáveis com mesmo nome, em diferentes funções, pois o escopo delas é local, mas, e se existir uma variável global e uma local com mesmo nome, por exemplo:

```
int x = 10;
int main(){
    int x = -1;
    printf("\n Valor de x = %d",x);
}
```

Qual valor será impresso na variável x? A variável local sempre sobrescreverá o valor da global, portanto, nesse caso, será impresso o valor -1 na função principal.

Na linguagem C, para conseguirmos acessar o valor de uma variável global, dentro de uma função que possui uma variável local com mesmo nome, devemos usar a instrução **extern** (MANZANO, 2015). No Quadro 4.12, como utilizar variáveis globais e locais com mesmo nome na linguagem C. Veja que foi necessário criar uma nova variável chamada "b", com um bloco de instruções (linhas 6 – 9), que atribui a nova variável o valor "externo" de x.

Quadro 4.12 | Variável global e local com mesmo nome

```
1.  #include<stdio.h>
2.
3.  int x = 10;
4.
5.  int main(){
6.      int x = -1;
7.      int b;
8.      {
9.          extern int x;
10.         b = x;
11.     }
12.     printf("\n Valor de x = %d",x);
13.     printf("\n Valor de b (x global) = %d",b);
14.     return 0;
15. }
```

Fonte: elaborado pela autora.

Passagem de parâmetros para funções

Para criar funções usamos a seguinte sintaxe:

```
<tipo de retorno> <nome> (<parâmetros>)
{
    <Comandos da função>
    <Retorno> (não obrigatório)
}
```

Com exceção dos parâmetros que uma função pode receber, todos os demais já foram apresentados, portanto, nos dedicaremos a entender esse importante recurso.

Ao definir uma função, podemos também estabelecer que ela receberá informações “de quem” a invocou. Por exemplo, ao criar uma função que calcula a média, podemos definir que “quem chamar” deve informar os valores, sobre os quais o cálculo será efetuado.



Refleta

O uso de funções que recebem parâmetros pode substituir a criação de variáveis globais em alguns casos, e você como desenvolvedor tem a liberdade de escolher qual técnica utilizará. Qual ou quais os melhores critérios para serem levados em consideração na hora de optar por uma determinada técnica de programação em detrimento de outra?

Passagem por valor

Na sintaxe, para criar uma função que recebe parâmetros é preciso especificar qual o tipo de valor que será recebido. Uma função pode receber parâmetros na forma de valor ou de referência (SOFFNER, 2013). Na passagem parâmetros por valores, a função cria variáveis locais automaticamente para armazenar esses valores e após a execução da função essas variáveis são liberadas. Veja, no Quadro 4.13, um exemplo de definição e chamada de função com passagem de valores. Observe que, na linha 2, a função *somar()* foi definida para receber dois valores inteiros. Internamente, serão criadas as variáveis “a” e “b” locais, para guardar esses valores até o final da função. Na linha 7, a função *somar* foi invocada, passando os dois valores inteiros que a função espera receber e, o resultado do cálculo será guardado na variável “*result*”.

Quadro 4.13 | Chamada de função com passagem de valores

```
1. #include<stdio.h>
2. int somar(int a, int b){
3.     return a + b;
4. }
```

```

5.  int main(){
6.      int result;
7.      result = somar(10,15);
8.      printf("\n Resultado da soma = %d",result);

9.      return 0;
10. }
```

Fonte: elaborado pela autora.



Assimile

A técnica de passagem de parâmetros para uma função é extremamente usada em todas as linguagens de programação. Ao chamar uma função que espera receber parâmetros, caso todos os argumentos esperados não sejam informados, será gerado um erro de compilação.

Ao utilizar variáveis como argumentos, na passagem de parâmetros por valores, essas variáveis não são alteradas, pois é fornecido uma cópia dos valores armazenados para a função (SOFFNER, 2013). Para ficar claro essa importante definição, veja o código no Quadro 4.14. A execução do programa começa na linha 9, pela função principal, na qual são criadas duas variáveis "n1" e "n2". Na linha 13, o comando determina a impressão dos valores das variáveis, na linha 14, a função *testar()* é invocada passando como parâmetro as duas variáveis. Nesse momento, é criada uma cópia de cada variável na memória para utilização da função. Veja que dentro da função, o valor das variáveis é alterado e impresso, mas essa alteração é local, ou seja, é feita na cópia dos valores e não afetará o valor inicial das variáveis criadas na função principal. Na linha 16, imprimimos novamente os valores após a chamada da função. A Figura 4.3 apresenta o resultado desse programa.

Quadro 4.14 | Variáveis em chamada de função com passagem de valores

```

1.  #include<stdio.h>

2.  int testar(int n1, int n2){
3.      n1 = -1;
4.      n2 = -2;
```

```

5.     printf("\n\n Valores dentro da funcao testar() :
        ");
6.     printf("\n n1 = %d e n2 = %d",n1,n2);
7.     return 0;
8.     }
9.     int main(){
10.        int n1 = 10;
11.        int n2 = 20;
12.        printf("\n\n Valores antes de chamar a funcao:
        ");
13.        printf("\n n1 = %d e n2 = %d",n1,n2);

14.        testar(n1,n2);

15.        printf("\n\n Valores depois de chamar a funcao:
        ");
16.        printf("\n n1 = %d e n2 = %d",n1,n2);

17.        return 0;
18.     }

```

Fonte: elaborado pela autora.

Figura 4.3 | Resultado do código do Quadro 4.15

```

Valores antes de chamar a funcao:
n1 = 10 e n2 = 20

Valores dentro da funcao testar():
n1 = -1 e n2 = -2

Valores depois de chamar a funcao:
n1 = 10 e n2 = 20

```

Fonte: elaborada pela autora.

