Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais
Campus São João Evangelista
Curso Técnico em Manutenção e Suporte em Informática

Apostila de Introdução à Programação

Karina Dutra de Carvalho Lemos
Rosinei Soares de Figueiredo

São João Evangelista – MG
2016

SUMÁRIO

1 INTRODUÇÃO

Os computadores eletrônicos atuais são máquinas com gigantesca capacidade de processamento. Eles são capazes de processar milhões, bilhões, até mesmo trilhões de operações de ponto flutuante por segundo, dependendo do modelo dos componentes. Em um piscar de olhos os computadores são capazes de resolver problemas que humanos demorariam muito tempo, talvez até uma vida inteira, ou talvez nunca fossem capazes de terminar.

Porém, se não receber instruções, se não for programado, os computadores não são capazes de fazer nada. Eles não são capazes de compreender problemas e definir soluções, assim como os humanos fazem, pelo contrário, eles apenas são capazes de executar tarefas predefinidas, formalizadas, por isso precisam ser programados pelos humanos.

Quando existe um problema, primeiramente os humanos precisam entendê-lo, selecionar uma forma de resolvê-lo, definir o conjunto de passos necessário para alcançar a solução, escrever estes passos de maneira formal e rigorosa, utilizando alguma linguagem que o computador possa entender. Só aí, já com as instruções bem definidas, o computador poderá executar os passos e buscar a solução para o problema.

Em termos mais técnicos, primeiramente é necessário entender o problema e definir uma estratégia lógica, montar um algoritmo que represente tal estratégia e, depois, escrever tal algoritmo em uma linguagem de programação. Estes termos serão abordados com detalhes nas próximas seções.

No contexto da disciplina de Introdução à Programação, serão abordados os conceitos e técnicas relacionados com a lógica de programação, a construção de algoritmos e sua programação em uma linguagem específica.

2 CONCEITOS BÁSICOS

Como comentado na seção anterior, quando se tem um problema, pensa-se em uma estratégia de solução lógica e definem-se os passos formais desta solução (algoritmo), para depois criar o programa propriamente dito. É importante, então, que se tenha em mente, de maneira clara, os seguintes conceitos iniciais:

a. **Lógica de Programação**: é a técnica de encadear pensamentos para atingir determinado objetivo;
b. **Algoritmo**: " Um conjunto finito de regras que provê uma sequência de operações para resolver um tipo de problema especifico" conforme Knuth;
c. **Programa**: um conjunto de instruções escrito em uma linguagem computacional;
d. **Implementação**: é a tradução de algoritmos ou outros modelos de solução em programas;
e. **Linguagem de Programação**: linguagem utilizada para se traduzir algoritmos ou outros modelos de solução em programas.

O entendimento da lógica de programação e a concepção do algoritmo são tarefas fundamentais da programação, pois a partir do conjunto de passos definidos é que se implementa efetivamente o programa. Um algoritmo bem elaborado pode ser facilmente implementado na maioria das linguagens de programação. Por outro lado, um algoritmo mal elaborado pode tornar a implementação complexa ou, até mesmo, levar à construção de um programa que não resolve o problema que deveria resolver.

As seções seguintes abordam os tópicos introdutórios da lógica de programação, a construção de algoritmos e, posteriormente, sua implementação em uma linguagem de programação.

2.1 Introdução à lógica de programação

Como se sabe, os computadores precisam receber instruções para executar seu trabalho. Tais instruções precisam ser formais e escritas em um idioma computacional, que é extremamente mecanicista. Desta forma, ao criar algoritmos, é necessário que o pensamento seja expresso de maneira formal, coerente e logicamente encadeada.

Alguns exemplos de pensamentos logicamente corretos são apresentados a seguir:

a. Todo mamífero é um animal.
   Todo cavalo é um mamífero.
   Portanto, todo cavalo é um animal.

b. Kaiton é país do planeta Stix.
   Todos os Xinpins são de Kaiton.
   Logo, todos os Xinpins são Stixianos.

c. A gaveta está fechada.
   A caneta está dentro da gaveta.
   Precisamos primeiro abrir a gaveta para depois pegar a caneta.

d. Anacleto é mais velho que Felisberto.
   Felisberto é mais velho Marivaldo.
   Portanto, Anacleto é mais velho que Marivaldo.

Nos exemplos apresentados, os pensamentos possuem um sentido lógico, este tipo de pensamento é fundamental na criação de algoritmos, cada etapa/instrução presente em um algoritmo precisa estar disposta na ordem correta e ser coerente com as possibilidades de um computador eletrônico.

Alguns exemplos de pensamentos expressos de maneira encadeada são apresentados a seguir, eles ainda não são algoritmos computacionais, são apenas situações convencionais tomadas como analogia.

Passo a passo para uma troca de lâmpada:

1. pegar uma escada;
2. posicionar a escada embaixo da lâmpada;
3. buscar uma lâmpada nova;
4. subir na escada;
5. retirar a lâmpada velha;
6. colocar a lâmpada nova.

Passo a passo para se utilizar um telefone público:

1. tirar o fone do gancho;
2. ouvir o sinal de linha;
3. introduzir o cartão;
4. teclar o número desejado;
5. se der o sinal de chamar
    5.1 conversar;
    5.2 desligar;
    5.3 retirar o cartão.
6. senão
    6.1 repetir;

Nas descrições apresentadas anteriormente, os passos definidos para se resolver as situações estão ordenados e coerentes (fazem sentido lógico). Assim também precisam ser os algoritmos computacionais, feitos de maneira ordenada e coerente para que o computador possa executá-los corretamente e alcançar a solução.

Não existe uma única maneira de se resolver um problema. É possível que alguém troque uma lâmpada de maneira diferente da apresentada anteriormente, significa também que podem existir mais de um algoritmo para

resolver um determinado problema. Indiferente disso, a coerência e a ordenação lógica devem permanecer. Por exemplo, trocar uma lâmpada colocando a nova antes de remover a velha, seria uma situação incoerente, se algo similar aparecesse em um algoritmo computacional, ele estaria errado.

A lógica de programação apresenta mecanismos para formalização, dedução e análise, que possibilitam a criação de algoritmos coerentes e válidos. Existem mecanismos para representação de tipos primitivos de dados, para declaração e utilização de variáveis e constantes, expressões lógicas e aritméticas, movimentação/atribuição de dados, entrada e saída de dados, estruturas para controle do fluxo de execução e repetição, estruturas de dados, entre outros. Alguns destes mecanismos serão abordados nesta apostila, sendo que, primeiramente, serão feitos os algoritmos utilizando uma linguagem formal semelhante ao Português, denominada Portugol[1], para posterior implementação em C/C++[2].

2.2 Dados

Os computadores somente conseguem enxergar e processar informações na forma de dados. O programa processa os dados de acordo com as instruções definidas no algoritmo, e gera as saídas que representam a solução do problema ou o resultado de alguma tarefa.

Os dados são, na verdade, os valores que serão utilizados na resolução de um problema. Esses valores podem ser fornecidos pelo usuário do programa ou podem ser originados a partir de processamentos (cálculos) ou ainda a partir de arquivos, bancos de dados ou outros programas.

Em Portugol, os tipos de dados básicos são:

---

[1] Portugol é uma pseudolinguagem que permite ao programador pensar no problema em si e não no equipamento que irá executar o algoritmo. Devem ser considerados a sintaxe (em relação à forma) e a semântica (em relação ao conteúdo ou seu significado). Em portugol a sintaxe é definida pela linguagem e a semântica depende do significado que quer se dar ao algoritmo.

[2] Existem várias outras linguagens de programação, muitas delas possuem fortes semelhanças com a linguagem C ou até foram baseadas nela. Assim, iniciar o aprendizado nessa linguagem pode facilitar o aprendizado de outras linguagens posteriormente.

a. **Inteiro**: Qualquer número inteiro (negativo, nulo ou positivo).
   Exemplo: - 100, 0, 1, 2, 1250.
b. **Real**: qualquer número real, nulo ou positivo.
   Exemplo: - 10, - 1.5, 11.2, 0,1, 2, 50.
c. **Caractere**: caracteres alfanuméricos.
   Exemplo: casa, Win31, 123, alfa#2, etc...
d. **Lógico**: valor lógico verdadeiro ou falso
   Exemplo: x > y?

A principal maneira de armazenar dados que serão executados pelos programas é através da utilização de variáveis que será vista no próximo tópico.

2.3 Variáveis

Uma variável é um espaço de memória, identificado através de um nome, reservado na memória principal do computador para armazenar dados de algum tipo.

Os computadores somente conseguem enxergar e processar informações na forma de dados. Assim, por mais complexo que seja o contexto para o qual um programa seja construído, este contexto precisa ser apresentado aos computadores na forma de dados a serem processados. O programa, então, processa os dados de acordo com as instruções definidas no algoritmo, e gera as saídas que representam a solução do problema ou o resultado de alguma tarefa.

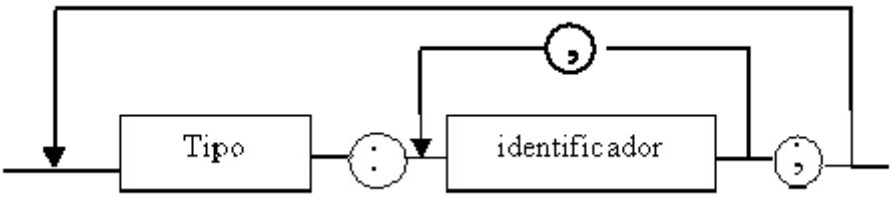
A principal maneira de armazenar dados que serão executados pelos programas é através da utilização de variáveis. Quando um programa é executado, ele solicita ao sistema operacional que reserve espaços na memória principal do computador para o armazenamento de dados, esses espaços são chamados, inicialmente, de variáveis.

Através do identificador da variável, o programa pode facilmente guardar, ler e alterar dados na memória do computador. Quando o programa é

encerrado, aquele espaço de memória fica livre, podendo ser utilizado para outros propósitos.

Em Portugol, para se criar variáveis, utiliza-se o esquema ilustrado na Figura 1:

Figura 1: Esquema de criação de variáveis.



Na criação (declaração) de variáveis é necessário definir o tipo de dado que será guardado nesta variável, colocar dois pontos (:), depois colocar o identificador (nome) da variável, ao final, termina-se a declaração com ponto e vírgula (;). Caso seja necessário declarar mais de uma variável do mesmo tipo, elas podem ser colocadas na mesma declaração, apenas separando por vírgula (,) o seus nomes.

O tipo da variável refere-se ao tipo de dado que será guardado nela. O programa precisa saber qual tipo de dado para definir o tamanho do espaço de memória que deverá alocar, isto porque um número inteiro pode ocupar um espaço com tamanho diferente do espaço necessário para guardar um número decimal, por exemplo.

O identificador da variável refere-se ao nome que será utilizado para referenciá-la. Este nome deve obedecer a algumas restrições, sendo as principais:

a. o nome da variável é *case sensitive*, ou seja, diferencia maiúsculo de minúsculo, por exemplo, **`Media`** é diferente de **`media`** e de **`MEDIA`**;
b. o nome deve ter como primeiro caractere uma letra ou sublinhado (_);
c. após a primeira letra ou sublinhado, o nome só pode conter letras, dígitos ou sublinhados;
d. o nome não pode conter espaços;

e. em um mesmo contexto não é possível definir mais de uma variável, indiferente do tipo, com o mesmo nome (`real: A` e `inteiro: A`, por exemplo, daria erro);
f. o nome não pode ser uma palavra reservada da linguagem utilizada.

Alguns exemplos de declaração de variáveis em Portugol:

```
inteiro: valor;      // a variável valor é do tipo inteiro
real: media;         // a variável media é do tipo real
caractere: nome;     // a variável nome é do tipo caractere
lógico: maior;       // a variável maior é do tipo booleano
```

Cada variável definida no programa usa um local da memória, acessível através do nome dado a variável. O espaço de memória ocupado pelo conteúdo da variável depende do tamanho destes tipos de dados, que variam de acordo com a plataforma sobre o qual o programa foi construído e/ou irá executar. Por exemplo, programas construídos na linguagem C, na especificação ANSI C, consideram 2 bytes para o tipo inteiro, 4 bytes para o tipo real e 1 byte para o tipo caractere.

2.4 Constantes

Constantes são recursos muito similares às variáveis, tendo propósito parecido, guardar algum dado que será processado pelo programa. A diferença é que uma constante, quando definida, não pode mais ter seu valor alterado, ao contrário de uma variável, que pode ser alterada a qualquer momento no programa. Um exemplo clássico de constante é o PI, um valor fixo que não se altera, independente das condições do programa. Um algoritmo que faz algum cálculo utilizando o número PI, poderia declará-lo como uma constante.

Em Portugol, as constantes são declaradas utilizando-se a palavra **`const`** seguida do nome da constante e do valor que ela vai assumir, separados por

espaço. O nome da constante segue as mesmas restrições dos nomes de variáveis, porém, é comum manter os nomes em maiúsculo, como forma de diferenciar mais facilmente as constantes e as variáveis de um algoritmo ou programa.

Alguns exemplos de declaração de constantes em Portugol são apresentados abaixo.

```
const M 10;
const PI 3.1416;
```

2.5 Acesso e atribuição de dados às variáveis

O simples uso do nome da variável faz referência ao valor que está armazenado nela, assim, se for necessário utilizar este valor em alguma operação, basta utilizar o nome da variável no local em que seu valor for necessário.

Por exemplo, considerando que existe uma variável `preco`, e que é necessário somar o valor guardado na variável com um outro número qualquer, isso poderia ser feito da seguinte maneira:

```
45 + preco
```

O valor da variável é obtido automaticamente quando o seu nome é utilizado. A expressão acima não é uma instrução completa, pois ainda faltam elementos que serão vistos mais a frente.

Porém, muitas vezes, os programas precisam guardar nas variáveis os valores que serão processados posteriormente, para isso é necessário a utilização de um operador específico, chamada operador de atribuição, um símbolo que indicará ao programa que algum dado deve ser armazenado na variável. Em Portugol, utiliza-se o símbolo (`<-`), que se assemelha a uma seta, indicando que na variável à esquerda do operador deve ser armazenado algum valor, especificamente, ou o resultado de alguma expressão, apresentada à

direita do operador. Alguns exemplos da utilização do operador de atribuição são apresentados abaixo:

```
real: nota1, nota2, soma;
nota1 <- 3.5;
nota2 <- 6.1;
soma <- nota1 + nota2;
```

No exemplo acima é apresentado um fragmento de programa onde, inicialmente, são declaradas três variáveis do tipo `real`, posteriormente estas variáveis recebem valores, sendo que as duas primeiras são preenchidas com valores constantes e a terceira é preenchida com o resultado de uma operação sobre as duas anteriores, no caso, uma soma.

Assim como na matemática, várias outras operações aritméticas podem ser aplicadas sobre os dados em processamento por um programa, este operadores são apresentados na seção seguinte.

2.6 Operadores aritméticos

Os símbolos das operações aritméticas fundamentais são:

a. A multiplicação é dada através do operador `*` (asterisco);
   Exemplo: `z <- x * y;`
b. A soma é realizada através do operador `+`;
   Exemplo: `z <- x + y;`
c. A subtração é dada através do operador `-`;
   Exemplo: `z <- x - y;`
d. A divisão (considerando números reais) utiliza o operador `/`;
   Exemplo: `z <- x / y;`
e. A divisão (considerando números inteiros) é dada por `div`;
   Exemplo: `z <- x div y;`
f. O resto de uma divisão é dada pelo comando `mod`.
   Exemplo: `z <- x mod y;`

g. O cálculo de x^y (potenciação) é dado pelo símbolo `^`.

   Exemplo: `z <- x^y;`

h. A raiz de um valor é extraída através do comando `raiz()`.

   Exemplo: `z <- raiz(x);`

Assim como na matemática, em algoritmos as expressões podem ser compostas por mais de um operador, nesses casos, a prioridade natural de execução destes operadores é mostrado na Tabela 1:

Tabela 1: Ordem de prioridade dos operadores matemáticos

1ª prioridade	`^ raiz`
2ª prioridade	`* / DIV MOD`
3ª prioridade	`+ -`

Porém, assim como acontece na matemática, estas prioridades podem ser alteradas. Há, contudo, uma pequena diferença, enquanto na Matemática são utilizados parênteses (), colchetes [] e chaves {} para se alterar as prioridades dos operadores, em algoritmos são utilizados apenas parênteses, que podem ser encadeados. Outro detalhe é que as expressões devem ser escritas em linha, conforme os exemplos a seguir:

Em Matemática se faz:	Em Portugol se faz:
$x = 1 + \left[\frac{2}{3} + (5 - 3)\right]$	$x = 1 + (2/5 + (5 - 3))$
$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$	$x = (-b + \text{raiz}(b\text{^}2 - 4 * a * c))/2 * a$

2.7 Entrada e saída de dados

Os programas de computador, sobretudo os softwares aplicativos, comumente interagem com os usuários, recebendo informações e enviando informações ao utilizador. Existem diversas maneiras de um programa se comunicar com seus usuários, e cada linguagem de programação trata esse aspecto de maneira peculiar. Inicialmente serão considerados os comandos básicos de escrita e leitura de tela, sendo eles:

a. `escreva()`: escreve na tela as informações (variável, valor ou expressão) colocadas dentro do parêntese, separadas por vírgula, caso sejam mais de uma.

   Exemplo:

```
inteiro: a, b;
a <- 10;
b <- 5;
escreva("Valores");
escreva(a);
escreva(b);
escreva("A soma deu ", a + b);
```

Este exemplo cria duas variáveis, atribui valores fixos a elas, depois escreve na tela a palavra `"Valores"`, seguida dos valores das duas variáveis. Em seguida, escreve na tela o termo `"A soma deu "` seguido do valor da soma dos valores das variáveis.

b. `leia()`: lê valores digitados pelo usuário e os armazena nas variáveis colocadas dentro do parêntese, separadas por vírgula, caso sejam mais de uma.

   Exemplo:

```
inteiro: a, b, soma;
escreva("Digite dois números para a soma:");
leia(a);
```

```
leia(b);
soma <- a + b;
escreva("A soma deu ", soma);
```

Este exemplo cria três variáveis, mas não atribui valores fixos a elas. Ao contrário, ele escreve uma mensagem na tela solicitando que o usuário informe os números a serem somados. Depois disso, ele lê os números digitados pelo usuário. Em seguida, soma esses valores, jogando resultado na variável `soma`. Por fim, escreve na tela a expressão `"A soma deu "` seguida do valor da variável `soma`.

2.8 Comentários

Comentários são recursos utilizados para se indicar que certos trechos do código não devem ser executados. Eles são utilizados para inserir explicações sobre os comandos em meio ao código, manter exemplos ou informações sobre o arquivo, enfim, para inserir informações adicionais junto aos códigos de maneira que estas informações não gerem erro na interpretação dos comandos.

Serão utilizados, durante a elaboração de algoritmos em Portugol, os símbolos indicativos de comentários que ocorrem na maioria das linguagens de programação, sendo eles:

a. Comentários de linha: duas barras (//) no início no comentário.

   Exemplo:

```
// Comentário ocupando uma linha inteira.
caracter: op;
op <- 'R'; //Comentário em parte de uma linha.
```

b. Comentários de múltiplas linhas: barra e asterisco (/*) para abrir o comentário e asterisco e barra (*/) para fechar o comentário.

Exemplo:

```
/*
Comentário ocupando várias linhas.
Poderia ser, por exemplo, a data de criação do
arquivo, junto com o nome do autor e uma
explicação geral sobre o que está contido nele.
Eles podem ser colocados em qualquer lugar do
arquivo, não necessariamente no início.
*/
caracter: op;
op <- 'R';
```

2.9 Regras básicas para elaboração de algoritmo

A estrutura básica de um algoritmo em Portugol é montada da seguinte maneira:

```
início
     <comandos de declaração de variáveis>
     <comandos de processamento e iteração>
fim
```

As palavras `início` e `fim` demarcam o início e o fim do algoritmo, definindo um bloco onde são inseridos os códigos. Estes códigos podem ser as declarações de variáveis, geralmente feitas no início do algoritmo, e os demais comandos de processamento e iteração com usuários, dentre outros.

Como a elaboração de algoritmos em Portugol é uma tarefa formal, algumas regras precisam ser obedecidas, sendo elas:

a. incluir comentários pelo menos nas linhas mais importantes do algoritmo;
b. usar nomes significativos para as variáveis e constantes, de maneira que se possa entender facilmente o que elas significam;

c. grifar as palavras chaves do Portugol (início, fim, escreva, leia, por exemplo);
d. colocar ponto e vírgula ao final de cada comando, exceto nos comandos de abertura e fechamento de bloco (início e fim, por exemplo);
e. identar os comandos (recuar os blocos de comandos em relação ao seu elemento pai), o que facilita a legibilidade do algoritmo e reduz a possibilidade de erros.

2.10 Estruturas de controle

Até o momento, todos os exercícios e exemplos apresentados foram de algoritmos cujo fluxo de instruções é direto, porém isso não é o que acontece na maioria das vezes. É comum os programas terem que decidir entre executar ou não determinada tarefa, ou escolher executar um bloco de comandos entre mais de uma possibilidade. Neste casos são utilizadas as chamadas estruturas de controle (ou desvio), que permitem ao programa decidir qual fluxo deve seguir.

Porém, antes de começar a trabalhar com as estruturas de controle, é necessário conhecer os princípios básicos das expressões lógicas e seus operadores, recursos necessários a este tipo de estrutura.

2.10.1 Expressões lógicas

Como mencionado anteriormente, os computadores precisam tomar decisões de acordo com as orientações contidas nos programas. Estas orientações adicionadas aos programas geralmente são resultantes da análise de expressões lógicas, o que pode ser entendido, de maneira simples, como comparações (relações) entre dados do mesmo tipo, estas comparações podem resultar em verdadeiro ou falso.

Para a representação destas comparações, em Portugol são utilizados os operadores relacionais (baseados nas operações relacionais convencionais da matemática), mostrados na Tabela 2:

Tabela 2: Operadores relacionais

Operador	Função	Exemplos
=	Igual a	`3 = 3, x = y`
>	Maior que	`5 > 4, x > y`
<	Menor que	`3 < 6, x < y`
>=	Maior ou igual a	`5 >= 3, x >= 0`
<=	Menor ou igual a	`3 <= 5, x <= y`
<>	Diferente de	`8 <> 9, x <> y`

Abaixo são apresentados alguns exemplos mais elaborados, incluindo sua análise, para verificação do resultado:

EXEMPLO 1: Considere as variáveis x, y e z com valores 3, 7 e 2, respectivamente, para análise das expressões.

a. Expressão 1

```
2 * 4 = 24 / x
2 * 4 = 24 / 3
    8 = 8
      V
```

b. Expressão 2

```
17 mod x < 22 mod y
17 mod 3 < 22 mod 7
       2 < 1
         F
```

c. Expressão 3

```
x * 5 div 4 <= 3^z / 0,5
3 * 5 div 4 <= 3^2 / 0,5
   15 div 4 <= 9 / 0,5
          3 <= 18,0
             V
```

d. Expressão 4

```
z + 8 mod y >= x * 6 – 15
2 + 8 mod 7 >= 3 * 6 – 15
      2 + 1 >= 18 -15
          3 >= 3
             V
```

Nas expressões lógicas anteriores, percebe-se que, indiferente se os operandos são valores, variáveis ou expressões, ao final é gerado um valor (lógico) de verdadeiro ou falso. É esse resultado de verdadeiro ou falso que será utilizado pelas estruturas de controle para tomar decisões dentro do programa.

Antes de apresentar as estruturas de controle, é ainda importante analisar uma outra situação, quando as expressões lógicas precisam ser combinadas, formando expressões lógicas compostas. Como exemplo, pode-se considerar um programa que precisa verificar se um número é par e, também, se esse número é maior que 100.

Nesse caso, existiriam duas expressões independentes utilizando operadores relacionais (considerando que o número está guardado em `n`):

`n mod 2 = 0` resulta em `V` se `n` for par e em `F` se n for ímpar.