Passagem por referência

A utilização de funções com de passagem de parâmetros por referência está diretamente ligada aos conceitos de ponteiro e endereço de memória. A ideia da técnica é análoga à passagem por valores, ou seja, a função será definida de modo a receber certos parâmetros e “quem” faz a chamada do método deve informar esses argumentos. Entretanto, o comportamento e o resultado são diferentes. Na passagem por referência, não será criada uma cópia dos argumentos passados, na verdade, será passado o endereço

da variável e função trabalhará diretamente com os valores ali armazenados (SOFFNER, 2013).

Como a função utiliza o endereço (ponteiros), na sintaxe, serão usados os operadores * e & (MANZANO, 2015). Na definição da função os parâmetros a serem recebidos devem ser declarados com *, por exemplo: `int testar(int* parametro1, int* parametro2)`. Na chamada da função, os parâmetros devem ser passados com o &, por exemplo: `resultado = testar(&n1, &n2)`. Veja o código no Quadro 4.15, um exemplo de uma função que passa variáveis como referência. Observe que a única diferença desse código para o do Quadro 4.14 é a utilização dos operadores * e &, porém, o resultado completamente diferente (Figura 4.4). Com a passagem por referência os valores das variáveis são alterados. Nas linhas 3 e 4, usamos asterisco para acessar o conteúdo guardado dentro do endereço que ele aponta, pois se usássemos somente n1 e n2 iríamos acessar o endereço que eles apontam.

Quadro 4.15 | Variáveis em chamada de função com passagem de referência

```
1.  #include<stdio.h>
2.  int testar(int* n1, int* n2){
3.      *n1 = -1; //tem que usar o * para acessar o conteúdo
4.      *n2 = -2;
5.      printf("\n\n Valores dentro da funcao testar(): ");
6.      printf("\n n1 = %d e n2 = %d",*n1,*n2);
7.      return 0;
8.  }
9.  int main(){
10.     int n1 = 10;
11.     int n2 = 20;
12.     printf("\n\n Valores antes de chamar a funcao: ");
13.     printf("\n n1 = %d e n2 = %d",n1,n2);
14.     testar(&n1,&n2);
15.     printf("\n\n Valores depois de chamar a funcao: ");
16.     printf("\n n1 = %d e n2 = %d",n1,n2);
17.     return 0;
18. }
```

Fonte: elaborado pela autora.

Figura 4.4 | Resultado do código do Quadro 4.15

```
Valores antes de chamar a funcao:  
n1 = 10 e n2 = 20  
  
Valores dentro da funcao testar():  
n1 = -1 e n2 = -2  
  
Valores depois de chamar a funcao:  
n1 = -1 e n2 = -2
```

Fonte: elaborada pela autora.

Passagem de vetor

Para finalizar esta seção, vamos ver como passar um vetor para uma função. Esse recurso pode ser utilizado para criar funções que preencham e imprimem o conteúdo armazenado em um vetor, evitando a repetição de trechos de código. Na definição da função, os parâmetros a serem recebidos devem ser declarados com colchetes sem especificar o tamanho, por exemplo: `int testar(int v1[], int v2[])`. Na chamada da função, os parâmetros devem ser passados como se fossem variáveis simples, por exemplo: `resultado = testar(n1,n2)`.

Na linguagem C, a passagem de um vetor é feita implicitamente por referência. Isso significa que mesmo não utilizando os operadores “*” e “&”, quando uma função que recebe um vetor é invocada, o que é realmente passado é o endereço da primeira posição do vetor.

Para entender esse conceito, vamos criar um programa que, por meio de uma função, preencha um vetor de três posições e em outra função percorre o vetor imprimindo o dobro de cada valor do vetor. Veja, no Quadro 4.16, o código que começa a ser executado pela linha 15. Na linha 18, a função `inserir()` é invocada, passando como parâmetro o vetor “`numeros`”. Propositamente, criamos a função na linha 2, recebendo como argumento um vetor de nome “`a`”, para que você entenda que o nome da variável que a função recebe não precisa ser o mesmo nome usado na chamada. Na grande maioria das vezes, utilizamos o mesmo nome como boa prática de programação. Na função `inserir()`, será solicitado ao usuário digitar os valores que serão armazenados no vetor “`numeros`”, pois foi passado como referência implicitamente. Após essa função finalizar,

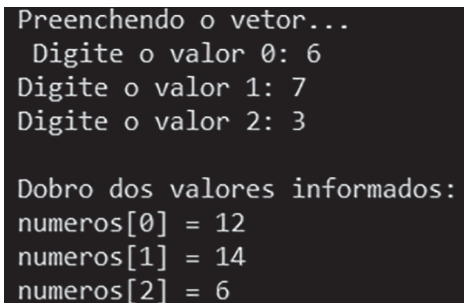
a execução vai para a linha 18 e depois 19, na qual é invocada a função *imprimir()*, novamente passando o vetor "numeros" como argumento. Mais uma vez, criamos a função na linha 9 com o nome do vetor diferente para reforçar que não precisam ser os mesmos. Nessa função, o vetor é percorrido, imprimindo o dobro de cada valor. Veja na Figura 4.5 o resultado desse programa.

Quadro 4.16 | Função com passagem de vetor

```
1.  #include<stdio.h>
2.  void inserir(int a[]){
3.      int i=0;
4.      for(i=0;i<3;i++){
5.          printf("Digite o valor %d: ",i);
6.          scanf("%d",&a[i]);
7.      }
8.  }
9.  void imprimir(int b[]){
10.     int i=0;
11.     for(i=0;i<3;i++){
12.         printf("\n numeros[%d] = %d",i,2*b[i]);
13.     }
14. }
15. int main(){
16.     int numeros[3];
17.     printf("\n Preenchendo o vetor... \n ");
18.     inserir(numeros);
19.     printf("\n Dobro dos valores informados:");
20.     imprimir(numeros);
21.     return 0;
22. }
```

Fonte: elaborado pela autora.

Figura 4.5 | Resultado do código do Quadro 4.16



```
Preenchendo o vetor...
  Digite o valor 0: 6
  Digite o valor 1: 7
  Digite o valor 2: 3

Dobro dos valores informados:
numeros[0] = 12
numeros[1] = 14
numeros[2] = 6
```

Fonte: elaborada pela autora.