`n > 100` resulta em `V` se `n` for maior que `100` e em `F`, caso contrário

Porém, essas duas expressões precisam ser combinadas para gerar apenas um resultado, que represente a condição como um todo. Nesses casos, são utilizados os operadores lógicos, que permitem a combinação e alteração

das expressões ou valores lógicos. Na Tabela 3 temos os operadores lógicos básicos:

Tabela 3: Operadores lógicos

Operador	Função	Exemplo
não	Negação	não (a > b)
e	Conjunção	a <= b e c > d
ou	Disjunção	x = y ou y = z

O operador **'não'** é aplicável quando se deseja inverter um valor lógico, diferente dos demais operadores, que são utilizados quando é necessário combinar valores lógicos. Este operador inverte um valor lógico resultante de alguma expressão ou armazenado em uma vaiável ou constante. O V – verdadeiro se transforma em F – falso, e vice-versa, quando sofre o efeito de um operador **'não'**, conforme se observa na Tabela 4 a seguir:

Tabela 4: Negação de proposição

Valor de uma expressão **A**	Valor da expressão negado **não A**
V	F
F	V

O operador **'e'** (aparece como **∧** em algumas definições) relaciona valores lógicos armazenados em variáveis ou constantes ou resultantes de expressões lógicas. Como resultado, ele devolve V – verdadeiro, caso os dois valores sejam verdadeiros ou F – falso, em qualquer outro caso, conforme se observa na Tabela 5:

Tabela 5: Conjunção de proposições

Valor da expressão 1 **A**	Valor da expressão 2 **B**	Conjunção das expressões **A e B**
V	V	V

V	F	F
F	V	F
F	F	F

O operador **'ou'** (aparece como **V** em algumas definições) relaciona valores lógicos armazenados em variáveis ou constantes ou resultantes de expressões lógicas. Como resultado, ele devolve V – verdadeiro, caso algum dos dois valores seja verdadeiros ou F – falso, caso nenhum deles seja verdadeiro, conforme se observa na Tabela 6 seguinte:

Tabela 6: Disjunção de proposições

Valor da expressão 1 A	**Valor da expressão 2 B**	**Disjunção das expressões A ou B**
V	V	V
V	F	V
F	V	V
F	F	F

O sentido expresso na condição indica qual operador deve ser utilizado. No exemplo apresentado anterirormente "*um programa que precisa verificar se um número é par e, também, se esse número é maior que 100*", é possível identificar duas condições isoladas conectadas por um 'e', formando uma expressão composta. Assim, a expressão final seria:

`(n mod 2 = 0)` **e** `(n > 100)` retorna V só se os dois lados forem V.

Numa expressão com mais de um operador lógico, esses operadores são analisados obedecendo a seguinte ordem de prioridade, conforme Tabela 7:

Tabela 7: Ordem de prioridade

Prioridade	**Operadores**
1ª	não
2ª	e
3ª	ou

Em expressões compostas por vários tipos de operadores, eles são analisados conforme a ordem de prioridade mostrada na Tabela 8:

Tabela 8: Ordem de prioridade de vários operadores

Prioridade	Operadores
1ª	parênteses mais internos
2ª	Operadores aritméticos
3ª	Operadores relacionais
4ª	Operadores lógicos

Quando o conector não puder ser identificado diretamente, é necessário analisar bem a situação para se escolher corretamente os conectores a serem utilizados. Até agora foram apresentados apenas os princípios básicos relacionados com as expressões lógicas, mas é importante saber que existem outros operadores e, além disso, mecanismos avançados de manipulação (simplificação, por exemplo) de expressões, que não foram incluídos no escopo deste material.

As estruturas de controle (desvio) que utilizam as expressões lógicas para tomarem decisão serão apresentadas nas seções seguintes.

2.10.2 Desvio condicional simples

O desvio condicional simples é uma estrutura de controle que permite ao programa **executar ou não um bloco de comandos, dependendo do resultado da avaliação de uma expressão lógica** (o bloco é executado se o resultado da expressão for V - verdadeiro).

Um bloco de desvio condicional simples é formado utilizando os termos **se**, **então** e **fim se**, dispostos da seguinte forma:

```
se (condição) então
     lista de comandos...
fim se
```

Como exemplo, considera-se um programa que irá escolher o maior entre dois números inteiros digitados pelo usuário, mostrando o resultado ao final. Uma possível versão para esse algoritmo seria o seguinte:

Algoritmo 1: Máximo

```
início
     inteiro: a, b;
     escreva("Algoritmo de Máximo.");
     escreva("Digite o primeiro número: ");
     leia(a);
     escreva("Digite o segundo número: ");
     leia(b);
     se (a < b) então
          a <- b;
     fim se
     escreva("O maior é: ",  a);
fim
```

A parte em destaque do algoritmo possui uma estrutura de desvio e, dentro dela, uma tarefa que poderá ou não ser executada, dependendo da situação.

O objetivo do algoritmo é guardar o número maior na própria variável `a` (já utilizada na leitura dos dados), o que não tem problema, pois essa variável não será utilizada para mais nada além disso.

Caso o usuário tenha digitado um primeiro número (que está em `a`) maior ou igual ao segundo (que está em `b`), o comando contido no bloco de desvio **não será** executado, uma vez que a expressão lógica resultaria em F – falso.

Por outro lado, caso o usuário tenha digitado um primeiro número (em `a`) menor que o segundo (em `b`), o comando contido no bloco de desvio **será** executado, uma vez que a expressão lógica resultaria em V – verdadeiro.

Quando executa, o comando atribui à variável `a` o valor que está contido na variável `b`, garantido que `a` esteja sempre com o valor maior.

2.10.3 Desvio condicional composto

O desvio condicional composto é uma estrutura de controle que permite ao programa **selecionar, entre dois ou mais blocos de comandos, aquele que deverá ser executado, dependendo do resultado da avaliação de uma expressão lógica** (o bloco a ser executado será aquele cujo resultado da expressão for V – verdadeiro, ou o último bloco, se nenhum dos anteriores foi executado).

Um bloco de desvio condicional composto é formado utilizando os termos **`se`**, **`então`**, **`senão se`**, **`senão`** e **`fim se`**.

Quando há apenas duas condições excludentes, a formação é mais simples, os elementos ficam dispostos da seguinte forma:

```
se (condição1) então
     lista principal de comandos...
senão
     lista alternativa de comandos...
fim se
```

Neste caso, se o resultado da condição for verdadeiro, será executada a lista principal de comandos, caso contrário, por exclusão, será executada a lista alternativa. Sempre que o programa for acionado, apenas um dos blocos será executado, dependendo do resultado da condição para o contexto (valor guardado em uma variável, por exemplo) atual.

Um exemplo em que há uma estrutura de desvio com dois possíveis blocos de execução é apresentado a seguir:

Algoritmo 2: Par ou ímpar

```
início
     inteiro: n;
     escreva("Algoritmo de Par ou Ímpar.");
     escreva("Digite um número: ");
     leia(n);
     se (n mod 2 = 0) então
          escreva("O número é PAR.");
     senão
          escreva("O número é ÍMPAR.");
     fim se
fim
```

A parte em destaque do algoritmo apresenta uma estrutura de controle com dois blocos de comandos. Se o resultado da condição for V, o primeiro bloco será executado (o usuário será informado que o número digitado por ele é par), caso contrário, o segundo bloco será executado (o usuário será informado que o número é ímpar).

Porém, existem outras situações em que a quantidade de alternativas pode ser maior que duas, nestes casos os elementos ficam dispostos da seguinte forma:

```
se (condição1) então
     Primeira lista de comandos...
senão se (condição2) então
     Segunda lista de comandos...
senão se (condição 3) então
     Terceira lista de comandos...
senão se (condição 4) então
     Quarta lista de comandos...
senão
     Lista alternativa de comandos...
fim se
```

Nestes casos, a estrutura possui várias condições que serão analisadas em sequência. O bloco a ser executado será aquele cujo resultado da expressão for V – verdadeiro. Como a estrutura só permite a execução de um bloco, se houver mais de um com condição verdadeira, apenas o primeiro será executado.

Caso não haja nenhum bloco com resultado verdadeiro para a expressão, o último bloco de comando será executado, aquele que não precisa obedecer uma condição e serve como bloco alternativo.

Um exemplo em que há uma estrutura de desvio com mais de dois possíveis blocos de execução é apresentado a seguir:

Algoritmo 3: Situação do Estudante

```
início
     real: nota1, nota2, nota3, nota4, soma;
     escreva("Algoritmo Situação do Estudante.");
     escreva("Nota do primeiro bimestre: ");
     leia(nota1);
     escreva("Nota do segundo bimestre: ");
     leia(nota2);
     escreva("Nota do terceiro bimestre: ");
     leia(nota3);
     escreva("Nota do quarto bimestre: ");
     leia(nota4);
     soma <- nota1 + nota2 + nota3 + nota4;
     se (soma < 45) então
          escreva("Estudante REPROVADO.");
     senão se (soma >= 45 e soma < 60)
          escreva("Estudante em RECUPERAÇÃO.");
     senão
          escreva("Estudante APROVADO.");
     fim se
fim
```

A parte em destaque do algoritmo apresenta uma estrutura de controle com três blocos de comandos. Se o resultado da primeira condição for V, o primeiro bloco será executado (aparecerá a mensagem de estudante reprovado). Caso contrário, ainda existirão duas possibilidades, então a segunda condição será analisada, se ela for V, o segundo bloco será executado (mensagem de estudante em recuperação), caso contrário, o último bloco será executado como bloco alternativo (mensagem de estudante aprovado).

2.11 Estruturas de repetição

Estruturas de repetição (loops) são usadas para reduzir linhas de código repetitivo. Para isso, são utilizados comandos que indicam ao programa que ele deve repetir a execução de determinado trecho de código, sem que o programador tenha que escrever tais códigos várias vezes.

Serão apresentadas as duas principais estruturas de repetição existentes, mas é importante esclarecer que as linguagens de programação possuem, além destas, outras variações de estruturas de repetição.

2.11.1 Comando enquanto/faça

Estra estrutura de repetição, o `enquanto/faça`, define um bloco de comandos que deve ser executado repetidamente enquanto uma determinada condição (expressão lógica) for V – verdadeiro, tal condição é avaliada antes do início de cada repetição, deixando que a repetição seja feita somente se o resultado da expressão for verdadeiro.

A estrutura é construída da seguinte maneira:

```
enquanto (condição) faça
     ...
     Lista de comandos;
     ...
fim enquanto
```

Como exemplo, considere um programa que precisa escrever os números de zero a cem na tela. São apresentados dois algoritmos que geram o mesmo resultado, porém um utiliza estrutura de repetição e outro não.

Exemplo sem loop

```
início
     inteiro: num;
     num <- 0;
     escreva(num);
     num <- num + 1;
     escreva(num);
     num <- num + 1;
     escreva(num);
     num <- num + 1;
     ...
     ...
     ...
     num <- num + 1;
     escreva(num);
     num <- num + 1;
     escreva(num);
fim
```

Exemplo com loop

```
início
     inteiro: num;
     num <- 0;
     enquanto (num < 101) faça
          escreva(num);
          num <- num + 1;
     fim enquanto
fim
```

A quantidade de linhas de código do algoritmo que não utiliza estrutura de repetição (esquerda) é expressivamente maior do que a do algoritmo que utiliza (direita), mesmo a visualização tendo suprimindo várias linhas do primeiro, afim de comportá-lo na página. Estes algoritmos apresentados são simples, mas a situação pode se agravar muito mais em algoritmos maiores, que podem se tornar inviáveis de serem construídos sem a utilização de recursos como as estruturas de repetição.

2.11.2 Comando para/até/faça

A estrutura de repetição `para/até/faça` é uma estrutura bastante utilizada, ele pode ser aplicada nas mesmas situações em que a estrutura anterior se aplica, porém, é aplicada com mais intuitividade em algoritmos que precisam processar valores gerados em sequencias e/ou armazenados em vetores e matrizes (serão vistos posteriormente).