As técnicas apresentadas nesta seção, fazem parte do cotidiano de um desenvolvedor em qualquer linguagem de programação. Leia o capítulo 10 do livro *Linguagem C: acompanhada de uma xícara de café*. E aprofunde seu conhecimento.

MANZANO, J. A. N. G. **Linguagem C: acompanhada de uma xícara de café**. São Paulo: Érica, 2015.

Com esta seção, demos um passo importante no desenvolvimento de softwares, pois os conceitos apresentados são utilizados em todas as linguagens de programação. Continue estudando e aprofundando.

Sem medo de errar

Você foi contratado por um laboratório de pesquisa que presta serviço para diversas empresas e, agora, você precisa fazer um programa para a equipe de químicos. Para saber a massa exata de um determinado composto, eles precisam calcular a massa de cada elemento que será usado, bem como a proporção que será usada. Foi solicitado a você automatizar o cálculo de uma reação chamada de proteção. Nessa reação, um composto A, de massa 321,43 g/mol será somando a um composto B de massa 150,72 g/mol. Seu programa, além de calcular o composto com base nas informações do usuário, deverá também exibir os valores de referência das combinações: (1,2 : 1,0), (1,4 : 1,0) e (1,0 : 1,6) (valores da Quadro 4.8).

Para implementar essa solução você pode criar uma função que recebe como parâmetro a quantidade, em mol, dos componentes A e B e retorna a massa da reação.

O primeiro passo é criar a função que fará o cálculo da função. Você pode criá-la da seguinte forma:

- `float calcularMassa(float a, float b)`

Nessa função, deverão ser impressos os valores de referência conforme Quadro 4.8, e também deverá ser feito o cálculo da massa e retornar o resultado.

Em seguida, é preciso criar a função principal, na qual será solicitado ao usuário que informe a quantidade dos elementos A e B. Em posse dos valores, a função `calcularMassa()` deverá ser invocada e os valores passados, o resultado dessa função deve ser guardado dentro de uma variável, da seguinte forma:

- resultado = `calcularMassa(a,b)`

Como último passo, deverá ser impressa a massa da reação.



Você poderá verificar uma opção de código-fonte para o problema proposto por meio do link < https://cm-kls-content.s3.amazonaws.com/ebook/embed/qr-code/2018-2/ algoritmos-e-tecnicas-de-programacao/u4/s2/resolucaoSP_atp.pdf > ou QR Code.

Avançando na prática

Rendimento de investimentos

Descrição da situação-problema

Você foi contratado por uma agência de créditos pessoais para implementar um programa que calcula o total de rendimentos (usando juros simples) que um cliente terá em determinado investimento. O cliente informará o valor que pretende investir, qual o plano e quanto tempo pretende deixar o dinheiro investido. No “plano A”, o rendimento é de 2%, porém, o cliente não pode solicitar o resgate antes de 24 meses. Já no “plano B”, o rendimento é de 0,8% e o tempo mínimo para resgate é de 12 meses. Faça um programa que peça as informações para o usuário e, a partir de uma função, calcule o rendimento que o cliente terá.

Resolução da situação-problema

Para implementar a solução, primeiro você deve saber a fórmula de juros simples $j = C \cdot i \cdot t$, na qual j é o juros, C é o capital inicial, i é a taxa e t é o tempo. Veja no Quadro 4.17 uma possível implementação para o problema.

```
#include<stdio.h>

float calcularInvestimento(float valor, char plano, int meses){
    if((plano == 'A' || plano == 'a') && meses >=24){
        return valor * 0.02 * meses;
    }else{
        printf("\n Dados invalidos!");
        return 0;
    }
    if((plano == 'B' || plano == 'b') && meses >=12){
        return valor * 0.008 * meses;
    }else{
        printf("\n Dados invalidos!");
        return 0;
    }
}

int main(){
    float valorInv = 0;
    float rendimento = 0;
    char plano;
    int tempo = 0;

    printf("\n Digite o plano: ");
    scanf("%c",&plano);

    printf("\n Digite o valor a ser investido: ");
    scanf("%f",&valorInv);

    printf("\n Digite o tempo para resgate: ");
    scanf("%d",&tempo);

    rendimento = calcularInvestimento(valorInv, plano,
tempo);
    printf("\n Seu rendimento sera = %.2f",rendimento);

    return 0;
}
```

Fonte: elaborado pela autora.

Faça valer a pena

1. O uso de funções permite criar programas mais organizados, sem repetição de códigos e ainda com possibilidade de reutilização, pois, caso você implemente uma função de uso comum, poderá compartilhá-la com outros desenvolvedores. Em linguagens do paradigma orientado a objetos, as funções são chamadas de métodos, mas o princípio de construção e funcionamento é o mesmo.

A respeito das funções, analise cada uma das afirmativas e determine se é verdadeira ou falsa.

I - () Funções que retornar um valor do tipo *float*, só podem receber como parâmetros valores do mesmo tipo, ou seja, *float*.

II - () Funções que trabalham com passagem de parâmetros por referência, não criam cópias das variáveis recebidas na memória.

III - () Funções que trabalham com passagem de parâmetros por valor criam cópias das variáveis recebidas na memória.

a) I – V; II – V; III – V.

b) I – V; II – F; III – V.

c) I – F; II – V; III – V.

d) I – F; II – F; III – V.

e) I – F; II – V; III – F.

2. Além de retornar valores, as funções também podem receber parâmetros de “quem” a chamou. Essa técnica é muito utilizada e pode economizar na criação de variáveis ao longo da função principal. Os tipos de parâmetros que uma função/procedimento pode receber são classificados em passagem por valor e passagem por referência.

Análise o código a seguir e escolha a opção que contém o que será impresso na linha 11.

```
1.  #include<stdio.h>
2.  int pensar(int a, int b){
3.      a = 11;
4.      b = 12;
5.      return 0;
6.  }
7.  int main(){
8.      int a = -11;
9.      int b = -12;
10.     pensar(a,b);
11.     printf("\n a = %d e b = %d",a,b);
12.     return 0;
13. }
```

a) a = -11 e b = -12.

b) a = 11 e b = 12.

c) a = -11 e b = 12.

d) a = 11 e b = -12.

e) Será dado um erro de compilação.

3. Uma função pode receber parâmetros por valor ou por referência. No primeiro caso, são criadas cópias das variáveis na memória e, nesse caso, o valor original não é alterado. Para trabalhar com passagem por referência é preciso recorrer ao uso de ponteiros, pois são variáveis especiais que armazenam endereços de memória.

Análise o código a seguir e escolha a opção que contém o que será impresso na linha 11.

```
1.  #include<stdio.h>
2.  int pensar(int* a, int* b){
3.      a = 10;
4.      b = 20;
5.      return 0;
6.  }
7.  int main(){
8.      int a = -30;
9.      int b = -40;
10.     pensar(&a, &b);
11.     printf("\n a = %d e b = %d", a, b);
12.     return 0;
13. }
```

- a) a = -30 e b = -40.
- b) a = 10 e b = 20.
- c) a = -30 e b = 20.
- d) a = 10 e b = -40.
- e) Será dado um erro de compilação.

Seção 4.3

Recursividade

Diálogo aberto

Caro estudante, bem-vindo a última seção no estudo dos algoritmos e das técnicas de programação. A caminhada foi longa e muito produtiva. Você teve a oportunidade de conhecer diversas técnicas de programação e agora continuaremos avançando no estudo das funções. Você se lembra desse símbolo da matemática: \sum_a^b ? É o símbolo de somatória e representa que um certo termo será somado, uma certa quantidade de vezes, repetindo o processo no intervalo definido. Para você implementar computacionalmente uma solução desse tipo, você já conhece as estruturas de repetição, como o *for*, entretanto, essa técnica não é a única opção, você também pode utilizar a recursividade, assunto central dessa seção.

Após trabalhar com a equipe de engenheiros civis e químicos, agora é hora de enfrentar um novo desafio com a equipe de matemáticos do laboratório em que atua. Foi solicitado a você, implementar um programa que calcula a raiz quadrada de um número usando o método de aproximações sucessivas de Newton, portanto você deverá entregar o arquivo compilado para que eles possam utilizar. Os matemáticos o ajudaram, fornecendo a fórmula usada para tal tarefa: $x_n = \frac{x_{n-1}^2 + n}{2x_{n-1}}$, onde x_n é o valor procurado, ou seja, a raiz quadrada. O parâmetro x_{n-1} é a raiz anterior calculada pelo método e n é o número que se deseja calcular raiz. Após conhecer sua nova tarefa, seu gerente, que também é programador, pediu para que utilizasse a técnica de recursividade na implementação, portanto, o usuário precisa informar o número que deseja calcular a raiz quadrada. Quais valores precisam ser informados para que o programa execute de maneira correta? O que determinará o término do cálculo? Esse método exige algum tipo de valor específico?

Para que você cumpra essa missão, você aprenderá o que é uma função recursiva, como ela deve ser implementada e qual sua relação com as estruturas iterativas.

Aproveite mais essa oportunidade de aprimoramento e bons estudos!

Não pode faltar

Nesta unidade apresentamos as funções a você. Vimos como criar uma função, qual sua importância dentro de uma implementação, estudamos a saída de dados de uma função, bem como a entrada, feita por meio dos parâmetros.

Entre as funções existe uma categoria especial, chamada de funções recursivas. Para começarmos a nos apropriar dessa nova técnica de programação, primeiro vamos entender o significado da palavra recursão. Ao consultar diversos dicionários, por exemplo, o Houaiss ou o Aulete, temos como resultado que a palavra recursão (ou recursividade) está associada à ideia de recorrência de uma determinada situação. Quando trazemos esse conceito para o universo da programação, nos deparamos com as funções recursivas. Nessa técnica, uma função é dita recursiva quando ela chama a si própria ou, nas palavras de Soffener: *"recursividade é a possibilidade de uma função chamar a si mesma."* (SOFFENER, 2017, p. 107).

A sintaxe para implementação de uma função recursiva, nada difere das funções gerais, ou seja, deverá ter um tipo de retorno, o nome da função, os parênteses e os parâmetros quando necessário. A diferença estará no corpo da função, pois a função será invocada dentro dela mesma. Observe a Figura 4.6, nela ilustramos a construção da função, bem como a chamada dela, primeiro na função principal e depois dentro dela mesma.

Figura 4.6 | Algoritmo para função recursiva

```
<tipo> funcaoRecursiva(){
  //comandos
  funcaoRecursiva(); ← Chamando a si próprio
  //comandos
}
void main(){
  //comandos
  funcaoRecursiva(); ①
  //comandos
}
```

Fonte: elaborada pela autora.



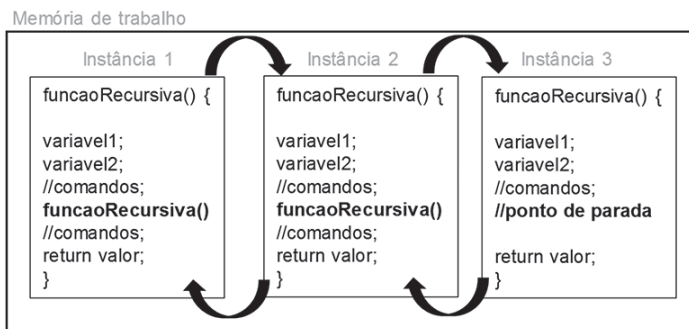
Em termos de sintaxe, uma função recursiva se difere de outras funções simplesmente pelo fato dessa função possuir em seu conjunto de comandos uma chamada a si própria.