A estrutura é construída da seguinte maneira:

```
para variável de valor inicial até valor final faça
     ...
     Lista de comandos;
     ...
fim para
```

Considere o mesmo exemplo utilizado na estrutura anterior: um programa que precisa escrever os números de zero a cem na tela. Utilizando o comando `para/até/faça`, ele ficaria da seguinte maneira:

```
início
     inteiro: num;
     para num de 0 até 100 faça
          escreva(num);
     fim enquanto
fim
```

Em Portugol, a condição lógica já fica implícita no comando, diferentemente da estrutura `enquanto/faça`, onde ela é inserida explicitamente. No comando `para/até/faça`, o que fica estipulado é um intervalo de repetição. Apesar das diferenças, a maioria das situações pode ser resolvida com um ou com outro, restando ao programador escolher aquele que for mais intuitivo e adequado.

2.11.3 O problema do loop infinito

Ao construir algoritmos utilizando estruturas de repetição, é importante ficar atento se às condições foram definidas corretamente. A falta de uma expressão que estabeleça uma condição ou uma expressão que não seja adequada pode acarretar erros no programa. Um destes erros é chamado de loop infinito. Ele ocorre quando o bloco de comandos a ser repetido fica se repetindo infinitamente, bloqueando os recursos do computador e deixando de finalizar o processamento necessário. Um exemplo é apresentado abaixo.

```
início
     inteiro: i;
     i <- 0;
     enquanto (i < 5) faça
          escreva (i);
     fim enquanto
fim
```

O algoritmo anterior irá escrever o número 0 na tela, infinitamente, pois a condição será sempre V. Para corrigi-lo, seria necessário alterar o valor da variável `i` a cada repetição para que, em algum momento, a condição se torne F.

```
início
     inteiro: i;
     i <- 0;
     enquanto (i < 5) faça
          escreva(i);
          i <- i + 1; //Altera o valor da variável i.
     fim enquanto
fim
```

Este problema pode acontecer com qualquer estrutura de repetição, se as condições de controle não forem estabelecidas corretamente.

3 INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO C/C++

"A linguagem C foi criada inicialmente por Dennis M. Ritchie e Ken Thompson no laboratório Bell em 1972, baseada na linguagem B, de Thompson" (MIZRAHI, 2008).

Após sua criação e popularização, C passou a ser uma da linguagens mais utilizadas no mundo. Até hoje, aparece entre as mais utilizadas, muitas vezes dividindo espaço com outras linguagem que, inclusive, foram desenvolvidas com base na linguagem C.

Por ser se tornado popular, novos recursos foram sendo adicionados à linguagem, o que deu origem à linguagem C++. C++ nasceu como uma evolução da linguagem C. Por este motivo, C e C++ são, parte uma da outra, preservando os recursos que o C já possuía.

Quando um programa é construído em C++, ele pode então usar recursos mais antigos, originais da linguagem C, além de recursos mais novos, introduzidos pela linguagem C++. Por este motivo é normal referir à linguagem C/C++, como esta mescla das duas, o que é o caso deste material.

3.1 Visão geral do processo de criação de programas

O processo de criação de programas em C/C++ é composto pelos seguintes passos fundamentais:

a. Codificação do programa: utilizando algum editor de texto, o programador escreve os comandos que farão parte do programa, obedecendo a sintaxe da linguagem;
b. Compilação do programa: os comandos digitados pelo programador são analisados e, caso estejam sintaticamente corretos, são transformados em linguagem de máquina;
c. Link edição do programa: para gerar o programa final, os comandos digitados pelo programador, já transformados, são combinados com

os comandos existentes nas bibliotecas básicas da linguagem, ou outras bibliotecas adicionais.

A codificação de programas em C/C++ pode ser feita em qualquer editor de texto. Após a escrita dos códigos, o programador pode solicitar sua compilação e link edição via linha de comandos, desde que estas ferramentas estejam instaladas no sistema.

Porém, é possível fazer tudo isso de maneira mais simples utilizando algum IDE (ambiente integrado de desenvolvimento). Os IDEs são ferramentas que integram várias ferramentas necessárias ao desenvolvimento de programas, oferecendo ainda uma série de recursos adicionais, como coloração de código, inserção e correção automática de código, análise de erros de sintaxe, entre outros.

Existem um gama de IDEs para o desenvolvimento em C/C++, dos quais citam-se:

a. Code::Blocks;
b. Bloodshed Dev C++;
c. Microsoft Visual C++.

O Code::Blocks e o Dev C++ são IDEs mais simples, enquanto o Visual C++ (parte do software Microsoft Visual Studio) possui uma interface mais cheia, em razão da maior quantidade de componentes que possui. Cada um dos IDEs possui suas particularidades, embora todos eles suportem os recursos básicos de programação em C/C++.

3.2 Estrutura básica de um programa em C

Os programas em C/C++ são compostos de uma ou mais funções (funções serão abordadas com mais detalhes nas seções seguintes deste documento) , sendo que uma delas é obrigatória, a função `main`. Esta função especial deve existir em todo programa, pois é ela o ponto de partida da

execução do programa. A função `main` é um bloco de comandos formado da seguinte maneira:

```
int main ()
{
     declaração de variáveis;
     instrução;
     ...................
     instrução;
     return 0;
}
```

A função `main` pode ser escrita de algumas maneiras diferentes, o que poderá ser observado na sequência dos estudos. Especificamente, se o programa for feito no Visual C++, ela precisa ser escrita da seguinte maneira:

```
int _tmain(int argc, _TCHAR* argv[])
{
     declaração de variáveis;
     instrução;
     ...................
     instrução;
     return 0;
}
```

OBS: O Visual C++ pode ser configurado para aceitar, também, a versão anterior da função.

Muitas vezes é necessário incluir bibliotecas[3] de funções ao programas, e isso acaba sendo considerado parte da estrutura básica também. Um exemplo é apresentado a seguir.

[3] Bibliotecas, neste contexto, são arquivos que contêm conjuntos de funções já implementadas. Estas funções podem ser utilizadas pelos programadores para agilizar o desenvolvimento de programas, uma vez que eles não precisam refazer códigos que já existem, apenas incorporá-los.

```
#include <iostream>

int main()
{
     declaração de variáveis;
     instrução;
     ...................
     instrução;
     return 0;
}
```

Assim como em Portugol, a elaboração de programas em C/C++ é uma tarefa formal, algumas regras precisam ser obedecidas, entre elas:

a. incluir comentários pelo menos nas linhas mais importantes do algoritmo;
b. usar nomes significativos para as variáveis e constantes, de maneira que se possa entender facilmente o que elas significam;
c. colocar ponto e vírgula ao final de cada comando, exceto nos comandos de abertura e fechamento de bloco (`if`, `else` e `for`, por exemplo);
d. identar os comandos (recuar os blocos de comandos em relação ao seu elemento pai), o que facilita a legibilidade do algoritmo e reduz a possibilidade de erros.

3.3 Declaração de variáveis

Como já citado previamente, uma variável é um espaço de memória reservado para armazenar algum certo tipo de dados e possui um nome como identificador. Em C/C++, elas são declaradas da seguinte maneira:

```
tipo nome;
```

O primeiro termo refere-se ao tipo de dado que será guardado na variável e o segundo termo é o nome dado à variável. É possível declarar mais de uma variável do mesmo tipo em uma mesma linha de declaração, separando-as por vírgula. É possível, também, atribuir valores às variáveis no momento da sua declaração. Alguns exemplos foram adicionados abaixo.

```
int num1;
int num2;
float nota1, nota2, nota3, soma = 0.0;
char sexo;
```

A expressão `soma = 0.0` cria a variável `soma` e atribui o valor `0.0` a esta variável no momento em que ela é criada, as demais ficam com valores indefinidos. É importante ressaltar que o símbolo = é um operador de atribuição em C/C++, diferentemente do símbolo do atribuição usado em Portugol (os operadores serão apresentados mais a frente).

3.4 Principais tipos de dados

Como mencionado na seção anterior, para se declarar uma variável é necessário que se indique o tipo do dado que será guardado nela. Os principais tipos de dados que podem ser utilizados na linguagem C/C++ são apresentados no Quadro 1:

Quadro 1: Tipos de dados da linguagem C/C++

Grupo	Nome do tipo*	Observações sobre tamanho e precisão
Tipo para caracteres	**char**	Exatamente um byte de tamanho. Ao menos 8 bits.
	char16_t	Não menor que o char. Ao menos 16 bits.
	char32_t	Não menor que char16_t. Ao menos 32 bits.
	wchar_t	Pode representar o maior

		conjunto de caracteres suportado.
Tipos para números inteiros (com sinal)	**signed char**	Mesmo que char. Ao menos 8 bits.
	signed **short** *int*	Não menor que o char. Ao menos 16 bits.
	signed **int**	Não menor que o short. Ao menos 16 bits.
	signed **long** *int*	Não menor que o int. Ao menos 32 bits.
	signed **long long** *int*	Não menor que o long. Ao menos 64 bits.
Tipos para números inteiros (sem sinal)	**unsigned char**	Mesmo tamanho dos seus equivalentes com sinal.
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Tipos para números de ponto flutuante	**float**	
	double	Precisão não menor que float.
	long double	Precisão não menor que double.
Tipo lógico	**bool**	
Tipo para vazio	**void**	Sem armazenamento.
Ponteiro para nulo	**decltype(nullptr)**	

OBS: Os nomes de alguns dos tipos podem ser escritos de maneira abreviada, dispensando os termos signed e int escritos em itálico na tabela acima, o que não causa ambiguidade no código. Ex.:* signed *short* int *pode ser escrito como* signed *short*, *short* int*, ou simplesmente* short*; todas essas formas se referem ao mesmo tipo fundamental.

3.5 Declaração de constantes

A criação de constantes em C/C++ é feita utilizando-se uma das diretivas[4] que a linguagem oferece, no caso, o comando `#define`, da seguinte forma:

```
#define NOME VALOR
```

[4] Diretivas são comandos permitem ao programador fazer indicações especiais ao pré-processador, um programa que axamina o código antes da compilação. As diretivas, podem, por exemplo, indicar a inclusão de uma biblioteca, definir a criação de constantes, entre outras coisas.

Nesse caso, uma constante `NOME` é criada, armazenado o que for colocado como `VALOR`. A linguagem só permite a criação de constantes para armazenar os tipos básicos, já apresentados anteriormente. Elas devem ser criadas fora dos blocos de comandos, geralmente no início do arquivo.

Como é uma diretiva, não é necessário colocar o comando de atribuição e o ponto e vírgula. Alguns exemplos são apresentados abaixo:

```
#define PI 3.14
#define X 100
#define TAM_MAX 1024
#define FEMININO 'F'
```

Também é possível declarar constantes em C/C++ utilizando uma maneira parecida com a declaração de variáveis, apenas adicionando a palavra const à declaração, como no exemplo a seguir:

```
const int TAM_MAX = 10;
```

3.6 Operadores

A linguagem C/C++ possui uma série de operadores que podem ser utilizados nos programas. A maioria desses operadores é baseada nos operadores matemáticos, assim como em Portugol, com pequenas diferenças.

Um dos operadores mais utilizados nos programas é o operador de atribuição. Em C/C++, a atribuição é feita utilizando o operador = como podemos visualizar no exemplo abaixo:

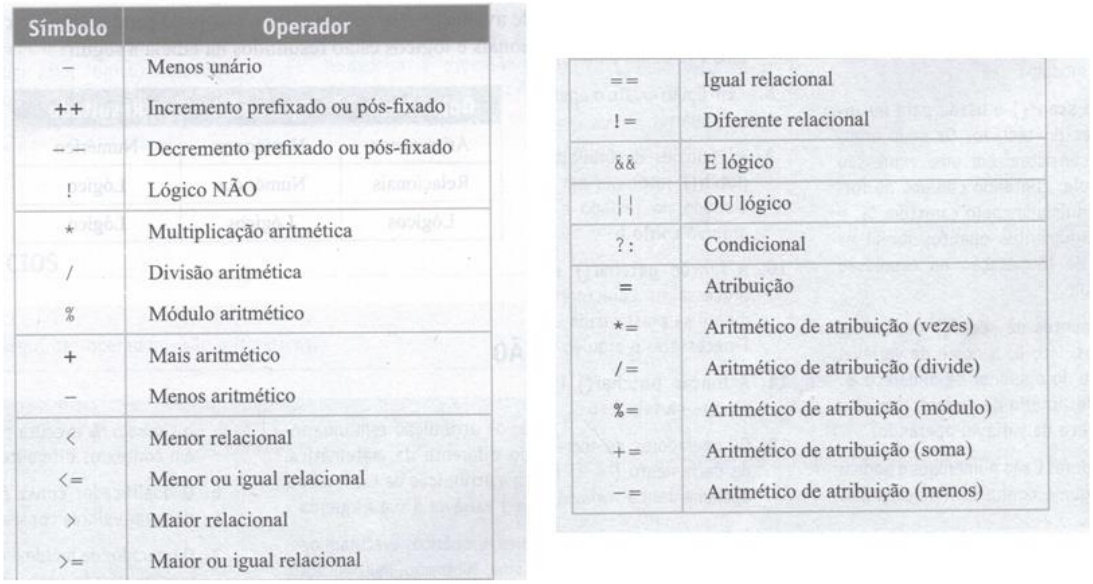
```
int a;
a = 10;   //Se lê: "a recebe 10".
```

O exemplo anterior cria uma variável `a` e, em seguida, atribui a esta variável o valor `10`. Para não haver confusão entre os operadores, o igual

relacional (quando um dado está sendo comparado com outro) é feito utilizando o operador == (dois iguais).

Os operadores de atribuição e de igual relacional são as duas principais diferenças entre os operados da linguagem Portugol para os operadores básicos da linguagem C/C++. Como os demais são muito parecidos, eles são apresentados, em ordem de precedência, de maneira resumida na Figura 2:

Figura 2: Ordem de precedência dos operadores em C/C++



Símbolo	Operador
–	Menos unário
++	Incremento prefixado ou pós-fixado
– –	Decremento prefixado ou pós-fixado
!	Lógico NÃO
*	Multiplicação aritmética
/	Divisão aritmética
%	Módulo aritmético
+	Mais aritmético
–	Menos aritmético
<	Menor relacional
<=	Menor ou igual relacional
>	Maior relacional
>=	Maior ou igual relacional
==	Igual relacional
!=	Diferente relacional
&&	E lógico
||	OU lógico
?:	Condicional
=	Atribuição
*=	Aritmético de atribuição (vezes)
/=	Aritmético de atribuição (divide)
%=	Aritmético de atribuição (módulo)
+=	Aritmético de atribuição (soma)
–=	Aritmético de atribuição (menos)

Como se pode observar na figura, a linguagem C/C++ possui novos operadores que simplificam determinadas operações. Entre esses novos operadores estão os operadores de incremento e decremento, que adicionam ou subtraem um unidade em alguma variável numérica. Além desses, existem ainda os operadores aritméticos de atribuição, que executam uma operação aritmética e atribuem o resultado à variável na forma de uma mesma instrução.

Outras diferenças podem ser notadas nos operadores lógicos, que usam símbolos ao invés de nomes para definirem as operações.

O operador condicional ternário '?:', que ainda não havia sido apresentado, funciona como um desvio condicional simples, podendo ser aplicado em algumas situações básicas. Ele é composto por três partes, separadas por um ponto de interrogação (?) e dois pontos (:).

```
exp1 ? exp2 :  exp3;
```

A primeira expressão é uma condição, se ela for verdadeira, a segunda expressão é executada, senão, a terceira expressão é executada. A seguir são apresentados alguns exemplos de utilização.

```
//Atribui o maior valor em max.
max = (a > b) ? a : b;

//Atribui o valor absoluto de x em abs.
abs = (x > 0) ? x : -x;
```

3.7 Entrada e saída de dados em C puro

A entrada e saída de dados nativa da Linguagem C é feita através das funções:

a. `printf()`:permite a escrita de dados na saída padrão do sistema operacional (geralmente console);
b. `scanf()`: permite a leitura de dados da entrada padrão do sistema operacional (geralmente digitados no teclado).

Para a utilização destas funções é preciso utilizar símbolos adicionais especificando o tipo e/ou formato dos dados que serão lidos ou exibidos na tela, como se observa no exemplo abaixo:

```
int main()
{
     float p1, p2, p3, p4, soma;
     printf("Primeiro Programa.\n");
     printf("Digite as notas: ");
     scanf("%f%f%f%f", &p1, &p2, &p3, &p4);
     soma = p1 + p2 + p3 + p4;
```

```
        printf("O total da nota foi %f pontos.", soma);
        return 0;
}
```

Os códigos que podem ser utilizados para a leitura de dados (`scanf()`) e seus significados são apresentados no Quadro 2:

Quadro 2: Códigos utilizados na função `scanf ()`

Códigos de formatação para scanf()	Significado
%c	Caractere simples.
%d	Inteiro decimal com sinal.
%i	Inteiro decimal, hexadecimal ou octal.
%e	Notação científica.
%f	Ponto flutuante em decimal.
%g	Usa %e ou %f, o que for menor.
%o	Inteiro octal.
%s	String de caracteres.
%u	Inteiro decimal sem sinal.
%x	Inteiro hexadecimal.
%ld	Inteiro decimal longo.
%lf	Ponto flutuante longo (double).
%Lf	Double longo.

Da mesma forma, os códigos que podem ser utilizados para a escrita de dados (`printf()`) e seus significados são apresentados no Quadro 3 a seguir:

Quadro 3: Códigos utilizados na função `printf ()`

Códigos de formatação para printf()	Significado
%c	Caractere simples.
%d	Inteiro decimal com sinal.
%i	Inteiro decimal com sinal.
%e	Notação científica (e minúsculo).
%E	Notação científica (E maiúsculo).
%f	Ponto flutuante em decimal.
%g	Usa %e ou %f, o que for menor.
%G	Usa %E ou %f, o que for menor.
%o	Inteiro octal sem sinal.
%s	String de caracteres.
%u	Inteiro decimal sem sinal.
%x	Inteiro hexadecimal sem sinal (letras minúsculas).
%X	Inteiro hexadecimal sem sinal (letras maiúsculas).
%p	Ponteiro (endereço).
%n	Ponteiro inteiro.
%%	Imprime um caractere %.

Além dos códigos apresentados anteriormente, alguns códigos adicionais com funções específicas também podem ser utilizados na escrita de dados como mostrado no Quadro 4:

Quadro 4: Códigos adicionais utilizados na função `printf ()`

Códigos especiais	Significado
\n	Nova linha.
\t	Tabulação.
\b	Retrocesso (usado para impressora).
\f	Salto de página de formulário.
\a	Beep – Toque do auto-falante.
\r	CR – Retorno do cursor para o início da linha.
\\	\ – Barra invertida.
\0	Zero.
\'	Aspas simples (apóstrofo).
\"	Aspas dupla.
\xdd	Representação hexadecimal.
\ddd	Representação octal.

Para usar `printf()` e `scanf()` é necessário incluir a biblioteca `stdio.h` ao programa.

3.8 Entrada e saída de dados em C++

A entrada e saída de dados padrão da linguagem C pode ser considerada complexa por programadores iniciantes, porém, um recurso mais recente, próprio da linguagem C++, facilita esta tarefa.

Assim, por conveniência, é comum usar um recurso de C++ para fazer este tipo de entrada e saída de dados, mesmo sem a necessidade de entrar no mérito da Orientação a Objetos, que é o principal recurso adicional da linguagem em relação à linguagem C.

A linguagem C++ possui objetos de *stream*[5] que possibilitam a leitura e escrita padrão de maneira mais simplificada, estes objetos são:

a. `cout`: objeto de *stream* que permite o envio de dados para a saída padrão (geralmente monitor);
b. `cin`: objeto de *stream* que permite a leitura de dados da entrada padrão (geralmente teclado) e o seu armazenamento em variáveis ou manipulação.

Uma das grandes vantagens de se utilizar estes objetos é que eles já possuem implementados mecanismos de detecção automática de tipos, evitando que o programador tenha que definir os tipos no momento da leitura e escrita.

Para usar os objetos `cin` e `cout` é necessário incluir a biblioteca `iostream` ao programa. A forma como os dois são utilizados é apresentada nas seções seguintes.

[5] Stream são sequências de dados que podem ser gerados e consumidos por diferentes agentes. Por exemplo, um programa envia dados para um stream de vídeo, esse stream é acessado por outros agentes que irão apresentá-los na tela. O acesso deve ser sincronizado de alguma maneira.

3.8.1 Objeto std::cout

Para escrever alguma saída na tela utilizando o `cout`, basta acrescentar o comando `std::cout`, seguido de dois símbolos de menor (<<), seguido do dado que se deseja apresentar (texto direto, constantes, variáveis, etc.). Abaixo são apresentados alguns exemplos de sua utilização:

```
#include <iostream>

int main()
{
     int i;
     float p;
     i = 80;
     p = 60.8;
     std::cout << "Olá Mundo\n";
     std::cout << "Eu tenho " << i << " anos.\n";
     std::cout << "Meu peso é " << p << " quilos.\n";
     return 0;
}
```

É importante observar que quando mais de um dado precisa ser apresentado na mesma sentença, o símbolo << também precisa ser utilizado mais vezes, para que os dados sejam concatenados (juntados) na saída.

Além disso, alguns códigos especiais apresentados anteriormente, como o `\n`, por exemplo, para quebra de linha, também podem ser adicionados ao comando `cout`.

3.8.2 Objeto std::cin

Para ler dados da entrada padrão utilizando o `cin` basta acrescentar o comando `std::cin`, seguido de dois símbolos de maior (>>), seguido da

variável onde o dado será armazenado. Abaixo são apresentados alguns exemplos de sua utilização:

```
#include <iostream>

int main()
{
     int i;
     float p;
     std::cout << "Digite a sua idade: ";
     std::cin >> i;
     std::cout << "Digite o seu peso: ";
     std::cin >> p;
     std::cout << "Você tem " << i << " anos.\n";
     std::cout << "Seu peso é " << p << " quilos.\n";
     return 0;
}
```

3.8.3 Utilizando o namespace std

A utilização do `cin` e do `cout` é feita utilizando-se o termo `std`, seguindo de quatro pontos, seguido do nome do objeto `cin` ou `cout`. Isso acontece porque a biblioteca é divida em namespaces (espécie de pacotes), e os códigos estão se referindo ao namespace `std`.

É possível evitar a repetição do termo acrescentando, no início do programa, depois das inclusões das bibliotecas, o comando `using namespace std`. Tal comando informa ao programa que o namespace `std` está sendo utilizado pelo arquivo inteiro, não sendo mais necessário informar seu nome em todas as utilizações.

Abaixo é apresentado um exemplo em que o referido comando foi utilizado.

```
#include <iostream>
```

```
using namespace std;

int main()
{
     int i;
     float p;
     cout << "Digite a sua idade: ";
     cin >> i;
     cout << "Digite o seu peso: ";
     cin >> p;
     cout << "Você tem " << i << " anos.\n";
     cout << "Seu peso é " << p << " quilos.\n";
     return 0;
}
```

3.9 Comentários

Comentários em linguagem C/C++ são feitos da mesma forma como são feitos em Portugol:

a. `//` para comentários de uma única linha;
b. `/*` e `*/` para comentário de bloco.

Um exemplo ilustrando a utilização de comentários de linha e de bloco é apresentado a seguir:

```
/*
File: principal.cpp. Criado em: 01/03/2016
Autor: Rosinei Soares de Figueiredo
*/
#include <iostream>

int main()
{
```

```
        cout << "Olá Mundo";      // Exibe uma mensagem.
        return 0;
}
```

No exemplo, tem-se incialmente um comentário de bloco, composto por várias linhas. Um marcador é colocado no início no bloco de comandos e outro é colocado no final, assim, o compilador entende que deve ignorar o que estiver escrito no bloco.

Em sequência, na linha do `cout`, tem-se outro comentário, porém ocupando apenas uma linha. Nesse caso, ele foi feito utilizando o símbolo de comentário de linha, que deve ser colocado apenas no início do comentário, diferentemente do comentário de bloco. Ao encontrar o símbolo `//`, o compilador entende que, naquela linha apenas, tudo que estiver após o símbolo deve ser ignorado.