Embora a sintaxe da função recursiva seja similar as não recursivas, o funcionamento de ambas é bastante distinto e o mau uso dessa técnica pode acarretar em uso indevido de memória, muitas vezes chegando a travar a aplicação e o sistema (MANZANO, 2015). Para entender todo o processo, vamos estabelecer alguns pontos de atenção:

- A função recursiva chama a si própria até que um ponto de parada seja estabelecido. O ponto de parada poderá ser alcançado por meio de uma estrutura condicional ou por meio de um valor informado pelo usuário.
- Uma função possui em seu corpo variáveis e comandos, os quais são alocados na memória de trabalho. No uso de uma função recursiva, os recursos (variáveis e comandos) são alocados (instâncias) em outro local da memória, ou seja, para cada chamada da função novos espaços são destinados a execução do programa. É justamente por esse ponto que o ponto de parada é crucial.
- As variáveis criadas em cada instância da função na memória são independentes, ou seja, mesmo as variáveis tendo nomes iguais, cada uma tem seu próprio endereço de memória e a alteração do valor em uma não afetará na outra.

Para auxiliá-lo na compreensão desse mecanismo observe a Figura 4.7. A instância 1 representa a primeira chamada à função *funcaoRecursiva()*, esta, por sua vez, possui em seu corpo um comando que invoca a si mesma, nesse momento é criada a segunda instância dessa função na memória de trabalho. Veja que um novo espaço é alocado, com variáveis e comandos, e como a função é recursiva, novamente ela chama a si mesma, criando, então, a terceira instância da função. Dentro da terceira instância, uma determinada condição de parada é satisfeita, nesse caso, a função deixa de ser instanciada e passa a retornar valores.

Figura 4.7 | Mecanismo da função recursiva



Fonte: elaborada pela autora.

- A partir do momento que a função recursiva alcança o ponto de parada, cada instância da função passa a retornar seus resultados para a instância anterior (a que fez a chamada). No caso da Figura 4.7, a instância três retornará para a dois e, a dois, retornará para a um. Veja que, se o ponto de parada não fosse especificado na última chamada, a função seria instanciada até haver um estouro de memória.

Toda função recursiva, obrigatoriamente, tem que ter uma instância com um caso que interromperá a chamada a novas instâncias. Essa instância é chamada de **caso base**, pois representa o caso mais simples, que resultará na interrupção.



Refleta

Toda função recursiva tem que possuir um critério de parada. A instância da função que atenderá a esse critério é chamada de caso base. Um programador que implementa de maneira equivocada o critério de parada, acarretará em um erro somente na sua aplicação, ou tal erro poderá afetar outras partes do sistema?

Vamos implementar uma função recursiva que faz a somatória dos antecessores de um número inteiro positivo, informado pelo usuário, ou seja, se o usuário digitar 5, o programa deverá retornar o resultado da soma $5 + 4 + 3 + 2 + 1 + 0$. A partir desse exemplo, você consegue determinar quando a função deverá parar de executar?

Veja, a função deverá somar até o valor zero, portanto esse será o critério de parada. Veja a implementação da função no Quadro 4.18 e a seguir sua explicação.

Quadro 4.18 | Função recursiva para soma

```
1. #include<stdio.h>
2. int somar(int valor){
3.     if(valor != 0){ //critério de parada
4.         return valor + somar(valor-1); //chamada
           recursiva
5.     }
6.     else{
7.         return valor;
8.     }
9. }
10. int main(){
11.     int n, resultado;
12.     printf("\n Digite um numero inteiro positivo : ");
13.     scanf("%d",&n);
14.     resultado = somar(n); //fazendo a primeira cha-
           mada da função
15.     printf("\n Resultado da soma = %d",resultado);
16.     return 0;
17. }
```

Fonte: elaborado pela autora.

A execução do programa no Quadro 4.18 começará na linha 10, pela função principal. Na linha 14 a função *somar()* é invocada, passando como parâmetro um número inteiro digitado pelo usuário. Nesse momento, a execução "salta" para a linha 2, onde a função é criada. Observe que ela foi criada para retornar e receber um valor inteiro. Na linha 3, o condicional foi usado como critério de parada, veja que se o valor for diferente (!=) de zero, a execução passa para a linha 4, na qual a função é invocada novamente, mas dessa vez passando como parâmetro o valor menos 1. Quando o valor for zero, as instâncias passam a retornar o valor para a que chamou.



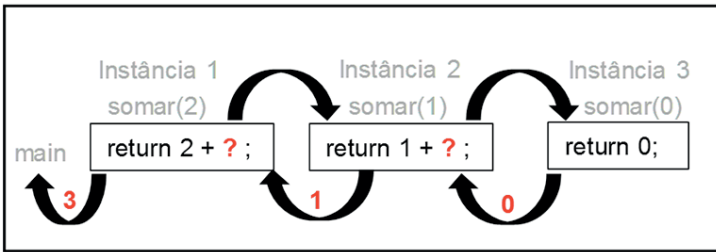
Exemplificando

A Figura 4.8 exemplifica o funcionamento na memória de trabalho da função *somar()*. Nessa ilustração o usuário digitou o valor 2, então a

função *main()* invocará a função *somar(2)*, criando a primeira instância e passando esse valor. O valor 2 é diferente de zero na primeira instância, então, o critério de parada não é satisfeito e a função chama a si própria criando a segunda instância, mas, agora, passando o valor 1 como parâmetro *somar(1)*. Veja que na primeira instância o valor a ser retornado é $2 + ?$, pois ainda não se conhece o resultado da função. Na segunda instância, o valor também é diferente de zero, portanto a função chama a si mesmo novamente, agora passando como parâmetro zero (valor $- 1$). Veja que o retorno fica como $1 + ?$, pois também não se conhece, ainda, o resultado da função. Na terceira instância o critério de parada é satisfeito, nesse caso a função retorna zero. Esse valor será usado pela instância anterior, que após somar $1 + 0$, retornará seu resultado para a instância 1, que somará $2 + 1$ e retornará o valor para a função principal. Fechando o ciclo de recursividade.