3.10 Estruturas de controle

Estruturas de controle são comandos que permitem ao programa decidir o que deve ser executado à seguir. Geralmente avaliam alguma expressão lógica e, sobre o resultado da expressão, decidem por executar ou não algum bloco de comando, ou decidem entre possíveis blocos qual deve ser executado.

A avaliação das expressões lógicas segue a mesma ideia já expressa nas seções iniciais deste documento. O que não se deve esquecer é que alguns operadores utilizam símbolos diferentes, como por exemplo, o igual relacional, que em Portugol é um igual (=) e em C/C++ são dois iguais (==), e o diferente, que em Portugol é um menor e um maior (<>) e em C/C++ é um exclamação e um igual (`!=`). Assim, é sempre importante estar atento à tabela de operadores da linguagem que se está utilizando.

Em C/C++, as principais estruturas de controle são:

a. `if`
b. `if-else`
c. `else-if`

d. `switch`

Destas estruturas, a primeira (`if`) é a tradução do desvio simples visto em Portugol, enquanto a segunda e a terceira (`if-else` e `else-if`) são traduções dos desvios compostos. A quarta (`switch`) é apenas uma maneira alternativa (com algumas restrições) de resolver desvios compostos. As estruturas serão exemplificadas mais a frente.

A estrutura `if` instrui o computador a tomar uma decisão simples, ela é formada da seguinte maneira:

```
if (Condição)
{
     Intrução;
     Instrução;
     ...
}
```

Exemplo de utilização da estrutura `if`:

```
...
int ano;
cout << "Digite um ano: ";
cin >> ano;
if (ano % 4 == 0  && ano % 100 != 0 && ano % 400 == 0)
{
     cout << "Ano bissexto.";
}
...
```

O comando `if-else` é uma expansão do comando `if`, ele permite escolher uma ação caso a condição seja atendida ou outra caso a condição não seja atendida, ou seja, quando se tem dois blocos como opção de execução. Esta estrutura é montada da seguinte maneira:

```
if (Condição)
{
     Intrução;
     Intrução;
     ...
}
else
{
     Intrução;
     Intrução;
     ...
}
```

Exemplo de utilização da estrutura `if-else`:

```
...
int ano;
cout << "Digite um ano: ";
cin >> ano;
if (ano % 4 == 0  && ano % 100 != 0 && ano % 400 == 0)
{
     cout << "Ano bissexto.";
}
else
{
     cout << "Ano normal.";
}
...
```

O comando `else-if` é uma forma de evitar comandos `if` aninhados quando se tem mais de dois possíveis blocos de execução. Ele permite a adição de quantos blocos forem necessários. Esta estrutura é formada da seguinte maneira:

```
if (Condição 1)
{
     Intrução;
     Intrução;
     ...
}
else if (Condição 2)
{
     Intrução;
     Intrução;
     ...
} else
{
     Intrução;
     Intrução;
     ...
}
```

Exemplo de utilização da estrutura `else-if`:

```
...
float soma = nota1 + nota2 + nota3 + nota4;
if (soma < 45)
{
     cout << "Estudante REPROVADO.";
}
else if (soma >= 45 e soma < 60)
{
     cout << "Estudante em RECUPERAÇÃO.";
}
else
{
     cout << "Estudante APROVADO.";
}
```

```
...
```

O comando `switch` permite a seleção entre várias alternativas de uma forma mais elegante do que em uma estrutura de `if-else`.

```
switch (variável ou expressão)
{
     case constante 1:
          Instrução;
          Instrução;
          ...
          break;
     case constante 2:
          Instrução;
          Instrução;
          ...
          break;
     case constante 3:
          Instrução;
          Instrução;
          ...
          break;
     default:
          Instrução;
          Instrução;
          ...
}
```

A seguir é apresentado um exemplo que implementa, incialmente, um trecho de programa utilizando uma estrutura `switch` e, posteriormente, um programa correspondente construído utilizando as `if`, `else-if` e `else`.

Exemplo com switch:

```
...
switch (x) {
     case 1:
          cout << "x é 1";
          break;
     case 2:
          cout << "x é 2";
          break;
     default:
          cout << "valor de x é desconhecido";
}
...
```

Exemplo correspondente utilizando `if`, `else-if` e `else`:

```
...
if (x == 1) {
     cout << "x é 1";
}
else if (x == 2) {
     cout << "x é 2";
}
else {
     cout << "valor de x é desconhecido";
}
...
```

3.11 Estruturas de repetição

Uma estrutura de repetição é uma construção que permite a execução repetida de instruções enquanto uma condição estiver sendo satisfeita. Em C/C++, bem como na maioria das linguagens, as estruturas básicas de repetição são:

a. `while`
b. `do-while`
c. `for`

Os dois primeiros comandos (`while` e `do-while`) são formas utilizadas para se implementar a estrutura `enquanto/faça` da linguagem Portugol. Já o terceiro comando (`for`) é a implementação do comando `para/até/faça`.

O laço `while` é usado geralmente quando se quer repetir um bloco de comandos vinculado a alguma condição simples, que sofre efeito da execução dos comandos do bloco. Um bloco `while` é formado da seguinte maneira:

```
while (Condição)
{
     Instrução;
     Instrução;
     ...
}
```

Enquanto a avaliação da condição resultar em verdadeiro, as instruções que compõem o bloco serão executadas. Um exemplo é apresentado a seguir:

```
...
int num, fat;
cout << "Digite um numero inteiro maior zero: ";
cin >> num;
fat = num;
```

```
while (num > 1)
{
     fat = fat * (num – 1);
     num = num – 1;
}
cout << "O fatorial do numero é: " << fat;
...
```

O laço `do-while` é semelhante ao laço `while`, a diferença é que o laço `do-while` executa as instruções uma vez, antes mesmo de verificar a condição de teste, ao contrário do laço `while`, que testa a condição antes mesmo da primeira execução. O comando `do-while` é montado da seguinte maneira:

```
do
{
     Instrução;
     Instrução;
     ...
} while (Condição);
```

A condição é avaliada depois de cada execução do bloco de comandos, assim, eles serão executados ao menos uma vez. Enquanto a condição for verdadeira, uma nova execução será feita. Um exemplo de utilização da estrutura `do-while` é apresentado na sequência:

```
...
int opcao;
do
{
     cout << "Opções:\n";
     cout << "1. Cadastrar.\n";
     cout << "2. Listar.\n";
     cout << "3. Excluir.\n";
```

```
        cout << "0. Sair.\n";
        cout << "Escolha: ";
        cin >> opcao;
    } while (opcao != 0);
    ...
```

O laço `for` é usado geralmente quando se quer repetir algo por uma quantidade fixa de vezes (manipulação de vetores e matrizes, que serão vistos mais a frente, utilizam muito a estrutura `for`), embora também possa ser adaptado para outras situações. A estrutura `for` é codificada de acordo com o modelo a seguir:

```
for (int cont = 0; cont < 10; cont = cont + 1)
{
    Instrução;
    Instrução;
    ...
}
```

O `for` é um pouco diferente das estruturas anteriores, ele não tem apenas uma condição, na verdade, ele tem três expressões dentro do parêntese após a palavra `for`.

A primeira expressão representa o ponto de partida da repetição, definido antes da repetição começar, no caso, foi declarada uma variável `cont` com valor `0`, significa que a repetição começará a contar a partir de `0`.

A segunda expressão é a condição que deve ser avaliada antes de cada repetição. Se esta expressão for verdadeira, a repetição segue, senão, ela é interrompida.

A terceira expressão refere-se ao tamanho do passo dado de uma repetição para outra, essa instrução é executada ao final de cada repetição. No caso, ela define que o contador receberá uma unidade a mais ao final de cada iteração, mas isso pode ser diferente, dependendo do programa.

Abaixo são apresentados alguns exemplos de utilização da estrutura `for`:

```
// Mostra na tela todos os números de 0 a 100.
for (int cont = 0; cont <= 100; cont = cont + 1)
{
     cout << cont << "\n";
}

// Mostra na tela todos os números pares de 0 a 100.
for (int cont = 0; cont <= 100; cont = cont + 2)
{
     cout << cont << "\n";
}
```

3.12 O problema do loop infinito

Assim como em Portugol, em C/C++ é possível a ocorrência do problema do loop infinito quando as condições de parada da estrutura não são bem elaboradas. Independente da linguagem em que se está programando, é sempre importante escrever os códigos com cuidado para que o resultado seja aquele que se espera do programa. Abaixo é apresentado um exemplo com ocorrência do problema do loop infinito:

```
/*
A ideia é que a repetição seja feita enquanto a
variável cont for diferente de 100, porém ela nunca
será 100, pois começa de 1 e aumenta de 2 em dois,
então pulará de 99 para 101, e a repetição será
infinita.
*/
...
int cont = 1;
```

```
while (cont != 100)
{
     cout << cont << "\n";
     cont = cont + 2;
}
...
```

4 ESTRUTURAS DE DADOS

Como já discutido anteriormente, a forma básica para se armazenar dados que são manipulados em programas é através da utilização de variáveis e constantes.

Porém, em algumas situações, a quantidade de dados é muito grande, criando a necessidade de se declarar muitas variáveis, o que torna o código desorganizado e complexo.

Para evitar este problema, os dados que são manipulados por um programa podem ser organizados em estruturas de dados. Estruturas de dados são mecanismos utilizados para armazenamento de conjuntos de dados utilizados nos programas.

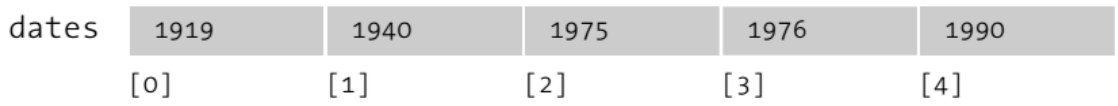
As principais estruturas de dados utilizadas nas das linguagens de programação são: **vetores**, **matrizes** e **registros**. Estas estruturas serão abordadas nas seções seguintes.

4.1 Estrutura homogênea de dados unidimensional - Vetores

Em programação, vetor (também conhecidos como arranjo ou arrays), é utilizado para se referir a um conjunto de dados de mesmo tipo, agrupados de maneira linear, e que podem ser acessados através de uma única variável ao invés de várias variáveis. Nos vetores podem ser guardados conjuntos de números, de letras, de valores lógicos, entre outros.

O vetor é criado para armazenar um conjunto de dados, mas cada dado pode ser acessado de maneira independente, indicando a posição que se quer acessar através de um número entre colchetes. Na Figura 3, tem-se um vetor “dates”, que guarda cinco valores. Note que temos as posições de 0 a 4, pois os vetores começam a serem numerados a partir do 0. Isto significa que em um vetor criado para guardar 10 elementos, existem as repartições de 0 a 9.

Figura 3: Exemplo de vetor em C/C++



Os vetores permitem que uma coleção de dados seja guardada e manipulada através de uma única variável, conforme o exemplo abaixo:

```
float notas[4];

notas[0] = 20.3;
notas[1] = 19.8;
notas[2] = 15.5;
notas[3] = 24.0;
```

Neste exemplo, quatro notas são manipuladas através de uma única variável chamada notas. São necessários alguns símbolos adicionais para que isso possa acontecer, tais símbolos são explicados mais a frente.

A maneira como os dados são arranjados dentro de um vetor (ordenados ou não, por exemplo) dependem da preferência do programador, mas é necessário considerar que os dados serão processados posteriormente, então a criação dos vetores deve facilitar tal manipulação.

A criação de um vetor é semelhante à criação de uma variável comum, porém, é necessário adicionar um colchete (**`[]`**), contendo a quantidade de valores que se deseja guardar no vetor. Essa quantidade deve ser um valor constante.

```
//Modelo
tipo nome [qtdDeElementos];

// Exemplo
int vet[10];
```

Esta linha cria um vetor chamado `vet`, onde será possível guardar até dez valores, indexados de 0 a 9. Outros exemplos, utilizando outros tamanhos e outros tipos de dados são apresentados abaixo:

```
float notas[5];
char nome[50];
double precos[10];
```

Uma vez criados, os vetores podem ser manipulados, escrevendo-se e lendo-se os valores que eles guardam. Isso deve ser feito acessando-se cada posição do vetor, como por exemplo:

```
float notas[4];
float soma;

notas[0] = 20.3;
notas[1] = 19.8;
notas[2] = 15.5;
notas[3] = 24.0;

soma = notas[0] + notas[1] + notas[2] + notas[3];
```

As linhas de código anteriores criam um vetor para guardar cinco números de ponto flutuante, depois acessam as cinco posições do vetor, guardando valores nelas.

Também é criada uma variável soma, esta variável recebe a soma de todos os valores guardados no vetor. O acesso a estes valores no momento da soma também é feito indicando-se o índice de cada posição do vetor.

Os vetores também podem receber valores no momento da sua declaração, isso pode ser feito utilizando umas das seguintes maneiras:

```
// Colocando-se a lista de valores entre {}
```

```
int numeros [5] = {16, 2, 77, 40, 120};

// Mesma forma anterior, mas omitindo a atribuição e a
// quantidade
int vetor[] {10, 20, 30};
```

Na programação, vetores combinam muito bem com laços de repetição, pois muitas vezes uma determinada ação deve ser feita repetidamente para todos os elementos de um vetor, o que pode ser feito mais facilmente com uma estrutura de repetição, como no exemplo a seguir.

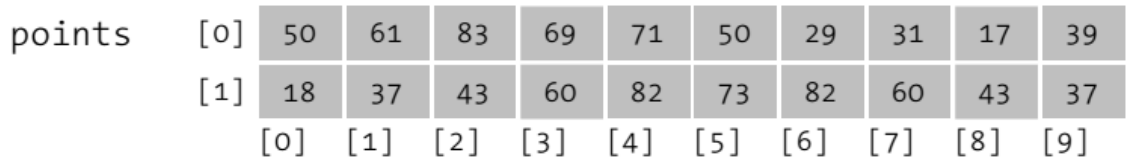
```
int v[5] = {16, 2, 77, 40, 120};

// Mostra o valor de cada número elevado ao quadrado.
for(int cont = 0; cont < 5; cont = cont + 1){
     cout << v[i] << ": " << v[i] * v[i] << "\n";
}
```

Existem, ainda, outras formas mais avançadas de manipulação de vetores que não fazem parte do escopo deste material.

4.2 Estrutura homogênea de dados bidimensional - Matrizes

Matrizes funcionam de maneira semelhante e possuem os mesmos propósitos que os vetores, sendo também utilizadas para organizar dados de mesmo tipo, porém dispostos de maneira bidimensional, semelhante ao conceito de matriz da matemática. Na Figura 4 tem-se um exemplo com duas linhas de dados, representando uma coleção de pontos geométricos bidimensionais, cada linha com dez elementos, organizados na forma de uma matriz 2 x 10 (duas linhas e dez colunas).

Figura 4: Exemplo de matriz em C/C++



A criação de uma matriz é semelhante à criação de um vetor, porém, é necessário adicionar um colchete (`[]`) a mais, sendo o primeiro para indicar o número de linhas e o segundo para indicar o número de colunas, de acordo com a quantidade de valores que se deseja guardar na matriz. O número de linhas e colunas da matriz devem ser valores constantes.

```
//Modelo
tipo nome [qtdDeLinhas][qtdDeColunas];

// Exemplos
int mat[10][10];
```

Esta linha cria uma matriz chamada `mat`, onde será possível guardar até cem valores, dispostos em 10 linhas e 10 colunas, cada linha e cada coluna é indexada de 0 a 9. Outros exemplos são apresentados abaixo:

```
float notas[5][6];
char nomes[10][50];
double precos[45][3];
```

Uma vez criadas, as matrizes podem ser manipuladas, escrevendo-se e lendo-se os valores que elas guardam. Isso deve ser feito acessando-se cada célula da matriz através do número de sua linha e do número de sua coluna, colocando-se estes índices dentro dos colchetes, como por exemplo:

```
float pontos[10][2];
```

```
pontos[0][0] = 3.0;
pontos[0][1] = 0;
pontos[1][0] = 9.8;
pontos[1][1] = 5.6;
pontos[2][0] = 5.1;
pontos[2][1] = 12.7;
...
pontos[9][0] = 13.4;
pontos[9][1] = 19.7;
```

As linhas de código anteriores criam uma matriz de 10 linhas e 2 colunas para guardar até 20 valores decimais (10 pontos de um plano cartesiano, por exemplo), depois acessam as posições da matriz, guardando valores nelas.

O acesso aos valores guardados na matriz também é feito utilizando-se o seu nome seguido de dois colchetes, indicando a linha e coluna da célula que se deseja acessar.

Assim como nos vetores, as matrizes também podem receber valores no momento da sua declaração, isso pode ser feito da seguinte maneira:

```
// Colocando-se os dados de cada linha dentro de {} e
// colocando estas linhas, separadas por vírgula,
// dentro de outra {}.
float notas[3][5] = {
     {5.7, 3.5, 9.6, 5.3, 7.9},
     {5.5, 8.0, 1.5, 6.6, 5.3},
     {4.1, 3.1, 7.2, 8.5, 7.0}
};
```

Na programação, as matrizes também combinam muito bem com laços de repetição, pois muitas vezes uma determinada ação deve ser feita repetidamente para todos os elementos de uma matriz, o que pode ser feito mais facilmente com uma estrutura de repetição.

```
float notas[3][5] = {
     {5.7, 3.5, 9.6, 5.3, 7.9},
     {5.5, 8.0, 1.5, 6.6, 5.3},
     {4.1, 3.1, 7.2, 8.5, 7.0}
};

// Mostra a soma dos valores de cada linha da matriz.
// Note que é necessário um for para percorrer cada
// coluna dentro das linhas, mas esse for deve ser
// colocado dentro de outro for, para que as linhas
// também sejam percorridas.
float totalLinha;
for(int lin = 0; lin < 3; lin = lin + 1){
     totalLinha = 0;
     for(int col = 0; col < 5; col = col + 1){
          totalLinha = totalLinha + notas[lin][col];
     }
     cout << "Total na linha: " << totalLinha << "\n";
}
```

Existem, ainda, outras formas mais avançadas de manipulação de matrizes que não fazem parte do escopo deste material.

4.3 Estrutura heterogênea de dados - Registros

Algumas vezes, dependendo da situação, os vetores e matrizes não são suficientes para se armazenar os dados necessários ao programa, principalmente quando o conjunto possui dados que não são do mesmo tipo.

Para estes casos, existe outro recurso que pode ser utilizado, o registro. Registro é um grupo de elementos de dados, que podem ser de tipos e tamanhos diferentes, agrupados sob um mesmo nome.

Em C/C++, a utilização de um determinado registro só é possível se existir um tipo vinculado a ele. Este tipo é que define quantos e quais tipos de

dados poderão ser guardados no registro (seus membros). Isso é feito utilizando a palavra `struct`.

```
struct Aluno
{
     int matricula;
     float notas[3];
     float media;
};
```

O trecho de código acima cria um novo tipo de dados (apenas para o programa atual) chamado `Aluno`. Ao declarar uma variável do tipo `Aluno`, o programador tem acesso a três variáveis contidas nele: `matricula`, `notas` (que é um vetor) e `media`. A declaração é feita da seguinte maneira:

```
Aluno al;
```

O trecho acima cria uma variável `al`, do tipo `Aluno`, esta variável dá acesso aos membros do registro através do operador `ponto(.)` da seguinte maneira:

```
Aluno a;

// colocar dados no registro
a.nmat = 456;
a.notas[0] = 7.5;
a.notas[1] = 5.2;
a.notas[2] = 8.4;
a.media = (a.notas[0]  + a.notas[1] + a.notas[2])/3;

// ler dados do registro
cout << "A media deu:  " <<  a.media;
```

Da mesma forma, pode se declarar muitas variáveis do tipo inteiro. Também é possível declarar quantas variáveis forem necessárias de um tipo registro.

5 FUNÇÕES E PROCEDIMENTOS

Para começar a construir programas em C/C++, é necessário criar um bloco principal de comandos usando a seguinte estrutura básica:

```
int main()
{
     declaração de variáveis;
     instrução;
     ...................
     instrução;
     return 0;
}
```

Um bloco de código construído conforme o modelo anterior é um subprograma. Subprogramas são muito utilizadas em programas C/C++, como forma de deixá-los modularizados, deixando os códigos mais organizados e estruturados. Em um programa podem existir várias subprogramas.

Um subprograma é um conjunto de comandos agrupados em um bloco que recebe um nome através do qual o bloco pode ser invocado/chamado.

Os subprogramas são divididos em funções e procedimentos. Funções são subprogramas que geram e devolvem algum resultado. Já procedimentos são subprogramas que executam alguns comandos, mas não devolvem resultados.

Como em C/C++, a estrutura de uma função e de um procedimento não apresenta diferença relevante, é comum o uso do termo função de maneira genérica, abrangendo as duas categorias.

As funções são muito importantes para:

a) Permitir o reaproveitamento de código já construído (pelo programador ou por terceiros);
b) Evitar que um trecho de código seja repetido várias vezes dentro de um mesmo programa;

c) Permitir a alteração de um trecho de código de uma forma mais rápida, sendo necessário alterar apenas dentro da função;
d) Permitir que os blocos do programa não fiquem grandes demais e, por consequência, mais difíceis de entender;
e) Facilitar a leitura do programa-fonte;
f) Separar o programa em partes (blocos) que possam ser logicamente compreendidas de forma isolada.

5.1 Primero contato

A primeira função apresentada é a função `main`. Como já mencionado, ela é uma função especial, pois é ponto de partida da execução do programa, sendo executada automaticamente, simplesmente é construída pelo programador, mas é executada automaticamente quando o programa executa.

```
int main()
{
     cout << "Alô Terezinha!";
     return 0;
}
```

Porém, os programas reais possuem muitas linhas de código que podem ficar desorganizadas e/ou repetitivas se colocadas todas dentro da função main, para evitar isso, são utilizadas as funções[6], ente outros recursos que serão estudados mais a frente.

Muitas funções fazem parte da estrutura básica da linguagem, disponíveis através de bibliotecas, mas muitas vezes é necessário que o programador crie mais funções para efetuar processamentos específicos dos programas que estiver implementando.

[6] Existem também muitas funções disponível em bibliotecas de funções. Estas bibliotecas podem ser adicionadas aos programas e, com isso, suas funções se tornam disponíveis nele.

5.2 Formato geral de uma função

Para a criação de funções e procedimento em linguagem C/C++, utiliza-se a seguinte estrutura:

```
tipoDafuncao  nomeDaFuncao (listaDeParametros)
{
     // corpo da função
}
```

O tipo da função serve para indicar qual tipo de dado (dentre os tipos convencionais da linguagem ou outros definidos pelo programador) a função retorna (devolve) para a função que a chamou, essa devolução é feita utilizando a palavra `return`.

O nome da função segue basicamente as regras padrões para nomes de variáveis e deve ser algo que faça lembrar a razão de existência da função, sua funcionalidade.

A lista de parâmetros, ou argumentos, é opcional e serve para indicar se a função precisa ou não receber algum valor para processar e quais são os tipos desses valores. Funciona como declarações de variáveis (uma variável para cada parâmetro), sendo que as variáveis assumirão os dados passados para a função no momento de sua invocação.

O corpo da função contém os comandos que pertencem à função, eles ficam dentro de um bloco delimitado por chaves.

Funções que não retornam nada, também conhecidas como procedimentos, devem usar o tipo `void`, podendo ainda, possuir internamente a palavra `return` sem nada a seguir, ou omitir a palavra `return`.

Funções que não recebem nenhum argumento devem deixar os parênteses após o nome vazio.