Figura 4.8 | Exemplo função *somar()*

Memória de trabalho



Fonte: elaborada pela autora.

Uma das grandes dúvidas dos programadores é quando utilizar a recursividade em vez de uma estrutura de repetição. A função *somar()*, criada no Quadro 4.18 poderia ser substituída por uma estrutura de repetição usando *for*? No exemplo dado, poderia ser escrito algo como:

```
for(i=0;i<=2;i++) {  
    resultado = resultado + i  
}
```

A verdade é que poderia sim ser substituído. A técnica de recursividade pode substituir o uso de estruturas de repetição,

tornando o código mais elegante, do ponto de vista das boas práticas de programação. Entretanto, como você viu, funções recursivas podem consumir mais memória que as estruturas iterativas. Para ajudar a elucidar quando optar por essa técnica, veja o que diz um professor da Universidade de São Paulo (USP):

Muitos problemas têm a seguinte propriedade: cada instância do problema contém uma instância menor do mesmo problema. Dizemos que esses problemas têm estrutura recursiva. Para resolver um tal problema, podemos aplicar o seguinte método: se a instância em questão for pequena, resolva-a diretamente (use força bruta se necessário); senão, reduza-a a uma instância menor do mesmo problema, aplique o método à instância menor, volte à instância original. A aplicação desse método produz um algoritmo recursivo. (FEOFILOFF, 2017, p. 1)

Portanto, a recursividade é indicar quando um problema maior pode ser dividido em instâncias menores do mesmo problema, porém, considerando a utilização dos recursos computacionais que cada método empregará.

Não poderíamos falar de funções recursivas sem apresentar o exemplo do cálculo do fatorial, um clássico no estudo dessa técnica. O fatorial de um número qualquer N consiste em multiplicações sucessivas até que N seja igual ao valor unitário, ou seja, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que resulta em 120. Observe o código no Quadro 4.19 que implementa essa solução usando uma função recursiva, em seguida a explicação.

Quadro 4.19 | Função recursiva para fatorial

```
1. #include<stdio.h>
2. int fatorial(int valor){
3.     if(valor != 1){ //critério de parada
4.         return valor * fatorial(valor - 1); //cha-
        mada recursiva
5.     }
6.     else{
```

```

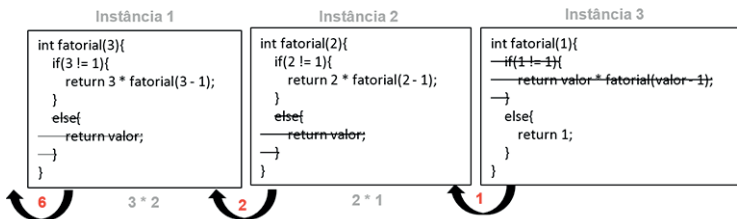
7.         return valor;
8.     }
9. }
10. int main(){
11.     int n, resultado;
12.     printf("\n Digite um numero inteiro positivo : ");
13.     scanf("%d",&n);
14.     resultado = fatorial(n);
15.     printf("\n Resultado do fatorial = %d",resultado);
16.     return 0;
17. }

```

Fonte: elaborado pela autora.

A execução do código no Quadro 4.19 inicia pela função principal, a qual solicita um número ao usuário, e na linha 14 invoca a função *fatorial()*, passando o valor digitado como parâmetro. Dentro da função *fatorial()*, enquanto o valor for diferente de 1, a função chamará a si própria, criando novas instâncias nas memória, cada vez passando como parâmetro o valor decrementado de um. Quando o valor chegar a um, a função retorna à multiplicação dos valores encontrados em cada instância. É importante entender bem o funcionamento. Observe a Figura 4.9, que ilustra as instâncias quando o usuário digita o número 3. Veja que os resultados só são obtidos quando a função chega no caso base e, então, começa a “devolver” o resultado para a instância anterior.

Figura 4.9 | Exemplo função *fatorial()*



Fonte: elaborada pela autora.

Esse mecanismo é custoso para o computador, pois tem que alocar recursos para as variáveis e comandos da função, procedimento chamado de empilhamento, e tem também que

armazenar o local onde foi feita a chamada da função (OLIVEIRA, 2018). Para usar a memória de forma mais otimizada, existe uma alternativa chamada recursividade em cauda. Nesse tipo de técnica a recursividade funcionará como uma função iterativa (OLIVEIRA, 2018). Uma função é caracterizada como recursiva em cauda quando a chamada a si mesmo é a última operação a ser feita no corpo da função. Nesse tipo de função, o caso base costuma ser passado como parâmetro, o que resultará em um comportamento diferente. Para entender vamos implementar o cálculo do fatorial usando essa técnica, veja o código no Quadro 4.20. Veja na linha 3 que a função recursiva em cauda retorna o fatorial, sem nenhuma outra operação matemática, e que passa o número a ser calculado e o critério de parada junto. Também é possível perceber que foi preciso criar uma função auxiliar para efetuar o cálculo.