A seguir são apresentados alguns exemplos de funções:

```
/*
Uma função que calcula a soma de dois números inteiros, recebidos como parâmetro e devolve o resultado para quem a chamar.
*/
int somaDoisNumeros(int a, int b)
{
     int soma = a + b;
     return s;
}


/*
Uma função que exibe uma mensagem de alerta para o usuário através de um código recebido como parâmetro, mas não retorna nada (procedimento).
*/
void mostraAlertaDeErro(int codigoErro)
{
     switch (codigoErro)
     {
     case 1:
          cout << "Erro 1: Arquivo inacessível.";
          break;
     case 2:
          cout << "Erro 2: Arquivo corrompido.";
          break;
     case 3:
          cout << "Erro 3: Dados fora do padrão.";
          break;
     default:
          cout << "Erro não identificado.";
     }
}
```

```
/*
Uma função que executa algum processamento sem receber
parâmetros nem retornar valores (procedimento).
*/
void desenhaMenu()
{
     cout << "Opções:\n";
     cout << "1. Cadastrar.\n";
     cout << "2. Listar.\n";
     cout << "3. Excluir.\n";
     cout << "0. Sair.\n";
}
```

Importante relembrar que os códigos acima apenas criam as funções, mas isso não é suficiente para que os comandos pertencentes a elas sejam executados, apenas a função `main` é executada automaticamente, as demais precisam ser invocadas.

Indiferente se uma função já existe na linguagem, ou se foi criada durante a programação, a forma de invocar a função é a mesma, escreve-se o nome da função e um abre e fecha parênteses. Se for necessário passar algum dado para a função, esses dados são inseridos entre os parênteses, senão, os parênteses ficam vazios.

Alguns exemplos de invocação de funções, considerando as funções criadas anteriormente, são apresentados a seguir.

```
// Chama uma função com parâmetros e retorno.
int main()
{
     int resultado;
     /*
     A função é chamada, e o resultado que ela
     retornar (devolver), é jogado na variável
     resultado.
```

```
     */
     resultado = somaDoisNumeros(23, 65);
     cout << "Soma dos números deu: " << resultado;
     return 0;
}

// Chama uma função com parâmetros, mas sem retorno.
int main()
{
     int erro;
     erro = 2;
     /*
     A função é chamada com o envio de um parâmetro,
     ela faz algum processamento, mas nenhum resultado
     é devolvido.
     */
     mostraAlertaDeErro(erro);
     return 0;
}

// Chama uma função sem parâmetros e sem retorno.
int main()
{
     int opcao;
     /*
     A função é chamada, faz algum processamento, mas
     nenhum resultado é devolvido.
     */
     desenhaMenu();
     cin >> opcao;
     cout << "Você escolheu a opção " << opcao;
     return 0;
}
```

5.3 Escopo de variáveis

Relembrando, uma variável é um espaço de memória reservado para armazenar algum determinado tipo de dados e possui um nome como identificador. É possível declarar variáveis dentro das funções, a forma de declaração segue a forma padrão.

```
int num1, num2, num3;
int pX;
char sexo;
boolean dia;
```

Porém, quando se começa a trabalhar com funções, é importante observarmos o escopo das variáveis. Por **escopo de uma variável** entende-se o bloco de código onde esta variável é válida/visível. Com base nisto, tem-se as seguintes afirmações:

a) as variáveis somente são visíveis/válidas no bloco em que são definidas;
b) as variáveis definidas dentro de uma função recebem o nome de **variáveis locais**, enquanto as declaradas fora das funções são **variáveis globais**;
c) os parâmetros de uma função valem também somente dentro da função;
d) uma variável definida dentro de uma função não é acessível em outras funções, mesmo se tiverem o mesmo nome.

6 QUADRO RESUMO DE EQUIVALÊNCIAS

No Quadro 5 são apresentados os recursos básicos da linguagem Portugol e seus equivalentes em C/C++.

Quadro 5: Equivalências de Portugol e C/C++

QUADRO DE EQUIVALÊNCIAS	
PORTUGOL	C/C++
Comentários	

`// Comentário de linha.`
 	
`/*`
`Comentário de bloco.`
`*/`
	`// Comentário de linha.`
	
	`/*`
	`Comentário de bloco.`
	`*/`

Delimitadores de blocos	
início e fim	{ e }
Declaração de variáveis	

`TIPO: nome1;`
`TIPO: nome2, nome3, ...;`
	`TIPO nome1;`
	`TIPO nome2, nome3, ...;`
	`TIPO nome4 = valor;`

Declaração de constantes	

`const NOME VALOR;`	`#define NOME VALOR`
	
	ou
	
	`const TIPO NOME = VALOR;`

Principais tipos de dados	
Caractere	**char** **char16_t** **char32_t** **wchar_t**
Inteiro (com sinal)	**signed char** *signed* **short** *int* *signed* **int** *signed* **long** *int*

	signed **long long** *int*
Inteiro (sem sinal)	**unsigned char** **unsigned short** *int* **unsigned** *int* **unsigned long** *int* **unsigned long long** *int*
Real	**float** **double** **long double**
Lógico	**bool**
Operador de atribuição	
<-	=
Operadores aritméticos	

+
-
*
/ DIV
MOD
^
RAIZ
	+
	-
	*
	/
	%
	// função pow
	// função sqrt

Operadores relacionais	
= > < >= <= <>	== > < >= <= !=
Operadores lógicos	
não e ou	! && ||
Entrada de dados padrão	
leia(variável);	cin >> variável;
Saída de dados padrão	
escreva(variável ou valor);	cout << variável ou valor;

Estrutura de controle	

Desvio condicional simples
 	
se (condição) então
  lista de comandos...
fim se
	Desvio condicional simples
	
	if (condição)
	{
	  lista de comandos...
	}

Desvio condicional composto – dois blocos
 	
se (condição) então
  lista principal...
senão
  lista alternativa...
fim se
	Desvio condicional composto – dois blocos
	
	if (condição)
	{
	  lista principal...
	}
	else
	{
	  lista alternativa...
	}

Desvio condicional composto – três ou mais blocos
 	
se (condição1) então
  Intrução;
  Intrução;
  ...
senão se (condição2) então
  Intrução;
  Intrução;
  ...
senão se (condição 3) então
  Intrução;
  Intrução;
  ...
senão
  Intrução;
	Desvio condicional composto – três ou mais blocos
	
	if (condição1)
	{
	  Intrução;
	  Intrução;
	  ...
	}
	else if (condição2)
	{
	  Intrução;
	  Intrução;
	  ...
	}
	else if (condição3)
	{

Intrução;
...
fim se
	Intrução;
	Intrução;
	...
	}
	else
	{
	Intrução;
	Intrução;
	...
	}
	
	ou (em algumas situações)
	switch (variável ou expressão)
	{
	case constante1:
	Instrução;
	Instrução;
	...
	break;
	case constante2:
	Instrução;
	Instrução;
	...
	break;
	case constante3:
	Instrução;
	Instrução;
	...
	break;
	default:
	Instrução;
	Instrução;
	...
	}

Estruturas de repetição

Comando enquanto/faça

```
enquanto (condição) faça
     ...
     Lista de comandos;
     ...
fim enquanto
```

Comandos while e do-while

```
while (condição)
{
     Instrução;
     Instrução;
     ...
}
```

ou

```
do
{
     Instrução;
     Instrução;
     ...
} while (condição);
```

Comando para/até/faça (Portugol)

```
para variável de valor inicial até valor final faça
     Instrução;
     Instrução;
     ...
fim para
```

Comando for (C/C++)

```
for (ponto de partida; condição; incremento)
{
     Instrução;
     Instrução;
     ...
}
```

Exemplo:

```
for (int cont = 0; cont < 10; cont = cont + 1)
{
     cout << cont;
}
```

7 FUNÇÕES ÚTEIS EM C/C++

A seguir são apresentadas algumas funções úteis da biblioteca <*cstdlib*> (*stdlib.h*) da linguagem C/C++. Esta lista não é completa e não representa todos os recursos da linguagem, são apenas alguns recursos básicos compatíveis à natureza introdutória deste material.

7.1 Conversão de String

atof

Converte string para double.

```
double atof (const char* str);
```

atoi

Converte string para integer.

```
int atoi (const char * str);
```

atol

Converte string para long integer.

```
long int atol (const char * str);
```

atoll

Converte string to long long integer.

```
long long int atoll (const char * str);
```

7.2 Números aleatórios

rand

Gera um número aleatório.

```
int rand (void);
```

srand

Inicializa o gerador de números aleatórios.

```
void srand (unsigned int seed);
```

7.3 Ambiente

abort

Aborta o processo corrente.

```
void abort();
```

atexit

Indica uma função que será executada na saída do programa.

```
int atexit (void (*func)(void)) noexcept;
```

exit

Termina o processo chamado.

```
void exit (int status);
```

system

Executa comandos de sistema.

```
int system (const char* command);
```

7.4 Aritmética inteira

abs

Extrai o valor absoluto de um número inteiro.

```
int abs (int n);
long int abs (long int n);
long long int abs (long long int n);
```

A seguir são apresentadas algumas funções úteis da biblioteca <*cmath*> (*math.h*) da linguagem C/C++. Esta lista não é completa e não representa todos os recursos da linguagem, são apenas alguns recursos básicos compatíveis à natureza introdutória deste material.

7.5 Funções trigonométricas

cos

Calcula o cosseno de um ângulo expresso em radianos.

```
double cos (double x);
float cos (float x);
```

```
long double cos (long double x);
```

sin

Calcula o seno de um ângulo expresso em radianos.

```
double sin (double x);
float sin (float x);
long double sin (long double x);
```

tan

Calcula a tangente de um ângulo expresso em radianos.

```
double tan (double x);
float tan (float x);
long double tan (long double x);
```

acos

Calcula o arco cosseno de um ângulo expresso em radianos.

```
double acos (double x);
float acos (float x);
long double acos (long double x);
```

asin

Calcula o arco seno de um ângulo expresso em radianos.

```
double asin (double x);
float asin (float x);
long double asin (long double x);
```

atan

Calcula o arco tangente de um ângulo expresso em radianos.

```
double atan (double x);
float atan (float x);
long double atan (long double x);
```

7.6 Funções exponenciais e logarítmicas

exp

Calcula o exponencial na base e: e^x.

```
double exp (double x);
float exp (float x);
```

```
long double exp (long double x);
```

log

Calcula o logaritmo natural de x, que é na base *e*.

```
double log (double x);
float log (float x);
long double log (long double x);
```

log10

Calcula o logaritmo na base 10.

```
double log10 (double x);
float log10 (float x);
long double log10 (long double x);
```

7.7 Funções de potenciação

pow

Calcula o valor da base elevada a uma potência.

```
double pow (double base, double exponent);
float pow (float base, float exponent);
long double pow (long double b, long double e);
```

sqrt

Calcula a raiz quadrada de um valor.

```
double sqrt (double x);
float sqrt (float x);
long double sqrt (long double x);
```

cbrt

Calcula a raiz cúbica de um valor.

```
double cbrt (double x);
float cbrt (float x);
long double cbrt (long double x);
```

hypot

Calcula a hipotenusa de um triângulo retângulo de catetos x e y.

```
double hypot(double x, double y);
float hypot(float x, float y);
```

```
long double hypot(long double x, long double y);
```

7.8 Funções de arredondamento e resto

ceil

Arredonda um valor para cima.

```
double ceil (double x);
float ceil (float x);
long double ceil (long double x);
```

floor

Arredonda um valor para baixo.

```
double floor (double x);
float floor (float x);
long double floor (long double x);
```

trunc

Remove a parte decimal de um valor, deixando a parte inteira.

```
double trunc (double x);
float trunc (float x);
long double trunc (long double x);
```

round

Arredonda um valor para o valor inteiro mais próximo.

```
double round (double x);
float round (float x);
long double round (long double x);
```

7.9 Funções de máximo e mínimo

fmax

Encontra o maior entre dois valores.

```
double fmax (double x, double y);
float fmax (float x, float y);
long double fmax (long double x, long double y);
```

fmin

Encontra o menor entre dois valores.

```
double fmin (double x, double y);
float fmin (float x, float y);
long double fmin (long double x, long double y);
```

REFERÊNCIAS

CPLUSPLUS. **C++**. Disponível em <http://www.cplusplus.com/>. Acessado em 13 de março de 2016.

FORBELLONE, Andé Luiz Villar; EBERSPÄCHER, Henri Frederico. **Lógica de Programação**: a construção de algoritmos e estruturas de dados. 3. ed. São Paulo: Pearson Prentice Hall, 2005.

LOPES, Anita; GARCIA, Guto. **Introdução à Programação**. Rio de Janeiro: Elsevier, 2002.

MIZRAHI, Victorine Viviane. **Treinamento em Linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008.