Quadro 4.20 | Recursividade em cauda

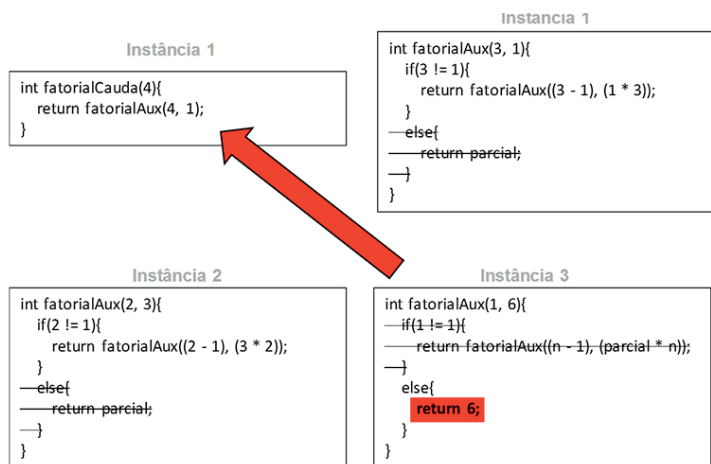
```
1.  #include<stdio.h>
2.  int fatorialCauda(int valor){
3.      return fatorialAux(valor, 1);
4.  }
5.  int fatorialAux(int n, int parcial){
6.      if(n != 1){
7.          return fatorialAux((n - 1), (parcial * n));
8.      }else{
9.          return parcial;
10.     }
11. }
12. int main(){
13.     int n, resultado;
14.     printf("\n Digite um numero inteiro positivo : ");
15.     scanf("%d", &n);
16.     resultado = fatorialCauda(n); //fazendo a primeira
17. chamada da função
18.     printf("\n Resultado do fatorial = %d", resultado);
19.     return 0;
20. }
```

Fonte: elaborado pelo autor.

Observe na Figura 4.10 que o mecanismo de funcionamento da recursividade em cauda é diferente. A função *fatorialCauda()* possui apenas uma instância, a qual invoca a função *fatorialAux()*, passando

como parâmetro o valor a ser calculado e o critério de parada. A partir desse ponto inicia a maior diferença entre as técnicas, veja que as instâncias vão sendo criadas, porém, quando chega na última (nesse caso a instância 3), o resultado já é obtido, as funções não precisam retornar o valor para “quem” invocou, gerando otimização na memória, pois não precisa armazenar nenhum ponto para devolução de valores.

Figura 4.10 | Exemplo recursividade em cauda



Fonte: elaborada pela autora.



Pesquise mais

O professor Paulo Feofiloff possui uma página com diversos exercícios. Acesse esse conteúdo e exercite mais essa técnica. Disponível em <<https://goo.gl/QWTMqi>>. Acesso em: 16 jul. 2018, e faça os exercícios propostos para exercitar o uso de funções recursivas.

Com essa seção encerramos o estudo dos algoritmos e técnicas de programação. Não se esqueça que quanto mais você exercitar, mais algoritmos implementar, mais desafios solucionar, mais sua lógica irá se desenvolver e você terá maior facilidade para desenvolver novos programas.

Sem medo de errar

Após conhecer a técnica de recursividade, chegou o momento de implementar o programa para os matemáticos do laboratório onde trabalha. Lembrando que foi solicitado a você implementar o método de Newton para o cálculo da raiz quadrada, porém, usando funções recursivas. Os matemáticos lhe forneceram a seguinte fórmula: $x_n = \frac{x_{n-1}^2 + n}{2x_{n-1}}$, onde x_n é o valor procurado, ou seja, a raiz quadrada e n o valor informado para obter a raiz.

Para implementar essa solução você precisa de um programa que solicite ao usuário um número. Você também deve especificar um valor inicial para a raiz e um critério de parada. Após pesquisar, você verificou que o método iterativo de Newton, consiste em fazer comparações entre os resultados obtidos em cada instância. Será preciso comparar o valor calculado da raiz, com o obtido no passo anterior, até que a diferença entre eles seja menor que o critério de parada. Por isso, também será necessário usar as funções *fabs()* e *pow()*, ambas pertencentes a biblioteca *math.h*. A primeira retorna o valor absoluto de um número e a segunda usada para potenciação.

Como valor inicial você pode adotar a metade do valor informado pelo usuário e como critério de parada o valor 0.001. Quanto menor o valor, mais exato será o cálculo.

Você pode seguir os seguintes passos para a implementação:

1) Crie a função *calcularRaiz()*, de modo que ela receba dois parâmetros, o valor a ser calculado e a raiz anterior que, na primeira chamada, será o chute inicial:

```
float calcularRaiz(float n, float raizAnt)
```

2) Implemente a fórmula passada pelos matemáticos usando a função *pow()*:

```
raiz = (pow(raizAnt, 2) + n)/(2 * raizAnt);
```

 Veja que a raiz anterior é elevada à segunda potência pela função.

3) Calcule a diferença entre o valor obtido no passo 2, com o valor da raiz anterior e veja se é menor que o critério de parada.

```
if (fabs(raiz - raizAnt) < 0.001)
```

4) Caso seja, o valor é satisfatório e pode ser retornado, caso não satisfaça, a função *calcularRaiz()* deve ser novamente

invocada, passando como parâmetro o número e a raiz obtida no passo 2.

Veja no Quadro 4.21 uma possível implementação para essa solução.

Quadro 4.21 | Cálculo da raiz quadrada por aproximações sucessivas

```
1.  #include<stdio.h>
2.  #include<math.h>
3.  float calcularRaiz(float n, float raizAnt)
4.  {
5.      float raiz = (pow(raizAnt, 2) + n)/(2 * raizAnt);
6.      if (fabs(raiz - raizAnt) < 0.001)
7.          return raiz;
8.      return CalcularRaiz(n, raiz);
9.  }
10. void main(){
11.     float numero, raiz;
12.     printf("\n Digite um número para calcular a raiz: ");
13.     scanf("%f", &numero);
14.     raiz = calcularRaiz(numero, numero/2);
15.     printf("\n Raiz quadrada funcao = %f", raiz);
16. }
```

Fonte: elaborado pela autora.

Avançando na prática

Máximo divisor comum

Descrição da situação-problema

Você foi contratado como professor de programação e em conversa com o professor de matemática instrumental, este relatou que os alunos têm dúvidas quanto ao mecanismo para calcular o máximo divisor comum (MDC) e sugeriu que você implementasse essa solução com os alunos, para que eles possam de fato compreender o algoritmo. Como os alunos já têm conhecimento em programação, como você irá implementar a solução?

Resolução da situação-problema

Para implementar essa solução, como os alunos já possuem conhecimento de técnicas de programação, você deve optar

por funções recursivas, a fim de desenvolver ainda mais o raciocínio lógico. Para isso, você terá que recorrer ao método numérico de divisões sucessivas (PIRES, 2015). Para entender o mecanismo, considere como exemplo encontrar o MDC entre 16 e 24.

1ª) É preciso dividir o primeiro número pelo segundo e guardar o resto:

$$\frac{16}{24} = 1, \text{ com resto } 16$$

2ª) O divisor da operação anterior deve ser dividido pelo resto da divisão.

$$\frac{24}{16} = 1, \text{ com resto } 8$$

3ª) O segundo passo deve ser repetido, até que o resto seja nulo e, então, o divisor é o MDC.

$$\frac{16}{8} = 2, \text{ com resto } 0$$

Veja no Quadro 4.22 uma possível implementação para o MDC

Quadro 4.22 | MDC com recursividade

```
1.  #include<stdio.h>
2.  int calcularMDC(int a, int b) {
3.      int r = a % b;
4.      if(r == 0){
5.          return b;
6.      }else{
7.          return calcularMDC(b,r);
8.      }
9.  }
10. void main(){
11.     int n1, n2, resultado;
12.     printf("\n Digite dois números inteiros posi-
13.     tivos: ");
14.     scanf("%d %d",&n1,&n2);
15.     resultado = calcularMDC(n1,n2);
16.     printf("\n MDC = %d",resultado);
    }
```

Fonte: elaborado pela autora.

Faça valer a pena

1. A recursividade é uma técnica de programação usada para tornar o código mais elegante, organizado, o que pode facilitar a manutenção. Essa técnica, em muitos casos, pode ser usada para substituir uma estrutura de repetição iterativa, por exemplo, uma que use o *for*.

Analise as asserções a seguir e a relação proposta entre elas.

I. As estruturas de repetição sempre podem ser substituídas por funções recursivas.

PORQUE

II. Uma função recursiva funciona como um laço de repetição, o qual será interrompido somente quando o caso base for satisfeito.

- a) As asserções I e II são proposições verdadeiras, e a II é uma justificativa correta da I.
- b) As asserções I e II são proposições verdadeiras, mas a II não é uma justificativa correta da I.
- c) A asserção I é uma proposição verdadeira, e a II é uma proposição falsa.
- d) A asserção I é uma proposição falsa, e a II é uma proposição verdadeira.
- e) As asserções I e II são proposições falsas.

2. Para criar uma função recursiva, a sintaxe nada difere das funções gerais, portanto, é necessário informar o tipo de retorno, o nome, e se recebe ou não parâmetros. O grande diferencial das funções recursivas e tradicionais é um comando no corpo da função, que invoca a si própria.

Analise o código a seguir e escolha a opção que representa o que será impresso na linha 11.

```
1.  #include<stdio.h>
2.  int somar(int valor){
3.      if(valor != 0){
4.          return valor + somar(valor - 1);
5.      }
6.      else{
7.          return valor;
8.      }
9.  }
10. int main(){
11.     printf("\n Resultado  = %d",somar(6));
12.     return 0;
13. }
```

- a) Resultado = 21
- b) Resultado = 0
- c) Resultado = 12
- d) Resultado = 6
- e) Resultado = 5

3. A recursividade é uma técnica de programação que deve ser usada com cautela, pois a cada chamada à função novos recursos são alocados na memória, em um processo chamado de empilhamento, que cresce rapidamente com as chamadas, podendo acarretar em um estouro de memória.

A respeito de funções recursivas analise as afirmações e escolha a opção correta.

- I. Existe uma classe específica de funções recursivas chamada de recursividade em cauda, que embora possua a mesma sintaxe no corpo da função, o comportamento é diferente das demais funções.
- II. Uma função é caracterizada como recursiva em cauda quando a chamada a si mesma é a última operação a ser feita no corpo da função.
- III. Em uma função que implementa a recursividade em cauda, a instância que fez a chamada recursiva, depende do resultado da próxima.
- IV. O uso da recursividade em cauda torna opcional o uso do caso base, pois a última instância retornará o valor final esperado.

- a) I – V; II – V; III – V; IV – F.
- b) I – V; II – F; III – F; IV – F.
- c) I – F; II – V; III – V; IV – F.
- d) I – F; II – F; III – F; IV – V.
- e) I – F; II – V; III – F; IV – F.

Referências

ALMEIDA, R.; ZANLORENSSI, G. **A evolução do número de linhas de telefone fixo e celular no mundo**. 2018. Disponível em: <<https://www.nexojornal.com.br/grafico/2018/05/02/A-evolu%C3%A7%C3%A3o-do-n%C3%BAmero-de-linhas-de-telefone-fixo-e-celular-no-mundo>>. Acesso em: 7 jul. 2018.

COUNTER, L. **Lines of code of the Linux Kernel Versions**. Disponível em: <<https://www.linuxcounter.net/statistics/kernel>>. Acesso em: 30 jun. 2018.

FEOFILOFF, P. **Recursão e algoritmos recursivos**. 2017. Disponível em: <<https://www.ime.usp.br/~pf/algoritmos/aulas/recu.html>>. Acesso em: 2 jan. 2018.

MANZANO, J. A. N. G. **Linguagem C**: acompanhada de uma xícara de café. São Paulo: Érica, 2015. 480 p.

_____. MATOS, E.; LOURENÇO, A. E. **Algoritmos**: técnicas de programação. São Paulo: Érica, 2015.

OLIVEIRA, R. de. **Algoritmos e programação de computadores**. Disponível em: <http://www.ic.unicamp.br/~oliveira/doc/mc102_2s2004/Aula19.pdf>. Acesso em: 16 jul. 2018.

PIRES, A. de A. **Cálculo numérico**: prática com algoritmos e planilhas. São Paulo: Atlas, 2015.

SOFFNER, R. **Algoritmos e programação em linguagem C**. São Paulo: Saraiva, 2013.

ISBN 978-85-522-1079-5



9 788552 210795 >